The Dissertation Committee for Nader Al Awar
certifies that this is the approved version of the following dissertation:

PyKokkos: a Performance Portability Framework for Python

Committee:

Milos Gligoric, Supervisor

George Biros

Derek Chiou

Vijay K. Garg

Christopher J. Rossbach

PyKokkos: a Performance Portability Framework for Python

by

Nader Al Awar

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2024

Acknowledgments

This dissertation would not have been possible without the help and support of many people throughout my time as a Ph.D. student.

My advisor, Milos Gligoric, has been an excellent guide during the past five years. His support and mentorship throughout the this time were invaluable to the development of PyKokkos as well as my professional development. I am forever grateful to him for all I have accomplished and for all my future professional endeavors.

I would also like to thank the members of my dissertation committee, George Biros, Derek Chiou, Vijay K. Garg, and Christopher J. Rossbach, for their encouragement and feedback on my research. My discussions with George in particular were a great influence on the design of PyKokkos.

My journey as a researcher began during my undergraduate studies at the American University of Beirut, where I worked under Wassim Masri and Fadi Zaraket. Their guidance was essential for introducing me to the world of research. Additionally, during a summer research internship at the University of Illinois Urbana-Champaign, I worked under Darko Marinov and Owolabi Legunsen, whose mentorship has been a huge influence on my abilities as a researcher.

I also want to thank all the people I have collaborated with, including Steven Zhu, Kush Jain, James Almgren-Bell, Hannan Naeem, and Zijian Yi, among others. Their support was critical to the work I did.

I would also like to thank all of my mentors during my internships: Giorgis Georgakoudis and Todd Gamblin at Lawrence Livermore National Laboratory and Min Zhao, Cody Addison, Andrei Alexandrescu, Cédric Augonnet, and Michael Garland at NVIDIA. Their unwavering support, guidance, and mentorship provided me with many valuable lessons that have profoundly shaped my skills, perspectives, and aspirations, leaving a lasting and meaningful impact on the trajectory of my career.

My work on PyKokkos would not have been possible without Kokkos and other open source projects. I am extremely grateful to all members of the open source community who contributed to these projects and to the Kokkos developers particularly, specifically Christian Trott, Damien Lebrun-Grandie, Sivasankaran Rajamanickam, and Jonathan Madsen among others.

I am extremely grateful to the staff at the Electrical and Computer Engineering department at UT Austin, including Cayetana Garcia, Melanie Gulick, Barry Levitch, and Thomas Atchity, whose assistance was extremely helpful for navigating the administrative side of things.

Parts of this dissertation were published at the International Conference on Supercomputing (ICS) 2021 [9] and the International Conference on Software Engineering, Tool Demonstrations Track (ICSE Demo) [10] (Chapter 2). I want to thank all the anonymous reviewers and the audiences at these conferences for their feedback.

I received financial support from the US National Science Foundation under Grant No. CCF-1652517 and the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003969.

I am eternally grateful to all my friends over the past five years, who deserve huge thanks for their constant support. To all my cousins and family, and to Omar, Reebal, Kareem, Omar, Daniel, Aiman, Jamil, and Zayyan (among others), your support was essential to my work.

Lastly, but most importantly, I would like to thank my family for their unwavering love and support. My father Mahmoud; my mother Salwa; my brother and his family, Faisal, Rita, and Kareem; and my sister and her family, Reem, Ayoub, and Haya. None of this work would have been possible without them.

Abstract

PyKokkos: a Performance Portability Framework for Python

Nader Al Awar, PhD
The University of Texas at Austin, 2024

SUPERVISOR: Milos Gligoric

High-performance computing (HPC) hardware is becoming increasingly heterogeneous, with most modern supercomputers containing different types of processors, such as central processing units (CPUs) and graphics processing units (GPUs), from a variety of different hardware vendors, such as NVIDIA, AMD, and Intel. To enable programmers to write software that extracts the maximum possible performance from their processors, hardware vendors typically provide programming frameworks that specifically target their own hardware.

Developing software with these frameworks results in code that is tightly coupled to the targeted processor since the frameworks have different application programming interfaces (APIs) and usage guidelines. Using these frameworks, programmers write parallel, high-performance functions, which are known as kernels. The APIs allow programmers to interface with the processors while the usage guidelines provide directions on how to write kernel code that extracts the highest possible performance. Differences in these APIs and usage guidelines means that porting code from one type of processor to another requires considerable effort from programmers: they must rewrite their code to use the new framework's API and learn its usage guidelines and best practices in order to achieve good performance on the new processor. Finally, they have to maintain two versions of the same code, one for each

processor. As new processors and programming frameworks are constantly emerging, programmers must keep updating their code to take advantage of the new hardware and software, which is not a scalable approach to software development.

An alternative approach is to use programming frameworks that enable writing code that runs on different types of processors with good performance, a concept known as performance portability. One such framework is Kokkos, a performance portable programming model with a C++ implementation which aims to provide a single API that runs efficiently on different hardware. While Kokkos achieves its goals of performance portability, its availability as a C++-only library negatively impacts usability. C++ is a powerful and widely used programming language but is notorious for being difficult to use. This is especially true for scientists with no formal training in software development, a group that forms a large portion of Kokkos's user base. Instead, these users prefer higher level languages such as Python, a high-level, dynamically-typed, and interpreted language that has historically prioritized usability over performance.

This dissertation presents PyKokkos, a Python framework for writing parallel performance portable kernels, as well as PyFuser, a kernel fusion framework which provides further speedups.

Unlike C++ Kokkos, PyKokkos enables performance portability in Python by providing software abstractions that allows programmers to write their kernels entirely in Python. Internally, PyKokkos translates the Python kernel code to C++ Kokkos code, and automatically generates language bindings to allow for interoperability between Python and the generated C++ code. Using PyKokkos, we ported a number of existing C++ Kokkos examples to Python and showed that the PyKokkos kernels match the original kernels in terms of performance while being easier to write. These examples include ExaMiniMD, a $\sim$3k lines of code molecular dynamics mini-application. Furthermore, PyKokkos achieves better performance than Numba, the state-of-the-art Python library for writing kernels.

The dissertation then introduces PyFuser, a kernel fusion framework for PyKokkos. PyFuser first uses lazy evaluation to delay PyKokkos kernel execution and stores them in a trace. When the output of a kernel is accessed later, PyFuser automatically extracts the sequence of kernels that need to be executed to produce that output. PyFuser does not require any modifications to the PyKokkos code it operates on and is able to achieve speedups of 3.8× on average over the original unfused kernels.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

High performance computing (HPC) relies heavily on multi-core processors that enable high levels of parallelism. In order to fully utilize all cores on these processors, programmers must use low-level, shared memory parallel programming frameworks such as OpenMP [52], CUDA [21], HIP [33] and others. These frameworks require the user to be aware of architecture-specific details in order to write efficient and high performance code. For example, the optimal memory access pattern on a CPU typically relies on *cached* memory accesses, whereas GPUs perform better with *coalesced* memory accesses. Therefore, the optimal memory layout for multi-dimensional arrays varies across different types of processors. Additionally, each framework has its own syntax for expressing parallel execution patterns, so that an application written with one framework is tightly coupled to that framework's syntax and idioms and is not portable across different frameworks and processors.

Recently, there has been a paradigm shift in shared-memory parallel programming models to account for the issues mentioned above. Kokkos [25] and RAJA [13] are two models that provide *layers of abstraction* over existing frameworks to enable programmers to write *performance portable* code, i.e., code that runs on different types of processors with good performance. Both models include high-level abstractions for expressing common parallel execution patterns and memory layouts while hiding low-level details about the processor from the user.

While Kokkos and RAJA have achieved their goal of performance portability [30], they are implemented as C++ libraries, meaning that general *usability remains an issue*. Templates, cryptic error messages, manual memory management, complicated build processes, and other aspects of C++ make for a high barrier of entry for scientists with limited backgrounds in computer science and programming, despite scientific computing being the main use-case for HPC.

Due to these shortcomings, higher level dynamic languages such as Python and Julia [15] are preferred to C++ in the scientific computing and machine learning communities [51], both for algorithmic exploration but also increasingly for production. In the past decade, numerous libraries have been developed for writing high-performance Python code [8, 31, 50, 55, 69]. For example, the NumPy library [31] provides a high-performance multi-dimensional array type that is at the core of scientific computing in Python, while the CuPy library [50] provides the same API but targets GPUs.

We believe that a performance portability framework for Python is essential to enable more widespread usage of modern parallel processors. Performance portability is needed to enable usage of different types of processors, while the use of Python enables faster and easier development and makes these processors available to a wider audience. Furthermore, the use of a dynamic language such as Python enables opportunities for dynamic program optimizations and transformations that are typically not easily achievable with C++.

We introduce PyKokkos, the first framework for writing performance portable applications in (a subset of) Python. PyKokkos is an implementation of the Kokkos programming model. It provides an API that enables developers to write high-performance, device-portable code entirely in Python. Additionally, PyKokkos interoperates with NumPy and CuPy arrays, allowing for easy integration with existing scientific applications written in Python.

PyKokkos translates Python kernel code to C++ Kokkos. Furthermore, it automatically generates the necessary Python language bindings to interoperate between the users' Python code and the C++ code it generates. It also makes use of existing (manually-written) Kokkos bindings for memory allocations. Crucially, PyKokkos makes no changes to the Python language or its interpreter.

We then introduce an automated kernel fusion framework dubbed PyFuser to further improve the performance of PyKokkos applications. PyFuser delays execution of kernels and dynamically records *traces* of kernel invocations within a Python en-

vironment. Once a delayed kernel's outputs are accessed by the application, PyFuser lazily executes the recorded kernels by *fusing* them into a single kernel. PyFuser replaces the kernel calls recorded in the trace with a single call to the fused kernel. This fused kernel is expected to perform better than the originally recorded kernels as it will benefit from reuse of data loaded from memory, improved compiler optimizations, and reduced kernel launch overhead.

To augment the performance of the fused kernels generated by PyFuser, we also introduce optimizations during the PyKokkos code generation phase that further improve the performance of the fused kernels. Since PyKokkos generates Kokkos kernels that are optimized statically by a C++ compiler, data sharing patterns between the kernel arguments are not known, which greatly reduces the compiler's ability to optimize code. In a typical PyKokkos kernel, programmers are expected to hand optimize kernels which gives them more control over how the compiler generates code. For the automatically fused kernels PyFuser generates, programmers do not have direct access to the fused code and so these optimizations must be applied again to obtain good performance. We therefore *dynamically* analyze arguments passed to kernels and implement code transformations in PyKokkos that enable the C++ compiler to further optimize the code.

The key contributions of this dissertation include the following:

⋆ Design of a framework, dubbed PyKokkos, for writing performance portable Python code. PyKokkos is designed to closely follow the Kokkos programming model while being more concise and easier to use than C++ Kokkos. We implemented PyKokkos by combining code translation and automatic binding generation.

⋆ Design and implementation of PyFuser, a framework for automatic kernel fusion. PyFuser records traces of PyKokkos kernel calls and fuses them lazily to generate more performant kernels. We also introduce dynamic code transformations in the PyKokkos code generation phase in order to further improve the performance of the fused kernels.

- ⋆ We perform an extensive evaluation of PyKokkos and PyFuser. We evaluate PyKokkos by manually porting C++ Kokkos applications to Python and PyKokkos, including ExaMiniMD [4], a scientific application for molecular dynamics. Our results show that the kernels generated by PyKokkos can match the performance of manually written C++ kernels. We also evaluated PyFuser on the same benchmarks as well as a Particle-in-cell code originally implemented in PyKokkos [11]. Additionally, we evaluated PyFuser on a PyKokkos implementation of scikit-learn's Gaussian Naive Bayes classifier as well as NPBench, a collection of Python benchmarks. We assess the benefits of PyKokkos and PyFuser using multiple CPUs and GPUs. We also perform a deep dive into our results and report our findings for various (kernel, processor) pairs. The insights gained here are broadly applicable to other frameworks, including Kokkos and its underlying backends.
- ⋆ PyKokkos and PyFuser are both open source and are publicly available at `https://github.com/kokkos/pykokkos`.

Chapter 2: A Performance Portability Framework for Python

In this chapter, we present PyKokkos, a Python implementation of the Kokkos programming model. Kokkos provides abstractions for data management and common parallel operations, allowing developers to write portable high performance code with minimal knowledge of architecture-specific details. Kokkos was originally implemented as a heavily-templated C++ library. However, C++ is not ideal for rapid prototyping and quick algorithmic exploration. An increasing number of developers use Python for scientific computing, machine learning, and data analytics. PyKokkos enables writing performance portable applications entirely in Python. It provides Kokkos-like abstractions that are easier to use and more concise than the C++ interface. We implemented PyKokkos by building a translator from a subset of Python to C++ Kokkos and bridging necessary function calls via automatically generated Python bindings. PyKokkos is also compatible with NumPy and CuPy, two widely-used high performance Python libraries. By porting several existing Kokkos applications to PyKokkos, including ExaMiniMD (∼3k lines of code in C++), we show that the latter can achieve efficient execution with low performance overhead. [1]

2.1 Introduction

PyKokkos is a framework that enables *writing performance portable kernels entirely in Python*. It provides a domain specific language and abstractions that allow developing high-performance Python applications. Unlike existing Python libraries that operate within the same domain [8, 31, 55, 69], PyKokkos enables developers to write custom, high-performance, parallel kernels that can run efficiently on different

[1] Parts of this chapter are published at ICS 2021 [9] and ICSE DEMO 2022 [10]. I led the design, implementation, and evaluation of PyKokkos, as well as the analysis of the data and writing of the papers.

```
import pykokkos as pk

@pk.workunit
def yAx(
  m: pk.TeamMember, acc: pk.Acc[int],
  N: int, M: int, y: pk.View2D[int], x: pk.View2D[int], A: pk.View3D[int]
):
  e: int = m.league_rank()

  def team_reduce(j: int, team_acc: pk.Acc[int]):
    def vector_reduce(i: int, vector_acc: pk.Acc[int]):
      vector_acc += self.A[e][j][i] * self.x[e][i]

    tempM: int = pk.parallel_reduce(
      pk.ThreadVectorRange(m, self.M), vector_reduce)
    team_acc += self.y[e][j] * tempM

  tempN: int = pk.parallel_reduce(
    pk.TeamThreadRange(m, self.N), team_reduce)

  def single():
    nonlocal acc
    acc += tempN

  pk.single(pk.PerTeam(m), single)

# Assume E, N, M are given on command line and parsed before use
if __name__ == "__main__":
  pk.set_default_space(pk.OpenMP)
  y = pk.View([E, N], dtype=int)
  x = pk.View([E, M], dtype=int)
  A = pk.View([E, N, M], dtype=int)

  policy = pk.TeamPolicy(pk.Default, E, pk.AUTO, M)
  result = pk.parallel_reduce(policy, yAx, N=N, M=M, y=y, x=x, A=A)
```

Figure 2.1: An example of a matrix-weighted inner product kernel from the Kokkos tutorial written in PyKokkos.

processors. It *dynamically* (i.e., at run-time) translates the PyKokkos kernels into C++ Kokkos, while also automatically generating language bindings to interface between the two languages. PyKokkos compiles the C++ code it generates using any supported C++ Kokkos compiler (e.g., GCC, Clang, NVCC, etc.) and imports it into Python. The compiled code is cached on the file system so that it can be re-used during later runs instead of re-compiling. All of this is done dynamically and transparently to the programmer by the PyKokkos Runtime. Figure 2.1 shows an example of a PyKokkos kernel which is discussed in detail in Section 2.2. We implemented PyKokkos as a Python library that integrates seamlessly and naturally with existing Python code.

Prior to PyKokkos, calling a high-performance kernel from Python requires implementing it in C or C++ instead of Python itself for performance reasons. These kernels are then wrapped in manually written language bindings for interoperability with other languages, including Python. This is commonly done in practice and can be seen in some of the most popular Python packages, including SciPy [69], a Python library for scientific computing, and machine learning libraries such as TensorFlow [8] and PyTorch [55]. However, if a C++ implementation of a kernel is not available, developers have to look for alternatives.

Numba [39] is a just-in-time compiler for Python that targets LLVM [41]. Numba can target a number of different processors but does not provide high-level abstractions to hide processor-specific code, so portability remains an issue. Cython [14] is a static compiler that extends Python with C-like syntax to achieve better performance. However, these extensions make Cython a superset of Python, which may not be desirable, and Cython supports only OpenMP for parallelism at this point.

PyKokkos is the first framework for writing performance portable applications through Python. PyKokkos is an implementation of the Kokkos programming model. It provides an API that enables developers to write high-performance, device-portable code entirely in Python. Additionally, PyKokkos provides interoperability

with NumPy and CuPy arrays, allowing for easy integration with existing high performance Python applications.

To demonstrate the efficacy of PyKokkos in writing performance portable kernels, we ported a number of existing C++ Kokkos applications to Python and PyKokkos. Our results show that the kernels generated by PyKokkos match their respective C++ Kokkos kernels in terms of run-time performance while also being easier to write, while the overhead introduced by PyKokkos itself is negligible.

The main contributions of the PyKokkos project include:

⋆ Design of a framework, dubbed PyKokkos, for writing performance portable Python code. PyKokkos is designed to closely follow the Kokkos programming model while being more concise and easier to use than C++ Kokkos.

⋆ Implementation of the framework by combining code translation and automatic binding generation. PyKokkos supports three styles to write PyKokkos applications and can currently run on both CPUs and GPUs.

⋆ Evaluation of PyKokkos using a number of applications, including existing high-performance kernels and ExaMiniMD, which is a large-scale molecular dynamics application. Our results show that the kernels generated by PyKokkos can match the performance of manually written C++ kernels.

The PyKokkos source code and applications that we wrote are available at `https://github.com/kokkos/pykokkos`.

2.2 Background and Example

In this section, we first provide some background on Kokkos (Section 2.2.1), then we introduce PyKokkos via an example (Section 2.2.2).

2.2.1 Kokkos

Kokkos is a programming model that provides abstractions for writing performance portable HPC code. The two major components of the Kokkos model are *execution spaces* and *memory spaces*. Given a computing node, the processors are modeled as execution space instances, and the different memory locations are modeled as memory spaces. For example, on a machine with a CPU and a GPU, there could be two (or more) execution spaces, the CPU and the GPU, and two corresponding memory spaces, main memory and GPU memory. The other main abstractions provided by Kokkos include:

- **Execution patterns**: an execution pattern represents a *parallel operation* including parallel for, parallel reduce, and parallel scan, as well as task-based programming abstractions.
- **Execution policies**: an execution policy specifies *how* a parallel operation runs. The simplest policy is `RangePolicy`, which specifies that an operation will run for all values in a range. Another policy is the `TeamPolicy` that can be used for *hierarchical* (also known as nested) parallelism. The execution policy can also be used to set the execution space.
- **Memory layouts**: the memory layout specifies how data buffers are laid out in memory. For example, Kokkos supports column-major and row-major layouts among others.
- **Memory traits**: the memory trait specifies access properties of data buffers. For example, this could be set to `Atomic`, so that all accesses to elements of the data buffer are atomic.

