Bridging Control-Centric and Data-Centric Optimization

Tal Ben-Nun*
ETH Zurich
Zurich, Switzerland
talbn@inf.ethz.ch

Berke Ates*
ETH Zurich
Zurich, Switzerland
beates@student.ethz.ch

Alexandru Calotoiu
ETH Zurich
Zurich, Switzerland
acalotoiu@inf.ethz.ch

Torsten Hoefler
ETH Zurich
Zurich, Switzerland
htor@inf.ethz.ch

Abstract

With the rise of specialized hardware and new programming languages, code optimization has shifted its focus towards promoting data locality. Most production-grade compilers adopt a control-centric mindset — instruction-driven optimization augmented with scalar-based dataflow — whereas other approaches provide domain-specific and general purpose data movement minimization, which can miss important control-flow optimizations. As the two representations are not commutable, users must choose one over the other. In this paper, we explore how both control- and data-centric approaches can work in tandem via the Multi-Level Intermediate Representation (MLIR) framework. Through a combination of an MLIR dialect and specialized passes, we recover parametric, symbolic dataflow that can be optimized within the DaCe framework. We combine the two views into a single pipeline, called DCIR, showing that it is strictly more powerful than either view. On several benchmarks and a real-world application in C, we show that our proposed pipeline consistently outperforms MLIR and automatically uncovers new optimization opportunities with no additional effort.

CCS Concepts: • **Software and its engineering** → **Correctness**; **Software performance**; **Compilers**.

Keywords: MLIR, DaCe, data-centric programming
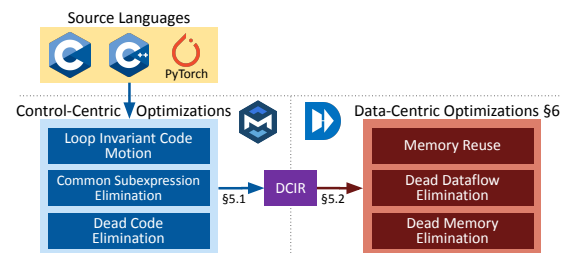
*Both authors contributed equally to the paper.

Figure 1. Overview of DCIR.

1 Introduction

Overcoming general-purpose optimization barriers is supported today by software optimization frameworks and intermediate representations (IRs). Whether driven by domain-specific optimizations [7, 11, 15, 34] or by specializing to emergent hardware [21, 29], each such framework delivers benefits that cannot be covered by general compilers.

The Multi-Level Intermediate Representation (MLIR) [19] project aims to standardize such efforts via the use of *dialects* and a homogenized pass infrastructure. Some classes of full-program optimizations, however, cannot natively fall into the category of a dialect. Data locality within operators, but also across entire applications, becomes crucial to manage [33], which led to the emergence of *data-centric* abstractions [1, 3, 35, 36] and optimization methodologies [13]. Such abstractions are promising in that they can verify programs in novel ways (bounds analysis, potential race condition detection), and mutate data movement (global layout management, distributed domain decomposition, local caching, etc.) and allocation (e.g., eliding memory subregions or entire arrays). However, to operate, data-centric optimizations require the visibility, whether constant or parametric, of all memory accesses for their analysis.

```
int example() {
  int *A = (int*)malloc(100000 * sizeof(int));
  int *B = (int*)malloc(100000 * sizeof(int));
  for (int i = 0; i < 100000; ++i) {
    A[i] = 5;
    for (int j = 0; j < 100000; ++j)
      B[j] = A[i];
    for (int j = 0; j < 10000; ++j)
      A[j] = A[i];
  }
  int res = B[0];
  free(A);
  free(B);
  return res;
}
```
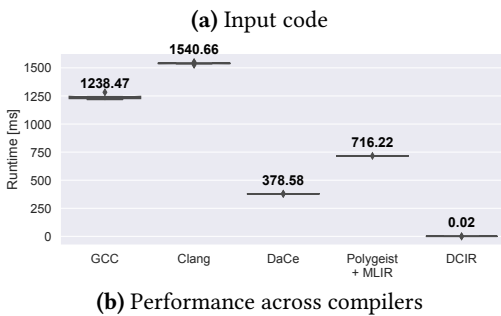
(a) Input code

(b) Performance across compilers

Figure 2. Mixed control- and data-centric analysis.

In this work, we build a general-purpose bridging infrastructure between existing compiler infrastructure and data-centric representations, called **DCIR**, benefiting from both aspects in a single compilation pipeline (Fig. 1). MLIR serves as a natural "connective tissue", with lowering and conversion from frontends in different languages and to backends covering a plethora of hardware architectures. For the data-centric representation, we choose the DaCe framework and its stateful dataflow multigraph (SDFG) IR [3], which is capable of optimizing a wide variety of applications through data movement minimization [4, 6, 8, 17, 28, 37, 38].

The usefulness of mixed-mode optimization can be demonstrated with a simple C example, shown in Fig. 2. Both production compilers and data-centric IRs produce suboptimal code on their own. Optimizations applied by GCC and LLVM are able to fuse the first two loops, setting every entry of A and B to 5, with MLIR further optimizing the write schedule to A. However, the third loop remains intact, although the entire memory allocation is unnecessary. Similarly, the DaCe framework can potentially elide the third loop, but requires global code motion to find the false dependency across the two arrays. By combining both optimization classes, DCIR elides all loops, reducing the code to a single statement.

To convert existing MLIR dialects to the SDFG IR, we augment global data movement analysis in MLIR. DCIR employs a multi-stage approach on both IRs, starting from introducing symbolic sizes to MLIR memrefs and propagating them to recover parametric subsets of moved data. We then lift and regroup computations to mitigate reliance on link-time optimization. Lastly, we introduce specialized data-centric transformations to the DaCe framework to better expose statically analyzable data movement generated by MLIR.

To evaluate the representational strength of mixed-mode optimization, we run a variety of applications with a standard optimizing pipeline (-O2, MLIR→SDFG) and *no domain-specific techniques*. The results indicate that the DCIR pipeline can recover the semantics necessary for standard MLIR dialects to match and outperform the original input codes, as well as expose new optimization opportunities automatically. The paper makes the following contributions:

- Introducing data-centric optimizations to a standard multi-level compilation pipeline through an MLIR dialect (§3) and a complementary set of passes (§6);
- Addressing MLIR limitations that inhibit global data movement analysis (§3.1) and efficiency (§7.2);
- Demonstrating performance results (§7) on small and real-world benchmarks, exhibiting 1.59× geomean speedup in Polybench/C over MLIR, 2.33× speedup for a deep learning activation function, and 7× speedup in MILC over the best general-purpose compiler.

2 Background

We first discuss the approaches we build upon, namely the MLIR [19] and DaCe [3] frameworks.

2.1 MLIR

Most compiler frameworks such as LLVM [18] and JVM [20] rely on a single abstraction level — programs are translated from a given language into an intermediate representation, and all transformation, optimization, and verification happens on it. By providing a framework in which multiple level of abstraction can coexist, MLIR [19] aims to make the development and interoperability of different IRs easier.

An intermediate representation within MLIR is called a *dialect*. Dialects provide specialized operators and data types, as well as conversion points from higher-level IRs (lowering) and equivalent-level IRs (converters). MLIR contains several core dialects and the rest are defined by users. Most dialects provide lowering capabilities one level down, thus saving redundant work in compiler infrastructure development.

