Dynamically Fusing Python HPC Kernels

NADER AL AWAR, The University of Texas at Austin, USA
MUHAMMAD HANNAN NAEEM, The University of Texas at Austin, USA
JAMES ALMGREN-BELL, The University of Texas at Austin, USA
GEORGE BIROS, The University of Texas at Austin, USA
MILOS GLIGORIC, The University of Texas at Austin, USA

Recent trends in high-performance computing show an increase in the adoption of performance portable frameworks such as Kokkos and interpreted languages such as Python. PyKokkos follows these trends and enables programmers to write performance-portable kernels in Python which greatly increases productivity. One issue that programmers still face is how to organize parallel code, as splitting code into separate kernels simplifies testing and debugging but may result in suboptimal performance. To enable programmers to organize kernels in any way they prefer while ensuring good performance, we present `PyFuser`, a program analysis framework for automatic fusion of performance portable PyKokkos kernels. `PyFuser` dynamically traces kernel calls and lazily fuses them once the result is requested by the application. `PyFuser` generates fused kernels that execute faster due to better reuse of data, improved compiler optimizations, and reduced kernel launch overhead, while not requiring any changes to existing PyKokkos code. We also introduce automated code transformations that further optimize the fused kernels generated by `PyFuser`. Our experiments show that on average `PyFuser` achieves speedups compared to unfused kernels of 3.8× on NVIDIA and AMD GPUs, as well as Intel and AMD CPUs.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; • **Computing methodologies → Parallel programming languages**.

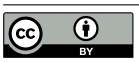
Additional Key Words and Phrases: Kokkos, PyKokkos, kernel fusion, performance, portability

ACM Reference Format:
Nader Al Awar, Muhammad Hannan Naeem, James Almgren-Bell, George Biros, and Milos Gligoric. 2025. Dynamically Fusing Python HPC Kernels. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA082 (July 2025), 23 pages. https://doi.org/10.1145/3728959

1 Introduction

The constant evolution of computer hardware architecture and high-performance computing (HPC) libraries represents a challenge to programmers, who must constantly update their code to take advantage of new hardware and library features. Performance portability frameworks [14], such as Kokkos [12, 33, 34], address this issue by providing a high-level parallel programming API which hides architecture and library-specific details from the programmer. This API is implemented with different HPC framework backends, such as CUDA, HIP, and OpenMP, which allows Kokkos to support a variety of major hardware architectures, including NVIDIA and AMD GPUs, as well as Intel and AMD CPUs, across many generations.

Authors' Contact Information: Nader Al Awar, The University of Texas at Austin, Austin, USA, nader.alawar@utexas.edu; Muhammad Hannan Naeem, The University of Texas at Austin, Austin, USA, hannan@utexas.edu; James Almgren-Bell, The University of Texas at Austin, Austin, USA, jalmgrenbell@utexas.edu; George Biros, The University of Texas at Austin, Austin, USA, biros@oden.utexas.edu; Milos Gligoric, The University of Texas at Austin, Austin, USA, gligoric@utexas.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2994-970X/2025/7-ARTISSTA082
https://doi.org/10.1145/3728959

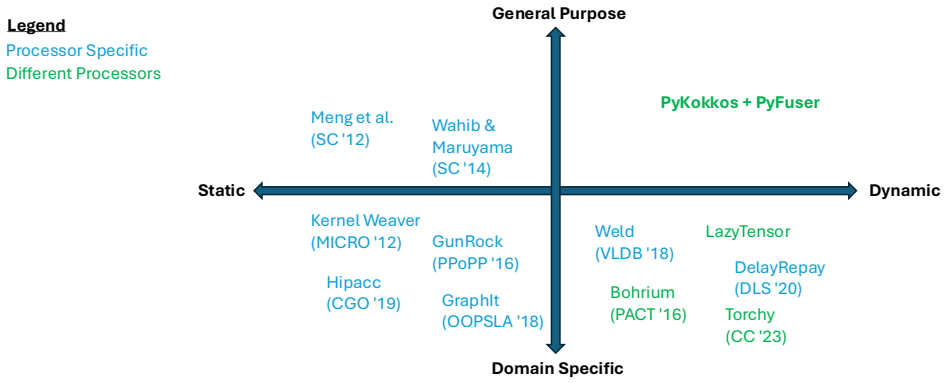


Fig. 1. PyFuser is the first framework for dynamic fusion of general purpose kernels on different processors.

Kokkos code is typically written in heavily-templated C++, which conflicts with another recent trend in HPC: increased adoption of high-level interpreted languages, such as Python [5, 9, 19, 39]. The PyKokkos framework [5, 6] enables HPC programmers to write performance portable parallel code entirely in Python by providing a statically typed domain-specific language (DSL) that it *dynamically* (at invocation time) transpiles to C++ Kokkos. PyKokkos, which is part of the Linux Foundation, has been used to write large HPC applications entirely in Python [7].

HPC frameworks such as Kokkos and PyKokkos require programmers 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 simplify testing and debugging. While this greatly improves usability, separating code in this way might result in sub-optimal performance, because compilers do not know (statically) how 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 programmer to *fuse* different kernels into one. In practice, manually finding opportunities for kernel fusion can be tedious, especially in larger codebases with dozens of kernels. Furthermore, such fused monolithic kernels are hard to test, debug, and maintain.

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 while enabling programmers to organize code in any way they prefer. `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.

Figure 1 shows how `PyFuser` compares to existing work on kernel fusion. The majority of existing work deals with fusing domain specific kernels, specifically *pre-existing kernels* that are typically smaller in size and mostly restricted to tensor arithmetic operations and deep learning operators, whereas `PyFuser` can fuse arbitrary and custom parallel kernels that are not restricted to any particular domain, such as machine learning or scientific computing. The majority of existing work on kernel fusion deals with fusing smaller kernels in deep learning programs, which is relatively easier to obtain speedups with compared to fusing custom kernels. Deep learning programs typically call a large number of smaller kernels that are shorter running (since they implement simple operations individually), which means that fusing them eliminates a large amount of run-time overhead associated with each call, greatly improving performance compared to unfused

code. In contrast, custom kernels are likely to be larger in size, longer running, and fewer in number, so obtaining speedups requires that the fused kernels be optimized properly. In addition to fusing custom kernels, `PyFuser` is the only framework that does this dynamically at run-time. This allows us to detect a larger amount of opportunities for kernel fusion compared to static approaches. Combining fusion of custom kernels with dynamic lazy evaluation is a novel approach.

