Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures

Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, Torsten Hoefler
Department of Computer Science, ETH Zurich, Switzerland
{talbn,definelicht,alziogas,timos,htor}@inf.ethz.ch

ABSTRACT

The ubiquity of accelerators in high-performance computing has driven programming complexity beyond the skill-set of the average domain scientist. To maintain performance portability in the future, it is imperative to decouple architecture-specific programming paradigms from the underlying scientific computations. We present the Stateful DataFlow multiGraph (SDFG), a data-centric intermediate representation that enables separating program definition from its optimization. By combining fine-grained data dependencies with high-level control-flow, SDFGs are both expressive and amenable to program transformations, such as tiling and double-buffering. These transformations are applied to the SDFG in an interactive process, using extensible pattern matching, graph rewriting, and a graphical user interface. We demonstrate SDFGs on CPUs, GPUs, and FPGAs over various motifs — from fundamental computational kernels to graph analytics. We show that SDFGs deliver competitive performance, allowing domain scientists to develop applications naturally and port them to approach peak hardware performance without modifying the original scientific code.

CCS CONCEPTS

• **Software and its engineering → Parallel programming languages**; **Data flow languages**; *Just-in-time compilers*; • **Human-centered computing → Interactive systems and tools**.

ACM Reference Format:
Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3295500.3356173

1 MOTIVATION

HPC programmers have long sacrificed ease of programming and portability for achieving better performance. This mindset was established at a time when computer nodes had a single processor/core and were programmed with C/Fortran and MPI. The last decade, witnessing the end of Dennard scaling and Moore's law, brought a flurry of new technologies into the compute nodes. Those range from simple multi-core and manycore CPUs to heterogeneous GPUs and specialized FPGAs. To support those architectures, the complexity of OpenMP's specification grew by more than an order of magnitude from 63 pages in OpenMP 1.0 to 666 pages in OpenMP 5.0. This one example illustrates how (performance) programming complexity shifted from network scalability to node utilization. Programmers would now not only worry about communication (fortunately, the MPI specification grew by less than 4x from MPI-1.0 to 3.1) but also about the much more complex on-node heterogeneous programming. The sheer number of new approaches, such as OpenACC, OpenCL, or CUDA demonstrate the difficult situation in on-node programming. This increasing complexity makes it nearly impossible for domain scientists to write portable and performant code today.

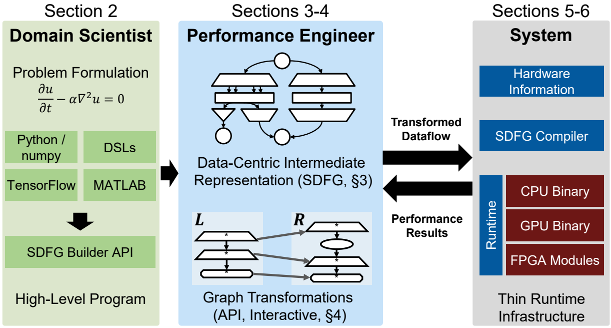


Figure 1: Proposed Development Scheme

The growing complexity in performance programming led to a specialization of roles into domain scientists and performance engineers. Performance engineers typically optimize codes by moving functionality to performance libraries such as BLAS or LAPACK. If this is insufficient, they translate the user-code to optimized versions, often in different languages such as assembly code, CUDA, or tuned OpenCL. Both libraries and manual tuning reduce code maintainability, because the optimized versions are not only hard to understand for the original author (the domain scientist) but also cannot be changed without major effort.

Code annotations as used by OpenMP or OpenACC do not change the original code that then remains understandable to the domain programmer. However, the annotations must re-state (or modify) some of the semantics of the annotated code (e.g., data placement or reduction operators). This means that a (domain scientist) programmer who modifies the code, *must* modify some annotations or she may introduce hard-to-find bugs. With heterogeneous target devices, it now becomes common that the complexity of annotations is higher than the code they describe [56]. Thus, scientific programmers can barely manage the complexity of the code targeted at heterogeneous devices.

The main focus of the community thus moved from scalability to performance portability as a major research target [69]. *We call a code-base* ***performance-portable*** *if the domain scientist's view ("what is computed") does not change while the code is optimized to different target architectures, achieving consistently high performance. The execution should be approximately as performant (e.g., attaining*

SC '19, November 17–22, 2019, Denver, CO, USA
2019. ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356173

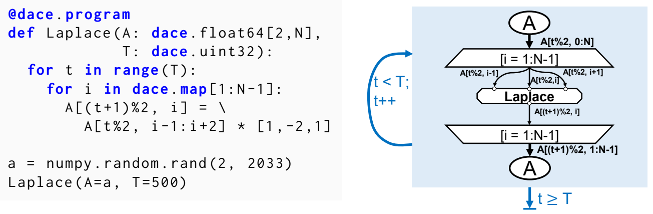
similar ratio of peak performance) as the best-known implementation or theoretical best performance on the target architecture [67]. As discussed before, hardly any existing programming model that supports portability to different accelerators satisfies this definition.

Our Data-centric Parallel Programming (DAPP) concept addresses performance portability. It uses a data-centric viewpoint of an application to separate the roles of domain scientist and performance programmer, as shown in Fig. 1. DAPP relies on Stateful DataFlow multiGraphs (SDFGs) to represent code semantics and transformations, and supports modifying them to tune for particular target architectures. It bases on the observation that data-movement dominates time and energy in today's computing systems [66] and pioneers the necessary fundamental change of view in parallel programming. As such, it builds on ideas of data-centric mappers and schedule annotations such as Legion [9] and Halide [58] *and extends them with a multi-level visualization of data movement, code transformation and compilation for heterogeneous targets, and strict separation of concerns for programming roles.* The domain programmer thus works in a convenient and well-known language such as (restricted) Python or MATLAB. The compiler transforms the code into an SDFG, on which the performance engineer solely works on, specifying transformations that match certain data-flow structures on all levels (from registers to inter-node communication) and modify them. Our transformation language can implement arbitrary changes to the SDFG and supports creating libraries of transformations to optimize workflows. Thus, SDFGs separate the concerns of the domain scientist and the performance engineers through a clearly defined interface, enabling highest productivity of both roles.

We provide a full implementation of this concept in our Data-Centric (DaCe) programming environment, which supports (limited) Python, MATLAB, and TensorFlow as frontends, as well as support for selected DSLs. DaCe is easily extensible to other frontends through an SDFG builder interface. Performance engineers develop potentially domain-specific *transformation libraries* (e.g., for stencil-patterns) and can tune them through DaCe's Interactive Optimization Environment (DIODE). The current implementation focuses on on-node parallelism as the most challenging problem in scientific computing today. However, it is conceivable that the principles can be extended beyond node-boundaries to support large-scale parallelism using MPI as a backend.