The C++ Kokkos library (Kokkos for short) is a concrete instantiation of the programming model described above. The main data structure is a multi-dimensional array referred to as a `View`. It is implemented as a C++ class templated on the data type, number of dimensions, memory space, memory layout, and memory trait. It maintains a memory buffer internally and uses reference counting for automatic mem-

ory management. The following code snippet shows an example of a one-dimensional `View` of size `N` holding elements of type `int`.

```
Kokkos::View<int*> v("v", N);
```

Kokkos uses C++ *functors* to define the *computational body* of parallel operations. Functors are classes or structs that define `operator()` as an instance method. The body of this method is therefore the kernel and it represents the operation that will be executed by each thread. The following code shows a simple example of a functor that performs a reduction over all the elements of a `View`.

```
struct Functor {
  Kokkos::View<int*> v;
  Functor(Kokkos::View<int*> v) : v(v) { }
  KOKKOS_FUNCTION void operator() (int tid, int& acc) const {
    acc += this->v(tid);
  }
};
```

`KOKKOS_FUNCTION` is a macro that abstracts framework-specific function type qualifiers for portability (e.g., `__host__ __device__` for CUDA). A *work index* (`tid` in the example above) parameter representing the thread ID is included in the `operator()` method signature. Since this is a reduction operation, a scalar result must be returned, so the definition includes an additional parameter, called an *accumulator*, that is passed by reference to hold that result. The scan operation additionally requires a boolean parameter to indicate whether the scan operation is on its final pass; the final pass is used to update the elements of a `View`. The parallel for operation only requires a work index as a parameter.

All the variables and `View`s needed by a functor are defined as instance variables (see `v` in the snippet above). An alternative to functors is C++ *lambdas*, or anonymous functions. Instead of instance variables, lambdas capture all the variables they need from the scope they are defined in. Lambdas are commonly more concise than functors, but the two are otherwise equivalent.

Kokkos provides a different function for each parallel operation: `parallel_for`, `parallel_reduce`, and `parallel_scan`. These functions accept as input an execution policy (or simply the number of threads) as the first argument and a functor object or a lambda as the second argument. As mentioned before, reduce and scan return a scalar result, so their functions accept as input a third argument passed by reference to hold that result. The following code shows how the functor defined earlier is used to call `parallel_reduce`, where `N` represents the number of elements of the `View`.

```
Functor f(v);
int acc = 0;
Kokkos::parallel_reduce(Kokkos::RangePolicy<>(0, N), f, acc);
```

Kokkos implements these operations for all the HPC backends it supports, including OpenMP, CUDA, and others. *The user selects which backends to enable when invoking the compiler.* During compilation, Kokkos selects the default execution spaces from the enabled backends, the corresponding memory spaces, and the optimal memory layouts for those spaces. An application can be ported to other devices by re-compiling with the needed execution spaces.

2.2.2 PyKokkos via an Example

PyKokkos is a Python implementation of the Kokkos model that enables developers to write *performance portable Python applications*. It is implemented as a Python framework and provides an API that is similar in structure to the Kokkos API, but is as easy to use as existing Python libraries. Internally, PyKokkos translates certain parts of the application into Kokkos and C++, automatically generates Python bindings for interoperability, and compiles and imports them. It also makes use of existing bindings to Kokkos to perform memory allocation.

Figure 2.1 shows an example written entirely in Python using PyKokkos. This example is taken from the `team_vector_loop` exercise in the Kokkos tutorials repository [2], and is used to demonstrate hierarchical parallelism in Kokkos. It calculates

a matrix-weighted inner product $y^T Ax$. We manually ported this example from C++ and Kokkos to Python and PyKokkos.

The first step in writing a PyKokkos application is to import the `pykokkos` package (line 1). The `as pk` statement added after the import statement indicates that `pk` is an alias for `pykokkos`.

A PyKokkos workunit represents the body of the parallel operation. It is defined by decorating a function definition with `@pk.workunit` (line 3). Since this particular workunit is a reduction operation, its first two parameters are a work index and an accumulator variable (line 5). The work index for this workunit is of type `pk.TeamMember` since it uses hierarchical parallelism. Since the accumulator is modified in the workunit, it cannot be a primitive type in Python, so we use the `pk.Acc` class type parameterized with a specific data type. The remaining parameters are user-specified and can be used to pass `Views` and scalars to the workunit. PyKokkos provides type annotations for `Views` that include the number of dimensions, i.e., `View1D`, `View2D`, etc. up to eight dimensions (the maximum allowed by Kokkos) as well as the data type. PyKokkos also provides type annotations for fixed width types (e.g., `float32`) but native Python types can be used as well. Specifying these type annotations in the workunit signature is optional as PyKokkos can determine the types dynamically from the arguments passed to the kernel call at run-time.

The `Views` `y`, `x`, and `A` are created by calling the `View()` constructor (lines 30-32). The first argument to the constructor is a list of the `View`'s dimensions. In this example, `y` and `x` are two dimensional `Views`, and `A` is three dimensional; `E`, `N`, and `M` are arbitrary integer values provided through the command line by the user. The second argument is the data type of the `View`. Additional arguments could include memory layouts, memory spaces, and memory traits. If not specified, these are set based on the current default execution space.

The execution policy of the functor is a `TeamPolicy` (line 34) since it uses hierarchical parallelism. The first argument is the execution space, `OpenMP` in this

case since it was set as the default. The second argument is the number of *thread teams*. In Kokkos, a single thread team is a group of threads that share a common *team index*. The third argument is the size of each team; `AUTO` tells Kokkos to select the appropriate team size based on the target architecture. The final argument is the vector length, i.e., the number of threads on the final level of parallelism.

To run the workunit, `parallel_reduce` is called with the execution policy, workunit, and user-specified kernel arguments passed as arguments (line 35). When the workunit finishes execution, `parallel_reduce` returns the result of the reduction operation. This is in contrast to Kokkos, which returns the result through a variable passed by reference.

We will now discuss the body of the workunit. On the outermost *team* level, each thread obtains its team index via `league_rank()` (line 8), a value shared across threads in the same team. The second level is the thread level and the third and final level is the vector level. The operations in the inner levels are defined using nested functions (lines 10 and 11). Nested functions capture the variables that are in scope when they are defined. In this case, both functions capture `e` (the team index), and the innermost function captures `j` (the thread index). The nested functions can then be invoked by calling `parallel_reduce` with the appropriate execution policy (lines 15 and 19). Finally, one thread per team member updates the outermost accumulator variable (line 25). The `nonlocal` statement is needed in Python so that `acc` is not redefined in the nested function. Once all threads are finished executing, the reduction result is returned through the original `parallel_reduce` on line 35.

This example can be executed with CUDA by simply changing the default execution space (line 29). PyKokkos takes care of setting the proper memory spaces and layouts in the `View` constructors.

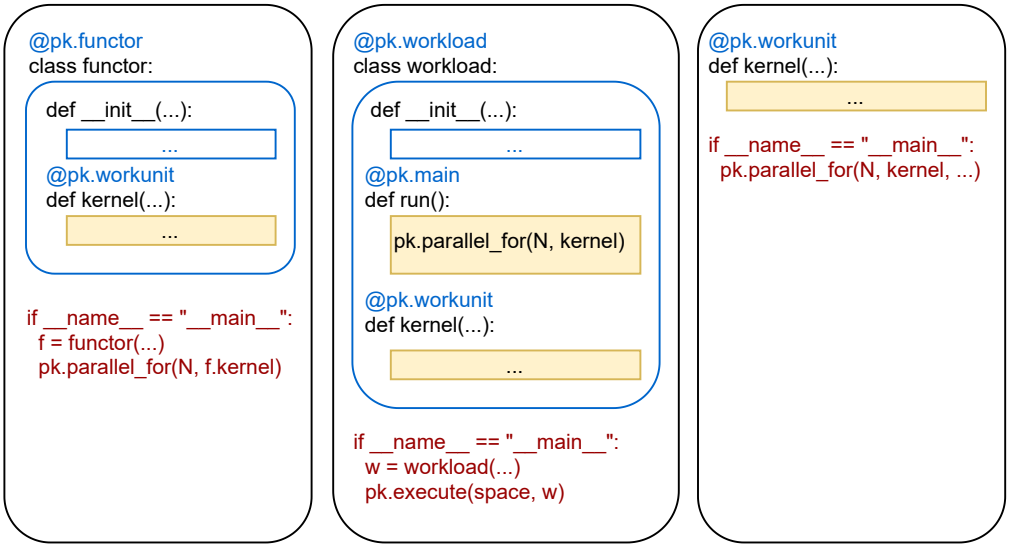


Figure 2.2: Visual summary of the three code styles supported in PyKokkos; the highlighted boxes represent the code that is translated to C++.

2.3 PyKokkos Programming Model

In this section, we first show three styles for writing PyKokkos workunits (Section 2.3.1), then we show the Kokkos features that are currently supported (Section 2.3.2), and finally we describe what Python syntax is allowed for the parts of the application that get translated to C++ (Section 2.3.3).

2.3.1 Code Styles

At present, PyKokkos supports three styles to organize workunits, which we call *ClassSty*, *ClassStyWithMain*, and *FunctionSty*. We show the differences between these styles in Figure 2.2. The highlighted boxes in each style represent the code that is translated to C++. In this section, we will describe each of three styles and show how they compare to the syntax of C++ Kokkos. Note that developers do not have to stick to a single style, as they are allowed to arbitrarily mix and match the styles across a single application.

PyKokkos uses Python decorators to annotate functions and classes that define workunits. Line 3 in Figure 2.1 illustrates the use of decorators available in PyKokkos.

2.3.1.1 ClassSty

In the ClassSty style, workunits are defined as methods, and a single class can contain one or more workunits. Each class is similar in style to a Kokkos functor, with the major difference being that workunits are annotated with `@pk.workunit` instead of the `operator()` method in C++. Only `Views` and other member variables that are defined with type-annotations in the constructor can be used in workunits. Additionally, Kokkos functions can be defined as methods inside a PyKokkos class using the `@pk.function` decorator. These methods can then be called from any workunit within the class.

2.3.1.2 ClassStyWithMain

The ClassStyWithMain style is similar to the ClassSty style except that it also contains a special method decorated with `@pk.main`, which we refer to as the PyKokkos *main method*. This method allows us to use parts of the Kokkos API for which we currently do not have bindings, such as BinSort. We add Python endpoints similar to the Kokkos API and translate those calls directly to the corresponding C++ version. This can also be used to call parallel operations, which similarly get translated to Kokkos. To execute the main method, the user calls `pk.execute(execution_space, instance)`, where `instance` is an instance of a `pk.workload` class.

2.3.1.3 FunctionSty

With this style (used in Figure 2.1), PyKokkos attempts to mimic C++ lambda usage in Kokkos. (Using Python lambdas is not an option since they are limited to a single expression unlike lambdas in C++.) The FunctionSty style al-

Table 2.1: Kokkos Features Supported in PyKokkos.

Feature	**Details**
Views	Multi-dimensional Views, Subviews, Dual Views
Memory Spaces	HostSpace, CudaSpace, CudaUVMSpace, HIPSpace
Memory Layouts	LayoutRight, LayoutLeft
Memory Traits	Atomic, RandomAccess, Restrict, Unmanaged
Execution Spaces	OpenMP, CUDA, HIP, Threads, Serial
Execution Patterns	parallel_for, parallel_reduce, parallel_scan
Execution Policies	RangePolicy, MDRangePolicy, TeamPolicy, TeamThreadRange, ThreadVectorRange, TeamThreadMDRange, WorkTag
Hierarchical Parallelism	Team Loops, Vector Loops
Atomic Operations	All atomic_fetch_[op] operations
Other	Kokkos Functions, BinSort, Timer, printf

lows standalone workunits that are defined as global functions (outside any class). In addition to the specific arguments required by each operation (e.g., accumulator for reduction), all `Views` and variables needed by the workunit are passed as type-annotated arguments. These arguments are passed to the workunit when the parallel operation is called.

We recommend using this style over the others as it is more familiar to Python programmers and will be the main style we support in the future.

2.3.2 Features

Table 2.1 shows what parts of Kokkos are supported in PyKokkos. The first column shows the names of the key Kokkos features and the second column shows the parts that are supported in PyKokkos.

PyKokkos `Views` are created through a regular constructor call (see lines 30-32 in Figure 2.1). Multi-dimensional `Views` are supported, as well as Kokkos Subviews,

which are slices of `Views` that reference a subset of an existing `View`'s data, and `View` resizing. Kokkos `DualViews` contain both a host and device buffer and are used to easily transfer data between the two. PyKokkos does not provide an abstraction for `DualViews` explicitly; instead, data is copied implicitly to device memory when necessary, as we will show in Section 2.4.2.3. This avoids burdening the user with having to explicitly copy memory between host and device memory and is in line with our view that PyKokkos can be used for rapid prototyping.

PyKokkos `Views` can be allocated in `HostSpace` (main memory), `CudaSpace` (CUDA GPU global memory), `CudaUVMSpace` (CUDA GPU unified memory), or `HIPSpace` (HIP GPU global memory). The supported memory layouts are `LayoutRight` (row-major) and `LayoutLeft` (column-major). All memory traits that are available in Kokkos are supported.

The supported Kokkos backends are OpenMP, CUDA, HIP, Threads, and Serial. In the future, other backends can be supported simply by adding API endpoints that allow the user to select them. All major loop-based execution patterns are supported. There is also support for most execution policies, including `RangePolicy`, `MDRangePolicy` (multi-dimensional range), as well as the other policies needed for hierarchical parallelism shown in Figure 2.1.

In Kokkos, `WorkTags` are used as identifiers for `operator()` methods in functors, since these methods cannot have user-defined names and a functor could have multiple workunits. Unlike Kokkos, PyKokkos identifies workunits through the `@pk.workunit` decorator (line 3 in Figure 2.1), so the user-defined kernel names can be used instead of `WorkTags`.

There is also support for various Kokkos features including some atomic operations, Kokkos functions (functions called from workunits), BinSort, the Kokkos Timer, and `printf()` in workunits.

2.3.3 Syntax Rules

PyKokkos translates all functions and classes that are annotated with `@pk.functor`, `@pk.workunit`, and `@pk.function`, which we collectively refer to as *annotated code*, to C++ Kokkos. This forces restrictions on what is allowed in annotated code. In this section, we describe these restrictions in detail.

Python is a dynamically typed language, meaning that variable types can change at run-time. On the other hand, C++ is statically typed, meaning that all variable types need to be known at compile-time and cannot be altered at run-time. Therefore, annotated code must have type annotations for all variables and `Views`; this includes both local and instance variables. However, the type annotations for workunit parameters are optional as PyKokkos can determine them at run-time when the kernel is called. One other restriction is that variables cannot be assigned to values of a different type. These restrictions do not apply outside annotated code.

Another characteristic of Python that affects translation is scoping. Whenever a function is called in Python, it creates a new local scope. Variables defined inside control blocks like `if` and `for` are scoped to the containing function. If the body of a control block contains a variable definition, then that variable can be accessed after the control block provided that it is executed. If the body of the control block is not executed, accessing the variable results in a run-time error. In C++, variables defined in control blocks go out of scope at the end of those blocks. Attempting to access these variables outside the block they were defined in results in a compile-time error. Therefore, PyKokkos annotated code has to conform to the C++ scoping rules in this regard.

Finally, not all variable types are allowed in annotated code. As of now, the types allowed are `int`, `float`, `bool`, C++ integer and floating point types of different sizes (e.g., int32_t, double, etc.), `pk.View`, and some NumPy primitive types. PyKokkos also allows user-defined classtypes that can be used in annotated code. These classtypes are Python classes with constructors and methods decorated with

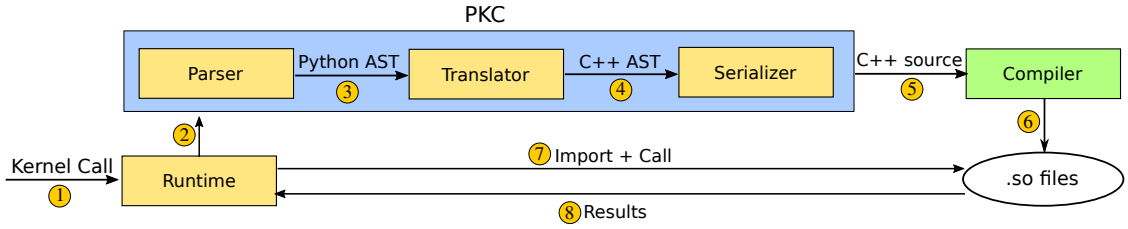


Figure 2.3: An overview of the PyKokkos framework implementation.

`@pk.function` (classtypes are therefore also considered as annotated code). Other types are not supported either because they are not necessary (strings), there is no clear C++ equivalent, or the C++ equivalent cannot be used in Kokkos code. Additionally, using modules from the Python Standard Library is not allowed in annotated code, except for several functions from the `math` module that can be mapped to C++ `cmath` functions.

In summary, PyKokkos annotated code is a subset of Python that adds restrictions to its dynamic typing, scoping rules, and allowed types in order to enable translation to C++.

2.4 PyKokkos Internals

In this section, we describe the PyKokkos framework internals. We implemented PyKokkos entirely in Python in order to allow for easy integration into existing Python codebases. Additionally, the Python Standard Library contains modules for working with the Python AST.

At a high level, PyKokkos first translates annotated code written in Python into C++ Kokkos code, compiles that code into a *shared object* file that can be imported as a Python module, and finally imports that module and calls the workunits as required. The process is illustrated in Figure 2.3. The main component of PyKokkos is PKC (Section 2.4.1), which generates C++ source files for each kernel call. At runtime, the PyKokkos Runtime component handles invoking PKC, importing modules, calling bindings, and creating `Views` (Section 2.4.2).

```
1 int bind_yAx(
2   int N, int M,
3   Kokkos::View<int **> y,
4   Kokkos::View<int **> x,
5   Kokkos::View<int ***> A,
6   int league_size, int team_size, int vector_length) {
7     // Functor is translated from Python
8     Functor functor(N, M, y, x, A);
9     int acc = 0;
10
11    Kokkos::parallel_reduce(
12      Kokkos::TeamPolicy<Functor::yAx>(
13        league_size, team_size, vector_length),
14      functor, acc);
15    return acc;
16 }
```

Figure 2.4: The wrapper function generated by PyKokkos for the inner product example.