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. In a typical PyKokkos kernel, the programmer is expected to hand optimize the code to assist the compiler. For automatically fused kernels generated by `PyFuser`, these optimizations must be applied again. We therefore dynamically analyze arguments passed to kernels to implement code transformations in PyKokkos, enabling better compiler optimizations.

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 [7], ExaMiniMD [2], a Gaussian Naive Bayes classifier, NPBench [38], and benchmarks from the PyKokkos repository. `PyFuser` achieves speedups of 3.8× on average across all processors.

The key contributions of this work include the following:

- ⋆ **Framework**. We present `PyFuser`, a framework for automatic kernel fusion. `PyFuser` records traces of kernel calls and fuses them to generate faster 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/tree/main/pykokkos/core/fusion.

2 Background

In this Section, we introduce Kokkos [34] and PyKokkos [5]. These frameworks enable programmers to write performance-portable parallel code, i.e., code that can run efficiently on different hardware platforms. Kokkos achieves this by providing a portable C++ API for writing parallel kernels that can be targeted to specific processors at compile-time. Internally, Kokkos contains backends that implement this API to run on different parallel processors, using frameworks such as OpenMP for CPUs, CUDA for NVIDIA GPUs, and HIP for AMD GPUs. The Kokkos API includes three main primitives: a parallel for loop, a parallel reduction, and a parallel scan. Furthermore, each framework is associated with a default memory layout (e.g., row-major or column-major) to achieve good performance for each processor. Kokkos also automatically selects kernel launch parameters at run-time (e.g., grid dimensions for CUDA and HIP).

PyKokkos is a Python framework built around C++ Kokkos that provides many of the same abstractions through Python. It dynamically (i.e., at run-time) transpiles Python kernels into C++ Kokkos, while also automatically generating language bindings to interface between the two languages. PyKokkos compiles 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 saved so that

```
1  @pk.workunit
2  def add(tid, A, B, N, S):
3    for i in range(N):
4      A[tid][i] = S + B[tid][i]
5
6  @pk.workunit
7  def mul(tid, A, B, C, N):
8    for i in range(N):
9      C[tid][i] = A[tid][i] * B[tid][i]
10
11 pk.parallel_for(threads, add, A, B, N, S)
12 pk.parallel_for(threads, mul, A, B, C, N)
```

(a) Example PyKokkos kernels.

```
1  @pk.workunit
2  def add_mul(tid,A0,B0,N0,S,A1,B1,C,N1):
3    for i in range(N0):
4      A0[tid][i] = S + B0[tid][i]
5    for i in range(N1):
6      C[tid][i] = A1[tid][i] * B1[tid][i]
```

(b) An example fused kernel.

```
1  @pk.workunit
2  def add_mul(tid, A, B, C, N, S):
3     for i in range(N):
4       A[tid][i] = S + B[tid][i]
5       C[tid][i] = A[tid][i] * B[tid][i]
```

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

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

(d) PTX for the fused kernel in Figure 2c.

Fig. 2. Fusion of two simple PyKokkos kernels where the compiler cannot fully optimize the code.

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 2a shows an example of two PyKokkos kernels. Users define kernels using the `pk.workunit` decorator. The parameters for each kernel include a thread ID (`tid` here) followed by the remaining user-defined parameters. Type annotations are optional, as PyKokkos specializes the kernel definition by inspecting the types of the arguments passed at run-time. The first kernel, `add` (line 2), adds a scalar `S` to a 2D array `B` and stores the result in the 2D array `A`. The second kernel, `mul` (line 7), multiplies arrays `A` and `B` element-wise and stores the result in `C`. Kernels are called with `pk.parallel_for()` (lines 11-12), which requires the number of threads, the name of the kernel, and the user provided arguments. When `parallel_for()` is called, PyKokkos internally transpiles the code to C++ and Kokkos, compiles it, and calls the kernel.

3 Motivation

In this Section, we discuss the need for kernel fusion in Kokkos and PyKokkos by discussing the performance benefits (Section 3.1). 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).

3.1 Benefits

Loop fusion is a classical compiler optimization that fuses multiple loops into one, improving data locality for arrays accessed in those loops [15]. PyKokkos kernels, which are expressed as parallel for loops, can be fused similarly.

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 that smaller kernels are easier to test, debug, and maintain. Additionally, the availability of existing kernel implementations in libraries (e.g., NumPy) 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 [32].

3.2 Example

At present, programmers must refactor their code to fuse Kokkos kernels, as C++ compilers do not fuse kernels automatically. Figure 2a shows two PyKokkos kernels that a programmer might want to fuse. These kernels 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 to get good performance proves to be challenging.

The first challenge is to actually write the fused kernel. Figure 2b 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 compilers know that the loop bounds are the same, 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.

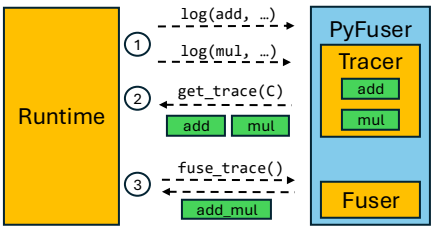
Suppose for the particular case in Figure 2a the programmer wrote the specialized fused implementation in Figure 2c, fusing the parameters and loops. Even here, the compiler cannot fully optimize the memory accesses as a programmer might expect: the second load from array B on line 5 cannot be safely removed. This can be confirmed by looking at the optimized PTX (low level NVIDIA ISA) generated by NVCC in Figure 2d (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 2c 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 2a 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. Programmers must also be careful not to introduce race conditions in fused kernels, i.e., only fuse kernels that are safe to fuse.

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

```
1  class Runtime:
2    self.tracer: Tracer
3    self.fuser: Fuser
4
5    def call_kernel(self, policy, kernel, args):
6      if self.tracing_enabled():
7        f = Future()
8        self.tracer.log(policy, kernel, args, f)
9        return f
10
11     handle = self.compile(kernel)
12     return self.call(policy, handle, args)
13
14   def flush_data(self, data):
15     ops = self.tracer.get_trace(data)
16     fused_ops = self.fuser.fuse_trace(ops)
17
18     for op in fused_ops:
19       h = self.compile(op.kernels)
20       result = self.call(op.policy, h, op.args)
21       op.future.value = result
```

(a) Tracer and Fuser additions to PyKokkos Runtime.



(b) PyFuser workflow for Figure 2a.