The key contributions of our work are as follows:

- We introduce the principle of Data-centric Parallel Programming, in which we use Stateful Dataflow Multigraphs, a data-centric Intermediate Representation that enables separating code definition from its optimization.
- We provide an open-source implementation[1] of the data-centric environment and its performance-optimization IDE.
- We demonstrate performance portability on fundamental kernels, graph algorithms, and a real-world quantum transport simulator — results are competitive with and faster than expert-tuned libraries from Intel and NVIDIA, approaching peak hardware performance, and up to five orders of magnitude faster than naïve FPGA code written with High-Level Synthesis, all from *the same program source code.*



```
@dace.program
def Laplace(A: dace.float64[2,N],
            T: dace.uint32):
  for t in range(T):
    for i in dace.map[1:N-1]:
      A[(t+1)%2, i] = \
        A[t%2, i-1:i+2] * [1,-2,1]

a = numpy.random.rand(2, 2033)
Laplace(A=a, T=500)
```

(a) Python Representation **(b) Resulting SDFG**

Figure 2: Data-Centric Computation of a Laplace Operator

2 DATA-CENTRIC PROGRAMMING

Current approaches in high-performance computing optimizations [66] revolve around improving data locality. Regardless of the underlying architecture, the objective is to keep information as close as possible to the processing elements and promote memory reuse. Even a simple application, such as matrix multiplication, requires multiple stages of transformations, including data layout modifications (packing) and register-aware caching [33]. Because optimizations do not modify computations and differ for each architecture, maintaining performance portability of scientific applications requires separating computational semantics from data movement.

SDFGs enable separating application development into two stages, as shown in Fig. 2. The problem is formulated as a high-level program (Fig. 2a), and is then transformed into a human-readable SDFG as an Intermediate Representation (IR, Fig. 2b). The SDFG can then be modified without changing the original code, and as long as the dataflow aspects do not change, the original code can be updated while keeping SDFG transformations intact. What differentiates the SDFG from other IRs is the ability to hierarchically and parametrically view data movement, where scopes in the graph contain overall data requirements. This enables reusing transformations (e.g., tiling) at different levels of the memory hierarchy, as well as performing cross-level optimizations.

*The modifications to the SDFG are **not** completely automatic.* Rather, they are made by the performance engineer as a result of informed decisions based on the program structure, hardware information, and intermediate performance results. To support this, a transformation interface and common optimization libraries should be at the performance engineer's disposal, enabling modification of the IR in a verifiable manner (i.e., without breaking semantics), either programmatically or interactively. The domain scientist, in turn, writes an entire application once for all architectures, and can freely update the underlying calculations without undoing optimizations on the SDFG.

Conceptually, we perform the separation of computation from data movement logic by viewing programs as data flowing between operations, much like Dataflow and Flow-Based Programming [40]. One key difference between dataflow and data-centric parallel programming, however, is that in a pure dataflow model execution is stateless, which means that constructs such as loops have to be unrolled. At the other extreme, traditional, control-centric programs revolve around statements that are executed in order. Data-centric parallel programming promotes the use of *stateful dataflow*, in which execution order depends first on data dependencies, but also on a global execution state. The former fosters the expression of

[1] https://www.github.com/spcl/dace

concurrency, whereas the latter increases expressiveness and compactness by enabling concepts such as loops and data-dependent execution. The resulting concurrency works in several granularities, from utilizing processing elements on the same chip, to ensuring overlapped copy and execution of programs on accelerators in clusters. A data-centric model combines the following concepts:

(1) **Separating Containers from Computation**: Data-holding constructs with volatile or non-volatile information are defined as separate entities from computations, which consist of stateless functional units that perform arithmetic or logical operations in any granularity.
(2) **Dataflow**: The concept of information moving from one container or computation to another. This may be translated to copying, communication, or other forms of movement.
(3) **States**: Constructs that provide a mechanism to introduce execution order independent of data movement.
(4) **Coarsening**: The ability to view parallel patterns in a hierarchical manner, e.g., by grouping repeating computations.

The resulting programming interface should thus enable these concepts without drastically modifying development process, both in terms of languages and integration with existing codebases.

2.1 Domain Scientist Interface

Languages Scientific applications typically employ different programming models and Domain-Specific Languages (DSLs) to solve problems. To cater to the versatile needs of the domain scientists, SDFGs should be easily generated from various languages. We thus implement SDFG frontends in high-level languages (Python, MATLAB, TensorFlow), and provide a low-level (builder) API to easily map other DSLs to SDFGs. In the rest of this section, we focus on the Python [30] interface, which is the most extensible.

Interface The Python interface creates SDFGs from restricted Python code, supporting numpy operators and functions, as well as the option to explicitly specify dataflow. In Fig. 2a, we demonstrate the data-centric interface on a one-dimensional Laplace operator. DaCe programs exist as decorated, strongly-typed functions in the application ecosystem, so that they can interact with existing codes using array-based interfaces (bottom of figure). The Python interface contains primitives such as `map` and `reduce` (which translate directly into SDFG components), allows programmers to use multi-dimensional arrays, and implements an extensible subset of operators from numpy [25] on such arrays to ease the use of linear algebra operators. For instance, the code `A @ B` generates the dataflow of a matrix multiplication.

Extensibility For operators and functions that are not implemented, a user can easily provide dataflow implementations using decorated functions (`@dace.replaces('numpy.conj')`) that describe the SDFG. Otherwise, unimplemented functions fall-back into Python, casting the array pointers (which may be defined internally in the DaCe program) into numpy arrays and emitting a “potential slowdown” warning. If the syntax is unsupported (e.g., dynamic dictionaries), an error is raised.

Explicit Dataflow If the programmer does not use predefined operators (e.g., for custom element-wise computation), dataflow “intrinsics” can be explicitly defined separately from code, in constructs which we call *Tasklets*. Specifically, tasklet functions cannot access data unless it was explicitly moved in or out using pre-declared operators (<<, >>) on arrays, as shown in the code.