MLIR is gaining community traction on front and back ends. Frontend projects such as Flang [22] translate FORTRAN code to MLIR dialects. For backends, Katel et al. [14] recently conducted a performance evaluation comparing MLIR to CUBLAS [26] on matrix-matrix multiplications and showed that MLIR could generate competitive code for GPUs.

In our workflow, we leverage Polygeist [24], a prototype frontend for MLIR that is capable of translating subsets of C and C++ to MLIR dialects. Polygeist emits MLIR code using the structured control flow (scf), arithmetic (arith), and the memory reference (memref) dialects. These three dialects are sufficient for Polygeist to express most C constructs: the structured control flow dialect encompasses loops and branch constructs, the arithmetic dialect can be used to express arithmetic and logical operations, and the memref

dialect describes data layout, array dimensions, and sizes. While Polygeist is not currently sufficiently stable to translate large scientific benchmarks, it can handle the Polybench suite [27] and snippets of real-world programs (§7).

2.2 DaCe

There are many optimization techniques aimed at improving data movement, though most are specific to particular hardware systems or even individual hardware configurations. At the same time, requiring high-level programs to consider data movement comes at the cost of added code complexity.

DaCe is a programming framework that addresses this issue by defining an IR called Stateful DataFlow multiGraphs (SDFG) [3], built around understanding data movement. DaCe provides frontends to translate code written in Python, Octave, or C into the SDFG IR. It also provides a transformation API on the IR to separate the concerns of the developer and the performance engineer. DaCe has succesfully improved the performance of applications in weather and climate models [4, 8], sparse linear algebra in quantum transport simulation [37], graph analytics [3], and full neural network optimization in deep learning [28].

Performance benefits in DaCe stem from a combination of local and global optimizations. Local optimizations are expressed as graph rewriting transformations, which perform a variety of operations: adding or removing explicit memory allocation, e.g., for increased cache utilization; SIMD vectorization; tiling parallel sections; and converting random-access memory into streaming (FIFO queues) when beneficial. For global optimizations, the same techniques exposed by the SDFG IR enable changing communication schemes in distributed applications and array layouts based on data movement modeling, or memory footprint reduction via memory region "live-set" analysis.

Optimization in the DaCe framework start by an SDFG simplification pass, which enlarges the pure dataflow regions and removes redundant memory allocation, equivalent to `-O1` in compilers. Subsequently, optimizations are applied via automated heuristics (`-O2`), followed by potential manual application of transformations by performance engineers.

The SDFG IR itself is represented by a control-flow graph (state machine) of dataflow acyclic multigraphs. SDFGs separate data containers and data movement (represented as dataflow edges called *memlets*) from their use in computational nodes (called *tasklets*). This allows the IR to express data movement explicitly — and even relies on data dependencies to create the execution order. Explicit control-flow is only used when dataflow cannot be otherwise inferred.

An important tool the SDFG IR relies on is parametric representation of data access patterns. By using symbolic expressions to represent array accesses, it becomes possible, for example, to determine whether or not accesses may overlap (whether sparse/indirect or dense).

```
func @fName (%A: memref<?xi32>,
             %B: memref<?xi32>) {
 memref.copy %A, %B : memref<?xi32>
 return
}
```

(a) Function that copies the entries of one `memref` to another

```
func @fName (%A: sdfg.array<sym("2*N")xi32>,
             %B: sdfg.array<sym("N")xi32>) {
 sdfg.copy %A, %B : !sdfg.array<sym("N")xi32>
 return
}
```

(b) Symbolic version of the same function detects the size mismatch

Figure 3. Parametric size verification with the `sdfg` dialect.

Conversely, SDFGs cannot natively represent passes such as loop-invariant code motion or general common subexpression elimination, which control-centric IRs routinely implement. Because the tasklet is seen as an atomic unit, its contents cannot be inspected for transformations. Additionally, some design patterns cannot be represented concisely in SDFGs, such as parametric-depth recursion or dynamic pointers, which could be adequately handled by other IRs. Therefore, a bridge between SDFG and MLIR is a logical choice that can aid optimization on both ends.

3 Data-Centric MLIR Dialect

At the core of our proposed bridge — DCIR — lies the `sdfg` dialect. Its purpose is to exist as a convertible target to/from the standard MLIR dialects within the MLIR dialect conversion framework, as well as a representation that is directly translatable to the DaCe SDFG IR. Conversion to the `sdfg` dialect, however, is not straightforward due to fundamental representational differences:

1. All SDFG data containers must be defined with a predetermined constant or parametric size;
2. Data movement granularity (subregions, indices) must be specified at each scope;
3. Memory operations in SDFG are divided into load, store, and update, as opposed to the load/store view in standard MLIR dialects.

Since SDFGs rely on parametric dataflow for their analyses (as indicated by the first two differences), we must first augment MLIR to enable expressing both parametric data movement and data container definition.

3.1 Symbols

MLIR introduces the concept of `memrefs`, which enable keeping track of the dimensionality of allocated arrays through multiple levels and enables memory-based optimizations. The memref type allows undefined dimensions with the question mark (?), which is useful to define functions that can work on arrays of arbitrary size. Such a function is presented

Table 1. sdfg Dialect Operations

Operation	Description
`sdfg.tasklet(%a: i32) -> i32 {...}`	**Computation:** Encapsulated unit of computation, with no external dataflow except for parameters and return values.
`sdfg.load %A[0] : !sdfg.array<2xi32> -> i32`	**Loading:** Loads a value from an array.
`sdfg.store %a, %A[0]: i32 -> !sdfg.array<2xi32>`	**Storing and Updating:** Stores a value to an array, or updates it if an update function is given.
`sdfg.alloc() : !sdfg.array<2xi32>`	**Allocation:** Allocates a data container (array, stream, etc.) of the specified size. May contain symbolic sizes as well.
`sdfg.map (%i) = (0) to (sym("N")) step (1) {...}`	**Parametric Parallelism:** Represents a scope that is executed in parallel.
`sdfg.state @state_0 {...}`	**States:** Groups multiple operations. The state machine ensures a correct order of execution and prevents data races.
`sdfg.edge @state_0 -> @state_1`	**State Transition:** Describes the edges of the state machine, linking states together to a directed graph.

in Fig. 3a, which copies one array's contents to another. However, this question mark also prevents statically checking for any size mismatches in the copy operation. The same issue arises with the star (*) representing arbitrary dimensionality.

DCIR introduces symbolic expressions directly in data type sizes for parametric dataflow analysis. *Regardless of the dialect*, MLIR inherently disallows defining an identifier used within the types of the function parameters (for example, a function parameter %N and another parameter of type memref<(%N)xf32>). We thus resort to maintaining a symbol store and introduce the `sym` keyword. Symbols are defined globally per module or scope by their name, retaining their (albeit unknown) value throughout their lifetime. Since MLIR disallows defining arbitrary parentheses-enclosed syntax such as $(N+1), nor non-affine expressions, we rely on strings to represent the expressions. Certain scopes (e.g., functions) may accept symbol mappings as attributes to keep correspondence across an application.