Fig. 3. PyFuser integration with PyKokkos Runtime.

4 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 4.1). Second, we describe the Tracer, which uses lazy evaluation to record and retrieve traces (Section 4.2). Third, we describe how the Fuser fuses multiple kernels into a one through code transformation (Section 4.3). Finally, we describe additional code transformations to realize the full benefits of fusion (Section 4.4).

4.1 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 3a 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

```
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(kernel, args, future)
 2:     read_arrays ← get_read_set(kernel, args)
 3:     dependencies ← get_dependencies(read_arrays)
 4:     op ← TracerOp(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, array, 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
```

Fig. 4. The algorithms to log kernel calls and retrieve kernel calls associated with some data.

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 3a), which given an array or `Future` returns the sequence of kernel calls, i.e., the trace, that needs to run in order to 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 future, its value is set at this point (line 21).

This sequence of events is illustrated in Figure 3b for the example in Figure 2a. 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).

4.2 Tracing

The Tracer's purpose is to log kernel calls and retrieve traces when data is requested by the user. Figure 4 shows the algorithms implemented in the Tracer. The `log` function (line 1) receives the kernel being called, the arguments for the kernel, and the `Future` object associated with that call

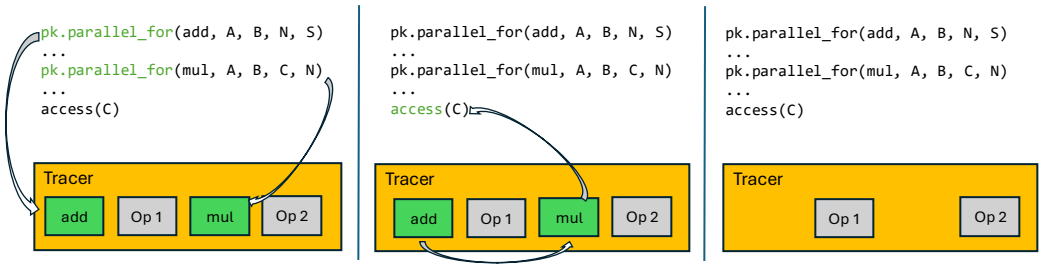


Fig. 5. The Tracer's internal state over time while executing the example in Figure 2a.

(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*.

We will explain the algorithm in Figure 4 by referring to the `mul` kernel in Figure 2a defined and called on lines 7 and 12 respectively. 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. For `mul`, this means storing the value of the scalar `N` and references to the array objects `A`, `B`, and `C` passed on line 12. Line 2 in Figure 4 extracts all the array parameters the kernel reads from and associates them with the array arguments. The `mul` kernel reads from the array parameters `A` and `B`, which are then associated with the array objects `A` and `B` passed as arguments. Since the arrays' contents could be different by the time `PyFuser` actually calls the kernel, the Tracer must also keep track of array *versions* 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). The `mul` kernel writes to the array `C`, so `C`'s version is incremented and the new version is associated with this invocation of the kernel (since this is the kernel call that will generate this version).

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).

To see what this would look like for our example in Figure 2a, assume we print the contents of array `C` following the call to `mul`. This will trigger a call to `get_trace(C, C.version)`. Since this version of `C` is generated by the call to `mul` on line 12, the Tracer adds this kernel call to the trace. The Tracer must then add the kernels responsible for generating `mul`'s read dependencies, which are the arrays `A` and `B`. The contents of `A` are generated by the call to the `add` kernel on line 11, so that kernel call gets added to the trace as well.

Figure 5 illustrates how Tracer's internal state changes during execution of the example in Figure 2a. 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.

4.3 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 [15]. 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 kernels, so only data in arrays can be shared between threads. We consider fusion to be unsafe if the same threads in different kernels access different elements from the same array when one of the accesses is a write.

The Fuser first retrieves all indexing expressions of arrays 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 two kernels writes to, the Fuser looks at the 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 they are accessing the same element from that array and the kernels 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 3a).

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, 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: 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 3a).