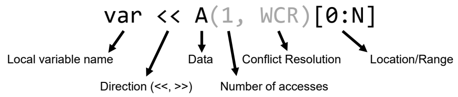


Figure 3: Anatomy of a Python Memlet

Data movement operations (*memlets*) can be versatile, and the Python syntax of explicit memlets is defined using the syntax shown in Fig. 3. First, a local variable (i.e., that can be used in computation) is defined, whether it is an input or an output. After the direction of the movement, the data container is specified, along with an optional range (or index). In some applications (e.g., with indirect or data-dependent access), it is a common occurrence that the *subset* of the accessed data is known, but not exact indices; specifying memory access constraints both enables this behavior and facilitates access tracking for decomposition, e.g., which data to send to an accelerator. Finally, the two optional values in parentheses govern the nature of the access — the number of data elements moved, used for performance modeling, and a lambda function that is called when write-conflicts may occur.

```
@dace.program
def spmv(A_row: dace.uint32[H + 1], A_col: dace.uint32[nnz],
         A_val: dace.float32[nnz], x: dace.float32[W],
         b: dace.float32[H]):
  for i in dace.map[0:H]:
    for j in dace.map[A_row[i]:A_row[i+1]]:
      with dace.tasklet:
        a << A_val[j]
        in_x << x[A_col[j]]
        out >> b(1, dace.sum)[i]
        out = a * in_x
```

Figure 4: Sparse Matrix-Vector Mult. with Memlets

Using explicit dataflow is beneficial when defining nontrivial data accesses. Fig. 4 depicts a full implementation of Sparse Matrix-Vector multiplication (SpMV). In the implementation, the access `x[A_col[j]]` is translated into an indirect access subgraph (see Appendix F) that can be identified and used in transformations.

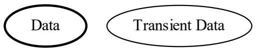
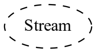
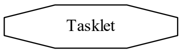
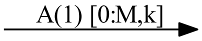
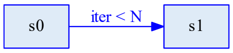
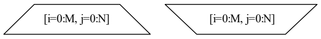
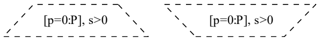
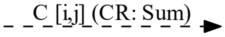
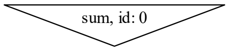
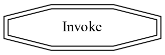
External Code Supporting scientific code, in terms of performance and productivity, requires the ability to call previously-defined functions or invoke custom code (e.g., intrinsics or assembly). In addition to falling back to Python, the frontend enables defining tasklet code in the generated code language directly. In Fig. 5 we see a DaCe program that calls a BLAS function directly. The semantics of such tasklets require that memlets are defined separately (for correctness); the code can in turn interact with the memory directly (memlets that are larger than one element are pointers). With this feature, users can use existing codes and benefit from concurrent scheduling that the SDFG provides.

Parametric Dimensions To support parametric sizes (e.g., of arrays and maps) in DaCe, we utilize symbolic math evaluation. In

```
@dace.program
def extmm(A: dace.complex128[M,K], B: dace.complex128[K,N],
          C: dace.complex128[M,N]):
  with dace.tasklet(language=dace.Language.CPP,
                    code_global='#include <mkl.h>'):
    a << A; b << B; in_c << C; out_c >> C
    '''
    dace::complex128 alpha(1, 0), beta(0, 0);
    cblas_zgemm(CblasRowMajor, 'N', 'N', M, N, K, &alpha, a, M,
                b, K, &beta, out_c, M);
    '''
```

Figure 5: External Code in DaCe

Table 1: SDFG Syntax

Primitive	Description
Data-Centric Model	
	**Data**: N-dimensional array container.
	**Stream**: Streaming data container.
	**Tasklet**: Fine-grained computational block.
	**Memlet**: Data movement descriptor.
	**State**: State machine element.
Parametric Concurrency	
	**Map**: Parametric graph abstraction for parallelism.
	**Consume**: Dynamic mapping of computations on streams.
	**Write-Conflict Resolution**: Defines behavior during conflicting writes.
Parallel Primitives and Nesting	
	**Reduce**: Reduction over one or more axes.
	**Invoke**: Call a nested SDFG.

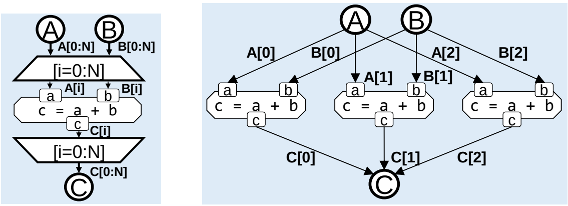
particular, we extend the SymPy [64] library to support our expressions and strong typing. The code can thus define symbolic sizes and use complex memlet subset expressions, which will be analyzed during SDFG compilation. The *separation of access and computation, flexible interface, and symbolic sizes* are the core enablers of data-centric parallel programming, helping domain scientists create programs that are amenable to efficient hardware mapping.

3 STATEFUL DATAFLOW MULTIGRAPHS

We define an SDFG as a *directed graph of directed acyclic multigraphs*, whose components are summarized in Table 1. Briefly, the SDFG is composed of acyclic dataflow multigraphs, in which nodes represent *containers* or *computation*, and edges (memlets) represent *data movement*. To support cyclic data dependencies and control-flow, these multigraphs reside in ***State*** nodes at the top-level graph. Following complete execution of the dataflow in a state, state transition edges on the top-level graph specify conditions and assignments, forming a state machine. For complete operational semantics of SDFGs, we refer to Appendix A.

3.1 Containers

As a data-centric model, SDFGs offer two forms of data containers: **Data** and **Stream** nodes. **Data** nodes represent a location in memory that is mapped to a multi-dimensional array, whereas **Stream** nodes are defined as multi-dimensional arrays of concurrent queues, which can be accessed using push/pop semantics. Containers are tied to a specific *storage location* (as a node property), which may be on a GPU or even a file. In the generated code, memlets between containers either generate appropriate memory copy operations or fail with illegal accesses (for instance, when trying to access paged CPU memory within a GPU kernel). In FPGAs, Stream nodes instantiate FIFO interfaces that can be used to connect hardware modules. Another property of containers is whether they are *transient*, i.e., only allocated for the duration of SDFG execution. This allows transformations and performance engineers to distinguish between buffers that interact with external systems, and ones that can be manipulated (e.g., data layout) or eliminated entirely, even across devices. This feature is advantageous, as standard compilers cannot make this distinction, especially in the presence of accelerators.



(a) Parametric **(b) Expanded ($N = 3$)**

Figure 6: Parametric Parallelism in SDFGs

3.2 Computation

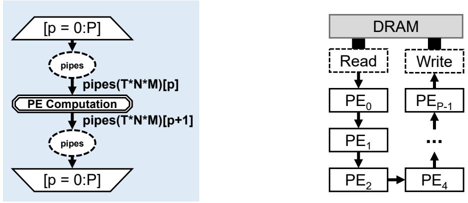
Tasklet nodes contain stateless, arbitrary computational functions of any granularity. The SDFG is designed, however, for fine-grained tasklets, so as to enable performance engineers to analyze and optimize the most out of the code, leaving computational semantics intact. Throughout the process of data-centric transformations and compilation, the tasklet code remains immutable. This code, provided that it cannot access external memory without memlets, can be written in any source language that can compile to the target platform, and is implemented in Python by default.