2.4.1 PKC

During translation, PyKokkos obtains the types of the workunit parameters statically if they are specified in the code or dynamically through the arguments passed to the kernel call. The types in the workunit body are obtained statically.

Parser. PKC first calls the Parser (step ① in Figure 2.3) passing as input the files containing the annotated code. The Parser uses the `ast` module to generate an AST from the Python source. It then scans the AST to find and obtain all annotated code. All of the relevant AST nodes are then passed to the Translator (step ②).

Translator. The next step is translating the AST into C++. First, the Translator checks that the PyKokkos annotated code does not use types and Standard Library functions that are not allowed. (PyKokkos does not currently type-check annotated code but this could be done by the user using type-checkers such as MyPy [1].)

The Translator proceeds by extracting type information for all kernel parameters. C++ `View`s are templated on data type, dimensionality, memory layout, memory

space, and memory traits, so PyKokkos has to collect this information per `View`, either statically or by examining the Python `View` objects. Since the memory space depends on the execution space, PyKokkos needs to generate a different template argument per memory space. To avoid generating multiple types per memory space, we use a C++ macro that we set to the enabled execution space's memory space during compilation.

Note that regardless of the PyKokkos style used, annotated code is always translated into Kokkos functors and not lambdas, as this simplifies the translation process. The member variables of the generated C++ Kokkos functor are the kernel parameters extracted in the previous step.

The final step is to generate bindings to call the translated workunits. Since there are no existing bindings for invoking the parallel operations, we cannot call them directly from Python. To solve this, the Translator creates *wrapper functions* that call the parallel operations internally. Figure 2.4 shows the wrapper function generated for the example shown in Figure 2.1. The arguments of the wrapper are the members extracted in the previous step and are passed to the functor constructor (line 8). The wrapper then calls parallel reduce (line 11) and returns the result (line 15). The Translator then binds these wrappers using the C++ `pybind11` library [58]. The Translator passes the C++ AST to a Serializer (step ③) which generates a source file and passes it to a C++ compiler (step ④) which compiles it into a shared object file (step ⑤).

Finally, PKC calls a C++ compiler to compile the generated source file. If a CPU execution space is selected (e.g., OpenMP), PKC uses a host compiler such as GCC or Clang. If a GPU space is selected (e.g., CUDA or HIP), PKC uses a device compiler such as NVCC or HIPCC.

2.4.2 Runtime

The PyKokkos API can be divided into two groups: an interface for executing code and an interface for `Views`. First, we show how the PyKokkos Runtime (and by extension Kokkos) is initialized. Second, we show how the Runtime invokes parallel operations. Third, we discuss how `Views` are created and shared between Python and C++. Finally, we describe how annotated code can be run sequentially in Python, which can help debug kernels.

2.4.2.1 Initialization

PyKokkos is initialized when the `import pykokkos` statement is executed. This creates all the necessary entities that are needed by PyKokkos at run-time: the Runtime, Parser, Translator, and Serializer. Additionally, PyKokkos internally calls `Kokkos::initialize()`. This initializes all Kokkos internal objects and acquires hardware resources. PyKokkos also registers `Kokkos::finalize()` to be called when Python terminates.

2.4.2.2 Parallel Execution

To call a parallel operation, the user has to pass in a workunit and execution policy. This workunit can either be a method in an initialized object, i.e., ClassSty, or a free function, i.e., FunctionSty. For the latter, the user also passes in all the necessary arguments. For the former, the Runtime automatically extracts these arguments from the class members. The ClassStyWithMain style does not require an execution policy since it can execute multiple workunits, each of which could potentially have a different execution policy specified at each kernel call in the `@pk.main` method.

The Runtime then checks whether a module (i.e., the shared object binary file containing the compiled kernel) corresponding to the workunit has already been generated with PKC. If not, this means that the compile-time phase was skipped by the user, so the Runtime has to call PKC (step ⑥).

The Runtime can then import the module and call the necessary wrapper function (step ⑦). For ClassSty and FunctionSty the execution policy passed by the user provides additional arguments that are passed on to the wrapper function, where they are used to construct the execution policy object (e.g., line 12 in Figure 2.4).

The wrapper function instantiates the Kokkos functor and execution policies, and then calls the necessary parallel operations. After execution terminates, the Runtime transfers the results of all reduction and scan operations back to Python. For FunctionSty and ClassSty there is only a single result that will be returned directly by the wrapper function (step ⑧). For ClassStyWithMain, there could be multiple calls to parallel reduce or scan, so the result of each operation is added to a `View` that the Runtime can access.

2.4.2.3 Views

PyKokkos `Views` are classes created through regular constructor calls (see lines 30-32 in Figure 2.1). Similar to Kokkos, the user is not expected to set the memory space and layout of a PyKokkos `View` for portability reasons. Instead, PyKokkos selects these based on the current default execution space. For the CPU execution spaces such as OpenMP, the memory space is always set to `HostSpace`. For GPU execution spaces such as CUDA or HIP, PyKokkos does not use `CudaSpace` or `HIPSpace` since they are not accessible from Python. It has to select a host accessible memory space i.e., `HostSpace` or `CudaUVMSpace` (Unified Virtual Memory [20]). At run-time, `HostSpace Views` are copied to `CudaSpace` or `HIPSpace` as needed. This approach allows the user to switch between different execution and memory spaces without worrying about where the data is located in memory.

When the PyKokkos `View` constructor is called, it invokes the C++ Kokkos `View` constructor internally through the available Python bindings [58]. This constructor allocates the memory for the `View` data buffer and the binding returns a Python object that provides access to the underlying data buffer through a NumPy

array. The returned object can be passed by reference between C++ and Python through `pybind11`.

Additionally, we implemented a binding for another Kokkos `View` constructor that accepts as input an allocated memory buffer. This allows users to create a `View` from a pre-existing NumPy or CuPy array, avoiding the cost of allocating memory again. Additionally, for CuPy arrays, PyKokkos uses the existing GPU memory buffer allocated by CuPy for the array to avoid re-allocating GPU memory and copying data on kernel calls.

The PyKokkos `View` type is therefore a wrapper over NumPy and CuPy array. Its purpose is to provide an interface that is similar to the Kokkos `View` interface, specifically the constructor. Otherwise, it behaves as a regular NumPy or CuPy array in Python. This allows PyKokkos to be easily added to existing Python codebases.

2.4.2.4 Pure Python Execution

Since valid annotated code is a subset of valid Python code, PyKokkos supports execution of workunits in Python. This is especially helpful for *debugging logic-based errors* in Python rather than C++ due to the dynamic nature of Python.

We implement calls to parallel operations using sequential for loops. In every iteration, we pass the current iteration counter to the workunit as the thread ID. To support hierarchical parallelism, we pass an object which provides access to the thread and team ID. `MDRangePolicy` iterates over multiple ranges, so we loop over a combination of two thread IDs. In reduce and scan operations, the `pk.Acc` object wraps the result as a substitute for Python's lack of reference types for primitives. We overloaded the arithmetic operators of `pk.Acc` so it can behave like a regular primitive type without any extra function calls.

2.5 Evaluation

In this section, we present the results of our evaluation of PyKokkos. First, we show how PyKokkos performance compares to C++ Kokkos for smaller applications where the running time is dominated by kernel execution. Second, we compare PyKokkos and Kokkos performance for a larger application. Third, we report the cost of pure Python execution of PyKokkos (i.e., Python sequential execution). Fourth, we compare the PyKokkos code to Kokkos code in terms of the lines of code and number of characters. Finally, we briefly compare PyKokkos with Numba.

2.5.1 Evaluation Setup

We ran all experiments on an Ubuntu 18.04.5 machine with a 6-core Intel i7-8700 3.20GHz CPU and 64GB RAM and an NVIDIA GeForce RTX 2080 GPU with 8GB of memory. For all our experiments, we used Python 3.8.3, Kokkos 3.1.01, OpenMP 4.5, CUDA 10.2, GCC 7.5, and Numba 0.51.

2.5.2 Subjects

For the purposes of our experiments, we ported existing C++ Kokkos applications to PyKokkos. We implemented 7 exercises from the official Kokkos tutorials repository [2]. All exercises follow a structure similar to the example in Figure 2.1: calculate a matrix-weighted inner product using an outer loop and inner loop, each of which performs a reduction operation. Each exercise introduces a feature that improves on the previous exercise. The exercises we did not port use features that we do not currently support or are not relevant to PyKokkos, e.g., 01 uses `malloc()` instead of `Views` (so it is not meaningful to port to Python). Specifically, we ported 02, 03, 04, subview, mdrange, team_policy, and team_vector_loop:

- **02**: Introduces `Views` and uses the `View` constructors instead of `malloc()` in 01.
- **03**: Introduces device (i.e., GPU) `Views` and shows how memory is copied between host and device.

- **04**: Introduces memory spaces, layouts, and `RangePolicy`.
- **mdrange**: Introduces `MDRangePolicy` to initialize matrix A.
- **subview**: Introduces `subview` to split each column of A into a one-dimensional `View` that can be accessed separately.
- **team_policy**: Introduces two-level hierarchical parallelism by replacing the inner sequential reduction with a parallel version that uses `TeamPolicy`.
- **team_vector_loop**: Increases the dimensionality of each view and introduces three-level hierarchical parallelism using `Team-Thread-Range` (shown in Figure 2.1).

We also implemented the NSTREAM, Stencil, and Transpose kernels from the Parallel Research Kernels (or PRK) repository [35]; the bytes_and_flops, gups, and gather benchmarks from the official Kokkos repository; and BabelStream [22]. Finally, we ported ExaMiniMD [4], a ∼3k lines of code molecular dynamics application, entirely to Python (and PyKokkos). We excluded code from the original implementation (which is written entirely in C++) that was not executed by the inputs provided in the repository. For all PyKokkos code, we used the ClassStyWithMain style. All kernel execution times were collected with the Simple Kernel Timer from the kokkos-tools repository [3].

2.5.3 Performance: Small Applications

In this section, we compare the performance of PyKokkos to Kokkos for smaller applications where the running time is dominated by kernel execution. All values shown (e.g., execution time) represent the mean of three runs. Additionally, each application runs the kernel 1,000 times. All CUDA execution times are using CUDA device memory (i.e., `CudaSpace`).

Tables 2.2 and 2.3 show execution time for all applications using OpenMP and CUDA respectively. The first column shows the name of the application. The second column shows the sizes we used for the largest input `View` per application. For the tutorial exercises, this `View` is A. The rest of each table shows the execution time of

Table 2.2: Comparison of Execution Time of PyKokkos and Kokkos Applications with OpenMP.

Application	**Size**	**OpenMP Time [s]**				
		Kernel			Total	
		PyKokkos	Kokkos	Ratio	PyKokkos	Kokkos
02	$2^{18}\times 2^{10}$	70.3	69.5	**1.01** ×	71.8	69.8
	$2^{19}\times 2^{10}$	140.5	139.2	**1.01** ×	142.4	139.8
03	$2^{18}\times 2^{10}$	69.8	69.5	**1.00** ×	71.3	69.8
	$2^{19}\times 2^{10}$	139.5	139.2	**1.00** ×	141.3	139.8
04	$2^{18}\times 2^{10}$	69.6	69.5	**1.00** ×	71.2	69.8
	$2^{19}\times 2^{10}$	139.5	139.2	**1.00** ×	141.5	139.8
mdrange	$2^{18}\times 2^{10}$	70.2	69.5	**1.01** ×	72.9	69.8
	$2^{19}\times 2^{10}$	141.2	139.2	**1.01** ×	145.4	139.8
subview	$2^{18}\times 2^{10}$	69.7	69.5	**1.00** ×	71.2	69.8
	$2^{19}\times 2^{10}$	139.9	139.2	**1.01** ×	141.7	139.8
team_policy	$2^{18}\times 2^{10}$	69.8	69.6	**1.00** ×	71.3	69.9
	$2^{19}\times 2^{10}$	139.9	139.4	**1.00** ×	141.6	140.0
team_vector_loop	$2^{8}\times 2^{10}\times 2^{10}$	70.6	70.4	**1.00** ×	72.1	70.7
	$2^{9}\times 2^{10}\times 2^{10}$	141.0	140.5	**1.00** ×	142.7	141.1
nstream	$2^{27}\times 1$	143.8	144.6	**0.99** ×	145.6	145.1
	$2^{28}\times 1$	286.7	287.9	**1.00** ×	289.0	288.9
stencil	$2^{12}\times 2^{12}$	15.9	15.7	**1.01** ×	26.5	25.3
	$2^{13}\times 2^{13}$	63.1	62.1	**1.02** ×	102.5	100.2
transpose	$2^{12}\times 2^{12}$	23.9	24.0	**1.00** ×	25.2	24.1
	$2^{13}\times 2^{13}$	95.4	95.8	**1.00** ×	96.8	96.1
bytes_and_flops	$2^{12}\times 2^{10}$	127.2	129.8	**0.98** ×	128.4	129.8
	$2^{13}\times 2^{10}$	254.4	259.5	**0.98** ×	255.6	259.5
gather	$2^{21}\times 2^{5}$	112.4	111.2	**1.01** ×	114.0	111.3
	$2^{22}\times 2^{5}$	223.3	222.7	**1.00** ×	225.4	222.9
gups	$2^{27}\times 1$	104.0	104.0	**1.00** ×	105.5	104.3
	$2^{28}\times 1$	207.2	204.9	**1.01** ×	209.0	205.7
BabelStream	$2^{24}\times 1$	71.3	71.5	**1.00** ×	72.5	71.9
	$2^{25}\times 1$	143.0	144.2	**0.99** ×	144.3	144.8

Table 2.3: Comparison of Execution Time of PyKokkos and Kokkos Applications with CUDA.

Application	**Size**	**CUDA Time [s]**				
		Kernel			Total	
		PyKokkos	Kokkos	Ratio	PyKokkos	Kokkos
02	$2^{18}\times 2^{10}$	5.2	5.3	**0.98** ×	7.6	6.2
	$2^{19}\times 2^{10}$	10.7	10.7	**1.00** ×	14.2	11.6
03	$2^{18}\times 2^{10}$	5.2	5.3	**0.98** ×	7.5	7.3
	$2^{19}\times 2^{10}$	10.7	10.7	**1.00** ×	14.0	13.8
04	$2^{18}\times 2^{10}$	5.2	5.3	**0.98** ×	7.5	7.2
	$2^{19}\times 2^{10}$	10.7	10.7	**1.00** ×	14.1	13.8
mdrange	$2^{18}\times 2^{10}$	5.2	5.3	**0.98** ×	7.3	6.2
	$2^{19}\times 2^{10}$	10.7	10.7	**1.00** ×	13.7	11.6
subview	$2^{18}\times 2^{10}$	5.2	5.3	**0.98** ×	7.5	7.3
	$2^{19}\times 2^{10}$	10.7	10.7	**1.00** ×	14.0	13.8
team_policy	$2^{18}\times 2^{10}$	5.3	5.3	**1.00** ×	7.6	7.3
	$2^{19}\times 2^{10}$	10.4	10.4	**1.00** ×	13.7	13.5
team_vector_loop	$2^{8}\times 2^{10}\times 2^{10}$	7.9	8.0	**0.99** ×	10.2	9.9
	$2^{9}\times 2^{10}\times 2^{10}$	15.9	15.9	**1.00** ×	19.3	19.1
nstream	$2^{27}\times 1$	10.6	10.6	**1.00** ×	13.2	11.5
	$2^{28}\times 1$	21.1	21.1	**1.00** ×	25.3	22.1
stencil	$2^{12}\times 2^{12}$	4.0	4.0	**1.00** ×	6.4	6.1
	$2^{13}\times 2^{13}$	15.7	16.0	**0.98** ×	22.0	21.5
transpose	$2^{12}\times 2^{12}$	1.7	1.7	**1.00** ×	2.9	2.6
	$2^{13}\times 2^{13}$	6.5	6.5	**1.00** ×	8.2	7.4
bytes_and_flops	$2^{12}\times 2^{10}$	53.0	53.7	**0.99** ×	54.2	54.5
	$2^{13}\times 2^{10}$	103.6	105.3	**0.98** ×	104.8	106.1
gather	$2^{21}\times 2^{5}$	32.3	32.6	**0.99** ×	34.0	33.4
	$2^{22}\times 2^{5}$	64.3	65.7	**0.98** ×	66.6	66.5
gups	$2^{27}\times 1$	2.5	2.5	**1.00** ×	4.7	4.1
	$2^{28}\times 1$	5.0	5.0	**1.00** ×	8.2	7.2
BabelStream	$2^{24}\times 1$	4.1	4.1	**1.00** ×	5.4	5.3
	$2^{25}\times 1$	8.1	8.1	**1.00** ×	9.6	9.7

the main kernel and the total execution time of PyKokkos and Kokkos. The Ratio columns show PyKokkos kernel execution time relative to Kokkos.

The results show that PyKokkos can achieve performance parity with Kokkos for these applications. By comparing kernel execution time for both PyKokkos and Kokkos across both backends, it can be seen that kernel code generated by PyKokkos can match the corresponding Kokkos version for performance. Any slight difference can likely be attributed to the overhead caused by running the Python interpreter concurrently with the kernels. For the CUDA backend, this effect is less pronounced since GPU execution is not as affected by the Python interpreter.

To measure the overhead introduced by PyKokkos, we compare the total running time to kernel execution time. It can be seen that for these applications, the overhead introduced by the PyKokkos Runtime and Python itself is minimal. (The stencil application total time is much longer than kernel time for both PyKokkos and Kokkos since it calls a different kernel to increment the input `View` each iteration.) Additionally, the overhead introduced by the Python interpreter on the total execution time is negligible, as these applications spend extremely little time in non-PyKokkos Python code.

In summary, PyKokkos can match Kokkos for smaller applications dominated by kernel execution time. We expect this solid performance for all applications where kernel execution time dominates the time spent inside the Python interpreter.