Symbolic expressions allow data-centric dialects to perform validation and track unknown sizes during passes. Fig. 3 shows a direct comparison of the same function in the `memref` and `sdfg` dialects. Because the `sdfg.array` type supports symbolic sizes, we can encode the relationship of array sizes while maintaining the flexibility of compile-time undefined sizes. As we can see in Fig. 3b, the `sdfg` dialect can statically check for size mismatches and raise an exception at compile-time. This enables catching errors early and avoiding out of bounds access, as well as let passes such as loop tiling track symbolic sizes to produce faster code.

3.2 Dialect Elements

We introduce the `sdfg` dialect, which closely follows the SDFG IR structure, is convertible from the standard dialects, and addresses all three aforementioned representational differences. Its operations are listed in Table 1.

As dictated by the definition of the SDFG IR (§2.2), computations (tasklet graph nodes) are defined separately from data movement (edges). Computations are encapsulated in the `sdfg.tasklet` operator, which contains an attached scope. It is an MLIR `IsolatedFromAbove` scope, which can only operate on scalars and memory references given to it as input arguments. A tasklet can also have zero or more scalars and/or memrefs as outputs. The contents of a tasklet scope can be arbitrary (including loops and other structured control flow constructs), and use any lowerable MLIR dialect, but cannot access any SSA value or memory address outside the ones given to it via memlets. This ensures convertibility to SDFG and analyzability in DaCe. As symbols are read-only throughout their lifetime, they can be readily accessed.

Data movement is abstracted differently than in the SDFG IR. In order to increase readability, as well as maintain a linear-time conversion pass from `memref`, the dialect provides the `sdfg.load` and `sdfg.store` operators. The latter operator can also perform updates (e.g., atomic operations or one-sided communication) via the `wcr` function attribute (Write-Conflict Resolution in SDFG jargon).

Memory allocation is simplified in our proposed dialect. Since DaCe defines its own allocation lifetime policies (e.g., to save memory footprint), allocation in the generated code is implicit. Thus, all that is necessary in the dialect is to define an `sdfg.alloc` operation for each data container that DaCe supports. The data container type (e.g., `sdfg.array`, `sdfg.stream` for FIFO queues) uses either constants or symbolic expressions to define its size. Attributes specify the allocation lifetime and whether the allocation management should be performed by the SDFG (called *transient* containers) or not, as well as any aliasing information on the latter.

Parametric parallelism in SDFGs is defined by *map* and *producer/consumer* scopes. The closest equivalent of `sdfg.map` is `affine.parallel`, both of which define parallel execution (implemented, e.g., by a GPU kernel, or by FPGA processing elements). Although there is no direct conversion for consume scopes, the construct exists for full commutability between data-centric and control-centric optimizations.

Lastly, when dataflow cannot imply the program order, control-flow constructs are expressed as a finite state machine (similarly to MLIR's `fsm` dialect). Provided by the `sdfg.state` and `sdfg.edge`, structured constructs and general CFGs can be represented via the induced state machine. As opposed to LLVM-style CFGs, edges encode conditions and assignments as symbolic expressions, enabling constant-time testing of data-dependent control flow and retaining semantics of, e.g., switch-case constructs.

In sum, the proposed dialect is, on the one hand, designed for simple conversion from matching components in existing MLIR dialects, and on the other, provides one-to-one compatibility with the SDFG IR. To use the dialect, however, we would need to lift parametric data movement semantics (e.g., symbolic sizes) from existing MLIR codes. Below, we discuss the conversion and translation to the data-centric IR.

4 Conversion Pipeline

Due to the fundamental differences between the representations, the translation process is not direct, requiring analysis and transformation passes in the process. Fig. 4 broadly describes the conversion pipeline we use, highlighting passes in the MLIR domain in blue and DaCe in red. As DaCe generates C code, we can take the resulting representation and feed it back to MLIR for bidirectional translation.

The process (demonstrated in Fig. 5) starts with input code and a frontend, for which we use C/C++ and Polygeist [24], which is the state of the art MLIR frontend for those languages. Our proposed pipeline starts with Polygeist generated code in the `builtin`, `func`, `memref`, `arith`, `math` and `affine` dialects. We then apply a suite of typical control-centric passes — loop-invariant code motion, dead code elimination, common subexpression elimination, function inlining — and lower code to adhere to the dialects `sdfg` can be converted from (Fig. 5b). The pipeline then generates the corresponding code in the `sdfg` dialect (Fig. 5c), described in Section 5, and translates it to the SDFG IR (Fig. 5d). Following translation, we must perform additional passes on the SDFG (§6) to ensure that data-centric optimizations can be applied. Lastly, we apply a suite of typical data-centric passes for data movement reduction and memory scheduling.

5 MLIR Conversion and Translation

To work, our figurative bridge needs to implement two interfaces on the MLIR and DaCe endpoints. In MLIR nomenclature, these components are *converters*, which transpile one dialect to another, and *translators*, which interface MLIR with other representations. We now describe the design decisions and considerations made in implementing both components.

5.1 Converter

A converter needs source dialects to convert from. We choose four core source MLIR dialects — `scf`, `arith`, `math`, and
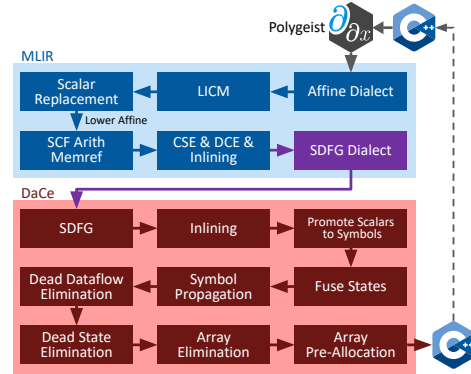

Figure 4. Overview of the DCIR conversion pipeline.

`memref` — in order to maximize compatibility (as most frontends and external dialects specify lowering to those). The conversion process is nearly direct: (a) memory allocation and load/store operations from the `memref` dialect are converted to `sdfg.{alloc,load,store}`; (b) arithmetic and mathematical computations, as well as unknown operations, are converted into separate `sdfg.tasklets` (see below); and (c) `scf` constructs are lowered to state machine subgraphs.

Any encountered question mark or star in `memref` sizes is replaced with a *unique identifier*, preserving original MLIR semantics. In the example in Fig. 5 this can be seen with `sym("s_0")` (❶). As conversion progresses on the MLIR side, the number of symbols is gradually reduced by propagating their values forward through references. The process is limited to assignments (rather than arbitrary subsets), but the process continues with DaCe's symbolic math engine as a data-centric transformation (§6.1).

It is crucial to track provenance of memory operations. Whenever a load or store is encountered, it is converted to data-centric scoping — propagating data dependencies outwards. Namely, an operation is injected in every scope (tasklet, control-flow construct, function, or parallel map-/consume) it is used, and all parent scopes must specify a *symbolic* subset of the data moved from the outer scope (❷). If analysis cannot determine the subregion, it is set to be equal to the outer region and refined later with DaCe's symbolic math engine.

To express computations in the `sdfg` dialect, the converter generates SDFG states and tasklets. In order to retain program order semantics, we first place every computation in its own `sdfg.state` (❸), which may be subsequently fused in DaCe (§6). We also split each computational operator into an individual tasklet, allowing DaCe to recover dataflow.