In order to support Python as a high-level language for tasklets, we implement a Python-to-C++ converter. The converter traverses the Python Abstract Syntax Tree (AST), performs type and shape inference, tracks local variables for definitions, and uses features from C++14 (such as lambda expressions and `std::tuples`) to create the corresponding code. Features that are not supported include dictionaries, dynamically-sized lists, exceptions, and other Python high-level constructs. Given that tasklets are meant to be fine-grained, and that our DaCe interface is strongly typed (§ 2.1), this feature-set is sufficient for HPC kernels and real-world applications.

3.3 Concurrency

Expressing parallelism is inherent in SDFGs by design, supported by the **Map** and **Consume** scopes. Extending the traditional task-based model, SDFGs expose concurrency by grouping parallel subgraphs (computations, local data, movement) into one symbolic instance, enclosed within two "scope" nodes. Formally, we define an enclosed subgraph as nodes dominated by a scope entry node and post-dominated by an exit node. The subgraphs are thus connected to external data only through scope nodes, which enables analysis of their overall data requirements (useful, e.g., for automatically transforming a map to GPU code).

Map scopes represent parallel computation on all levels, and can be nested hierarchically. This feature consolidates many parallel



(a) SDFG **(b) Hardware Modules**

Figure 7: Parametric Generation of Systolic Arrays

programming concepts, including multi-threading, GPU kernels, multi-GPU synchronization, and multiple processing elements on FPGAs. The semantics of a Map are illustrated in Fig. 6 — a symbolic integer set attribute of the scope entry/exit nodes called *range* (Fig. 6a) defines how the subgraph should be expanded on evaluation (Fig. 6b). Like containers, Maps are tied to *schedule*s that determine how they translate to code. When mapped to multi-core CPUs, Map scopes generate OpenMP `parallel for` loops; for GPUs, device schedules generate CUDA kernels (with the map range as thread-block indices), whereas thread-block schedules determine the dimensions of blocks, emitting synchronization calls (`__syncthreads`) as necessary; for FPGAs, Maps synthesize different hardware modules as processing elements. Streams can also be used in conjunction with Maps to compactly represent systolic arrays, constructs commonly used in circuit design to represent efficient pipelines, as can be seen in Fig. 7. Note that no data is flowing in or out of the Map scope (using empty memlets for the enclosed subgraph) — this would replicate the scope’s contents as separate connected components.

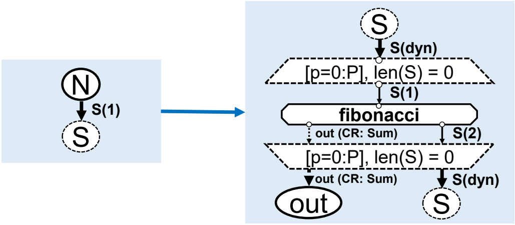
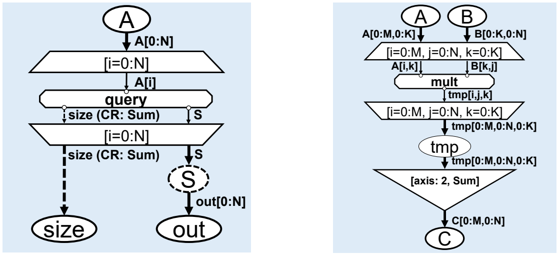


Figure 8: Asynchronous Fibonacci SDFG

Consume scopes enable producer/consumer relationships via dynamic processing of streams. Consume nodes are defined by the *number of processing elements*, an *input stream* to consume from, and a *quiescence condition* that, when evaluated to true, stops processing. An example is shown in Fig. 8, which computes the Fibonacci recurrence relation of an input N without memoization. In the SDFG, the value is first pushed into the stream S and asynchronously processed by P workers, with the memlet annotated as `dyn` for dynamic number of accesses. The tasklet adds the result to `out` and pushes two more values to S for processing. The consume scope then operates until the number of elements in the stream is zero, which terminates the program.

Consume scopes are implemented using batch stream dequeue and atomic operations to asynchronously pop and process elements. The potential to encompass complex parallel patterns like work stealing schedulers using high-performance implementations of this node dramatically reduces code complexity.

In order to handle concurrent memory writes from scopes, we define **Write-Conflict Resolution** memlets. As shown in Fig. 9a, such memlets are visually highlighted for the performance engineer using dashed lines. Implementation-wise, such memlets can be implemented as atomic operations, critical sections, or accumulator modules, depending on the target architecture and the function. **Reduce** nodes complement conflict resolution by implementing target-optimized reduction procedures on data nodes. An example can be seen with a map-reduce implementation of matrix multiplication (Fig. 9b), where a tensor with multiplied pairs of the input matrices is reduced to the resulting matrix. As we shall show in the next section, this inefficient representation can be easily optimized using data-centric transformations.



(a) Query **(b) Matrix Multiplication**

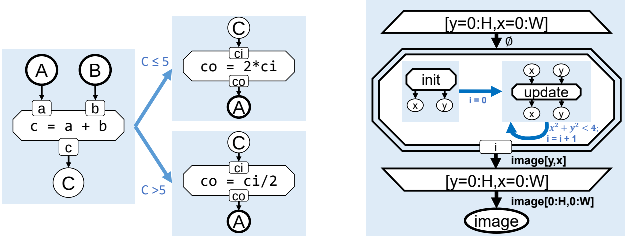
Figure 9: Write-Conflicts and Reductions

Different connected components within an SDFG multigraph also run concurrently (by definition). Thus, they are mapped to `parallel sections` in OpenMP, different CUDA streams on GPUs, or different command queues on FPGAs. These concepts are notoriously cumbersome to program manually for all platforms, where synchronization mistakes, order of library calls, or less-known features (e.g., `nowait`, non-blocking CUDA streams) may drastically impact performance or produce wrong results. Therefore, the SDFG’s automatic management of concurrency, and configurable fine-tuning of synchronization aspects by the performance engineer (e.g., maximum number of concurrent streams, nested parallelism) make the IR attractive for HPC programming on all platforms.