2.5.4 Performance: ExaMiniMD

In this section, we compare the performance of PyKokkos to Kokkos for ExaMiniMD. We measure PyKokkos execution time without including the translation and compilation time of the kernels, i.e., PyKokkos imports compiled kernels cached on the file system from a previous run. ExaMiniMD first reads an input file and initializes the position, velocity, and force `Views` in a sequential for loop. The size of these `Views` is $\#atoms \times 3$. It then executes another sequential for loop for 100 time

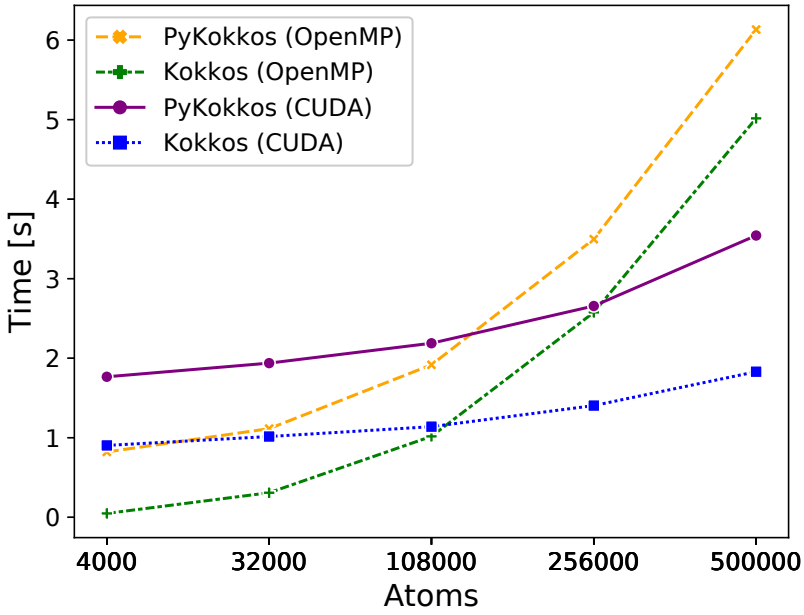


Figure 2.5: ExaMiniMD total execution time.

steps, updating the position, velocity, and force `Views` and calculating the temperature, potential energy, and kinetic energy values by calling parallel kernels.

In our initial PyKokkos implementation of ExaMiniMD we observed relatively large execution times, around 18s using OpenMP for the largest size (x-axis) shown in Figure 2.5. We profiled our implementation and discovered that the total execution time was dominated by the sequential for loop that initializes the `Views`, not the kernels written in PyKokkos. Since Python is an interpreted language, sequential

Table 2.4: ExaMiniMD Performance Metrics for the Largest Number of Atoms in Figure 2.5.

Metric	**OpenMP**		**CUDA**	
	PyKokkos	Kokkos	PyKokkos	Kokkos
Loop Time [s]	4.90	4.51	2.15	0.86
Total Time [s]	6.12	5.02	3.60	1.83
Atomsteps/s [1/s]	1.02e+07	1.11e+07	2.33e+07	5.78e+07

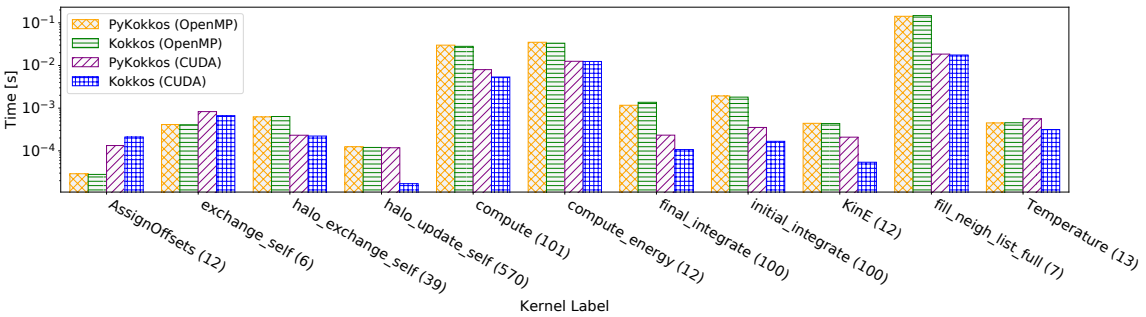


Figure 2.6: ExaMiniMD kernel time for the largest number of atoms in Figure 2.5. Number of kernel calls is shown in parentheses.

loops with large iteration counts (e.g., *#atoms* in ExaMiniMD) have significantly more overhead than in C++. We rewrote the initialization loop using Numba [39], a JIT compiler that translates Python to LLVM, to optimize the for loop. This resulted in performance comparable to the C++ for loop.

Figure 2.5 shows a plot of the total execution time vs. number of atoms. We used Unified Memory for all CUDA runs. For both OpenMP and CUDA, we observe performance comparable to Kokkos. The extra performance overhead in the PyKokkos implementation does not substantially increase as the size increases.

To understand this overhead, we first look at the kernel execution times shown in Figure 2.6. For all PyKokkos kernels, we observe minimal to no overhead compared to Kokkos. This result is in agreement with the results observed for the kernels in tables 2.2 and 2.3.

Table 2.4 shows performance metrics collected during execution: loop time is the amount of time spent in the main loop (that runs for 100 time steps), total time is end-to-end execution time, and atomsteps per second is the number of atoms multiplied by time steps per second. In addition to kernel execution time, these metrics include time spent during Python execution. Here, we observe larger performance differences between PyKokkos and Kokkos than in the kernels themselves. Thus the additional overhead observed in the loop time and total time can be attributed to time spent in the Python interpreter, outside of the generated kernels.

Table 2.5: Comparison of Pure Python Execution to OpenMP and CUDA in PyKokkos.

Application	**Size**	**PyKokkos Time [s]**		
		Python	OpenMP	CUDA
02	$2^8 \times 2^{10}$	169	1.1	1.2
03	$2^8 \times 2^{10}$	167	1.1	1.3
04	$2^8 \times 2^{10}$	169	1.1	1.2
mdrange	$2^8 \times 2^{10}$	173	1.1	1.3
subview	$2^8 \times 2^{10}$	139	1.1	1.2
team_policy	$2^7 \times 2^{10}$	194	1.1	1.1
team_vector_loop	$2^1 \times 2^7 \times 2^{10}$	245	1.2	1.1

2.5.5 Pure Python Execution

In this section, we report the cost of pure Python execution in PyKokkos (Section 2.4.2.4). Since all kernels are executed using Python sequential loops, we expect substantial performance overhead. We use the tutorial exercises to highlight the cost of each feature individually. Table 2.5 shows a comparison of total execution time using different PyKokkos backends. We set the timeout to 300s and show the largest size that completes within this budget. Clearly, this mode should be used only for debugging logical errors, as it does not escape the Python interpreter and provides users with a familiar debugging environment.

2.5.6 Code Characteristics

Table 2.6 shows basic code characteristics of the applications used in our experiments. The first column shows the source of the applications. We do not use the benchmarks since they include additional boilerplate for initialization or BabelStream since it includes code for other frameworks. The second and third columns show the lines of code (LOC) and number of characters (NOC) for Kokkos and PyKokkos, respectively. For the Tutorials and PRK rows, we show a single entry that is the

Table 2.6: Code Characteristics of PyKokkos and Kokkos Applications. Numbers for Tutorials and PRK show Total for all Applications in those Groups.

Application	**PyKokkos**		**Kokkos**		**Reduction [%]**	
	LOC	NOC	LOC	NOC	LOC	NOC
Tutorials	503	15758	592	18627	15	15
PRK	290	10004	385	11379	24	12
ExaMiniMD	2846	94811	3269	113210	12	16

summation of the values for each individual application. The fourth column shows the reduction in code size of the PyKokkos implementation compared to Kokkos.

Table 2.6 shows that PyKokkos code is more concise than Kokkos. We identify several reasons. First, Kokkos applications have to add code to initialize and finalize the Kokkos context. In PyKokkos, this is hidden from the user. Second, C++ naturally tends to be more verbose than Python. Static typing in particular contributes significantly to code clutter, even more so when templates and nested namespaces are involved. Some Kokkos applications include `typedef` and `using` declarations to avoid repeating long types, but even that still adds to the clutter. In contrast, type annotations are optional in Python (outside of PyKokkos annotated code), and dynamic typing subsumes the need for templates. Third, in C++, header files need to be included for string manipulation, IO, and other functionality, most of which is available in Python without any imports. Parsing command line arguments in C++ needs to be done through string comparison and large contiguous blocks of if statements, while in Python, this can be done with the `argparse` module from the Standard Library.

2.5.7 Numba Comparison

In this section, we compare PyKokkos to Numba. Specifically, we are interested in examining the effort required to write kernels targeting CPUs and GPUs in each framework. Of all of our test subjects, only the PRK applications have existing

Table 2.7: Comparison of Execution Time of PyKokkos and Numba Applications with OpenMP and CUDA.

Application	**Size**	**OpenMP Time [s]**		**CUDA Time [s]**	
		PyKokkos	Numba	PyKokkos	Numba
nstream	$2^{28}\times$ 1	289.0	290.2	25.3	25.3
stencil	$2^{13}\times2^{13}$	102.5	106.1	22.0	22.4
transpose	$2^{13}\times2^{13}$	96.8	103.1	8.2	8.3

Numba implementations. However, the kernels do not make use of the parallelism features in Numba, so we modified them by setting `parallel=True` and using `prange`. We also made further changes to get performance closer to the PyKokkos implementation, but we note once again that our goal is not to provide a complete performance comparison between the two, and that both implementations could be optimized further. For stencil and transpose, we manually implemented tiling in the Numba kernels to get better performance. This was not needed in the PyKokkos implementations due to the availability of `MDRangePolicy`, which provides a multi-dimensional iteration space with tiling.

We also implemented the kernels using CUDA through Numba. This required us to to use syntax specific to CUDA and to manually set the number of threads and blocks at each kernel launch.

Table 2.7 shows a comparison of total execution times. For all kernels, we observe similar execution times. All PyKokkos kernels use one common code for both OpenMP and CUDA, while for Numba, we had to re-implement the kernels for each device and add loop tiling for the CPU kernel. PyKokkos kernels are therefore more performance portable.

2.6 Conclusion

We presented PyKokkos, a new Python framework for writing performance portable applications entirely in Python. PyKokkos provides Kokkos-like abstractions that are easier to use and more concise than the C++ interface. We implemented PyKokkos by building a translator from PyKokkos annotated code to C++ Kokkos and bridging necessary function calls via automatically generated Python bindings. Our results showed that PyKokkos can obtain performance close to Kokkos for applications that are dominated by kernel execution time. PyKokkos applications are more concise than their Kokkos counterparts, and can achieve comparable performance in most cases. Kokkos provides a performance portability programming ecosystem, and we believe that PyKokkos enables developers to utilize such an ecosystem.

Chapter 3: Fusing Performance Portable Python Kernels

In this chapter, we present PyFuser, a framework for PyKokkos kernel fusion. While PyKokkos promises to increase productivity, the way developers organize their parallel code by splitting it into separate kernels may result in suboptimal performance. Kernels that operate on the same data will end up doing redundant work, since they will likely access the same memory and do similar computations. Furthermore, splitting kernels excessively leads to more kernel calls, which increases the runtime performance overhead introduced by PyKokkos and the underlying frameworks it uses. PyFuser addresses these issues by automatically fusing performance-portable PyKokkos kernels. PyFuser dynamically traces kernel calls and lazily fuses them once the result is requested by the application. The generated fused kernels are expected to execute faster due to better reuse of data, improved compiler optimizations, and reduced kernel launch overhead. We also introduce automated code transformations that further optimize the fused kernels generated by PyFuser. Our experiments on HPC applications show that PyFuser achieves substantial speedups over the original unfused kernels on NVIDIA and AMD GPUs, as well as Intel and AMD CPUs.

3.1 Introduction

HPC frameworks such as Kokkos and PyKokkos require developers to write their parallel code in standalone kernels (or functions) that are separate from the rest of the code. Dividing parallel code across different kernels is typically done to increase code reuse and improve maintenance. While this greatly improves usability, separating code in this way might result in suboptimal performance, because compilers do not know statically in what ways those kernels will be invoked and do not optimize across kernel boundaries. As a result, redundant computations and memory

accesses are frequently executed in different kernels, which would drive the developer to *fuse* different kernels into one. In practice, finding opportunities for kernel fusion can be tedious, especially in larger codebases with dozens of kernels.

We present PyFuser, *a framework for dynamic fusion of performance portable kernels written in PyKokkos*, with the goal to improve the performance of kernel code. PyFuser delays execution of kernels and dynamically records *traces* of kernel invocations within a Python environment. Once a delayed kernel's outputs are accessed by the application, PyFuser first *fuses* the recorded kernels. PyFuser then replaces kernel calls recorded in the trace with calls to the fused kernels, which should perform better than the original kernels as they benefit from reuse of data loaded from memory, improved compiler optimizations, and reduced kernel launch overhead.

We also introduce code transformations during the PyKokkos code generation phase to further improve the performance of the fused kernels. Since PyKokkos generates Kokkos kernels that are optimized statically by a C++ compiler, data sharing patterns between kernel arguments are not known, reducing the compiler's ability to optimize code. When writing a PyKokkos kernel, the programmer hand optimizes the code by fusing loops and eliminating redundant memory operations. For the fused kernels that PyFuser automatically generates, these optimizations cannot be applied by the programmer. We therefore implemented code transformations in PyKokkos that require run-time information about the arguments that are passed to the kernels. These code transformations help the C++ compiler optimize the code PyKokkos generates more effectively.

We assess the benefits and limitations of PyFuser using a number of existing HPC applications written in PyKokkos, including a Particle-in-cell (PIC) code [11], ExaMiniMD [4], a Gaussian Naive Bayes classifier, NPBench [77], and benchmarks we extracted from the examples in the PyKokkos repository. The fused kernels PyFuser generates achieve speedups of 3.8× on average over the original unfused kernels across all processors.

The key contributions of this chapter include:

⋆ **Framework**. We present PyFuser, a framework for automatic kernel fusion. PyFuser records traces of kernel calls and fuses them to generate more performant kernels. We also introduce dynamic code transformations to PyKokkos that further improve the performance of fused kernels.

⋆ **Evaluation**. We perform an extensive evaluation of PyFuser on a number of PyKokkos kernels. We assess the benefits of PyFuser using four different processors.

⋆ **Analysis**. We perform a deep dive into our results and report our findings for various (kernel, processor) pairs. The insights gained are broadly applicable to other frameworks, including Kokkos and its underlying backends.

The source code for PyFuser is available at `https://github.com/kokkos/pykokkos`.

3.2 Motivation

In this section, we discuss the need for kernel fusion in Kokkos and PyKokkos by discussing the performance benefits (Section 3.2.1). We then show a concrete example of two PyKokkos kernels a user might want to fuse motivating the need for automated kernel fusion in PyKokkos (Section 3.2.2).

3.2.1 Benefits

When fusing multiple parallel kernels into one, we expect the resultant *fused kernel* to perform better than the unfused kernels due to better data reuse, improved compiler optimizations, and lower total kernel launch overhead.

Despite the potential benefits of kernel fusion, there is no performance portable automated solution. It frequently falls to the programmer to decide how to organize parallel code across kernels. One factor influencing this decision is code reuse i.e., moving code to a separate kernel to reuse it in different parts of a codebase. Another factor is availability of existing kernel implementations in libraries (e.g., NumPy),

```
@pk.workunit
def add(tid, A, B, N, scalar):
    for i in range(N):
        A[tid][i] = scalar + B[tid][i]

@pk.workunit
def mul(tid, A, B, C, N):
    for i in range(N):
        C[tid][i] = A[tid][i] * B[tid][i]

pk.parallel_for(num_threads, add, A, B, N, scalar)
pk.parallel_for(num_threads, mul, A, B, C, N)
```

(a) Example PyKokkos kernels.

```
@pk.workunit
def add_mul(tid, A0, B0, N0, scalar, A1, B1, C, N1):
    for i in range(N0):
        A0[tid][i] = scalar + B0[tid][i]
    for i in range(N1):
        C[tid][i] = A1[tid][i] * B1[tid][i]
```

(b) An example fused kernel.

```
@pk.workunit
def add_mul(tid, A, B, C, N, scalar):
    for i in range(N):
        A[tid][i] = scalar + B[tid][i]
        C[tid][i] = A[tid][i] * B[tid][i]
```

(c) An example fused kernel with argument and loop fusion.

```
ld.global.u32   %r8, [%rd17];     (load B)
add.s32         %r9, %r8, %r4;
st.global.u32   [%rd18], %r9;     (store A)
ld.global.u32   %r10, [%rd17];    (load B)
mul.lo.s32      %r11, %r10, %r9;
st.global.u32   [%rd19], %r11;    (store C)
```

(d) PTX for the fused kernel above.

Figure 3.1: Fusion of two simple PyKokkos kernels where the compiler cannot fully optimize the code.

which forces programmers into having their code separated into different kernels at arbitrary boundaries with little room for customization. Finally, there is also the programmer's subjective interpretation of writing modular code with proper separation of concerns [68].

3.2.2 Example

At present, programmers are required to refactor their code to fuse Kokkos kernels, as C++ compilers do not fuse kernels automatically. Figure 3.1a shows two PyKokkos kernels that a programmer might want to fuse. These kernels both access common data, so we expect fusion to be beneficial: instead of loading each element of B twice (lines 4 and 9), we only need load it once and reuse the value later. Similarly, instead of loading each element of A in `mul`, we can reuse the value produced in `add`. However, fusing these two simple kernels and achieving good performance proves to be challenging.

The first challenge is to actually write the fused kernel. Figure 3.1b shows how a programmer might do this: the fused kernel parameters and body are formed by combining the lists of parameters and bodies of the unfused kernels and renaming the parameters to avoid name conflicts. In order to write a functionally correct fused kernel, the programmer must keep the parameters separate (e.g., A0 and A1) and the loops unfused, as these optimizations depend on the run-time values of the arguments. While this is a general-purpose solution, it does not reuse data effectively: the compiler cannot optimize the memory accesses as they occur in different loops (lines 3 and 5) and the arrays are accessed through different identifiers (A0 and A1). Even if loop bounds are known to be the same by compilers, we found that they will not consistently fuse loops, so users must do this manually.

Alternatively, the programmer could maintain multiple implementations of each fused kernel specialized to the run-time arguments. However, this approach will not scale, especially for larger kernels with more parameters.

Suppose for the particular case in Figure 3.1a the programmer wrote the specialized fused implementation in Figure 3.1c, fusing the parameters and loops. Even here, the compiler cannot fully optimize the memory accesses as a programmer might expect: the second load from array B on line 5 cannot be safely removed. This can be confirmed by looking at the optimized PTX (low level NVIDIA ISA) generated by NVCC in Figure 3.1d (we show PTX instead of SASS as PTX is higher level and easier to understand, but the generated SASS follows the same pattern). The reason is that the store to array A on line 4 in Figure 3.1c prevents the compiler from removing the second load from B, as it cannot prove that A and B do not alias, and so the store to A invalidates the previously loaded value from B. The programmer can avoid this issue by removing the redundant load or use the `restrict` keyword, which tells the compiler that the arrays do not alias.

The final remaining challenge is that the programmer must then locate all occurrences of consecutive calls to `add` and `mul` in the codebase in order to replace them with a single call to a fused kernel. The two kernel calls on lines 11 and 12 in Figure 3.1a are an example of one such occurrence. In this case, it is straightforward to replace these two calls. However, in larger codebases, this pattern can occur frequently, so replacing all occurrences by hand will not scale.