Conversion in the other direction poses fewer challenges. Similarly to SDFG code generation [3], tasklet contents can be inlined to MLIR core dialects, structured control flow can be raised from the state machine (e.g., using dominator analysis), and symbols become scalar identifiers.
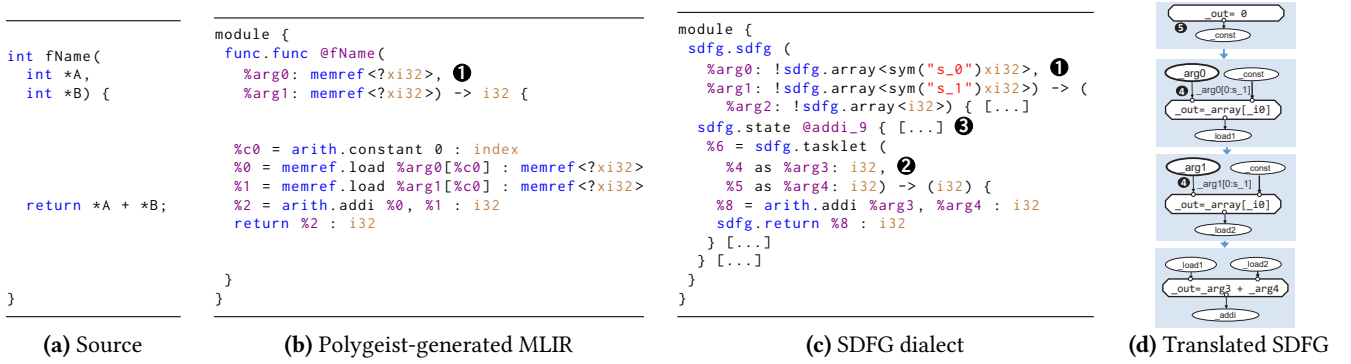
```
int fName(
  int *A,
  int *B) {




  return *A + *B;


}
```
(a) Source

```
module {
 func.func @fName(
   %arg0: memref<?xi32>, ❶
   %arg1: memref<?xi32>) -> i32 {


   %c0 = arith.constant 0 : index
   %0 = memref.load %arg0[%c0] : memref<?xi32>
   %1 = memref.load %arg1[%c0] : memref<?xi32>
   %2 = arith.addi %0, %1 : i32
   return %2 : i32


 }
}
```
(b) Polygeist-generated MLIR

```
module {
 sdfg.sdfg (
   %arg0: !sdfg.array<sym("s_0")xi32>, ❶
   %arg1: !sdfg.array<sym("s_1")xi32>) -> (
     %arg2: !sdfg.array<i32>) { [...]
  sdfg.state @addi_9 { [...] ❸
   %6 = sdfg.tasklet (
     %4 as %arg3: i32, ❷
     %5 as %arg4: i32) -> (i32) {
    %8 = arith.addi %arg3, %arg4 : i32
    sdfg.return %8 : i32
   } [...]
  } [...]
 }
}
```
(c) SDFG dialect

(d) Translated SDFG

Figure 5. Conversion and translation of a simple program.

5.2 MLIR-to-SDFG Translator

Once the input program is in the sdfg dialect, it is translated to the SDFG format in two passes. The first pass collects symbol, container, and scope metadata for constructing the graph; and the second pass creates and connects the graph.

MLIR operators that are not supported by our converter are kept as-is and compiled as *MLIR tasklets*. We add this functionality to DaCe by creating shims that convert data containers to memrefs, compiling the tasklet contents with mlir-opt and llc, and linking the resulting module.

As MLIR tasklets create multiple object files, they are only optimized via link-time optimization (LTO) and may thus yield lower performance than with single translation unit passes. During translation, we resolve this by *raising* MLIR tasklets to Python tasklets (which are native to DaCe) if possible. This includes parsing arithmetic operators into Python equivalents (e.g., arith.addi %a, %b to a + b) and built-in math library calls. Raising not only inlines the tasklet code during compilation in DaCe, but also enables data-centric analyses and transformations, including arithmetic intensity estimation and symbolic auto-differentiation [28].

6 Data-Centric Passes

Trivial translation of MLIR to SDFG may yield a data-centric representation that is functional, but not optimizable. We therefore make additional adaptations on the resulting IR to increase data movement analysis capabilities. Subsequently, we apply automatic optimizations (-O1 and -O2 compiler-flag equivalent) that reduce data movement to increase application performance.

6.1 Inference

We begin with global inference of symbolic expressions, update operations, and increasing the size of pure dataflow regions (states). The resulting recovered information enriches dataflow analysis and enables more optimizations.

Scalar to Symbol Promotion. This first pass elevates scalar values into symbolic expressions, if they can be represented as such, are used in indices/shapes, and do not change

during their lifetime (❹). In our example, _const is elevated to symbol, and therefore the memlet _arg0[0:s] becomes _arg0[_const]. This is crucial for MLIR codes, as every SSA value becomes a scalar data container. With this pass, index expressions, loop bounds, data-dependent memory allocations, and other elements are readily exposed to DaCe.

Symbolic Inference. Given DCIR's tendency to create many symbols (e.g., for each question mark), reducing their number is crucial for validation and optimization. The *symbol propagation* pass works similarly to constant propagation, forwarding values of symbolic expressions and replacing symbols if they are set once (❺). Propagation works in two passes over the code — one to collect symbols that are assigned once (via forward and reverse reachability), and another to make the necessary replacements. In the example from Fig. 5d, _const is detected to be 0, and that value is then propagated so the memlet _arg0[_const] becomes _arg0[0]. Additionally, on every function call, an attempt is made to reduce symbols by solving a system of equations.

Update Detection. SDFGs support a third mode of data movement: update. Differentiating between updates and writes is important for several optimizations, including automatic parallelization [6], detecting and improving reduction schedules, and choosing wait-free operations such as one-sided communication. We invoke the AugAssignToWCR DaCe transformation to detect updates via symbolic expression tracing around tasklets. If a read and a write operate on the same memory location, and it is expressible by an associative binary function, the read/write is converted to an update.

SDFG Simplification. The simplification pipeline [3] is an idempotent process that repeatedly fuses control-flow elements (states and nested CFGs) to enlarge pure dataflow regions. This functionality is given via the built-in DaCe API, which we invoke using sdfg.simplify(). More specifically, the method fuses SDFG states if their data dependencies can be expressed in one acyclic graph without introducing data races; and hoists control-flow that is situated inside dataflow (such as a branch inside parallel dataflow sections).

Once symbolic information is appropriately annotated, we can reduce unnecessary copies and memory allocation.

6.2 Data Movement Reduction (-O1)

In this work, we introduce novel data-centric passes into the DaCe framework, in order to support conversion of arbitrary MLIR codes to efficient programs.

Extended Dead Code Elimination (DCE). As a bridging pass between control- and data-centric transformations, we extend the notion of DCE in DaCe. Global dataflow and symbolic analysis lend their usefulness in DCE, which we can use to eliminate complete sequences of array operations. We perform the novel DCE in two separate passes: Dead State Elimination and Dead Dataflow Elimination. The former uses the propagated symbols to determine whether an expression will always be false, and eliminates unreachable state machine states. Dead Dataflow Elimination then traverses the state machine in reverse topological order, tracking future-reused data containers (arrays, scalars) and removing all computations that end up in unused temporary containers.