3.4 State

Sequential operation in SDFGs either implicitly occurs following data dependencies, or explicitly specified using multiple states. State transition edges define a condition, which can depend on data in containers, and a list of assignments to inter-state symbols (e.g., loop iteration). The concept of a state machine enables both complex control flow patterns, such as imperfectly nested loops, and data-dependent execution, as shown in Fig. 10a.

To enable control-flow within data-flow (e.g., a loop in a map), or a parametric number of state machines, SDFGs can be nested via the **Invoke** node. The semantics of Invoke are equivalent to a tasklet, thereby disallowing access to external memory without memlets. The Mandelbrot example (Fig. 10b) demonstrates nested SDFGs. In the program, each pixel requires a different number of iterations to converge. In this case, an invoke node calls another SDFG within the map to manage the convergence loop. Recursive calls to the same SDFG are disallowed, as the nested state machine may be inlined or transformed by the performance engineer.



(a) Branching **(b) Mandelbrot (Nested SDFG)**

Figure 10: Data-Dependent Execution

4 PERFORMANCE ENGINEER WORKFLOW

The Stateful Dataflow Multigraph is designed to expose application data movement motifs, regardless of the underlying computations. As such, the optimization process of an SDFG consists of finding and leveraging such motifs, in order to mutate program dataflow. Below, we describe the two principal tools we provide the performance engineer to guide optimization, followed by the process of compiling an SDFG into an executable library.

4.1 Graph Transformations

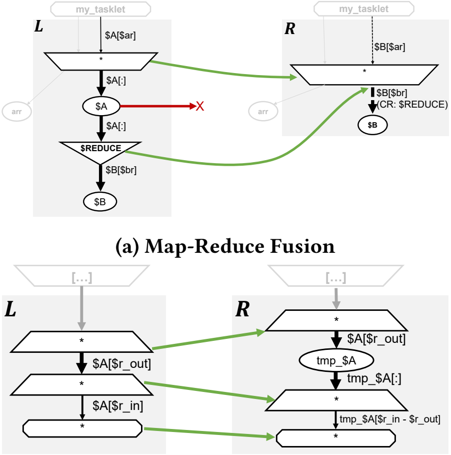
Informally, we define a graph transformation on an SDFG as a “find and replace” operation, either within one state or between several, which can be performed if all of the conditions match. For general optimization strategies (e.g., tiling), we provide a standard library of such transformations, which is meant to be used as a baseline for performance engineers. Transformations can be designed with symbolic expressions, or specialized for given sizes in order to fine-tune applications. A list of 16 transformations implemented in DaCe can be found in Appendix B.

Transformations consist of a *pattern* subgraph and a *replacement* subgraph. A transformation also includes a *matching* function, used to identify instances of the pattern in an SDFG, and programmatically verify that requirements are met. To find matching subgraphs in SDFGs, we use the VF2 algorithm [19] to find isomorphic subgraphs. Transformations can be applied interactively, or using a Python API for matching and applying transformations. An example of a full source code of a transformation is found in Appendix D. Using the transformation infrastructure, we enable seamless knowledge transfer of optimizations across applications.

Two examples of SDFG transformations can be found in Figures 11a and 11b. In Fig. 11a, the transformation detects a pattern *L* where Reduce is invoked immediately following a Map, reusing the computed values. In this case, the transient array $A can be removed (if unused later) and computations can be fused with a conflict resolution, resulting in the replacement *R*. In the second example (Fig. 11b), a local array, which can be assigned to scratchpad memory, is added between two map nodes. As a result, the relative indices are changed in all subsequent memlets.

4.2 DIODE

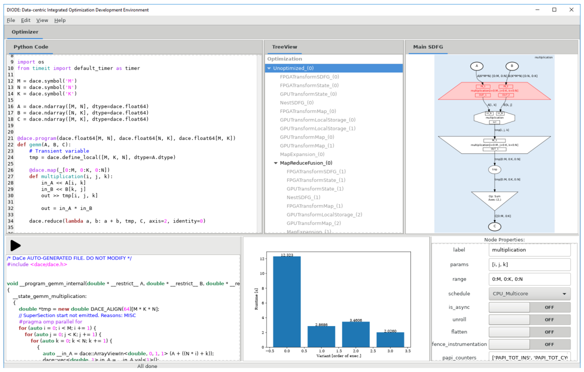
SDFGs are intended to be malleable and interactive, which we realize with an Integrated Development Environment (IDE). The Data-centric Integrated Optimization Development Environment,



(a) Map-Reduce Fusion

(b) Local Storage

Figure 11: SDFG Transformations



```python
import os
from timeit import default_timer as timer

M = dace.symbol('M')
N = dace.symbol('N')
K = dace.symbol('K')

A = dace.ndarray([M, N], dtype=dace.float64)
B = dace.ndarray([N, K], dtype=dace.float64)
C = dace.ndarray([M, K], dtype=dace.float64)


@dace.program(dace.float64[M, N], dace.float64[N, K], dace.float64[M, K])
def gemm(A, B, C):
    # Transient variable
    tmp = dace.define_local([M, K, N], dtype=A.dtype)

    @dace.map(_[0:M, 0:K, 0:N])
    def multiplication(i, j, k):
        in_A << A[i, k]
        in_B << B[k, j]
        out >> tmp[i, j, k]

        out = in_A * in_B

    dace.reduce(lambda a, b: a + b, tmp, C, axis=2, identity=0)
```

```cpp
/* DaCe AUTO-GENERATED FILE. DO NOT MODIFY */
#include <dace/dace.h>


void __program_gemm_internal(double * __restrict__ A, double * __restrict__ B, double * __re
{
  __state_gemm_multiplication:
  {
    double *tmp = new double DACE_ALIGN(64)[M * K * N];
    // SuperSection start not emitted. Reasons: MISC
    #pragma omp parallel for
    for (auto i = 0; i < M; i += 1) {
      for (auto j = 0; j < K; j += 1) {
        for (auto k = 0; k < N; k += 1) {
          {
            auto __in_A = dace::ArrayViewIn<double, 0, 1, 1> (A + ((N * i) + k));
```

label: multiplication
params: [i, j, k]
range: 0:M, 0:K, 0:N
schedule: CPU_Multicore
is_async: OFF
unroll: OFF
flatten: OFF
fence_instrumentation: OFF
papi_counters: ['PAPI_TOT_INS', 'PAPI_TOT_CY

Figure 12: DIODE Graphical User Interface

or **DIODE** (Fig. 12), is a specialized IDE for performance engineers that enables editing SDFGs and applying transformations in a guided manner, in addition to the programmatic interface. In particular, performance engineers can:

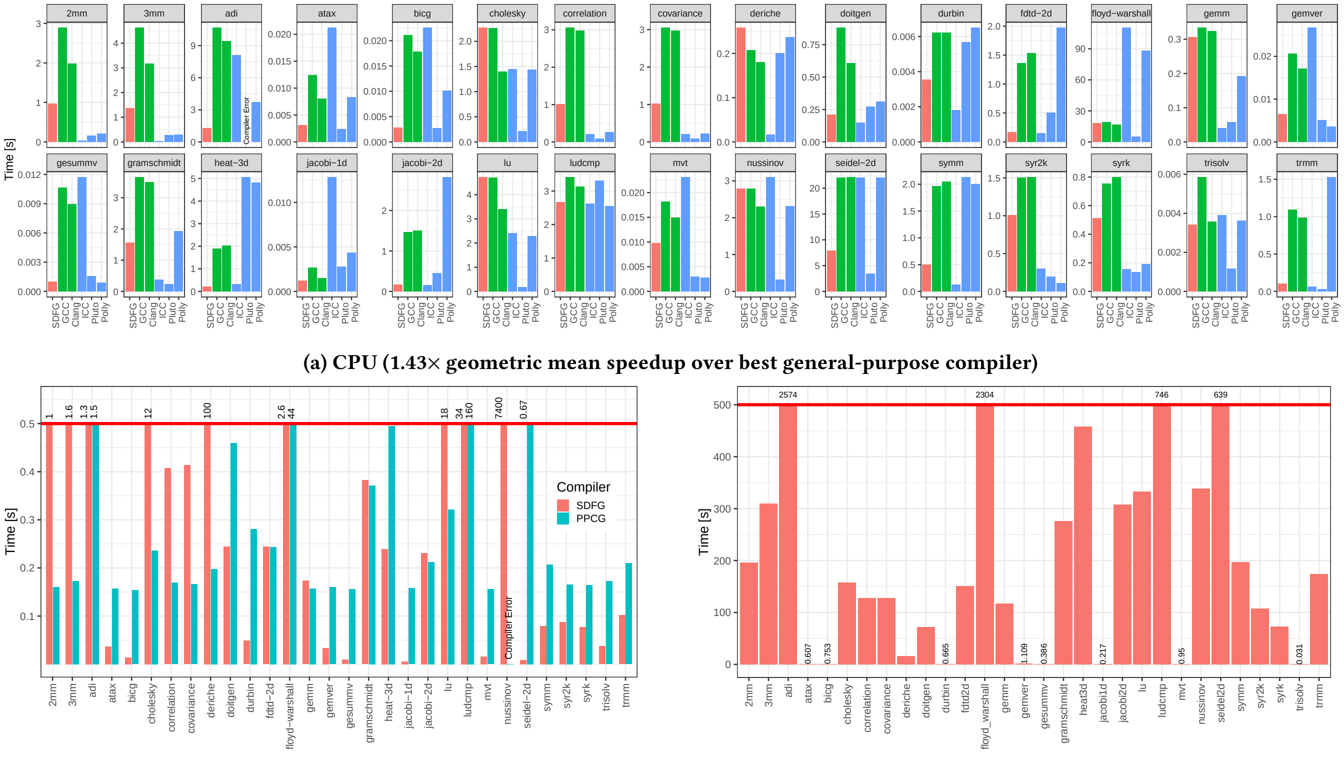
- interactively modify attributes of SDFG nodes and memlets;
- transform and tune transformation parameters;
- inspect an integrated program view that maps between lines in the domain code, SDFG components, and generated code;
- run and compare historical performance of transformations;
- save transformation chains to files, a form of “optimization version control” that can be used when tuning to different architectures (diverging from a mid-point in the chain); and
- hierarchically interact with large-scale programs, collapsing irrelevant parts and focusing on bottleneck subgraphs.

We demonstrate the process of interactively optimizing the matrix multiplication SDFG (Fig. 9b) in a video[2]. Using the IDE and the SDFG representation yields nearly the same performance as Intel’s optimized Math Kernel Library (MKL) [38] in minutes (§6.2).

4.3 Compilation Pipeline

Compiling an SDFG consists of three steps: ❶ data dependency inference, ❷ code generation, and ❸ compiler invocation.

[2] https://www.vimeo.com/301317247



(a) CPU (1.43× geometric mean speedup over best general-purpose compiler)

(b) GPU (1.12× geometric mean speedup over PPCG)

(c) FPGA

Figure 13: Polyhedral Application Runtime (lower is better, best viewed in color)

In step ❶, data dependencies on all levels are resolved. First, a validation pass is run on the graph to ensure that scopes are correctly structured, memlets are connected properly, and map schedules and data storage locations are feasible (failing when, e.g., FPGA code is specified in a GPU map). Then, memlet ranges are propagated from tasklets and containers outwards (through scopes) to obtain the overall data dependencies of each scope, using the image of the scope function (e.g., Map range) on the union of the internal memlet subsets. This information is later used to generate exact memory copy calls to/from accelerators.

The code generation process of an SDFG (step ❷) is hierarchical, starting from top-level states and traversing into scopes. It begins by emitting external interface code and the top-level state machine. Within each state, nodes are traversed in topological order, and a platform-specific dispatcher is assigned to generate the respective code, depending on the node’s storage or schedule type. The process continues recursively via map/consume scopes and nested SDFGs, generating heterogeneous codes using several dispatchers. Between states, transitions are generated by emitting for-loops and branches when detected, or using conditional `goto` statements as a fallback.

In step ❸, we invoke compiler(s) for the generated code according to the used dispatchers. The compiled library can then be used directly by way of inclusion in existing applications, or through Python/DaCe.

5 ASSESSING PERFORMANCE WITHOUT TRANSFORMATIONS

To understand how the inherently-concurrent representation of the SDFG creates reasonably performing naïve code, we reimplement and run the Polybench [57] benchmark over SDFGs, *without any optimizing transformations*, using the experimental setup of Section 6. We show that the representation itself exposes enough parallelism to compete with state-of-the-art polyhedral compilers, outperform them on GPUs, and provide ***the first complete set of placed-and-routed Polybench kernels on an FPGA***.

To demonstrate the wide coverage provided by SDFGs, we apply the `FPGATransform` automatic transformation to offload each Polybench application to the FPGA during runtime, use our simulation flow to verify correctness, and finally perform the full placement and routing process. The same applies for `GPUTransform`. We execute all kernels on the accelerators, including potentially unprofitable ones (e.g., including tasklets without parallelism).