4.4 Code Transformations

Looking at the output of the Fuser in Figure 2b, we observe that it suffers from the issues that prevent the compiler from 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 between them. 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

```
class add_mul {
 // Kernel arguments as member variables
 int S0, N0, N1;
 Kokkos::View<int**> A0, B0, A1, B1, C1;

 // Kernel definition
 void operator()(int tid) const {
  for (int i = 0; i < N0; i++) {
   A0(tid, i) = S0 + 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 {
 // Kernel arguments as member variables
 int S0, N0;
 Kokkos::View<int**> A0, B0, C1;

 // Kernel definition
 void operator()(int tid) const {
  for (int i = 0; i < N0; i++) {
   A0(tid, i) = S0 + 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 A0_S_0, int A0_S_1,
  int* __restrict__ B0, int B0_S_0, int B0_S_1,
  int* __restrict__ C1, int C1_S_0, int C1_S_1
 ) const {
  for (int i = 0; i < N0; i++) {
   A0[tid * A0_S_1 + A0_S_0] = S0 + B0[tid * B0_S_1 + B0_S_0];
   C1[tid * C1_S_1 + C1_S_0] = A0[tid * A0_S_1 + A0_S_0]*B0[tid * B0_S_1 + B0_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 restrict transformation.

Fig. 6. Applying transformations to the fused kernels

loaded values. To account for these issues, we implement three code transformations in PyKokkos to enable the compiler to optimize the fused kernel code better. Figure 6 shows the transpiled C++ code of the fused kernel before (Figure 6a) and after (Figures 6b-6c) applying the transformations.

4.4.1 Argument fusion. The first transformation fuses kernel parameters by identifying the passed arguments that refer to the same Python object (using the Python built-in `id()` function). Figure 6b shows this transformation on lines 3 and 4, 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 8-10). 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 a necessary code transformation that greatly improves performance when fusing kernels. Recall that when two kernels are fused, their arguments are renamed to prevent naming conflicts (Section 4.3). This means that two kernels that operate on the same array or scalar (passed as an argument) will appear to be operating on different values. This prevents the compiler from optimizing away redundant code, such as memory loads and stores from the arrays and arithmetic operations on the scalars. These operations are likely to be common across the original kernels since kernels within the same trace partition are chosen based on read dependencies, which means that the same arrays will likely be passed to multiple kernels that will end up being fused. By fusing arguments that refer to the same object, this allows the compiler to find more instances of redundant code which can be safely removed to improve performance.

4.4.2 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 [15]. Applying this to our kernel results in a single for loop replacing the original two loops (line 8 in Figure 6b). After we move all memory accesses to the same scope, the compiler can reuse the value stored to `A0` on line 9, eliminating a redundant load.

We utilize this code transformation to fuse for loops that were originally found in different kernels prior to fusion. Compilers do not optimize code across different scopes that are conditionally executed, such as if statements and loops, since they cannot know which of those code blocks will actually be executed at run-time. Since loop fusion is commonly disabled, this means for loops that were originally found in different kernels, which might contain common code, will not have their contents optimized properly. This is crucial since loops are hot spots where most of execution time is spent. By fusing the for loops in the fused kernel, we ensure that the compiler is able to optimize away redundant code.

4.4.3 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.

Applying the argument and loop fusion code transformations and compiling the kernel in Figure 6b results in the PTX shown in Figure 2d, 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 9 in Figure 6b 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++

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

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 not possible in C++ Kokkos kernels, which are defined as overloaded `operator()` methods 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. This transformation is shown in Figure 6c. 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 to preserve performance portability. This is what the original indexing method does internally, so this transformation essentially inlines the call to `operator()` to allow us to use `restrict`. 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).

This transformation gives the compiler necessary information to better optimize memory instructions. By adding the `restrict` keyword, we are telling the compiler that writes to one array cannot interfere with memory operations to other arrays, extending the validity of values loaded from memory which allows the compiler to reuse them instead of re-loading them from memory.

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 and 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 PyKokkos Runtime compiles and calls the fused kernel.

5 Evaluation

In this section we present our evaluation of `PyFuser`. We answer the following research questions:

RQ1. How does kernel fusion with `PyFuser` affect the performance of the kernels?

RQ2. In what cases is kernel fusion beneficial?

RQ3. What is the run-time overhead introduced by `PyFuser`?

First, we describe our evaluation setup (Section 5.1). Second, we introduce our test subjects (Section 5.2). Third, we answer our research questions (sections 5.3-5.5).

5.1 Evaluation Setup

We used Python 3.11, the latest version of PyKokkos (commit 3d4afd2), and Kokkos 3.7.02. We ran our experiments on multiple processors, including NVIDIA and AMD GPUs as well as Intel and AMD CPUs. Table 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 five runs. We used the simple kernel timer from the Kokkos Tools repository [1] 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.

Table 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

Table 2. Fusion speedup over unfused kernels, unoptimized (UO) and optimized (O), for PIC and ExaMiniMD.

Subject	Kernel	# Kernels	Speedup									
			V100		A100		MI250X		Xeon		EPYC	
			UO	O	UO	O	UO	O	UO	O	UO	O
PIC	DSMC	3	1.03	1.96	1.16	1.78	0.92	1.77	1.21	1.21	1.35	1.38
ExaMiniMD	F + C	2	0.98	0.99	0.87	0.88	0.97	1.02	1.01	1.10	1.03	1.01
ExaMiniMD	F + C + FI	3	0.87	0.91	0.94	0.88	0.90	0.96	1.17	1.16	1.04	1.00
ExaMiniMD	FI + II	2	1.40	1.52	1.60	1.60	1.28	1.62	1.51	1.43	1.60	1.67
ExaMiniMD	FI + II + E	3	1.24	1.61	1.67	1.69	1.18	1.30	1.82	1.69	1.88	1.92

5.2 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 originally in C++ Kokkos [2]. The PyKokkos version is ~3k lines of code and has 14 distinct kernels [5].
- **Particle-in-cell code**: a particle-in-cell (PIC) solver of the electron Boltzmann equation implemented originally in PyKokkos [7].
- **Gaussian Naive Bayes**: a PyKokkos implementation of scikit-learn's Gaussian Naive Bayes (GNB) classifier, which was originally written in NumPy [10].