Array Elimination. "Dead memory" does not only refer to unused arrays, but also to extraneous memory copies and subregions that remain unused. DaCe can already detect redundant copies [3], but we extend this feature from a local transformation to a pass that reduces memory usage via removing arrays and views through a linear-time traversal.

Memlet Consolidation. After converting MLIR dialects and propagating data dependencies (§5.1), we may end up with multiple memlets that refer to overlapping regions. For example, a stencil that reads A[i] and A[i + 1] would generate two separate memlets. We thus define a pass that unions memlets that refer to the same containers within the same scope, as a form of data movement common denominator.

6.3 Memory Scheduling Optimizations (-O2)

Our final step in the pipeline is to optimize the program and its order based on its relationship with its memory. In particular, we apply two optimizations on the program schedule:

Memory (Pre-)Allocation. DaCe already controls memory allocation and deallocation implicitly, based on data containers' lifetime. However, it can still yield suboptimal performance with arbitrary MLIR codes, e.g., if an array is allocated within a critical loop, causing many unnecessary calls. Two passes deal with this: one heuristic analyzes the program, deciding if a container could be allocated on the stack or registers rather than the heap; the other heuristic moves memory allocation to the outermost scope it can (if no data races occur), removing such calls from the critical path.

Memory-Reducing Loop Fusion. The inferred explicit, reduced symbolic expressions allow us to analyze computational schedules and mutate them to further minimize data movement. We greedily invoke the existing fusion transformations available in DaCe as a heuristic pass. Those transformations merge scopes that write and read from otherwise-unused intermediate data, if memory access pattern permits to do so. This reduces the size of the intermediate array to a scalar (or the common subregion, e.g., if sparse), promoting cache locality and reducing memory footprint.

Overall, the set of optimizations we perform in the pipeline is conservative. Other scheduling and footprint reduction passes (e.g., relayouting, memory pooling) can be beneficial for allocation but harm cache locality, thus they are only enabled manually. As we shall show, the above set still yields promising performance gains on general applications.

7 Evaluation

In this section, we show that a single pass through our pipeline can generate code that matches or outperforms established compilers.

7.1 Experimental Setup

For our evaluation, we use the DCIR pipeline, described in Section 4, without feeding the generated code back into the pipeline. We use DaCe (version 0.14), and disable auto-parallelization in order to ensure a uniform execution environment for all compilers and prevent conflating memory benefits with parallel efficiency. Unless otherwise mentioned, we use Clang++[1] to compile the generated C++ code from DaCe. We run all of our benchmarks on a server that contains an Intel 16-core Xeon Gold 6130 CPU (clocked at 2.80 GHz) and 1.5 TB DDR4 RAM. The server consists of 32 KB of L1 caches, 1 MB of L2 caches, and a 22 MB L3 cache.

We compare the generated code with the established compilers GCC (version 12.1.0) and Clang[1] with the -O3 -fPIC -march=native flags. For one application (gramschmidt), we instead use the -O2 flag, as due to the numerical sensitivity of the application, the baseline compilers generate wrong results on -O3. Additionally, we run the benchmarks directly using the DaCe C frontend[2] using the same flags. Performance counters were measured with PAPI [32] 7.0.0.

We generate MLIR[1] code from C via Polygeist[3], applying the same optimization passes as in our pipeline and lowering it to the LLVM IR, which we compile using llc[1] and Clang[1], with the same flags. This not only provides another pipeline to compare with, but also allows separating the performance gains of the control-centric and data-centric optimizations.

7.2 Fundamental Computational Kernels

In order to evaluate our pipeline on practical applications, we choose the Polybench/C [27] benchmark (version 4.2.1) because it contains numerical computations with static control flow from various application domains, such as linear algebra, image processing, physics simulation, dynamic programming, and statistics. This provides a wide range of practical applications and enables a more general evaluation. Since Polybench kernels allocate all memory in advance, it

---

[1]Commit: 00a12585933ef63ff1204bf5cd265f0071d04642

[2]Commit: e7fe56262ba20b6e7e664eef169dcf32b1907be9

[3]Commit: fc15676b30c80ac1adb10f4fc3e7d7e8fe3ed7b6

Tal Ben-Nun, Berke Ates, Alexandru Calotoiu, and Torsten Hoefler
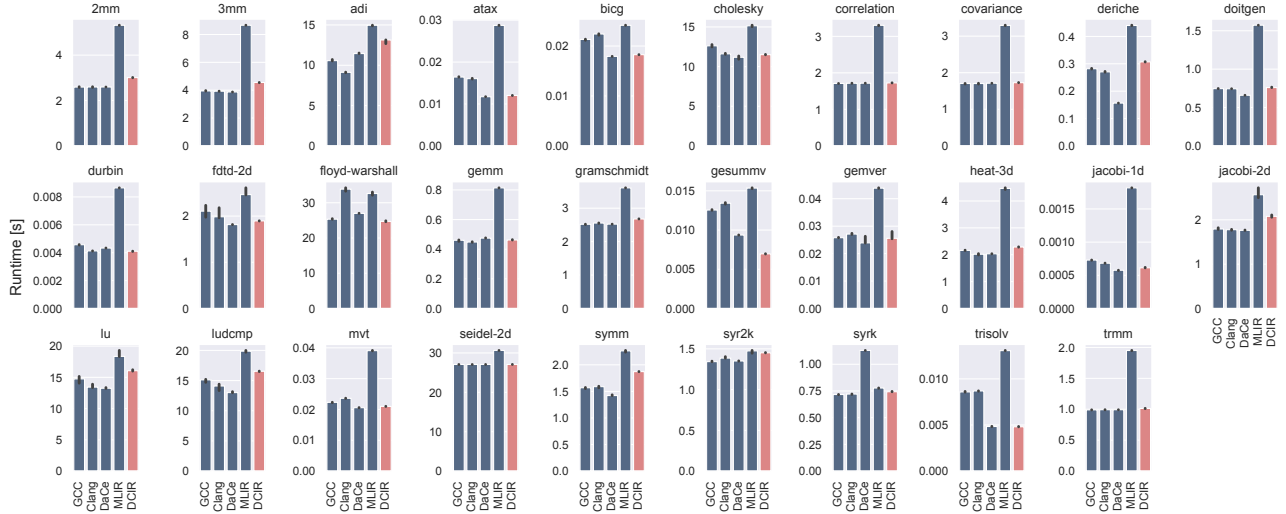
Figure 6. Polybench/C Benchmark comparing GCC, Clang, the DaCe C frontend, and MLIR (via Polygeist) with DCIR. Geometric mean of DCIR speedups: 1.59× over Polygeist with MLIR, 1.03× over GCC, 1.02× over Clang, 0.94× over DaCe.

is expected that the performance without schedule changes would be at the same level as the best compiler.