The results are shown in Fig. 13, comparing SDFGs both with general-purpose compilers (green bars in the figure), and with pattern-matching and polyhedral compilers (blue bars). We use the default Polybench flags, the `Large` dataset size, and select the best performance of competing compilers from the flags specified in the Appendix C. On the CPU, we see that for most kernels, the performance of unoptimized SDFGs is closer to that of the polyhedral compilers than to the general-purpose compilers. The cases where SDFGs are on the order of standard compilers are solvers (e.g., `cholesky`, `lu`) and simple linear algebra (e.g., `gemm`). In both cases, data-centric transformations are necessary to optimize the computations, which we exclude from this test in favor of demonstrating SDFG expressibility.

On the GPU, in most cases SDFGs generate code that outperforms PPCG, a tool specifically designed to transform polyhedral applications to GPUs. In particular, the bicg GPU SDFG is 11.8× faster than PPCG. We attribute these speedups to avoiding unnecessary array copies due to explicit data dependencies, as well as the inherent parallel construction of the data-centric representation.

6 PERFORMANCE EVALUATION

We evaluate the performance of SDFGs on a set of fundamental kernels, followed by three case studies: analysis of matrix multiplication, a graph algorithm, and a real-world physics application.

Experimental Setup We run all of our experiments on a server that contains an Intel 12-core Xeon E5-2650 v4 CPU (clocked at 2.20GHz, HyperThreading disabled, 64 GB DDR4 RAM) and a Tesla P100 GPU (16 GB HBM2 RAM) connected over PCI Express. For FPGA results, we use a Xilinx VCU1525 board, hosting an XCVU9P FPGA and four DDR4 banks at 2400 MT/s. We run the experiments 30 times and report the median running time (including memory copy), where the error-bars indicate 95% confidence interval of all runs, and points are outliers (Tukey fences, 1.5 IQR). For Polybench running times, we use the provided measurement tool, which reports the average time of five runs. All reported results were executed in hardware, including the FPGA.

Compilation Generated code from SDFGs is compiled using gcc 8 for CPU results, CUDA 9.2 for GPU, and Xilinx SDAccel 2018.2 for FPGAs. Compilation flags: -std=c++14 -O3 -march=native -ffast-math for CPU, -std=c++14 -arch sm_60 -O3 for GPU, and -std=c++11 -O3 for FPGA. Fundamental kernels use single-precision floating point types, Polybench uses the default experiment data-types (mostly double-precision), and graphs use integers.

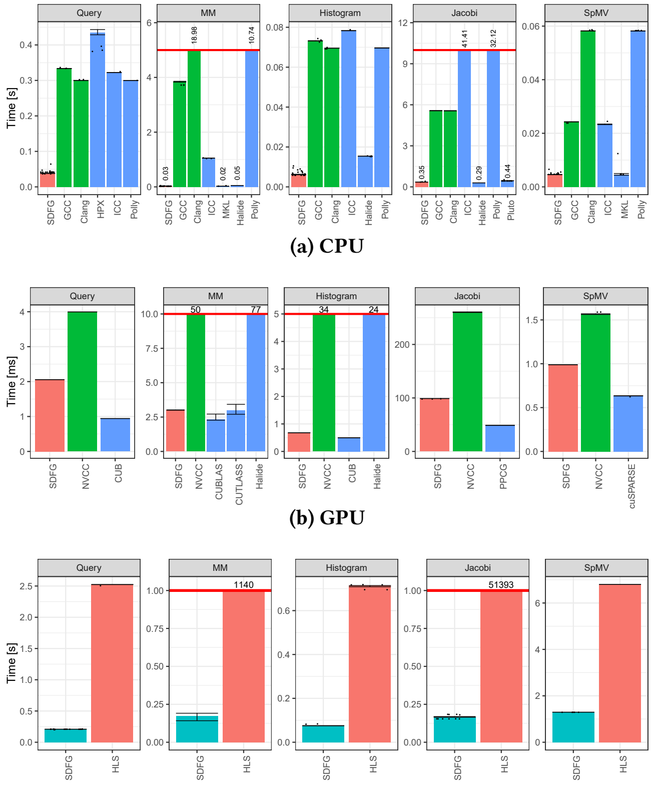
6.1 Fundamental Computational Kernels

We begin by evaluating 5 core scientific computing kernels, implemented over SDFGs:

- **Matrix Multiplication (MM)** of two 2,048×2,048 matrices.
- **Jacobi Stencil**: A 5-point stencil repeatedly computed on a 2,048 square 2D domain for T=1,024 iterations, with constant (zero) boundary conditions.
- **Histogram**: Outputs the number of times each value (evenly binned) occurs in a 8,192 square 2D image.
- **Query**: Filters a column of 67,108,864 elements according to a given condition (filters roughly 50%).
- **Sparse Matrix-Vector Multiplication (SpMV)** of a CSR matrix (8,192 square; 33,554,432 non-zero values).

Our results, shown in Fig. 14, are compared with naïve implementations of the code, compiled with GCC, Clang, NVCC, and ICC; Intel MKL [38] and HPX [41] corresponding library calls for CPU; NVIDIA CUBLAS [54], CUSPARSE [55], and CUB [20], as well as Hybrid Hexagonal Tiling over PPCG [68] for GPU; Halide [58] (auto- and manually-tuned) for CPU and GPU; and Xilinx Vivado HLS/SDAccel [71, 72] and Spatial [43] for FPGA.

On all applications, our SDFG results only employ data-centric transformations, keeping tasklets intact (§4.1). We highlight key results for all platforms below.



(a) CPU

(b) GPU

(c) FPGA

Figure 14: Fundamental Kernel Runtime (lower is better)

In MM, a highly tuned kernel, SDFGs achieve ~98.6% of the performance of MKL, ~70% of CUBLAS, and ~90% of CUTLASS, which is the upper bound of CUDA-based implementations of MM. On FPGA, SDFGs yield a result 4,992× faster than naïve HLS over SDAccel. We also run the FPGA kernel for 1024 × 1024 matrices and compare to the runtime of 878 ms reported for Spatial [43] on the same chip. We measure 45 ms, yielding a speedup of 19.5×.

We observe similar results in SpMV, which is more complicated to optimize due to its irregular memory access characteristics. SDFGs are on par with MKL (99.9% performance) on CPU, and are successfully vectorized on GPUs.