We will now present PyFuser, an automated kernel fusion framework for PyKokkos which handles these challenges.

3.3 Technique

In this section, we describe the design and implementation of PyFuser. PyFuser contains two components, the Tracer and the Fuser, which are integrated into the PyKokkos Runtime, and run prior to the PyKokkos transpilation and compilation steps. When enabled, PyFuser dynamically records all PyKokkos kernel calls and stores them in a *trace*. It then fuses kernels in the trace and replaces them with the fused kernels. We first describe how PyFuser is integrated with the PyKokkos

Runtime (Section 3.3.1). Second, we describe the Tracer, which uses lazy evaluation to record and retrieve traces (Section 3.3.2). Third, we describe how the Fuser fuses multiple kernels into a one through code transformation (Section 3.3.3). Finally, we describe additional code transformations that are needed to realize the full benefits of fusion (Section 3.3.4).

3.3.1 PyKokkos Runtime

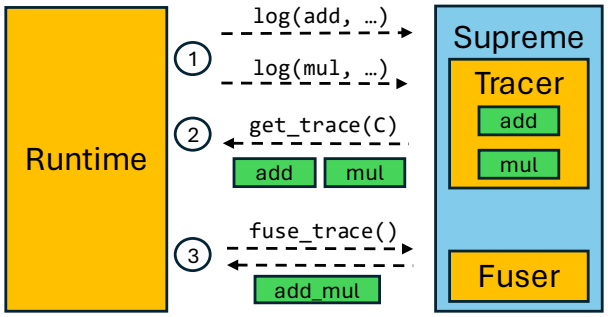
The Tracer and the Fuser contain the bulk of PyFuser's implementation. The Tracer dynamically records all kernel calls at run-time into traces while the Fuser generates the fused kernels. PyFuser integrates with PyKokkos by passing these fused kernels to the PyKokkos Compiler, which transpiles kernels to C++ and Kokkos before compiling them with a C++ compiler.

Figure 3.2a shows how PyFuser's Tracer is integrated into the PyKokkos Runtime. In PyKokkos, every kernel call invokes the `call_kernel()` Runtime method (line 5). If tracing is enabled by the user, PyKokkos logs the call using PyFuser's Tracer (line 8). For reduce and scan kernels, the caller expects that the scalar value of the reduction result is returned. Since PyFuser delays the kernel's execution, the result is not immediately available. Instead, PyFuser creates a `Future` object and returns it to the caller (line 9). The `Future` contains a field representing the scalar value and implements all the Python arithmetic operators (e.g., `__add__()`) and so behaves as a Python scalar type would.

Due to lazy evaluation, a kernel's output is not immediately visible following a kernel call. Kernels output data either by writing to arrays or by returning a scalar, which we replace with a `Future` when tracing is enabled. Therefore, we must ensure that the programmer receives the correct data when reading from an array or a `Future` by actually running the kernels that generate that data. Therefore, we introduce the `flush_data()` Runtime method (line 14 in Figure 3.2a), which given an array or `Future` returns the sequence of kernel calls, i.e., the trace, that must run in order to

```
class Runtime:
  self.tracer: Tracer
  self.fuser: Fuser

  def call_kernel(self, policy, kernel, args):
    if self.tracing_enabled():
      future = Future()
      self.tracer.log(policy, kernel, args, future)
      return future

    handle = self.compile(kernel)
    return self.call(policy, handle, args)

  def flush_data(self, data):
    trace = self.tracer.get_trace(data)
    fused_ops = self.fuser.fuse_trace(trace)

    for op in fused_ops:
      handle = self.compile(op.kernels)
      result = self.call(op.policy, handle, op.args)
      op.future.value = result
```

(a) Tracer and Fuser additions to PyKokkos Runtime.



(b) PyFuser workflow for Figure 3.1a.

Figure 3.2: PyFuser integration with PyKokkos Runtime.

generate that data. We modify the PyKokkos array and `Future` implementations to automatically call `flush_data()` whenever the user reads from them.

When an array or `Future` calls `flush_data()`, PyFuser first retrieves the trace associated with that data from the Tracer (line 15) and generates fused kernels using the Fuser (line 16). It then resumes the PyKokkos compilation process (i.e., transpilation, compilation, and invocation) for each fused kernel (lines 18-20). If the kernel returns a scalar result, i.e., it is a reduce or a scan, the Tracer retrieves the result after the kernel finishes execution and sets the `Future`'s value to the result at this point (line 21).

This sequence of events is illustrated in Figure 3.2b for the example in Figure 3.1a. With tracing enabled, the PyKokkos Runtime uses PyFuser to log the calls to `add` and `mul` (step 1). When the user reads from the `C` array, PyFuser retrieves the trace (step 2) and fuses the kernels (step 3).

3.3.2 Tracing

The Tracer's purpose is to log kernel calls and retrieve traces when data is requested by the user. Figure 3.3 shows the algorithms implemented in the Tracer. The `log` function (line 1) receives the execution policy of the kernel call, the kernel being called, the kernel arguments, and the `Future` object associated with that call (set to `None` for kernels that don't return scalars), all of which are needed for executing it later, while the `get_trace()` function (line 13) retrieves the part of the trace associated with the requested data, which we call a *trace partition*.

The first step in `log` is storing the arguments to allow executing the kernel later. This includes storing the values of scalar arguments and references to the array arguments. Line 2 extracts all the array parameters the kernel reads from and associates them with the array arguments. Each array the kernel reads from is a dependency that needs to be fulfilled when the kernel is executed later. In order to correctly replay execution later, the Tracer must keep track of their *versions* at the

```
Require: policy - The execution policy of the kernel call
Require: kernel - The kernel being called
Require: args - The arguments passed to the kernel
Require: future - The Future associated with the kernel
 1: function LOG(policy, kernel, args, future)
 2:     read_arrays ← get_read_set(kernel, args)
 3:     dependencies ← get_dependencies(read_arrays)
 4:     op ← TracerOp(policy, kernel, args, future, dependencies)
 5:     add_to_trace(op)
 6:     write_arrays ← get_write_set(kernel, args)
 7:     for array in write_arrays do
 8:         version ← get_current_version(array)
 9:         set_data_version(array, version + 1)
10:         map_array_to_op(op, version + 1)
11:     end for
12: end function
Require: data - The array or Future being requested
Require: version - Optionally specify the requested version
13: function GET_TRACE(data, version)
14:     if version is None then
15:         version ← get_current_version(data)
16:     end if
17:     op ← get_op(data, version)
18:     trace ← {}
19:     for d in op.dependencies do
20:         new_ops ← get_trace(d.array, d.version)
21:         trace.extend(new_ops)
22:     end for
23: return trace
24: end function
```

Figure 3.3: The algorithms to log kernel calls and retrieve kernel calls associated with some data.

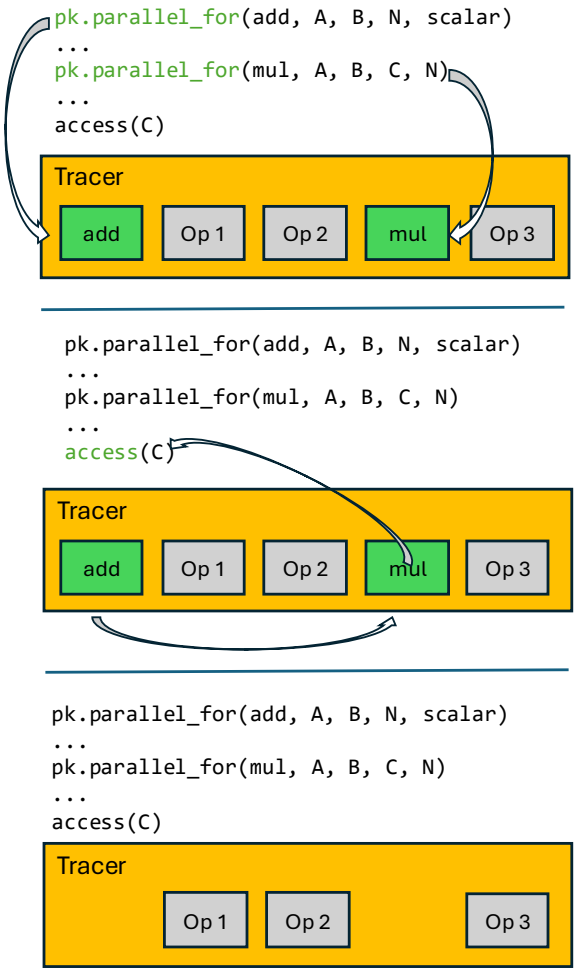


Figure 3.4: The Tracer's internal state over time while executing the example shown in Figure 3.1a.

kernel call site and record them in the trace. The array arguments plus their current versions form the read dependencies of the kernel (line 3). The tracer can then add the kernel call to the trace (line 5).

The next step is to update the version numbers of all arrays the kernel writes to. The Tracer identifies those arrays (line 6), iterates over each one to get its current version (line 8), increments that version (line 9), and finally maps the new version to this kernel call (line 10).

When the user requests some data associated with a trace (array or `Future`), the PyKokkos Runtime calls `get_trace()`. The Tracer first identifies the current version of the data (line 15) and then maps this version of the data to a delayed kernel call (line 17). This kernel call alone is not enough, as it might depend on arrays that are written to by earlier kernel calls. Therefore, the tracer finishes building the trace by iterating over the dependencies of the current kernel call (line 19) and recursively getting all the kernel calls needed (line 20), specifying the exact version of the dependency, before finally returning the trace (line 22).

Figure 3.4 shows how Tracer's internal state changes during execution of the example in Figure 3.1a. When execution reaches the calls to `add` and `mul`, the Tracer adds them to the trace, along with other kernel calls that might occur during execution (shown in gray). Later, when the user requests the contents of array `C` (through the `access()` function), the Tracer finds the current version and maps it to `mul`, which itself depends on the results of the `add` call. The other kernel calls in the trace are not needed to fulfill the user's request and so they are left untouched. The calls to `add` and `mul` form the trace partition retrieved by the Tracer. The Tracer removes these kernels from the trace following retrieval.

3.3.3 Fusion

Given a trace partition, the Fuser selects the kernels that can be fused and generates the code of the fused kernels. We refer to the process of selecting the

kernels to be fused as the *fusion strategy*. The current fusion strategy implemented in PyFuser's Fuser is greedy, i.e., it attempts to fuse as many kernels as possible. Ideally, this strategy would replace the entire trace partition with a single call to one fused kernel. However, certain factors prevent that: first, each fused kernel can contain at most one reduce or scan kernel. Since reduce and scan kernels return a single scalar, fusing more than one such kernels would require returning multiple scalars, which is currently not supported in PyKokkos and is left for future work. The second factor is safety, i.e., preserving the semantics of the original code, which is a similar condition to safely implement loop fusion [36]. In order to safely fuse two kernel calls, they must run the same number of threads (or iterations in loop terminology) and there should be no negative distance dependencies between the two kernels, i.e., one of the kernels uses a value in a thread that is computed by another thread in the other kernel.

In order to prevent unsafe fusion of kernels, the Fuser runs a safety check that inspects the current kernel and the next kernel to be fused. In PyKokkos, scalar arguments cannot be modified by the kernels, so only the data in arrays can be shared between threads. We consider fusion to be unsafe if the same threads from different kernels access different elements from the same array when one of those accesses is writing to the array.

The Fuser first retrieves all indexing expressions of arrays that are common between the two kernels, using the parser in PyKokkos to obtain the abstract syntax trees (ASTs). For all common arrays that at least one of the kernels writes to, the Fuser looks at expressions used to index the array. If the expression is the thread ID or a constant (or a function of the two), i.e., if each element of the array is accessed by at most one thread, and both kernels use the same expression, then the kernels are accessing the same element from that array and are safe to fuse.

The Fuser therefore iterates over the trace partition and partitions it further according to the above conditions. This process is initiated by the PyKokkos Runtime when data is requested by the user (line 16 in Figure 3.2a).

Finally, the Fuser generates a fused kernel from each trace partition in the form of a Python AST for easy integration with PyKokkos. It first gets the AST of each kernel called in the trace partition. Each kernel AST contains a list of statements in the kernel's body and a list of parameters passed to the kernel, so the Fuser forms the fused AST by concatenating all the body and parameter lists from the unfused ASTs. In order to prevent any naming conflicts in the fused code, the Fuser first pre-processes each unfused AST's body and renames all variables by adding a prefix unique to each kernel.

The fused Python AST then proceeds through the typical PyKokkos compilation pipeline (Section 2.4.1), which transpiles the kernel to C++, generates language bindings to call it, and compiles it using a C++ compiler. This results in a single fused kernel which the PyKokkos Runtime calls (lines 19-20 in Figure 3.2a).

3.3.4 Code Transformations

Looking at the output of the Fuser in Figure 3.1b, we observe that it suffers from the issues that prevents the compiler removing redundant memory instructions and reusing data effectively. First, the same array object is referred to with different identifiers (e.g., `A0` and `A1` on line 2), meaning that the compiler does not know it can reuse data loaded from one in another. Second, the memory accesses are in different loops (lines 3 and 5) and therefore scopes, so the compiler cannot optimize them together. Third, the potential for aliasing between the arrays prevents the compiler from reusing loaded values. To account for these issues, we implement three code transformations in PyKokkos that enable the compiler to optimize the fused kernel code better. Figure 3.5 shows the transpiled C++ code of the fused kernel before (Figure 3.5a) and after (Figures 3.5b-3.5c) applying the transformations.

```
class add_mul {
 int scalar0, N0, N1;
 Kokkos::View<int**> A0, B0, A1, B1, C1;
 void operator()(int tid) const {
  for (int i = 0; i < N0; i++) {
   A0(tid, i) = scalar0 + B0(tid, i);
  }
  for (int i = 0; i < N1; i++) {
   C1(tid, i) = A1(tid, i) * B1(tid, i);
  }
 }};
```

(a) Generated Kokkos kernel before transformations.

```
class add_mul {
 int scalar0, N0;
 Kokkos::View<int**> A0, B0, C1;
 void operator()(int tid) const {
  for (int i = 0; i < N0; i++) {
   A0(tid, i) = scalar0 + B0(tid, i);
   C1(tid, i) = A0(tid, i) * B0(tid, i);
  }
 }};
```

(b) After argument and loop fusion transformations.

```
class add_mul {
 int S0, N0;
 Kokkos::View<int**> A0, B0, C1;
 // Kokkos function with restrict
 KOKKOS_FUNCTION void kernel(
  int tid,
  int* __restrict__ A0, int A_S_0, int A_S_1,
  int* __restrict__ B0, int B_S_0, int B_S_1,
  int* __restrict__ C1, int C_S_0, int C_S_1
 ) const {
  for (int i = 0; i < N0; i++) {
   A0[tid * A_S_1 + A_S_0] = S0 + B0[tid * B_S_1 + B_S_0];
   C1[tid * C_S_1 + C_S_0] = A0[tid * A_S_1 + A_S_0] * B0[tid * B_S_1 + B_S_0];
  }
 }
 // Kernel calling Kokkos function
 void operator()(int tid) const {
  kernel(tid,
    A0.data(), A0.stride_0(), A0.stride_1()),
    B0.data(), B0.stride_0(), B0.stride_1()),
    C1.data(), C1.stride_0(), C1.stride_1());
 }
};
```

(c) After the restrict transformation.

Figure 3.5: Applying transformations to the fused kernels.

3.3.4.1 Argument fusion

The first transformation fuses kernel parameters by dynamically identifying the passed arguments that refer to the same Python object at run-time (using the Python built-in `id()` function). Figure 3.5b shows this transformation on lines 2 and 3, where `A1`, `B1`, and `N1` are all removed from the kernel arguments (which are listed as member variables in C++ Kokkos kernels). All references to `A1`, `B1`, and `N1` in the kernel body are replaced by `A0`, `B0`, and `N0` respectively (lines 5-7). This is a form of run-time specialization that depends on the input arguments, so we modify PyKokkos to handle multiple specialized versions of the same kernel. This is necessary as multiple calls of the same kernel at run-time might have different arguments which could be fused differently, and thus different calls might lead to different code transformations, with each being specialized to a specific kernel call.

3.3.4.2 Loop fusion

The second transformation fuses for loops in the kernels to move memory accesses to the same scope. Since compilers are not guaranteed to always fuse loops, we implement our own loop fusion code transformation in PyKokkos to fuse for loops in kernels, following the typical safety requirements [36]. Applying this to our kernel results in a single for loop replacing the original two loops (line 5 in Figure 3.5b). Moving all memory accesses to the same scope enables the compiler to recognize that the value stored to `A0` on line 6 can be reused, eliminating a redundant load.

3.3.4.3 Restrict

The third transformation applies the `restrict` keyword[1] to the arrays in the fused kernel code, which tells the compiler that the arrays do not point to overlapping regions of memory, enabling more memory optimizations.

[1] `https://en.cppreference.com/w/c/language/restrict`

Applying the argument and loop fusion code transformations and compiling the kernel in Figure 3.5b results in the PTX shown in Figure 3.1d, where the same memory location in `B0` is loaded twice. By default, compilers must assume that all function parameters passed as a pointer or reference could alias. Due to the presence of a store to another array (`A0`) between the two loads, the value loaded from `B0` on line 6 in Figure 3.5b cannot be reused on the next line.

We therefore implemented a code transformation to apply the `restrict` keyword to arrays. As with argument fusion, it depends on the run-time values of the input arguments. Before adding `restrict`, we inspect the input arrays in the PyKokkos Runtime and record those that do not alias.

Kokkos provides the `Kokkos::Restrict` memory trait for use as a template argument to its arrays. However, this information is not used by compilers such as NVCC and HIPCC during optimization, since the actual array pointer is defined as a member variable in a class (`Kokkos::View`, the C++ Kokkos array type). Adding `restrict` to pointers which are member variables is ignored by most compilers (likely due to `restrict` being part of the C but not the C++ standard). The only way to use `restrict` reliably across C++ compilers is to apply it to kernel parameters. However, this is currently not possible in C++ Kokkos kernels, which are defined as overloaded `operator()` methods or lambdas with no parameters (besides the thread ID and other Kokkos parameters).

We found a workaround to the Kokkos `restrict` issue by introducing a new Kokkos function (i.e., a function that can be called from a kernel) which accepts the array arguments as raw pointers, which we can then add `restrict` to. In order to index these array pointers, we need to obtain their strides, which tell us the distance in memory between two array elements in a particular dimension. Using these strides, we can calculate an element's location from a multi-dimensional index.

The restrict transformation is shown in Figure 3.5c. The new Kokkos function is defined on line 5. The parameters of this function are same as the parameters of

the kernel. We replace each array with a raw pointer and stride variables passed as parameters, instead of a `Kokkos::View` which would contain all this information. This requires that we replace each array indexing operation with C-style array indexing (line 12), taking the memory layout into account. Finally, we call the Kokkos function from the kernel, using the View `data()` method to access the raw pointer and the stride method for each dimension to get the corresponding stride value (line 18).