We use the benchmarks in their unaltered form, apart from increasing the printing precision to more accurately check the correctness of the outputs. We disable the auto-optimization pass for `doitgen`, `durbin`, and `gemver` because the pass produced wrong results due to its experimental status. For `floyd-warshall` we had to remove both the simplification and auto-optimization passes as they both raised exceptions. Instead, we apply a subset of those passes: optional array inference, symbol propagation, state fusion, and memory (pre-)allocation. Furthermore, we exclude `nussinov` from the benchmarks, as Polygeist was unable to generate MLIR core dialect operations for the entire application.

All benchmarks were run ten times on the large dataset defined by Polybench using double-precision floating-point numbers. The plots in Fig. 6 represent the median of the measured runtimes with 95% confidence intervals. In our benchmarks, we measured the runtimes of the whole applications, contrary to Polybench solely measuring the execution time of the kernels. This allows us to additionally consider program-wide data-centric optimization passes.

Compilation time for each of the benchmarks ranges between 19–64 seconds (median: 24 s), where the median DCIR optimization time is 3.46 s, and most of the time (13.94 s) is spent in CMake, which DaCe calls on the generated code.

DCIR strictly outperforms the MLIR pipeline by a factor of 1.59× on average, on par with the best general-purpose compilers. We highlight four key observations:

1. DCIR is never slower than MLIR, showing that it retains the control-centric optimizations of MLIR.
2. DCIR strictly outperforms the MLIR pipeline, proving that there is performance to be gained by applying data-centric optimizations.

```
for (i = 0; i < _PB_N; i++){
 for (k = 0; k < _PB_M; k++) {
  for (j = 0; j <= i; j++)
   C[i][j] += alpha * A[i][k]
            * A[j][k];
 }
}
```

```
for (i = 0; i < _PB_N; i++){
 for (k = 0; k < _PB_M; k++) {
  tmp = alpha * A[i][k]
  for (j = 0; j <= i; j++)
   C[i][j] += tmp * A[j][k];
 }
}
```

(a) Source   **(b)** DCIR-generated C++ code

Figure 7. `syrk` kernel on DaCe and DCIR.

3. DCIR is on par with GCC and Clang w.r.t. the geometric mean speedup. We thus successfully recover the performance that the MLIR pipeline leaves untapped.
4. The Polybench suite does not contain extraneous arrays or unnecessary regions, which would enable additional data-centric optimizations, so there is no significant speedup compared with GCC and Clang.

The cases with increased performance can be attributed to moving arrays to the stack and to improved scheduling. For example, on `gesummv`, DCIR moves one out of the five arrays to the stack, which improves the execution time for loading and storing operations. These benchmarks demonstrate the possible speedup provided by the novel data-centric optimization passes.

Furthermore, we notice that the DaCe C frontend underperforms on the symmetric rank-k update (`syrk`) benchmark compared to all the other compilers and pipelines. In Fig. 7a we can see a snippet of the `syrk` benchmark kernel. Because `alpha * A[i][k]` is completely independent of `j`, we can move it out of the innermost loop, which can be seen in Fig. 7b. The DaCe C frontend misses this optimization, because the generated tasklets are indivisible C++ codes — and these tasklets are treated as black box units of computation, preventing internal optimizations.

```
class Mish(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        x = torch.log(1 + torch.exp(x))
        return x
```

(a) Input Python code
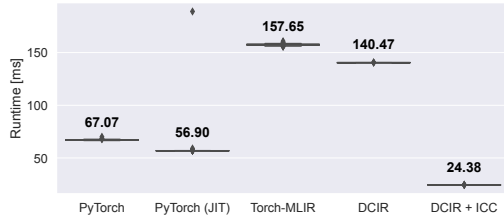
(b) Performance across compilers

Figure 8. The Mish operator in PyTorch and its performance.

The direct C-to-SDFG parser (DaCe) outperforms DCIR with a geometric mean speedup of 1.04×. This reduced performance can be seen most prominently on `deriche`. We can observe that DaCe outperforms DCIR on `deriche` by a factor of 1.7×, which is likely due to Polygeist and MLIR performing loop order inversion — transforming loops iterating using decrements into loops using increments[4]. Indeed, performance counters indicate that DCIR exhibits 2.41× L2 and 2.72× L3 cache misses over the direct C parser. This benchmark exemplifies the semantic information lost in the Polygeist-MLIR pipeline.

In general, the combined DCIR pipeline outperforms the MLIR standard `-O3` mode, resulting in it being on par with Clang and GCC, *all while retaining the flexibility to work with arbitrary high-level IRs.*

7.3 Case Studies

In addition to the Polybench benchmarks, we use code snippets from real-world applications and compare their performance across the same compilers. We only alter the code snippets in order to run them in an isolated environment.

Fig. 8a shows the Mish [23] activation function, which is used in object detection deep neural networks [5], written in PyTorch. The LLVM Torch-MLIR[5] project takes deep neural networks written in PyTorch and compiles them through MLIR's core dialects using the linalg-on-tensors and MHLO (also used in TensorFlow) dialects.

We benchmark PyTorch 1.14, its built-in `torch.jit` compiler, Torch-MLIR's optimizing pipeline, and DCIR, plotting the results in Fig. 8b. PyTorch, being an eager-execution framework, uses separate tensors for each intermediate value. The `torch.jit`-compiled version reports that it fused operators, mitigating the call overhead and reducing intermediate tensor allocation. We find that Torch-MLIR's generated IR
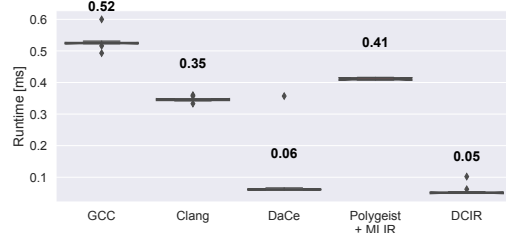
---

[4]The `scf` dialect is *inherently limited* as it defines the step of a for-loop as a strictly positive integer.
[5]https://github.com/llvm/torch-mlir

```
[...]
  for (j = 1; j < Norder; j++) {
    if (converged[j] == 0) {
      zeta_ip1[j] = zeta_i[j] * zeta_im1[j]
                  * beta_im1[0];
      c1 = beta_i[0] * alpha[0] * (zeta_im1[j]
          - zeta_i[j]);
      c2 = zeta_im1[j] * beta_im1[0] * (1.0 - (
          shift[j] - shift[0]) * beta_i[0]);
      zeta_ip1[j] /= c1 + c2;

      beta_i[j] = beta_i[0] * zeta_ip1[j] / zeta_i[j];
    }
  }
}
[...]
```

(a) Input code

(b) Performance across compilers (speedups: 8.4× over Polygeist with MLIR, 10.4× over GCC, 7× over Clang, 1.2× over DaCe)

Figure 9. MILC codebase case study performance.

also contains allocation operations, which add extraneous data movement and inhibit rescheduling. Running the DCIR pipeline removes *all* allocation calls and is able to successfully fuse the computations, improving the performance by 12% over the standard pipeline. We additionally noticed that Clang does not vectorize math library calls (namely, `exp` and `log`). Since PyTorch internally uses the SLEEF vector math library [30], we also compile the DCIR-generated code with the Intel C Compiler (ICC 2021.3). The produced binary contains vector math operations of the same length, and, combined with data movement reduction, outperforms the state of the art (JIT-compiled PyTorch) by 2.33×.