- **NPBench**: a collection of NumPy code samples for evaluating frameworks that accelerate NumPy [38]. 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`.
- **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 [5].

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

5.3 Kernel Speedups

RQ1. *How does kernel fusion with* `PyFuser` *affect the performance of the kernels?*

Table 2 and Figure 7 show the speedups we get by fusing the kernels in our test subjects. Table 2 shows speedups for the Particle-in-cell code and ExaMiniMD both with and without transformations applied to evaluate the impact of the compilers' optimizations. Columns 1 and 2 show the source of the 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

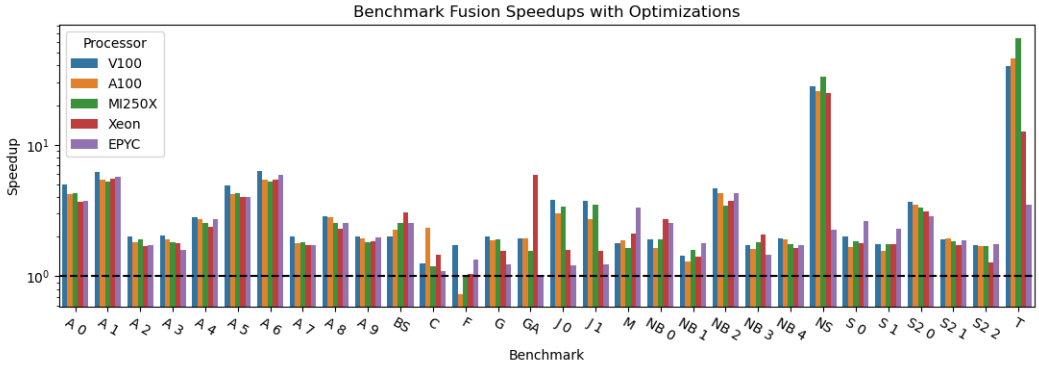


Fig. 7. Kernel fusion speedup over unfused kernels with our transformations for `adi` (A), `BabelStream` (BS), `covariance` (C), `fdtd_2d` (F), GUPS (G), `GUPS Atomic` (GA), `jacobi_1d` (J), `mvt` (M), Gaussian Naive Bayes (NB), `NSTREAM` (NS), `syrk` (S), `syr2k` (S2), and `Transpose` (T).

Table 3. Effectiveness of fusion and transformations on reducing redundant instructions and DRAM traffic.

Kernel	Arith. Inst. Saved [%]						Mem. Inst. Saved [%]						DRAM Bytes [%]					
	V100		A100		MI250X		V100		A100		MI250X		V100		A100		MI250X	
	UO	O	UO	O	UO	O	UO	O	UO	O	UO	O	UO	O	UO	O	UO	O
DSMC	0	28	1	28	0	40	0	52	3	52	-3	68	71	65	76	69	2	49
F + C	-3	-3	-4	-4	1	1	0	0	0	0	0	1	70	71	82	83	-2	4
F + C + FI	-3	-3	1	0	1	1	0	1	0	1	0	2	85	85	89	90	-3	4
FI + II	30	43	24	44	23	33	0	27	0	27	4	38	51	51	59	54	29	35
FI + II + E	43	46	39	46	39	47	-8	36	-8	36	12	44	68	67	73	69	37	39

to form a single kernel. The remaining columns show speedups on different processors when fusing these kernels, both Unoptimized (UO) and Optimized (O). Each row corresponds to a trace partition fused into a single kernel by `PyFuser`. Due to space constraints, we show the remaining speedups only with transformations applied in the bar plot in Figure 7 (the full results are shown in Appendix A). The x-axis shows the abbreviated name of each fused kernel and the y-axis shows the speedup. The kernel names are formed from the first letter of the source and the order of occurrence in the application. For example, for the `adi` benchmark, the first fused kernel is named `A 0`, the second `A 1`, etc. We calculate speedups by comparing execution times of the original unfused kernels called consecutively (i.e., how they were originally) to the time to execute the fused kernel.

Table 3 shows the reduction in instructions executed and memory traffic to DRAM. 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 [3] and AMD's Omniperf [22] profilers. We do not show CPU metrics as collecting them proved to be noisy due to non-kernel code. We use this table to explain the obtained speedups and slowdowns. We omit roofline analysis as it deals with execution rates while we look at speedups of wall clock times.

5.3.1 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 Table 2.

The results in Table 2 show that fusion alone does not always result in a large speedup. On the V100, A100, and MI250X GPUs, we see a speedup of 1.03×, a speedup of 1.16×, and a slowdown of 0.92× respectively without our transformations. After applying the transformations, we observe 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 attain their highest speedup without the need for transformations. Applying them does not result in a noticeable improvement.

In order to understand why we obtain these results, we examine the improvement in performance-related profiler metrics shown in Table 3. We first note that on all GPUs, better caching leads to significant reductions in DRAM traffic.

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.

These metrics indicate 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 compiler to eliminate a large number of redundant instructions, leading to large speedups for all GPUs.

In contrast to GPUs, CPUs benefit greatly from improved cache utilization, so reducing memory operations is not as essential. This explains why we obtained speedups on both CPUs even without our transformations, which proved to be unnecessary.

5.3.2 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 Table 2.

The first two fused kernels, F + C and F + C + FI, 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 slowdowns of 0.98×, 0.87×, and 0.97× on the V100, A100, and MI250X GPUs. Applying our transformations lead to small improvements on the NVIDIA GPUs, while the MI250X GPU rose to a 1.02× speedup. For the F + C + FI kernel, we see slowdowns of 0.87×, 0.94×, and 0.90× on the V100, A100, and MI250X GPUs. Applying our transformations lead to improvements on the V100 and MI250X, rising to 0.91× and 0.96× respectively, but decreased the speedup on the A100 to 0.88×.

Looking at Table 3, 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 Table 3). 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.

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 sub-optimal CUDA block size. Manually overriding it gives us similar results to F + C.

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). We have reached out to NVIDIA asking why that is the case.

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 show significant speedups, with the transformations further improving performance.

Table 3 shows large 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 have loops so the compiler optimizes the arithmetic operations even without our transformations. The amount of memory instructions is reduced after applying the restrict optimization, which further reduces the arithmetic instructions by eliminating memory address calculations.

The large reductions in arithmetic and memory instructions executed is the reason we observe larger speedups for these two kernels. The further reductions we observe after applying our transformations improve performance further.

As before, CPUs get large speedups even without our transformations due to better caching.

5.3.3 Gaussian Naive Bayes, NPBench, & Benchmarks. Gaussian Naive Bayes and NPBench [38] consist of kernels that perform NumPy-style array operations. The Benchmarks include less than 5 (unfused) kernels each, with most being small. In total, `PyFuser` generates 30 fused kernels.

From Figure 7 we see speedups for all kernels on all processors except for `fdtd_2d` (`F`) on the A100, which is 0.73× slower. On average, across all subjects on all processors, we see a 3.8× speedup.

We observe that the speedups are larger in the cases in which `PyFuser` is able to fuse large trace partitions. For example, `NSTREAM` (`NS`) and `Transpose` (`T`) call all kernels in a loop that runs for fifty iterations, which `PyFuser` fuses into one kernel, leading to large reductions in 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.

Table 4. PyFuser overhead from tracing, fusion, and transformations averaged for 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

5.4 Kernel Fusion Advantages

RQ2. *In what cases is kernel fusion beneficial?*