Together, these transformations enable the compiler to more effectively optimize code. Applying them manually to C++ Kokkos code is tedious as the run-time specialization transformations (i.e., argument fusion and restrict) require maintaining multiple implementations of each kernel to dispatch calls to the appropriate implementation according to run-time conditions. Run-time systems such as PyKokkos are naturally more suited to apply these transformations.

Following these transformations, the fused kernel is compiled and called by the PyKokkos Runtime.

3.4 Evaluation

In this section we present our evaluation of PyFuser. First, we describe our evaluation setup (Section 3.4.1) and introduce our test subjects (Section 3.4.2). Second, we show the kernel speedups we obtain by applying PyFuser to our test subjects before and after applying our code transformations (Section 3.4.3). Third, we study the impact of kernel fusion on performance related profiler metrics (Section 3.4.4). Fourth, we analyze the kernel speedups obtained by looking at the machine code generated by the compilers and the profiler metrics (Section 3.4.5). Fifth, we examine the run-time overhead of PyFuser (Section 3.4.6).

3.4.1 Evaluation Setup

We used Python 3.11, the latest version of PyKokkos at the time we conducted our experiments (commit 3d4afd2), and Kokkos 3.7.02. We ran our experiments on

Table 3.1: Processors used in our experiments.

Processor	DRAM	Backend	Compiler
NVIDIA V100	16 GB	CUDA 12.0	NVCC 12.0
NVIDIA A100	40 GB	CUDA 12.0	NVCC 12.0
AMD MI250X	128 GB	HIP 5.4.3	HIPCC 15.0
Intel Xeon E5-2620	128 GB	OpenMP	GCC 12.2
AMD EPYC 7763	256 GB	OpenMP	GCC 11.2

multiple processors, including NVIDIA and AMD GPUs as well as Intel and AMD CPUs. Table 3.1 shows the full list of processors we used, the size of the processor's memory, the relevant Kokkos backend, as well as the compilers we used.

All results shown are the arithmetic mean of data collected across three runs. We used the simple kernel timer from the Kokkos Tools repository [3] to measure kernel execution time. This measures the execution time of the generated C++ Kokkos kernel, which includes the raw kernel execution time and any performance overhead introduced by C++ Kokkos, but not Python, PyKokkos, or PyFuser overheads.

3.4.2 Test Subjects

Our test subjects include existing examples from the PyKokkos repository that contain multiple kernel calls, third-party PyKokkos applications, and new code we added to evaluate PyFuser. Much of these new subjects are ported from existing NumPy implementations by using PyKokkos as a drop-in replacement for those libraries. Our test subjects include:

- **ExaMiniMD**: a molecular dynamics mini-application that was originally implemented in C++ Kokkos [4]. The PyKokkos version has ∼3k lines of code and 14 distinct kernels (Section 2.5.2).
- **Particle-in-cell code**: a particle-in-cell (PIC) solver of the electron Boltzmann equation implemented originally in PyKokkos [11].

- **Gaussian Naive Bayes**: a PyKokkos implementation of scikit-learn's Gaussian Naive Bayes (GNB) classifier, which was originally written in NumPy [16].
- **NPBench**: a collection of NumPy code samples for evaluating frameworks that accelerate NumPy [77]. Of the original 52 samples, PyKokkos currently supports 12, of which 7 contain multiple kernel calls that can be fused. These include `adi`, `covariance`, `fdtd_2d`, `jacobi_1d`, `mvt`, `syrk`, and `syr2k`. Adding support for more NPBench subjects is possible but requires some additional engineering effort.
- **Benchmarks**: includes `BabelStream`, `GUPS`, `GUPS Atomic`, NSTREAM, and Transpose. Originally written in C++ the PyKokkos versions of these benchmarks achieved the same performance as Kokkos (Section 2.5).

We verified that the fused kernels produce the correct output by comparing with the unfused versions.

3.4.3 Kernel Speedups

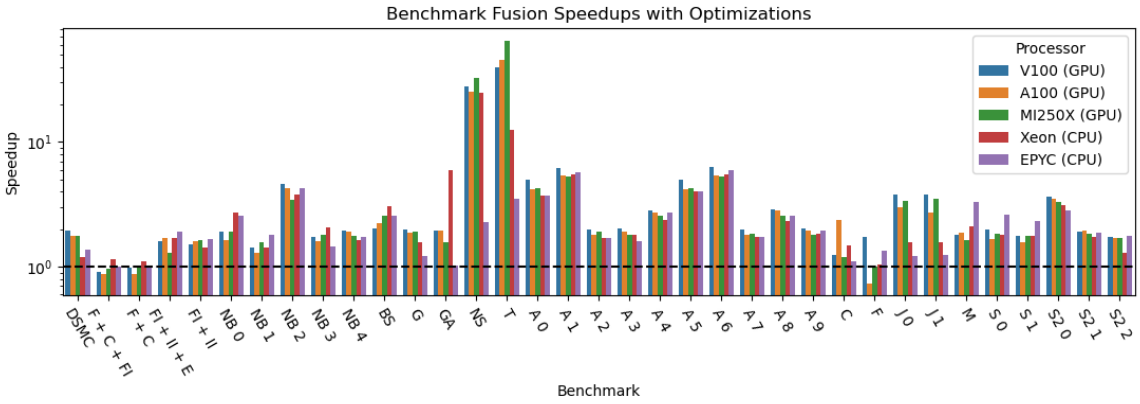


Figure 3.6: Kernel fusion speedup over unfused kernels with our transformations for `adi` (`A`), `BabelStream` (`BS`), `covariance` (`C`), `fdtd_2d` (`F`), `GUPS` (`G`), `GUPS Atomic` (`GA`), `jacobi_1d` (`J`), `mvt` (`M`), Gaussian Naive Bayes (`NB`), NSTREAM (`NS`), `syrk` (`S`), `syr2k` (`S2`), and Transpose (`T`).

Table 3.2: Kernel Fusion Speedup over Unfused Kernels on the GPUs.

Subject	**Kernel**	**# Kernels**	**Speedup**					
			V100		**A100**		**MI250X**	
			U/O	O	U/O	O	U/O	O
PIC	DSMC	3	1.03	1.96	1.16	1.78	0.92	1.77
ExaMiniMD	F + C	2	0.98	0.99	0.87	0.88	0.97	1.02
ExaMiniMD	F + C + FI	3	0.87	0.91	0.94	0.88	0.90	0.96
ExaMiniMD	FI + II	2	1.40	1.52	1.60	1.60	1.28	1.62
ExaMiniMD	FI + II + E	3	1.24	1.61	1.67	1.69	1.18	1.30
GaussianNB	NB 0	3	1.88	1.92	1.65	1.65	1.63	1.91
GaussianNB	NB 1	2	1.41	1.43	1.27	1.29	1.23	1.59
GaussianNB	NB 2	5	4.56	4.65	3.89	4.28	3.35	3.47
GaussianNB	NB 3	2	1.74	1.74	1.60	1.62	1.53	1.81
GaussianNB	NB 4	2	1.96	1.94	1.93	1.92	1.78	1.76
Microbenchmarks	BS	4	1.93	2.03	2.25	2.25	2.14	2.55
Microbenchmarks	G	2	2.02	2.01	1.94	1.88	1.88	1.90
Microbenchmarks	GA	2	1.96	1.96	1.94	1.94	1.86	1.56
Microbenchmarks	NS	50	4.67	27.70	5.12	25.41	3.23	32.86
Microbenchmarks	T	50	2.22	39.32	1.81	45.34	1.41	64.97
NPBench	A 0	5	5.00	5.00	4.28	4.22	4.06	4.29
NPBench	A 1	7	6.30	6.18	5.51	5.43	4.75	5.28
NPBench	A 2	2	2.00	2.00	1.95	1.82	1.80	1.91
NPBench	A 3	2	2.03	2.03	1.91	1.91	1.81	1.81
NPBench	A 4	3	2.73	2.81	2.77	2.72	2.50	2.55
NPBench	A 5	5	5.00	4.95	4.28	4.22	4.04	4.28
NPBench	A 6	7	6.30	6.30	5.45	5.45	4.77	5.25
NPBench	A 7	2	2.00	2.00	1.90	1.80	1.74	1.82
NPBench	A 8	3	2.73	2.87	2.91	2.81	2.53	2.56
NPBench	A 9	2	1.99	2.01	1.95	1.95	1.79	1.81
NPBench	C	3	1.12	1.25	1.24	2.35	1.03	1.20
NPBench	F	2	0.73	1.74	0.82	0.73	1.00	1.01
NPBench	J 0	4	3.86	3.79	3.08	2.99	3.40	3.40
NPBench	J 1	4	3.78	3.78	2.95	2.73	3.36	3.49
NPBench	M	2	1.84	1.79	1.63	1.87	1.63	1.63
NPBench	S 0	2	2.00	2.00	1.72	1.67	1.82	1.84
NPBench	S 1	2	1.76	1.76	1.61	1.57	1.69	1.76
NPBench	S2 0	4	3.64	3.68	3.55	3.49	2.99	3.32
NPBench	S2 1	2	1.94	1.92	1.95	1.95	1.84	1.84
NPBench	S2 2	2	1.71	1.72	1.83	1.69	1.69	1.69

Tables 3.2 and 3.3 and Figure 3.6 show the speedups we get by fusing the kernels in our test subjects. Tables 3.2 and 3.3 show the speedups both with and

Table 3.3: Kernel Fusion Speedup over Unfused Kernels on the CPUs.

Subject	Kernel	# Kernels	Speedup			
			Xeon		EPYC	
			U/O	O	U/O	O
PIC	DSMC	3	1.21	1.21	1.35	1.38
ExaMiniMD	F + C	2	1.01	1.10	1.03	1.01
ExaMiniMD	F + C + FI	3	1.17	1.16	1.04	1.00
ExaMiniMD	FI + II	2	1.51	1.43	1.60	1.67
ExaMiniMD	FI + II + E	3	1.82	1.69	1.88	1.92
GaussianNB	NB 0	3	2.88	2.72	2.33	2.57
GaussianNB	NB 1	2	1.59	1.42	1.70	1.80
GaussianNB	NB 2	5	3.64	3.76	3.93	4.30
GaussianNB	NB 3	2	2.60	2.08	1.47	1.47
GaussianNB	NB 4	2	1.63	1.64	1.58	1.73
Microbenchmarks	BS	4	3.01	3.07	0.89	2.57
Microbenchmarks	G	2	1.56	1.57	1.22	1.22
Microbenchmarks	GA	2	6.16	5.90	1.01	1.02
Microbenchmarks	NS	50	6.55	24.63	1.25	2.28
Microbenchmarks	T	50	7.06	12.62	1.09	3.52
NPBench	A 0	5	4.12	3.69	3.63	3.74
NPBench	A 1	7	5.71	5.55	6.00	5.68
NPBench	A 2	2	1.88	1.69	1.79	1.72
NPBench	A 3	2	1.92	1.80	1.95	1.59
NPBench	A 4	3	2.64	2.38	2.22	2.71
NPBench	A 5	5	4.18	4.02	4.19	4.02
NPBench	A 6	7	5.70	5.47	6.15	5.90
NPBench	A 7	2	1.87	1.74	1.90	1.74
NPBench	A 8	3	2.66	2.31	2.74	2.55
NPBench	A 9	2	1.96	1.85	1.93	1.97
NPBench	C	3	1.50	1.47	1.01	1.10
NPBench	F	2	1.04	1.04	1.36	1.33
NPBench	J 0	4	2.25	1.58	1.59	1.22
NPBench	J 1	4	2.25	1.56	1.62	1.24
NPBench	M	2	2.04	2.12	3.84	3.34
NPBench	S 0	2	1.81	1.79	2.73	2.64
NPBench	S 1	2	1.85	1.77	6.17	2.31
NPBench	S2 0	4	3.54	3.13	2.83	2.85
NPBench	S2 1	2	2.18	1.74	1.83	1.87
NPBench	S2 2	2	1.49	1.28	1.71	1.77

without transformations applied to evaluate the impact of the compilers' optimizations on our GPUs and CPUs respectively. Columns 1 and 2 show the source of the

Table 3.4: Effectiveness of Fusion and Optimization on Reducing Arithmetic Instructions. A Positive Number Means that the Number of Instructions Decreased while a Negative Number Means that they Increased.

Kernel	**Arithmetic Instructions Saved [%]**					
	V100		**A100**		**MI250X**	
	U/O	O	U/O	O	U/O	O
DSMC	0	28	1	28	0	40
F + C	-3	-3	-4	-4	1	1
F + C + FI	-3	-3	1	0	1	1
FI + II	30	43	24	44	23	33
FI + II + E	43	46	39	46	39	47

Table 3.5: Effectiveness of Fusion and Optimization on Reducing Memory Instructions. A Positive Number Means that the Number of Instructions Decreased while a Negative Number Means that they Increased.

Kernel	**Memory Instructions Saved [%]**					
	V100		**A100**		**MI250X**	
	U/O	O	U/O	O	U/O	O
DSMC	0	52	3	52	-3	68
F + C	0	0	0	0	0	1
F + C + FI	0	1	0	1	0	2
FI + II	0	27	0	27	4	38
FI + II + E	-8	36	-8	36	12	44

kernels and the name of the fused kernel respectively. Column 3 shows the number of kernel calls that PyFuser fused together to form the fused kernel, i.e., the size of the trace partition fused to form a single kernel. The remaining columns show speedups on different processors when fusing these kernels before and after applying our code transformations, corresponding to Unoptimized (U/O) and Optimized (O), respectively. Each row corresponds to a trace partition fused into a single kernel by PyFuser. To better visualize the impact of kernel fusion, we also show all speedups only with transformations applied in the bar plot in Figure 3.6. The x-axis shows the abbreviated name of each fused kernel and the y-axis shows the speedup.

Table 3.6: Effectiveness of Fusion and Optimization on Memory Loaded from DRAM.

Kernel	**DRAM Bytes Saved [%]**					
	V100		**A100**		**MI250X**	
	U/O	O	U/O	O	U/O	O
DSMC	71	65	76	69	2	49
F + C	70	71	82	83	-2	4
F + C + FI	85	85	89	90	-3	4
FI + II	51	51	59	54	29	35
FI + II + E	68	67	73	69	37	39

We calculate speedup by comparing the time to execute the original unfused kernels called consecutively (i.e., how they existed originally) to the time to execute the fused kernel. As we are interested in speedups of wall clock times, we do not show roofline analysis as it deals with execution rates.

3.4.3.1 Particle-in-cell Code

The PIC code uses a direct simulation Monte Carlo scheme (DSMC) to model particle collisions. Originally implemented in three kernels that are called consecutively, PyFuser fuses them into one kernel. We used 4M particles as the input size for the results in tables 3.2 and 3.3.

The results in tables 3.2 and 3.3 show that fusion alone does not always result in a large speedup when compared to the original unfused kernels. On the V100, A100, and MI250X GPUs, we see speedups of 1.03×, 1.16×, and 0.92× respectively without our transformations. After applying all of our transformations, we observe much larger speedups of 1.96×, 1.78×, and 1.77× on the V100, A100, and MI250X GPUs, respectively.

In contrast to the GPUs, both CPUs attained their highest speedup without the need for the transformations. Applying them does not result in a noticeable speedup improvement.

3.4.3.2 ExaMiniMD

Of the original 14 kernels in ExaMiniMD, PyFuser fuses 4 trace partitions into 4 new fused kernels, each represented in a separate row in tables 3.2 and 3.3.

The first two fused kernels, F + C and F + C + FI, both contain the `Force` (`F`) and the `Compute` (C) kernels and the latter also contains the `Final Integrate` (FI) kernel, as PyFuser dynamically creates and fuses trace partitions according to the kernels called, which differ in each time step. Fusion here has a relatively smaller impact compared to other kernels. For the F + C kernel, we see speedups of 0.98×, 0.87×, and 0.97× on the V100, A100, and MI250X GPUs. Applying our transformations resulted in negligible improvements on the NVIDIA GPUs, while the MI250X GPU rose to a 1.02× speedup. For the F + C + FI kernel, we see speedups of 0.87×, 0.94×, and 0.90× on the V100, A100, and MI250X GPUs. Applying our transformations resulted in improvements on the V100 and the MI250X, rising to 0.91× and 0.96× respectively, while decreasing the speedup on the A100 to 0.88×.

The two other kernels in ExaMiniMD are FI + II and FI + II + E. Both contain the `Initial Integrate` (II) and `Final Integrate` (FI) and the latter also contains `Exchange Self` (E). These kernels operate on the same arrays and do similar computations, leading to speedups on all devices, with transformations on the GPUs further improving performance.

As before, the results on our CPUs show strong speedups even without the need for our transformations.

3.4.3.3 Gaussian Naive Bayes, NPBench, & Benchmarks

The Gaussian Naive Bayes and NPBench [77] subjects consist primarily of kernels that perform NumPy-style array operations. The Benchmarks include less than five (unfused) kernels each, with most being small. In total, PyFuser generates 30 fused kernels.

Looking at Figure 3.6, we observe speedups for all kernels on all processors except for `fdtd_2d` (`F`) on the A100, which achieves a 0.73× speedup. On average, across all test subjects on all processors, we observe a speedup of 3.8×.

3.4.4 Profiler Metrics

In this section, we report relevant profiler metrics we obtain from kernel fusion and from our transformations. Recall that two main reasons we expect speedups from kernel fusion are improved compiler optimizations and better data reuse (Section 3.2.1). To test the validity of this hypothesis, we use profilers to examine the reduction in instructions executed and memory traffic to DRAM in tables 3.4, 3.5, and 3.6. The first column shows the name of the kernel. The second column shows the reduction in the total number of arithmetic instructions executed, both integer and floating point. The third column shows the reduction in the total number of memory instructions executed (i.e., loads and stores). The fourth column shows the reduction in the total number of bytes loaded from and stored to DRAM. We gathered these metrics using NVIDIA's NCU [5] and AMD's Omniperf [45] profilers. We do not show CPU metrics as collecting them proved to be noisy due to non-kernel code. We only show the PIC and ExaMiniMD kernels than others as they are larger.

We first note that for all kernels on all GPUs, better caching leads to significant reductions in DRAM traffic.

For the DSMC kernel, we initially observe no improvement in the number of arithmetic and memory instructions executed. The reason becomes apparent when looking at the fused kernel: the bulk of the work in each of the unfused kernels is done in a sequential for loop. Fusing the three kernels results in three separate for loops with three separate scopes. This prevents the compilers from effectively optimizing redundant computations and memory accesses. We even observe a slight increase in memory instructions executed on the MI250X (-3%) due to increased register pressure, which leads to register spills to memory. After applying our transformations, we see

significant reductions in the numbers of instructions executed: 28% and 52% for arithmetic and memory instructions respectively on both NVIDIA GPUs and 40% and 68% reductions on the MI250X GPU.