For our second case study, we use one of the solvers in the MILC lattice quantum chromodynamics scientific application[6]. The snippet in Fig. 9a is an implementation of a multi-mass conjugate gradient algorithm, which is an iterative method for solving sparse systems of linear equations.

Fig. 9b shows that DCIR achieves a speedup of 7× compared with general-purpose compilers. The snippet contains multiple array allocations and primarily consists of computing and moving their entries. DCIR and DaCe apply data-centric optimizations, eliminating two arrays, each containing 10,000 doubles, which explains the performance increase.

For our third benchmark, we use a memory bandwidth benchmarking repository[7], which we can see in Fig. 10a. This benchmark consists of allocating four arrays and performing computations, such as initializing the arrays, multiplying all entries with a scalar or summing all elements.

---

[6]https://github.com/milc-qcd/milc_qcd/blob/master/arb_overlap/congrad_multi_field.c
[7]https://github.com/RRZE-HPC/TheBandwidthBenchmark

```
[...]
 for (int k = 0; k < NTIMES; k++) {
  for (int i = 0; i < N; i++)
   a[i] = scalar;
  tmp = a[10];
  double sum = 0.0;
  for (int i = 0; i < N; i++)
   sum += a[i];
  a[10] = sum;
  a[10] = tmp;
  for (int i = 0; i < N; i++)
   a[i] = a[i] * scalar;
 }
[...]
```

(a) Input code
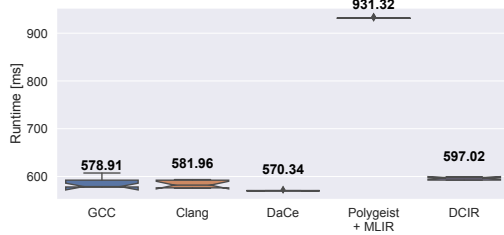
(b) Performance across compilers (speedups: 1.56× over Polygeist with MLIR, 0.97× over GCC and Clang, 0.96× over DaCe)

Figure 10. Memory bandwidth case study performance.

We can see in Fig. 10b that DCIR achieves a speedup of 1.56× compared with MLIR, and is on-par with GCC and Clang. As the results indicate, DCIR again recovers the performance lost by the MLIR-generated code.

In total, 63 arrays and scalars were eliminated from the three snippets, contributing most to the observed speedup.

In conclusion, both C applications in the wild and neural networks benefit from data-centric optimizations. DCIR not only successfully applies those, but also recovers semantics lost during conversion of the source language to MLIR, regaining or surpassing the performance of general compilers.

8 Related Work

In this section, we discuss other approaches that share some of the same goals or methods as our own.

Compilers with multiple abstraction layers. In this work, we focus on the Polygeist and Torch-MLIR frontends, but other frontends are being developed, which will likely become further candidates for our approach. Notable examples of emerging MLIR frontends are Flang[8] and Pylir[9]. At the moment of writing, these frontends use MLIR internally but lack the capability of emitting MLIR core dialect code directly. Once this changes, they could be used as a starting point to our workflow, expanding the scope of programs that can take advantage of DCIR.

SYCL [10] is a C++ cross-platform abstraction layer offering performance portability by leveraging template programming and lambda functions to create specialized code for different hardware architectures from the same high-level code. While not as extensible as MLIR, SYCL offers a way for applications to reach high performance on heterogeneous hardware without needing extensive rewrites. To bridge the two, SYCLops [31] converts SYCL LLVM IR into the `affine`, `arith`, and `memref` MLIR dialects.

Dataflow Representations. Several other data-centric representations exist, such as PDG [9], HPVM [16], Bamboo [36], Dryad [12], and Naiad [25]. SDFGs are unique because they encapsulate fine-grained data dependencies and differentiate between reads, writes and updates on memory. This distinction enables certain transformations which rely on such accesses. A further differentiating feature is the symbolic representation and tracking of memory access patterns throughout the application, a critical aspect that is key to understanding and optimizing dataflow.

Possibly the closest alternative to DCIR is the DaCe C frontend [6], a tool that generates SDFGs from C code. While it implements a number of AST transformations to pre-process the input program and make the translation to SDFG possible, it does not perform control-centric optimizations, nor enjoys the community support that the MLIR and LLVM environments provide. Therefore, it is unlikely that the DaCe C frontend could keep up with the versatile and domain-specific optimizations MLIR and its dialects provide.

9 Conclusion

The paper shows how data movement minimization can aid in general-purpose program optimization. Through the use of an MLIR dialect as a bridge between the SDFG IR and MLIR core dialects, data-centric representations become commutable with control-centric IRs on any source language that MLIR supports. The resulting performance on benchmarks and snippets from real codebases demonstrate that there is a necessity in optimizations such as dead memory elimination, and that the performance benefits can be substantial, at times exhibiting orders of magnitude of improvement.

Acknowledgments

---

[8]https://github.com/flang-compiler/flang
[9]https://github.com/zero9178/Pylir

A   Artifact Appendix

A.1   Abstract

The artifacts of this paper reproduce the results from Fig. 2 and the benchmarks in Section 7. The results include running the programs through the proposed compilation pipeline and its competitors, as well as outputting raw results and regenerating the figures in the paper. The benchmarks should run on any regular hardware, and a Docker container is provided for increased reproducibility. *All used software is publicly available and included in the container.*

A.2   Artifact Check-list (Meta-information)

- **Program:** PolyBench/C (Ver. 4.2.1 beta)
- **Compilation:** GCC (Ver. 11.3.0), Clang (Ver. 13.0.1), ICC (Ver. 2021.7.1), Polygeist (git revision e703db1)[10], Torch-MLIR (git revision eec9a7e)[11], DaCe (git revision c20e8ce)[12]
- **Transformations:** MLIR Optimizer and Translator (git revision 00a1258)[13], MLIR-HLO (git revision 2c4a384)[14], DCIR Optimizer and translator (git revision 91f067d)[15]
- **Run-time environment:** Tested on Ubuntu 22.04 and 22.10. Depending on Docker installation, root access may be required.
- **Metrics:** Runtime of generated code
- **Output:** CSV of runtimes for plots, raw paper results also included
- **Experiments:** Multiple scripts to run the experiments are provided
- **How much disk space required (approximately)?:** 3.8 GiB to pull Docker container (12.4 GiB uncompressed), ≈84 GiB to build container
- **How much time is needed to complete experiments (approximately)?:** Up to 5 hours for 10 repetitions and all compilers
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD 3-Clause License
- **Workflow framework used?:** No
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7519936

A.3   Description

A.3.1   How to Access. All the required files as well as instructions on how to execute them can be accessed on GitHub (https://github.com/Berke-Ates/dcir-artifact) and Zenodo [2]. The files themselves require 8.8 GiB of disk space, whereas building the Docker image from scratch additionally requires approximately 84 GiB of disk space (due to LLVM build files).

A.3.2   Software Dependencies. Running and building the Docker container requires an installation of Docker and a running instance of the Docker daemon.