We hypothesized that applying kernel fusion improves performance because first, the compiler optimizes the code further; second, it leads to better reuse of data loaded from memory; and third, it reduces kernel launch overhead. In this section, we examine the validity of this hypothesis.

We test the validity of the first point for our GPUs by looking at speedups in Table 2 and reductions in arithmetic and memory instructions executed in Table 3. Larger reductions in instructions executed mean that the compiler optimized the code further by eliminating redundant instructions. We can see that kernel fusion does not always lead to more optimizations, but that our transformations do assist in this metric. In those cases where the compiler eliminated a larger amount of redundant instructions, we did see larger speedups. Thus, we can see that kernel fusion does often lead to improved compiler optimizations, which in turn correlates with improved performance.

To test the validity of the second point, we compare speedups to the reduction in memory traffic from DRAM. We see that kernel fusion improves this metric, which we attribute to better cache utilization. However, this does not necessarily correlate to improved performance on our GPUs.

For our CPUs, we were unable to collect these metrics for our kernels due to interference from other CPU processes running simultaneously. We always observed speedups on our CPUs even prior to applying our transformations. We therefore attribute the speedups on our CPUs to the improved cache utilization, which is known to be more impactful compared to GPUs [12].

The third and final point applies to all our processors. We know that fusing larger trace partitions leads to less kernel calls and less overhead. The largest speedups were in our Benchmarks, where we fused fifty kernel calls into one. Thus large reductions in overhead lead to the largest speedups.

In summary, we believe that our experiments support our hypothesis with some slight differences for our GPUs and CPUs. First, kernel fusion does allow the compiler to optimize better; however, further code transformations are required in some cases. GPUs benefit more from this point than CPUs since memory accesses on GPUs are relatively more expensive. Second, kernel fusion does allow for better reuse of data loaded from memory. Conversely to the first point, CPUs benefit more from this point than GPUs since caching is more effective. Third, fusing larger trace partitions does mean reduced kernel launch overhead, which also correlates to better performance.

5.5 Run-time Overhead

RQ3. *What is the run-time overhead introduced by* PyFuser*?*

In this section, we examine the overhead introduced by PyFuser. Table 4 shows the name of the processor in the first column. The remaining columns show the overhead of different modes of PyFuser as a percentage of original running time, averaged across all subjects. The modes shown are tracing without fusion (only lazy evaluation) in the second column, tracing with fusion in the third column, and tracing with fusion and transformations 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.

6 Limitations and Future Work

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 [19] and Legate [9]. In the future, we will investigate how kernel fusion and task scheduling systems can be combined to further improve performance.

Lazy evaluation could lead to out of order execution which makes debugging code harder. For PyFuser, we expect users to turn off lazy evaluation and kernel fusion when debugging code.

PyFuser implements lazy evaluation and kernel fusion specifically for PyKokkos. The just-in-time compiler available in PyKokkos allows us to compile the fused kernels we generate dynamically at run-time and then import them from Python. The techniques presented in this paper can also be applied to C++ Kokkos, especially the code transformations. However, we would still need to implement lazy evaluation and dynamic kernel fusion in C++, which is more challenging since C++ is statically compiled. One possible approach is to generate traces by logging kernel calls in an initial run, as well as information about the arguments passed, and then subsequently fusing the kernels and applying code transformations offline following this initial run, so that later runs can use the fused kernels. However, this approach differs significantly from the current approach used in PyFuser. As for other kernel frameworks, those that are implemented in dynamic languages such as Python could use similar techniques such as those in PyFuser, while those that are implemented in static languages would require an approach similar to the one described for C++ Kokkos. Furthermore, different languages and frameworks could require different code transformations to enable better compiler optimizations.

Doing fusion dynamically at run-time requires a run-time system to record traces of kernel calls. This introduces performance overhead that could offset the performance gains obtained by kernel fusion, which could be a deterrent to using such systems. In this paper, we focused on the kernel speedups while also reporting the overhead of PyFuser. Our current implementation of PyFuser is in Python, an interpreted language, which adds additional overhead compared to a compiled language (e.g., C++). It is possible to implement some of the analyses done by PyFuser in a compiled language rather than Python while retaining PyFuser's dynamic nature, which will help in improving the performance of PyFuser itself.

Kernel fusion could lead to unpredictable performance and slowdowns in some cases. We used profilers to explain the characteristics of fused kernels that resulted in speedups and slowdowns. In the future, we plan on integrating some of this knowledge into our fusion strategy (instead of the greedy strategy) to anticipate the fused kernels' performance. Since the fused kernels' performance varies across processors, we will incorporate this knowledge into the fusion strategy in the future.

7 Related Work

Kernel fusion is an active research area [13, 20, 24, 25, 35, 37]. An early attempt at CUDA kernel fusion required users to express code as CPU skeletons which are used to extract dependencies between kernels [25], while also exploring the effectiveness of different fusion strategies using a performance model [24]. Kernel Weaver [36] 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 [37] also experimented with kernel fission, splitting kernels to overlap computation with data transfer from main memory to GPU memory. [35] formulated kernel fusion as a combinatorial optimization problem and used a genetic algorithm with performance projection to explore the space of potential fusions, focusing on memory-bound CUDA kernels. The search happens offline and the recommended fusions are applied manually to the code base. This approach scales better than prior approaches and shows performance improvements on a codebase with 142 kernels; however, the search still takes up to twelve minutes. KFCM [13] is a code motion based kernel fusion technique for CUDA that exposes opportunities for static kernel fusion by moving kernel calls together across a control flow graph. Unlike `PyFuser`, these frameworks analyze kernel calls statically at compile-time, missing opportunities for fusion compared to our run-time approach. They also employ expensive offline heuristics to make fusion decisions. Also, through PyKokkos, `PyFuser` is not restricted to CUDA and can fuse OpenMP and HIP kernels.