For the ExaMiniMD F + C and F + C + FI kernels, we initially see little improvement or even a small increase in arithmetic and memory instructions executed prior to applying transformations (the increases are due to register pressure forcing more loads and address calculations). For both fused kernels, the first kernel they are fused from (`Force`) initializes three arrays to zero, while the second kernel (`Compute`) further updates these arrays. Intuitively, it would seem that fusing these kernels allows the compiler to optimize the code further: first, the initial value stored in the arrays (in `Force`) does not have to be loaded when they are updated later (in `Compute`) as it is known at compile-time; second, since this value is zero, the compiler can eliminate an add instruction; third, the compiler can also eliminate the initial store as the second kernel overwrites it, making the first kernel's code completely redundant. Instead, the fused F + C kernel shows no improvement in arithmetic and memory instructions after fusion (second row in tables 3.4 and 3.5). Inspecting the assembly reveals that the compilers cannot perform any of the aforementioned optimizations due to the potential for aliasing between the three input arrays.

Our transformations help on the MI250X but not on the NVIDIA GPUs. Looking at the assembly with transformations enabled, we see that NVCC is not doing the aforementioned optimizations, while HIPCC is, resulting in a 1% reduction in memory instructions on the MI250X (Table 3.5). We have reached out to NVIDIA asking why that is the case.

For the fused FI + II and FI + II + E kernels, we see significant reductions in arithmetic instructions executed on all GPUs prior to applying our transformations. For FI + II, we see 30%, 24%, and 23% reductions on the V100, A100, and MI250X GPUs respectively, while for FI + II + E, we see 43%, 39%, and 39%. These kernels do not contain loops so the compiler optimizes the arithmetic operations in the fused

kernel even without our transformations. As for the memory instructions, we only see a reduction after applying the restrict optimization. This also reduces the arithmetic instructions further by eliminating the memory address calculations.

3.4.5 Performance Analysis

In this section, we analyze our results and explain the obtained speedups and slowdowns on our GPUs and CPUs.

3.4.5.1 GPUs

We initially observed low speedups or slowdowns for the DSMC kernel, which only improved after applying our transformations. From the observed profiler metrics (Section 3.4.4), we can see that the compiler cannot optimize the code effectively and remove redundant instructions due to each kernel body being completely wrapped in a different loop. Prior to applying our transformations, we expect some speedup due to reduced kernel launch overhead, as PyFuser fuses three kernel calls into one, and to improved caching. While this is true for the NVIDIA GPUs, we observe a small slowdown on the MI250X due to the slight increase in memory instructions executed. Enabling our transformations allows the C++ compiler to optimize the code further by eliminating a large number of redundant instructions, which leads to large speedups for all GPUs.

For the F + C kernel, we do not see speedups initially as the compilers' cannot optimize the fused code and remove redundant instructions. The slowdowns observed for the F + C kernel on the A100 compared to the V100 are due to differences in NVCC's register allocation strategies across the two generations of GPU, which leads to more register pressure and a slower kernel on the A100. Applying our transformations is not very effective on the NVIDIA GPUs as NVCC does not remove the redundant instructions, while HIPCC removes them for the MI250X leading to a small speedup in the fused kernel. The F + C + FI kernel exhibits similar behavior,

although we see worse performance on the V100 due to Kokkos selecting a suboptimal CUDA block size. Manually overriding it gives us similar results to F + C.

For the last two ExaMiniMD kernels, we see large speedups even in the unoptimized case as the compiler removes redundant instructions. The restrict transformation removes memory instructions and improves performance further.

For Gaussian Naive Bayes, NPBench, and benchmarks, we observe speedups for almost all kernels as shown in Figure 3.6, especially in cases where PyFuser is able to fuse large trace partitions. For example, NSTREAM (`NS`) and Transpose (`T`) call all their kernels in a loop that runs for fifty iterations, which PyFuser can fuse into one kernel each, leading to large reductions in kernel launch overhead and the largest speedups among all our benchmarks. In contrast, `GUPS` (`G`) and `GUPS Atomic` (`GA`) show relatively smaller speedups due to smaller trace partitions composed of two kernel calls.

We see a slowdown for one kernel: 0.73× for `fdtd_2d` on the A100. The assembly shows that NVCC reorders memory instructions in the fused kernel leading to more memory stalls and worse performance. Interestingly, NVCC selected a different order on the V100, which proved to be better.

In summary, kernel fusion leads to speedups when compilers can optimize away redundant instructions. Reduced kernel launch overhead and improved caching help but the main benefit of fusion comes from removing redundant instructions, which the compiler cannot always do without our transformations. Additionally, varying results across processors and compilers imply that further processor and compiler specific optimizations can provide even better results.

3.4.5.2 CPUs

We observe speedups for all kernels on both CPUs. Our transformations were not needed to obtain large speedups as the fused kernels already benefit greatly from improved cache utilization.

Table 3.7: Overhead with PyFuser from tracing, fusion, and transformations averaged across all subjects on all processors.

Processor	**Tracing** [%]	**Fusion (U/O)** [%]	**Fusion (O)** [%]
V100	2.1	4.5	5.3
A100	2.2	3.9	4.5
MI250X	4.1	7.4	8.2
Xeon	2.8	6.8	7.5
EPYC	7.4	16.1	16.8
Mean	3.7	7.7	8.5

3.4.6 Run-time Overhead

In this section, we examine the run-time overhead introduced by PyFuser during tracing, fusion, and applying the code transformations. Table 3.7 shows the overhead of different modes of PyFuser as a percentage of original running time, averaged across all subjects. The first column shows the name of the processor. The remaining columns show the measured overheads of each mode: tracing without fusion (only lazy evaluation) in the second column, tracing with fusion in the third column, and tracing with fusion and code transformations applied in the fourth column.

The results show that PyFuser introduces minor overhead. Tracing alone is lightweight, adding 3.7% to running time on average. Adding fusion to tracing increases overhead to 7.7%, largely due to applying the fusion safety check. Adding transformations on top of fusion is a relatively small increase to 8.5%, mostly due to applying the restrict transformation.

3.5 Conclusion

In this chapter, we presented PyFuser, a framework for dynamic fusion of Python Kokkos kernels. PyFuser dynamically traces kernel calls and lazily fuses kernels. Fused kernels are invoked when the application accesses the result of a sequence of kernel calls. We also introduced three code transformations to the PyKokkos to

enable further optimizations of fused kernels. These changes provide performance benefits to dynamically generated fused kernels due to improved compiler optimizations. Our experiments on HPC applications and benchmarks show that PyFuser achieves substantial speedups on NVIDIA and AMD GPUs, as well as Intel and AMD CPUs. We believe that PyFuser is a great step towards regaining performance lost due to language abstractions and common coding patterns.

Chapter 4: Related Work

In this chapter, we describe prior research work most related to HPC in Python and kernel fusion. First, we give a brief overview of Python HPC frameworks (Section 4.1). Second, we describe various techniques and libraries that implement kernel fusion (Section 4.2).

4.1 Python HPC Frameworks

There is a significant amount of prior work that aims to improve the performance of Python. Numba [39] compiles a subset of the language to LLVM IR and provides support for parallelism. Cython [14] extends Python with C types and translates code to C; at this point Cython supports only OpenMP for several parallel constructs. Shed Skin [63] compiles pure Python 2 programs to C++ but only supports a restricted subset of Python. Unlike prior work, PyKokkos enables performance portability across HPC frameworks by targeting the C++ Kokkos library and supports the latest version of Python. Dask [61] and Pygion [64] enable distributed task-based programming in Python. In contrast, PyKokkos focuses primarily on shared-memory parallelism instead.

There has been previous work on higher level abstractions to facilitate programmability and portability. PyTorch [55] and TensorFlow [8] are high performance libraries that provide abstractions for tensor computing and machine learning. Halide [60] is a domain specific language (DSL) embedded in C++ for writing portable, high performance image processing code. DiffTaichi [34] is a high-performance framework embedded in Python for building differentiable physical simulators. IrGL [53] is an intermediate representation for parallel graph algorithms that is compiled to CUDA. PyKokkos closely follows the Kokkos model for performance portability without necessarily specializing in a specific application domain.

Therefore, unlike the aforementioned frameworks, PyKokkos can be used to write general-purpose parallel kernels.

Java has also experienced some interest in the field of GPU computing [23]. Lime [24] and HJ-OpenCL [32] are Java-based DSLs that can access GPUs while providing limited support for various Java features. Lime is a Java-compatible object-oriented language capable of generating GPU code for OpenCL or CUDA. HJ-OpenCL generates OpenCL kernels from the Habanero-Java language, and further work [29] adds support for dynamic object allocation. Rootbeer [57] translates Java code that implements a specific kernel interface to CUDA workloads. Jacc [18] is another framework that translates native annotated Java code, but takes a different approach by directly generating NVIDIA PTX rather than OpenCL or CUDA. GVM [17] is a Java interpreter that runs entirely on GPUs. TornadoVM [19] is a Java framework for high-performance heterogeneous programming. PyKokkos is embedded in Python rather than Java, and is not limited to GPU execution since it targets Kokkos instead of device specific frameworks.

A recent approach to translating programming languages is unsupervised translation by training on monolingual source code [62]. PyKokkos takes a more traditional approach to translation that does not include machine learning. Combining the two approaches is worth exploring.

4.2 Kernel Fusion

Figure 4.1 shows how PyFuser compares to existing work on kernel fusion. The majority of this existing work deals with fusing domain specific kernels (shown beneath the horizontal line), specifically *pre-existing kernels* that are typically smaller in size and mostly restricted to tensor arithmetic operations and deep learning operators, whereas PyFuser is able to fuse arbitrary, general-purpose parallel kernels. Furthermore, PyFuser is the only framework that does this dynamically at run-time, which improves its ability to detect opportunities for kernel fusion.

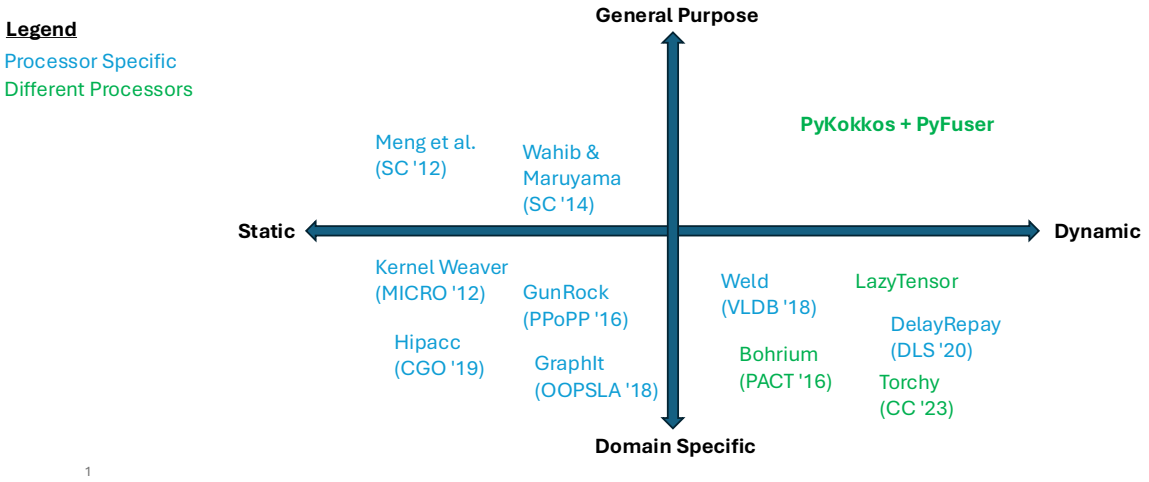


Figure 4.1: PyFuser is the first framework for dynamic fusion of general purpose kernels that runs on different processing units.

One of the earliest attempts at CUDA kernel fusion was a dataflow-driven approach [48]. Kernels are expressed as CPU code skeletons which are then analyzed to extract data dependencies between iterations across consecutive kernel calls. Different fusion strategies corresponding to different shared memory caching implementations are exhaustively searched and evaluated through a GPU performance model [47] at compile-time. The best projected implementation is then manually added back to the code. This framework requires rewriting the code into a skeleton and manual modifications to the users code, unlike PyFuser, which works with existing PyKokkos code with no modifications.

Kernel Weaver [73] applies fusion to kernels for relational algebra operators meant for use in data warehousing applications. Examples of these operators includes SELECT, JOIN, UNION, etc. They use a heuristic that estimates a kernel's expected resource usage to decide which operators to fuse. Their evaluation is focused solely on two queries that use some combination of these operators, through which they show performance speedups with fusion. Similar work by the authors [74] also experimented with kernel fission, splitting kernels to overlap computation with data transfer from main memory to GPU memory.

Wahib and Maruyama [70] formulated kernel fusion as a combinatorial optimization problem and used a genetic algorithm with a performance projection model to explore the space of potential fusions, focusing on memory-bound CUDA kernels. The kernels to be fused are extracted statically from the code, which requires imposing restrictions such as each kernel can only be called once, unlike PyFuser which uses a dynamic approach and works with existing code with no restrictions.

Filipovic et al. [26] present a framework for fusing CUDA BLAS routines. The programmer first writes down the routines to be fused in a separate script which the framework then parses and maps to the kernels in the BLAS library. The framework then selects which kernels to fuse by predicting the potential performance improvement. This work was later extended to evaluate the potential to fuse arbitrary kernels [27], but code generation and transformation was not automated.

Helium [46] is a framework for reordering and fusing OpenCL kernels. The kernels must be first written as strings and then used to create OpenCL kernel objects. Helium will then use lazy evaluation to delay the execution of the kernels. When data is requested by the user, Helium will reorder the kernels to achieve better performance, while optionally fusing some of them. The evaluation of Helium focuses mainly on the performance gains from reordering kernels and does not analyze the impact of fusion itself.

Gunrock [71] and GraphIt [75] are frameworks for GPU graph analytics. While evaluating Gunrock, the developers noted that excessive fragmentation of kernels leads to significantly lower performance, and kernel fusion is required in these situations. Similarly, GraphIt is a DSL for graph applications which allows users to specify which kernels should be fused. Both these frameworks require that the user explicitly specify which kernels to fuse. Furthermore, the DSLs used in these frameworks introduce high-level abstractions unfamiliar to most HPC developers.

Hipacc [59] is a DSL that allows programmers to write fused stencil-based image processing kernels. It decides which kernels to fuse together by predicting

the potential savings in executed cycles. Hipacc focuses mainly on image processing kernels and domain specific optimizations it can apply to those specific kernels, which are not applicable to PyFuser.

Kernel fusion was also implemented as an LLVM pass for C++ kernels [40]. The user specifies which kernels to be fused using C++ attributes. A loop fusion pass that works with skewed loops was also developed. This approach imposes severe restrictions on the user code, such as each kernel only being called once, and the body of each kernel can only contain a single loop, with no instructions appearing outside of the loop. A similar approach for OpenCL removed the need for user-specified attributes using lazy evaluation [65]; however, it focuses only on CPU execution.

Kernel fusion based on code motion [28] is a dataflow-based fusion technique that moves CUDA kernel calls to expose more opportunities for parallelism.

Tacker [76] is a framework for statically fusing CUDA core kernels with tensor core kernels. Fusing these two types of kernels allows for better utilization of an NVIDIA GPU's resources as each type of kernel uses different types of cores.

An extension of the SYCL API [56] allows the programmer to specify kernels that can be fused, which a JIT compiler then fuses at run-time. Thrust [6] is a C++ library for writing parallel algorithms that run with CUDA. Its API includes constructs that allow the programmer to specify multiple kernel calls as a call to a single fused kernel. These frameworks require the programmer to explicitly specify which kernels can be fused, unlike PyFuser, which automatically finds opportunities for fusion.

Bohrium [37] is a runtime system for automatic parallelization. Bohrium supports Python array programming through an API similar to NumPy. Subsequent work by the authors implemented fusion of Bohrium parallel array operations [38] by modeling the decision of which operations to fuse as a partitioning problem. Since the evaluation focuses on CPUs, the authors use a relatively simple cost function that

looks solely at the number of array accesses to evaluate candidate partitions. Furthermore, the NumPy API is more restrictive than the PyKokkos API, which allows writing general purpose kernels (note that PyKokkos also supports the NumPy API).

Weld [54] does lazy evaluation for a NumPy-like API meant specifically for data science applications. It generates an intermediate representation which is lightly optimized with classical compiler optimizations before passing it to LLVM. Additionally, Weld only supports CPU execution, whereas PyFuser supports a more general API than NumPy and supports both CPUs and GPUs.

DelayRepay [49] is a drop-in replacement for NumPy that provides delayed execution of universal functions and then maps them to CUDA kernels. At every call to a universal function, DelayRepay adds an AST node corresponding to that function, e.g., a unary or binary mathematical expression. When a certain array's data is requested, this triggers fusion of the accumulated AST and generation of CUDA code from that AST. The generated code is then compiled and executed and the results are returned to the user. As mentioned before, PyFuser supports a more general API than NumPy.

The increasing popularity of frameworks such as PyTorch [55] has reignited interest in kernel fusion research for deep learning operators [44, 66]. LazyTensor [66] implements delayed execution by building traces of PyTorch and Swift for TensorFlow operations to generate XLA HLO IR [7], which is then compiled and called when data is requested by the user. Torchy [44] is a tracing JIT compiler for PyTorch that delays execution of PyTorch tensor operations and stores them in a trace. This trace can be run directly or can be further optimized via the PyTorch neural network compiler, before being flushed when data is requested by the user. These frameworks address fusion of domain-specific deep learning operators and not general-purpose kernels written in PyKokkos like PyFuser.

Task fusion [67] is similar in spirit to kernel fusion, but deals with fusing tasks, which are higher level than kernels and can contain multiple kernel calls. The goal of

task fusion is mainly to reduce overheads of the tasking system, while in kernel fusion the goal is to generate more efficient kernels.

Horizontal fusion [43] is a CUDA kernel fusion technique that differs from classical (vertical) fusion. Instead of concatenating the contents of two consecutive kernel calls, horizontal fusion attempts to interleave the execution of the two kernels it fuses. The code from each original kernel is placed in a separate branch and each thread is dispatched to a specific branch based on its thread ID. This effectively results in each original kernel being executed simultaneously. This fusion technique could be integrated into PyFuser along with classical fusion to further improve performance.