A.4   Installation

There are three options for installation:

- Pull the docker image
- Manually build the docker image
- Manual installation

All three options are described in the README.md of the repository. The first option is recommended and can be invoked by: docker pull berkeates/dcir-cgo23:latest

A.5   Experiment Workflow

For our DCIR benchmarks, we first generate MLIR code from the C source code using Polygeist. After applying various optimizations within MLIR, we convert the MLIR code to an SDFG. Using DaCe we apply further optimizations, generate C++ code, which we compile using Clang (or ICC for torch-mlir). We then measure the execution time of the generated executable and compare it with GCC, Clang, Polygeist + MLIR and DaCe.

All the compilers, translators, and code generators are built as part of the Docker container, which can be run via docker run -it berkeates/dcir-cgo23.

A.6   Evaluation and Expected Results

The run_all.sh script inside the scripts folder will execute all benchmarks and generate the raw data as well as the plots seen in this paper. Calling the script with an output directory (for example, ./scripts/run_all.sh ae 10 outputs to the ae folder with 10 repetitions) would result in CSV files and PDF files called fig#.pdf for the appropriate figure number, or benchmark name for additional benchmarks. On similar hardware (see § 7.1), the plots are expected to approximately match the ones in the paper, which can be found in the artifact under the output directory.

A.7   Experiment Customization

Other benchmarks can be run by adding them in the same scheme as the ones already provided. The repository contains instructions on how to run a single benchmark.

References

[1] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.71

[2] Tal Ben-Nun, Berke Ates, Alexandru Calotoiu, and Torsten Hoefler. 2023. Data availability artifact. https://doi.org/10.5281/zenodo.7519936

[3] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference*

---

[10] https://github.com/llvm/Polygeist/tree/e703db1

[11] https://github.com/llvm/torch-mlir/tree/eec9a7e

[12] https://github.com/Berke-Ates/dace/tree/c20e8ce

[13] https://github.com/llvm/llvm-project/tree/00a1258

[14] https://github.com/tensorflow/mlir-hlo/tree/2c4a384

[15] https://github.com/spcl/mlir-dace/tree/91f067d

for High Performance Computing, Networking, Storage and Analysis (SC '19). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3295500.3356173

[4] Tal Ben-Nun, Linus Groner, Florian Deconinck, Tobias Wicky, Eddie Davis, Johann Dahm, Oliver Elbert, Rhea George, Jeremy McGibbon, Lukas Trümper, Elynn Wu, Oliver Fuhrer, Thomas Schulthess, and Torsten Hoefler. 2022. Productive Performance Engineering for Weather and Climate Modeling with Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*. IEEE Press. https://doi.org/10.48550/arXiv.2205.04148

[5] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. https://doi.org/10.48550/arXiv.2004.10934

[6] Alexandru Calotoiu, Tal Ben-Nun, Grzegorz Kwasniewski, Johannes de Fine Licht, Timo Schneider, Philipp Schaad, and Torsten Hoefler. 2022. Lifting C Semantics for Dataflow Optimization. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 17, 13 pages. https://doi.org/10.1145/3524059.3532389

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://doi.org/10.48550/arXiv.1802.04799

[8] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. 2021. StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 315–326. https://doi.org/10.1109/CGO51591.2021.9370315

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[10] Khronos Group. 2022. SYCL. https://www.khronos.org/sycl/

[11] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2021. Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-Accelerated Climate Simulation. *ACM Trans. Archit. Code Optim.* 18, 4, Article 51 (sep 2021), 23 pages. https://doi.org/10.1145/3469030

[12] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *SIGOPS Oper. Syst. Rev.* 41, 3 (mar 2007), 59–72. https://doi.org/10.1145/1272998.1273005

[13] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data Movement Is All You Need: A Case Study on Optimizing Transformers. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 711–732. https://doi.org/10.48550/arXiv.2007.00072

[14] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. 2021. High Performance GPU Code Generation for Matrix-Matrix Multiplication using MLIR: Some Early Results. https://doi.org/10.48550/arXiv.2108.13191 arXiv:2108.13191 [cs.DC]

[15] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (oct 2017), 29 pages. https://doi.org/10.1145/3133901

[16] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous Parallel Virtual Machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 68–80. https://doi.org/10.1145/3178487.3178493

[17] Grzegorz Kwasniewski, Tal Ben-Nun, Lukas Gianinazzi, Alexandru Calotoiu, Timo Schneider, Alexandros Nikolaos Ziogas, Maciej Besta, and Torsten Hoefler. 2021. Pebbles, Graphs, and a Pinch of Combinatorics: Towards Tight I/O Lower Bounds for Statically Analyzable Programs. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 328–339. https://doi.org/10.1145/3409964.3461796

[18] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. San Jose, CA, USA, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[19] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[20] Tim Lindholm and Frank Yellin. 1999. *Java Virtual Machine Specification* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[21] LLVM Project. 2022. Circuit IR Compilers and Tools (CIRCT). https://github.com/llvm/circt.

[22] LLVM Project. 2022. Flang: FORTRAN front end. https://github.com/llvm/llvm-project/tree/main/flang.

[23] Diganta Misra. 2020. Mish: A Self Regularized Non-Monotonic Activation Function. In *Proceedings of the 31st British Machine Vision Conference (BMVC)*. https://doi.org/10.48550/arXiv.1908.08681

[24] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event) *(PACT '21)*. Association for Computing Machinery, New York, NY, USA, 12 pages. https://doi.org/10.1109/PACT52795.2021.00011

[25] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[26] NVIDIA Corporation. 2022. cuBLAS: Basic Linear Algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas

[27] L. N. Pouchet. 2016. PolyBench: The Polyhedral Benchmark suite. https://sourceforge.net/projects/polybench

[28] Oliver Rausch, Tal Ben-Nun, Nikoli Dryden, Andrei Ivanov, Shigang Li, and Torsten Hoefler. 2022. A Data-Centric Optimization Framework for Machine Learning. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 36, 13 pages. https://doi.org/10.1145/3524059.3532364

[29] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A Multi-Level Intermediate Representation for Hardware Description Languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 258–271. https://doi.org/10.1145/3385412.3386024

[30] Naoki Shibata and Francesco Petrogalli. 2020. SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (2020), 1316–1327.

https://doi.org/10.1109/TPDS.2019.2960333

[31] Alexandre Singer, Frank Gao, and Kai-Ting Amy Wang. 2022. SYCLops: A SYCL Specific LLVM to MLIR Converter. In *International Workshop on OpenCL* (Bristol, United Kingdom, United Kingdom) *(IWOCL'22)*. Association for Computing Machinery, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/3529538.3529992

[32] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

[33] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H J Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, and Miquel Pericás. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 3007–3020. https://doi.org/10.1109/TPDS.2017.2703149

[34] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. https://doi.org/10.48550/ARXIV.1802.04730

[35] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/2851141.2851145

[36] Jin Zhou and Brian Demsky. 2010. Bamboo: A Data-Centric, Object-Oriented Approach to Many-Core Software. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 388–399. https://doi.org/10.1145/1806596.1806640

[37] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Guillermo Indalecio Fernández, Timo Schneider, Mathieu Luisier, and Torsten Hoefler. 2019. A Data-Centric Approach to Extreme-Scale Ab Initio Dissipative Quantum Transport Simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/3295500.3357156

[38] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 95, 13 pages. https://doi.org/10.1145/3458817.3476176