Helium [23] is an OpenCL framework that uses lazy evaluation but focuses on kernel reordering to achieve better performance, while optionally fusing some. The evaluation focuses mainly on speedups from reordering and does not analyze the impact of fusion. In contrast, `PyFuser` focuses on fusion and we do a deep dive to understand where and how we obtain speedups with fusion.

Horizontal fusion [20] is a fusion technique that differs from classical (vertical) fusion. Instead of concatenating the contents of two consecutive kernel calls, horizontal fusion interleaves execution of the two kernels it fuses by splitting each thread block to work on each kernel simultaneously. This fusion technique could be integrated into `PyFuser`'s Fuser along with classical fusion.

Kernel fusion has been applied to Python NumPy-style frameworks where kernels are simple array operations [16–18, 26, 27]. Bohrium [17, 18] is a virtual machine that records array operations into a bytecode which is JIT-compiled into fused CPU kernels. It models 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. DelayRepay [26] is a drop-in replacement for NumPy that builds an AST from executed universal functions which it maps 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. Kernel fusion has also been proposed for machine learning algorithms. [8] identifies a common computational pattern across different ML algorithms and presents a reusable, manually fused kernel. There are two versions of the fused kernel, one for sparse and one for dense data. These frameworks specifically fuse kernels that are simple array operations, while `PyFuser` fuses arbitrary user-defined kernels.

The increasing popularity of frameworks such as PyTorch [28] has reignited interest in kernel fusion research for deep learning operators [21, 29]. LazyTensor [29] implements delayed execution by building traces of operations to generate XLA HLO IR [4], which is then compiled and called when data is requested by the user. Torchy [21] is a tracing JIT compiler for PyTorch that delays execution of PyTorch tensor operations and stores them in a trace, which can be further optimized via the PyTorch neural network compiler, before being flushed when data is requested by the user. The lazy evaluation approach in `PyFuser` is similar to Torchy's, bust since we can define custom kernels in PyKokkos we add safety checks to ensure correctness. Furthermore, fusing kernels in deep learning programs leads to speedups mostly due to reduced kernel launch overhead, whereas for `PyFuser`, we had to introduce code transformations to allow the compiler to optimize the code further. Kernel fusion has also been applied to fuse deep learning operators implemented using the CUTLASS library [30]. This approach statically fuses two GEMM kernels and a tensor kernel,

which is a common pattern in deep learning. These frameworks address fusion of domain-specific deep learning operators and not custom kernels like `PyFuser`.

[11] proposes applying kernel fusion to the packing and unpacking GPU kernels found in MPI applications. In order to move non-contiguous data across processors, MPI adds calls to pack the data and then unpack it before sending and receiving it. By fusing these kernels, this approach can speed up existing MPI applications that frequently communicate data across processors and compute nodes. `PyFuser` fuses kernels written by users from any domain, and would be able to fuse packing and unpacking kernels if they were written in PyKokkos.

Task fusion [31] 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 and data movement in a distributed computing environment, while in kernel fusion the goal is to generate more efficient kernels.

8 Conclusion

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. Fused kernels perform better due to better reuse of loaded data, improved compiler optimizations, and reduced kernel launch overhead. 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.

9 Data Availability

The submitted artifact contains three top-level directories: `pykokkos/`, which contains `PyFuser` and its integration with PyKokkos. The `PyFuser` specific code is found in `pykokkos/core/fusion/` and `pykokkos/core/optimizations/`. The test subjects are located in another top-level directory, `examples/`. Scripts to process the data are found under the third top-level directory, `scripts/`.

Acknowledgments

We thank Cheng Ding, Ivan Grigorik, Tong-Nong Lin, Yu Liu, Hannan Naeem, Pengyu Nie, Aditya Thimmaiah, Zhiqiang Zang, Jiyang Zhang, Linghan Zhong, and the anonymous reviewers for their feedback on this work. This work was partially supported by the US National Science Foundation under Grant Nos. CCF-2107291, CCF-2217696, CCF-2313027, and CCF-2403036, as well as the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003969.

A Appendix: Kernel Speedups

Table 5 shows detailed speedups for all our kernels, including both unoptimized and optimized results.

Table 5. Kernel fusion speedup over unfused kernels both unoptimized (UO) and optimized (O) for all kernels.

Subject	Kernel	# Kernels	Speedup									
			V100		A100		MI250X		Xeon		EPYC	
			UO	O	UO	O	UO	O	UO	O	UO	O
PIC	DSMC	3	1.03	1.96	1.16	1.78	0.92	1.77	1.21	1.21	1.35	1.38
ExaMiniMD	F + C	2	0.98	0.99	0.87	0.88	0.97	1.02	1.01	1.10	1.03	1.01
ExaMiniMD	F + C + FI	3	0.87	0.91	0.94	0.88	0.90	0.96	1.17	1.16	1.04	1.00
ExaMiniMD	FI + II	2	1.40	1.52	1.60	1.60	1.28	1.62	1.51	1.43	1.60	1.67
ExaMiniMD	FI + II + E	3	1.24	1.61	1.67	1.69	1.18	1.30	1.82	1.69	1.88	1.92
GaussianNB	NB 0	3	1.88	1.92	1.65	1.65	1.63	1.91	2.88	2.72	2.33	2.57
GaussianNB	NB 1	2	1.41	1.43	1.27	1.29	1.23	1.59	1.59	1.42	1.70	1.80
GaussianNB	NB 2	5	4.56	4.65	3.89	4.28	3.35	3.47	3.64	3.76	3.93	4.30
GaussianNB	NB 3	2	1.74	1.74	1.60	1.62	1.53	1.81	2.60	2.08	1.47	1.47
GaussianNB	NB 4	2	1.96	1.94	1.93	1.92	1.78	1.76	1.63	1.64	1.58	1.73
Microbenchmarks	BS	4	1.93	2.03	2.25	2.25	2.14	2.55	3.01	3.07	0.89	2.57
Microbenchmarks	G	2	2.02	2.01	1.94	1.88	1.88	1.90	1.56	1.57	1.22	1.22
Microbenchmarks	GA	2	1.96	1.96	1.94	1.94	1.86	1.56	6.16	5.90	1.01	1.02
Microbenchmarks	NS	50	4.67	27.70	5.12	25.41	3.23	32.86	6.55	24.63	1.25	2.28
Microbenchmarks	T	50	2.22	39.32	1.81	45.34	1.41	64.97	7.06	12.62	1.09	3.52
NPBench	A 0	5	5.00	5.00	4.28	4.22	4.06	4.29	4.12	3.69	3.63	3.74
NPBench	A 1	7	6.30	6.18	5.51	5.43	4.75	5.28	5.71	5.55	6.00	5.68
NPBench	A 2	2	2.00	2.00	1.95	1.82	1.80	1.91	1.88	1.69	1.79	1.72
NPBench	A 3	2	2.03	2.03	1.91	1.91	1.81	1.81	1.92	1.80	1.95	1.59
NPBench	A 4	3	2.73	2.81	2.77	2.72	2.50	2.55	2.64	2.38	2.22	2.71
NPBench	A 5	5	5.00	4.95	4.28	4.22	4.04	4.28	4.18	4.02	4.19	4.02
NPBench	A 6	7	6.30	6.30	5.45	5.45	4.77	5.25	5.70	5.47	6.15	5.90
NPBench	A 7	2	2.00	2.00	1.90	1.80	1.74	1.82	1.87	1.74	1.90	1.74
NPBench	A 8	3	2.73	2.87	2.91	2.81	2.53	2.56	2.66	2.31	2.74	2.55
NPBench	A 9	2	1.99	2.01	1.95	1.95	1.79	1.81	1.96	1.85	1.93	1.97
NPBench	C	3	1.12	1.25	1.24	2.35	1.03	1.20	1.50	1.47	1.01	1.10