RAP [72] is a framework that improves GPU utilization in deep learning recommendation models (DLRMs) input pre-processing and training. It monitors GPUs and assigns input pre-processing kernels to GPUs that have idle resources. RAP will optionally horizontally fuse [43] certain pre-processing kernels to increase GPU utilization and improve performance.

Chapter 5: Future Work

In this chapter, we describe ways in which PyKokkos and PyFuser can be improved upon in multiple aspects, run-time performance, compilation-time, general usability, and ease of debugging.

5.1 Run-time performance

We plan on exploring techniques to further optimize the run-time performance of the kernels generated by PyKokkos and PyFuser.

5.1.1 Just-in-time Optimizations

Since PyKokkos kernels are compiled at run-time, it is possible to optimize the kernels further by utilizing information known about the kernels only at run-time. First, we can specialize each kernel call by replacing scalar variables used in the kernel with their run-time values in the generated C++ code, allowing the compiler to generate more optimal code. This can also be extended by specifying the dimensions of each view explicitly in the generated code. While these optimizations can easily be shown to produce more optimal code, specializing a kernel to a specific call means that it must be re-compiled if these scalar values change during later calls. We plan on exploring the trade-off between specialization and re-compilation in future work.

Another application of using run-time information to improve performance is in autotuning kernel launch parameters. For CUDA and HIP, this means selecting block sizes as well as other tunable knobs made available through their APIs. For OpenMP, this means selecting the schedule kind and chunk size. Currently, Kokkos handles these knobs using static information only, so there is potential for improvement. Additionally, all compilers offer tunable knobs that can impact code generation, which PyKokkos has the option of adjusting at run-time.

5.1.2 Kernel Fusion Optimizations

PyFuser fuses kernel calls in traces greedily. Once a kernel has been fused with another, it is removed from the trace and can no longer be fused with other kernels. In the future, we plan on exploring different fusion strategies that take into account kernel characteristics and explore trade-offs between different fusion decisions. We also plan on applying horizontal fusion [43] in PyFuser to fuse kernels which would not benefit from vertical fusion.

Recording kernel calls in traces allows us to potentially run these kernels concurrently if no dependencies exist between them. This is typically done by task scheduling systems such as Parla [42] and Legate [12] where tasks could contain multiple kernel calls. In the future, we will investigate how kernel fusion and task scheduling systems can be combined to further improve performance.

5.2 Usability

PyKokkos currently supports a subset of the Kokkos API so additional engineering work is needed to add other Kokkos features, such as scratch memory, scatter `Views`, etc. So far, we have focused on the most commonly used features.

One additional benefit of PyKokkos over Kokkos is that the translation to Kokkos happens dynamically during the execution of a program. This, for example, enables users to build a kernel during the execution of a program. So far, we have focused on migrating existing kernels to PyKokkos. However, it would be interesting to see how we can benefit further from dynamic compilation, and if such a style could lead to a novel way for writing kernels or supporting program analysis tools.

5.3 Debugging

Current support for debugging PyKokkos applications is limited to execution in Python. This approach is helpful for finding logic-based bugs but not concurrency

bugs. In the future, we plan to add support for running PyKokkos with a debugger by adding line number information to the generated C++ code. Optimizing pure Python execution would also improve debugging experience.

5.4 Code Translation

PyKokkos currently includes a code translator that generates C++ code from the Python kernels. Building the translator correctly requires significant engineering effort as the two languages have very different semantics. We plan on experimenting with using large language models to automatically translate PyKokkos kernels to C++ Kokkos.

Chapter 6: Conclusion

With modern HPC hardware becoming more heterogeneous, writing parallel code is increasingly difficult. This paper introduces PyKokkos, a framework for performance portable parallel programming in Python, as well as PyFuser, a framework that further enhances the performance of PyKokkos through kernel fusion.

PyKokkos allows programmers to write high-performance parallel code entirely through Python. It automatically translates user designated parallel functions into C++ and Kokkos, while generating language bindings to connect the two languages. Through PyKokkos, we were able to show that programmers can write various HPC programs entirely through Python, achieving performance parity with similar C++ and Kokkos implementations, which are more complex and harder to write.

PyFuser complement PyKokkos and further improves its performance through kernel fusion. It uses lazy evaluation to delay kernel calls and store them in traces. It then fuses the kernels called in the traces while also transforming the generated code to allow the compilers to optimize the code further and improve performance. PyFuser generates fused kernels that are 3.8× faster on average than their unfused constituent kernels across all processors.

Through PyKokkos and PyFuser, we have shown that writing performance portable, high-performance code is possible entirely through Python. We believe that programming languages of the future will combine dynamic languages with the ideas introduced in this dissertation to make HPC easier to use and more widely accessible to a broader audience.

Works Cited

[1] Mypy. https://github.com/python/mypy, 2012.

[2] Kokkos Tutorials. https://github.com/kokkos/kokkos-tutorials, 2015.

[3] KokkosP Profiling Tools. https://github.com/kokkos/kokkos-tools, 2016.

[4] ExaMiniMD. https://github.com/ECP-copa/ExaMiniMD, 2017.

[5] Nsight Compute CLI. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html, 2024.

[6] Thrust: The c++ parallel algorithms library. https://nvidia.github.io/cccl/thrust/, 2024.

[7] XLA:compiling machine learning for peak performance. https://openxla.org/xla, 2024.

[8] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[9] Nader Al Awar, Steven Zhu, George Biros, and Milos Gligoric. A performance portability framework for Python. In *International Conference on Supercomputing*, pages 467–478, 2021. https://doi.org/10.1145/3447818.3460376.

[10] Nader Al Awar, Steven Zhu, Neil Mehta, George Biros, and Milos Gligoric. PyKokkos: Performance portable kernels in Python. In *International Conference on Software Engineering, Tool Demonstrations Track*, pages 164–167, 2022. https://doi.org/10.1145/3510454.3516827.

[11] James Almgren-Bell, Nader Al Awar, Dilip S Geethakrishnan, Milos Gligoric, and George Biros. A multi-GPU Python solver for low-temperature non-equilibrium plasmas. In *International Symposium on Computer Architecture and High Performance Computing*, pages 140–149, 2022. https://doi.org/10.1109/SBAC-PAD55451.2022.00025.

[12] Michael Bauer and Michael Garland. Legate numpy: accelerated and distributed array computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–23, 2019. https://doi.org/10.1145/3295500.3356175.

[13] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. RAJA: Portable performance for large-scale scientific applications. In *International Workshop on Performance, Portability and Productivity in HPC*, pages 71–81, 2019. https://doi.org/10.1109/P3HPC49587.2019.00012.

[14] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science and Engineering*, 13(2):31–39, 2011. https://doi.org/10.1109/MCSE.2010.118.

[15] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. https://doi.org/10.1137/141000671.

[16] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[17] Ahmet Celik, Pengyu Nie, Christopher J. Rossbach, and Milos Gligoric. Design, implementation, and application of GPU-based Java bytecode interpreters. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–28, 2019. https://doi.org/10.1145/3360603.

[18] James Clarkson, Christos Kotselidis, Gavin Brown, and Mikel Luján. Boosting Java performance using GPGPUs. In *International Conference on Architecture of Computing Systems*, pages 59–70, 2017. https://doi.org/10.1007/978-3-319-54999-6_5.

[19] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Lujan. Exploiting high-performance heterogeneous hardware for Java programs using Graal. In *International Conference on Managed Languages and Runtimes*, pages 1–13, 2018. https://doi.org/10.1145/3237009.3237016.

[20] CudaUVM. Unified Memory in CUDA 6, 2013. https://developer.nvidia.com/blog/unified-memory-in-cuda-6.

[21] CUDAWebPage. CUDA Zone, 2024. https://developer.nvidia.com/cuda-zone.

[22] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core

processors across diverse parallel programming models. In *International Workshop on Performance Portable Programming models for Manycore or Accelerators*, pages 489–507, 2016. https://doi.org/10.1007/978-3-319-46079-6_34.

[23] Jorge Docampo, Sabela Ramos, Guillermo L. Taboada, Roberto R. Expósito, Juan Touriño, and Ramón Doallo. Evaluation of Java for general purpose GPU computing. In *International Conference on Advanced Information Networking and Applications Workshops*, pages 1398–1404, 2013. https://doi.org/10.1109/WAINA.2013.234.

[24] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). In *International Conference on Programming Language Design and Implementation*, pages 1–12, 2012. https://doi.org/10.1145/2345156.2254066.

[25] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. https://doi.org/10.1016/j.jpdc.2014.07.003.

[26] Jiri Filipovic, Matúš Madzin, Jan Fousek, and Luděk Matyska. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing*, 71(10):3934–3957, 2015. https://doi.org/10.1007/s11227-015-1483-z.

[27] Jirí Filipovic and Siegfried Benkner. OpenCL kernel fusion for GPU, Xeon Phi and CPU. In *International Symposium on Computer Architecture and High Performance Computing*, pages 98–105, 2015. https://doi.org/10.1109/SBAC-PAD.2015.29.

[28] Junji Fukuhara and Munehiro Takimoto. Automated kernel fusion for GPU based on code motion. In *International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 151–161, 2022. https://doi.org/10.1145/3519941.3535078.

[29] Max Grossman, Shams Imam, and Vivek Sarkar. HJ-OpenCL: Reducing the gap between the JVM and accelerators. In *International Conference on Principles and Practices of Programming on The Java Platform*, pages 2–15, 2015. https://doi.org/10.1145/2807426.2807427.

[30] Stephen Lien Harrell, Joy Kitson, Robert Bird, Simon John Pennycook, Jason Sewall, Douglas Jacobsen, David Neill Asanza, Abaigail Hsu, Hector Carrillo Carrillo, Hessoo Kim, and Robert Robey. Effective performance portability. In *International Workshop on Performance, Portability and Productivity in HPC*, pages 24–36, 2018. https://doi.org/10.1109/P3HPC.2018.00006.

[31] Charles R. Harris, K. Jarrod Millman, Stefan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernandez del Rio, Mark Wiebe, Pearu Peterson, Pierre Gerard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. https://doi.org/10.1038/s41586-020-2649-2.

[32] Akihiro Hayashi, Max Grossman, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Accelerating Habanero-Java programs with OpenCL generation. In *International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 124–134, 2013. https://doi.org/10.1145/2500828.2500840.

[33] HIPWebPage. HIP documentation, 2024. https://rocm.docs.amd.com/projects/HIP/en/latest/.

[34] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Fredo Durand. DiffTaichi: Differentiable programming for physical simulation. In *International Conference on Learning Representations*, 2020. https://openreview.net/forum?id=B1eB5xSFvr.

[35] Intel. PRK. https://github.com/ParRes/Kernels, 2013.

[36] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, 1993.

[37] Mads R.B. Kristensen, Simon A.F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: A virtual machine approach to portable parallelism. In *International Parallel and Distributed Processing Symposium Workshops*, pages 312–321, 2014. https://doi.org/10.1109/IPDPSW.2014.44.

[38] Mads R.B. Kristensen, Simon A.F. Lund, Troels Blum, and James Avery. Fusion of parallel array operations. In *International Conference on Parallel Architectures and Compilation*, pages 71–85, 2016. https://doi.org/10.1145/2967938.2967945.

[39] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015. https://doi.org/10.1145/2833157.2833162.

[40] Andrew Lamzed-Short, Timothy R. Law, Andrew Mallinson, Gihan R. Mudalige, and Stephen A. Jarvis. Towards automated kernel fusion for the optimisation of scientific applications. In *Workshop on the LLVM Compiler Infrastructure in HPC and Workshop on Hierarchical Parallelism for Exascale Computing*, pages 45–55, 2020. https://doi.org/10.1109/LLVMHPCHiPar51896.2020.00010.

[41] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86, 2004. https://doi.org/10.1109/CGO.2004.1281665.

[42] Hochan Lee, William Ruys, Ian Henriksen, Arthur Peters, Yineng Yan, Sean Stephens, Bozhi You, Henrique Fingler, Martin Burtscher, Milos Gligoric, Karl Schulz, Keshav Pingali, Christopher J. Rossbach, Mattan Erez, and George Biros. Parla: a Python orchestration system for heterogeneous architectures. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2022. https://doi.org/10.1109/SC41404.2022.00056.

[43] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for GPU kernels. In *International Symposium on Code Generation and Optimization*, pages 14–27, 2022. https://doi.org/10.1109/CGO53902.2022.9741270.

[44] Nuno P. Lopes. Torchy: A tracing JIT compiler for pytorch. In *International Conference on Compiler Construction*, pages 98–109, 2023. https://doi.org/10.1109/MCSE.2021.3098509.

[45] Xiaomin Lu, Cole Ramos, Fei Zheng, Karl W. Schulz, Jose Santos, Keith Lowery, Nicholas Curtis, and Cristian Di Pietrantonio. Amdresearch/omniperf: v1.1.0-pr1 (13 oct 2023), October 2023.

[46] Thibaut Lutz, Christian Fensch, and Murray Cole. Helium: a transparent inter-kernel optimizer for OpenCL. In *Workshop on General Purpose Processing Using GPUs*, pages 70–80, 2015. https://doi.org/10.1145/2716282.2716284.

[47] Jiayuan Meng, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D. Uram. GROPHECY: GPU performance projection from CPU

code skeletons. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011. https://doi.org/10.1145/2063384.2063402.

[48] Jiayuan Meng, Vitali A. Morozov, Venkatram Vishwanath, and Kalyan Kumaran. Dataflow-driven GPU performance projection for multi-kernel transformations. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012. https://doi.org/10.1109/SC.2012.42.

[49] John Magnus Morton, Kuba Kaszyk, Lu Li, Jiawen Sun, Christophe Dubach, Michel Steuwer, Murray Cole, and Michael F. P. O'Boyle. DelayRepay: Delayed execution for kernel fusion in Python. In *International Symposium on Dynamic Languages*, pages 43–56, 2020. https://doi.org/10.1145/3426422.3426980.

[50] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. CuPy: A NumPy-compatible library for NVIDIA GPU calculations. In *Workshop on Machine Learning Systems*, 2017. http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[51] Travis E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9(3):10–20, 2007. https://doi.org/10.1109/MCSE.2007.58.

[52] OpenMPWebPage. OpenMP, 2024. https://www.openmp.org.

[53] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on GPUs. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19, 2016. https://doi.org/10.1145/2983990.2984015.

[54] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk,

Saman Amarasinghe, Samuel Madden, and Matei Zaharia. Evaluating end-to-end optimization for data analytics applications in weld. *Proceedings of the VLDB Endowment*, 11(9):1002–1015, 2018. https://doi.org/10.14778/3213880.3213890.

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: an imperative style, high-performance deep learning library. In *International Conference on Neural Information Processing Systems*, pages 8026–8037, 2019.

[56] Víctor Pérez, Lukas Sommer, Victor Lomüller, Kumudha Narasimhan, and Mehdi Goli. User-driven online kernel fusion for SYCL. *Transactions on Architecture and Code Optimization*, 20(2):1–25, 2023. https://doi.org/10.1145/3571284.

[57] Philip C. Pratt-Szeliga, James W. Fawcett, and Roy D. Welch. Rootbeer: Seamlessly using GPUs from Java. In *International Conference on High Performance Computing and Communication*, pages 375–380, 2012. https://doi.org/10.1109/HPCC.2012.57.

[58] pybind11. Pybind11 Documentation, 2020. https://pybind11.readthedocs.io/en/stable/intro.html.

[59] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *International Symposium on Code Generation and Optimization*, pages 242–253, 2019. https://doi.org/10.1109/CGO.2019.8661176.

[60] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Programming Language Design and Implementation*, pages 519–530, 2013. https://doi.org/10.1145/2491956.2462176.

[61] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Python in Science Conference*, pages 126–132, 2015. https://doi.org/10.25080/Majora-7b98e3ed-013.

[62] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *International Conference on Neural Information Processing Systems*, pages 20601–20611, 2020. https://proceedings.neurips.cc/paper/2020/file/ed23fbf18c2cd35f8c7f8de44f85c08d-Paper.pdf.

[63] ShedSkin. Shed Skin, 2020. https://shedskin.github.io.

[64] Elliot Slaughter and Alex Aiken. Pygion: Flexible, scalable task-based parallelism with Python. In *Parallel Applications Workshop, Alternatives To MPI*, pages 58–72, 2019.

[65] John A. Stratton, Jyothi Krishna V. S., Jeevitha Palanisamy, and Karthikadevi Chinnaraju. Kernel fusion in opencl. In *Euro-Par 2021: Parallel Processing Workshops*, pages 191–202, 2022. https://doi.org/10.1007/978-3-031-06156-1_16.

[66] Alex Suhan, Davide Libenzi, Ailing Zhang, Parker Schuh, Brennan Saeta, Jie Young Sohn, and Denys Shabalin. LazyTensor: combining eager execution with domain-specific compilers, 2021. https://arxiv.org/abs/2102.13267.

[67] Shiv Sundram, Wonchan Lee, and Alex Aiken. Task fusion in distributed runtimes. In *Parallel Applications Workshop: Alternatives To MPI+X*, pages 13–25, 2022. https://doi.org/10.1109/PAW-ATM56565.2022.00007.

[68] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[69] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stefan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. https://doi.org/10.1038/s41592-019-0686-2.

[70] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound GPU applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202, 2014. https://doi.org/10.1109/SC.2014.21.

[71] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: a high-performance graph processing library on the GPU. In *Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2016. https://doi.org/10.1145/3016078.2851145.

[72] Zheng Wang, Yuke Wang, Jiaqi Deng, Da Zheng, Ang Li, and Yufei Ding. RAP: Resource-aware automated GPU sharing for multi-GPU recommendation model

training and input preprocessing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 964–979, 2024. https://doi.org/10.1145/3620665.3640406.

[73] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel Weaver: Automatically fusing database primitives for efficient GPU computation. In *International Symposium on Microarchitecture*, pages 107–118, 2012. https://doi.org/10.1109/MICRO.2012.19.

[74] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing data warehousing applications for GPUs using kernel fusion/fission. In *International Parallel and Distributed Processing Symposium Workshops and PhD Forum*, pages 2433–2442, 2012. https://doi.org/10.1109/IPDPSW.2012.300.

[75] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. GraphIt: a high-performance graph DSL. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–30, 2018. https://doi.org/10.1145/3276491.

[76] Han Zhao, Weihao Cui, Quan Chen, Youtao Zhang, Yanchao Lu, Chao Li, Jingwen Leng, and Minyi Guo. Tacker: Tensor-CUDA core kernel fusion for improving the GPU utilization while ensuring QoS. In *International Symposium on High-Performance Computer Architecture*, pages 800–813, 2022. https://doi.org/10.1109/HPCA53966.2022.00064.

[77] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. NPBench: a benchmarking suite for high-performance NumPy. In *International Conference on Supercomputing*, pages 63–74, 2021. https://doi.org/10.1145/3447818.3460360.