NPBench	F	2	0.73	1.74	0.82	0.73	1.00	1.01	1.04	1.04	1.36	1.33
NPBench	J 0	4	3.86	3.79	3.08	2.99	3.40	3.40	2.25	1.58	1.59	1.22
NPBench	J 1	4	3.78	3.78	2.95	2.73	3.36	3.49	2.25	1.56	1.62	1.24
NPBench	M	2	1.84	1.79	1.63	1.87	1.63	1.63	2.04	2.12	3.84	3.34
NPBench	S 0	2	2.00	2.00	1.72	1.67	1.82	1.84	1.81	1.79	2.73	2.64
NPBench	S 1	2	1.76	1.76	1.61	1.57	1.69	1.76	1.85	1.77	6.17	2.31
NPBench	S2 0	4	3.64	3.68	3.55	3.49	2.99	3.32	3.54	3.13	2.83	2.85
NPBench	S2 1	2	1.94	1.92	1.95	1.95	1.84	1.84	2.18	1.74	1.83	1.87
NPBench	S2 2	2	1.71	1.72	1.83	1.69	1.69	1.69	1.49	1.28	1.71	1.77

References

[1] 2016. KokkosP Profiling Tools. https://github.com/kokkos/kokkos-tools.

[2] 2017. ExaMiniMD. https://github.com/ECP-copa/ExaMiniMD.

[3] 2024. Nsight Compute CLI. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html.

[4] 2024. XLA : Compiling Machine Learning for Peak Performance. https://openxla.org/xla.

[5] Nader Al Awar, Steven Zhu, George Biros, and Milos Gligoric. 2021. A Performance Portability Framework for Python. In *International Conference on Supercomputing*. 467–478. doi:10.1145/3447818.3460376

[6] Nader Al Awar, Steven Zhu, Neil Mehta, George Biros, and Milos Gligoric. 2022. PyKokkos: Performance Portable Kernels in Python. In *International Conference on Software Engineering, Tool Demonstrations Track*. 164–167. doi:10.

1145/3510454.3516827

[7] James Almgren-Bell, Nader Al Awar, Dilip S Geethakrishnan, Milos Gligoric, and George Biros. 2022. A Multi-GPU Python Solver for Low-Temperature Non-Equilibrium Plasmas. In *International Symposium on Computer Architecture and High Performance Computing*. 140–149. doi:10.1109/SBAC-PAD55451.2022.00025

[8] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P. Sadayappan. 2015. On Optimizing Machine Learning Workloads via Kernel Fusion. In *Symposium on Principles and Practice of Parallel Programming*. 173–182. doi:10.1145/2688500.2688521

[9] Michael Bauer and Michael Garland. 2019. Legate NumPy: accelerated and distributed array computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. Article 23, 23 pages. doi:10.1145/3295500.3356175

[10] 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. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122. doi:10.48550/arXiv.1309.0238

[11] Ching-Hsiang Chu, Kawthar Shafie Khorassani, Qinghua Zhou, Hari Subramoni, and Dhabaleswar K. Panda. 2020. Dynamic Kernel Fusion for Bulk Non-contiguous Data Transfer on GPU Clusters. In *International Conference on Cluster Computing*. 130–141. doi:10.1109/CLUSTER49012.2020.00023

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

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

[14] 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. 2018. Effective Performance Portability. In *International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 24–36. doi:10.1109/P3HPC.2018.00006

[15] Ken Kennedy and Kathryn S. McKinley. 1993. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *International Workshop on Languages and Compilers for Parallel Computing*. 301–320.

[16] Andreas Klöckner. 2014. Loo.py: transformation-based code generation for GPUs and CPUs. In *Workshop on Libraries, Languages, and Compilers for Array Programming*. doi:10.1145/2627373.2627387

[17] Mads R.B. Kristensen, Simon A.F. Lund, Troels Blum, and James Avery. 2016. Fusion of Parallel Array Operations. In *International Conference on Parallel Architectures and Compilation*. 71–85. doi:10.1145/2967938.2967945

[18] Mads R.B. Kristensen, Simon A.F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. 2014. Bohrium: A Virtual Machine Approach to Portable Parallelism. In *International Parallel and Distributed Processing Symposium Workshops*. 312–321. doi:10.1109/IPDPSW.2014.44

[19] 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. 2022. Parla: a Python orchestration system for heterogeneous architectures. In *International Conference on High Performance Computing, Networking, Storage and Analysis*. Article 51, 15 pages.

[20] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic Horizontal Fusion for GPU Kernels. In *International Symposium on Code Generation and Optimization*. 14–27. doi:10.1109/CGO53902.2022.9741270

[21] Nuno P. Lopes. 2023. Torchy: A Tracing JIT Compiler for PyTorch. In *International Conference on Compiler Construction*. 98–109. doi:10.1145/3578360.3580266

[22] Xiaomin Lu, Cole Ramos, Fei Zheng, Karl W. Schulz, Jose Santos, Keith Lowery, Nicholas Curtis, and Cristian Di Pietrantonio. 2023. *AMDResearch/omniperf: v1.1.0-PR1 (13 Oct 2023)*. doi:10.5281/zenodo.7314631

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

[24] Jiayuan Meng, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D. Uram. 2011. GROPHECY: GPU performance projection from CPU code skeletons. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. doi:10.1145/2063384.2063402

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

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

[27] 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. 2018. Evaluating end-to-end optimization for data analytics applications in weld. *Proceedings of the VLDB Endowment* 11, 9 (may 2018), 1002–1015. doi:10.14778/3213880.3213890

[28] 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. 2019. PyTorch: an imperative style, high-performance deep learning library. In *International Conference on Neural Information Processing Systems*. Article 721, 12 pages. doi:10.5555/3454287.3455008

[29] Alex Suhan, Davide Libenzi, Ailing Zhang, Parker Schuh, Brennan Saeta, Jie Young Sohn, and Denys Shabalin. 2021. LazyTensor: combining eager execution with domain-specific compilers. doi:10.48550/arXiv.2102.13267 arXiv:2102.13267

[30] Wei Sun, Ang Li, Sander Stuijk, and Henk Corporaal. 2024. How Much Can We Gain From Tensor Kernel Fusion on GPUs? *IEEE Access* 12 (2024), 126135–126144. doi:10.1109/ACCESS.2024.3411473

[31] Shiv Sundram, Wonchan Lee, and Alex Aiken. 2022. Task Fusion in Distributed Runtimes. In *ACM Parallel Applications Workshop: Alternatives To MPI+X*. 13–25. doi:10.1109/PAW-ATM56565.2022.00007

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

[33] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. 2021. The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing. *Computing in Science Engineering* 23, 5 (2021), 10–18. doi:10.1109/MCSE.2021.3098509

[34] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. doi:10.1109/TPDS.2021.3097283

[35] Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 191–202. doi:10.1109/SC.2014.21

[36] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *International Symposium on Microarchitecture*. 107–118. doi:10.1109/MICRO.2012.19

[37] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. 2012. Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission. In *International Parallel and Distributed Processing Symposium Workshops and PhD Forum*. 2433–2442. doi:10.1109/IPDPSW.2012.300

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

[39] Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, and Torsten Hoefler. 2021. Productivity, portability, performance: data-centric Python. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. Article 95, 13 pages. doi:10.1145/3458817.3476176