For Histogram, SDFGs enable vectorizing the program, achieving 8× the performance of gcc, where other compilers fail due to the kernel's data-dependent accesses. We implement a two-stage kernel for the FPGA, where the first stage reads 16 element vectors and scatters them to 16 parallel processing elements generated with map unrolling (similar to Fig. 7), each computing a separate histogram. In the second stage, the histograms are merged on the FPGA before copying back the final result. This yields a 10× speedup in hardware.

In Query, SDFGs are able to use streaming data access to parallelize the operation automatically, achieving significantly better results than HPX and STL. On FPGA we read wide vectors, then use a deep pipeline to pack the sparse filtered vectors into dense vectors. This scheme yields a 10× speedup, similar to Histogram.

For Jacobi on CPU, we use a domain-specific transformation (DiamondTiling). We see that it outperforms standard implementations by up to two orders of magnitude, performing 90× faster than

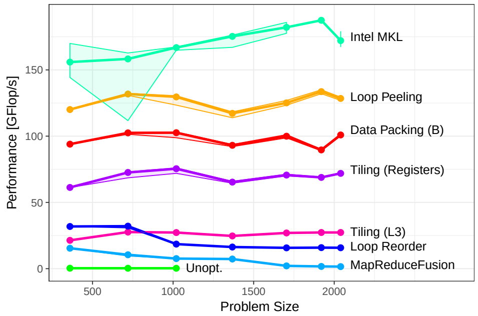


Figure 15: Performance of Transformed GEMM SDFG

Polly and marginally outperforming Pluto, which uses a similar tiling technique. In Halide, when all timesteps are hard-coded in the pipeline (which takes ~68 minutes to compile due to the stateless dataflow representation), its auto-tuner yields the best result, which is 20% faster than SDFGs. For FPGAs, Jacobi is mapped to a systolic array of processing elements, allowing it to scale up with FPGA logic to 139 GOp/s. Overall, the results indicate that data-centric transformations can yield competitive performance across both architectures and memory access patterns.

6.2 Case Study I: Matrix Multiplication

The transformation chain and performance results from Fig. 9b to the MM CPU SDFG in the previous section are shown in Fig. 15.

After fusing the map and reduction into a write-conflict resolution memlet (`MapReduceFusion` transformation), we largely follow the approach of Goto and van de Geijn [33]. Manually reordering the map dimensions in the SDFG reduces data movement and yields a marginal improvement. We then tile for the L3 cache (`MapTiling`) and tile again for mapping to vector registers. To use caches efficiently, we pack the tiles of B and store tiles of C using the `LocalStorage` transformation (twice), followed by `Vectorization` to generate code that uses vector extensions. Lastly, we apply a transformation to convert the internal write-conflict resolution memlet into a state machine, peeling the loop (`ReducePeeling`).

The figure shows that not all transformations yield immediate speedups, yet they are necessary to expose the next steps. Moreover, after only 7 steps the performance increases by ~536x (75% of MKL), and further increases to 98.6% of MKL after tuning transformation parameters for a specific size (Fig. 14).

6.3 Case Study II: Graph Algorithms

We implement an irregular computation problem on multi-core CPUs: Breadth-First Search (BFS). We use the data-driven push-based algorithm [12], and test five graphs with different characteristics as shown in Appendix E.

Due to the irregular nature of the algorithm, BFS is not a trivial problem to optimize. However, SDFGs inherently support constructing the algorithm using streams and data-dependent map ranges. The primary state of the optimized SDFG is shown in Fig. 16, which contains only 14 nodes (excluding input/output data) and is the result of **three** transformations from the base Python program. In particular, the initial generated OpenMP code does not map well to cores due to the dynamic scheduling imposed by the frontier. We mitigate this effect by applying ❶ `MapTiling` with T tiles. Since the accesses to the next frontier are performed concurrently through

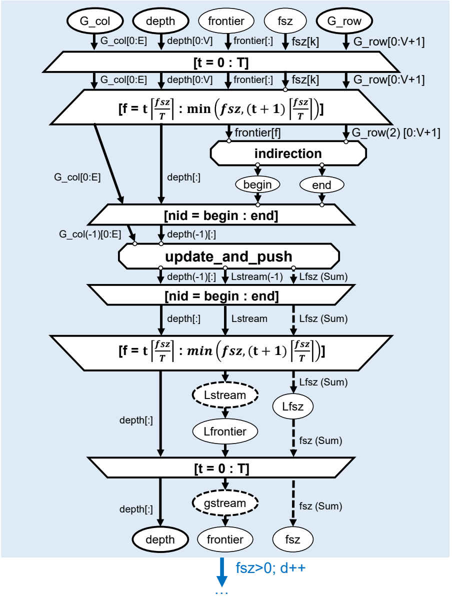


Figure 16: Main State of Optimized BFS SDFG

`gstream`, major overhead was incurred due to atomic operations. Using ❷ `LocalStream`, the results are accumulated to `Lfrontier` and bulk updates are performed to the global frontier. Similarly, we use ❸ `LocalStorage` for frontier size accumulation.

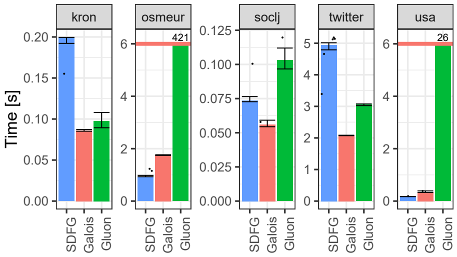


Figure 17: BFS Performance

We compare our results with two state-of-the-art CPU graph processing frameworks: Galois [53] and Gluon [23]. We use the default settings (`bfs_push` for Gluon, `SyncTile` for Galois) and use 12 threads (1 thread per core). In Fig. 17, we see that SDFGs perform on-par with the frameworks on all graphs, where Galois is marginally faster on social networks (~1.53× on `twitter`) and the Kronecker graph. However, in road maps (`usa`, `osm-eur`) SDFGs are up to 2× faster than Galois. This result could stem from our fine-grained scheduling imposed by the map scopes.

6.4 Case Study III: Quantum Transport

Quantum Transport (QT) Simulation is used for estimating heat dissipation in nanoscale transistors. OMEN [50] is an efficient (two-time Gordon Bell finalist) QT simulator based on a nonlinear solver, written in C++ and CUDA using MKL, CUBLAS, and CUSPARSE.

We use SDFGs and transformations to optimize the computation of OMEN, the full details of which are described by Ziogas et al. [74]. A significant portion of the OMEN runtime is spent in computing Scattering Self-Energies (SSE), which is given by the formula in Fig. 18 (top-left). Here we focus on the computation of $\Sigma^{\gtrless}$. Upon generating the SDFG from the formula, we see that

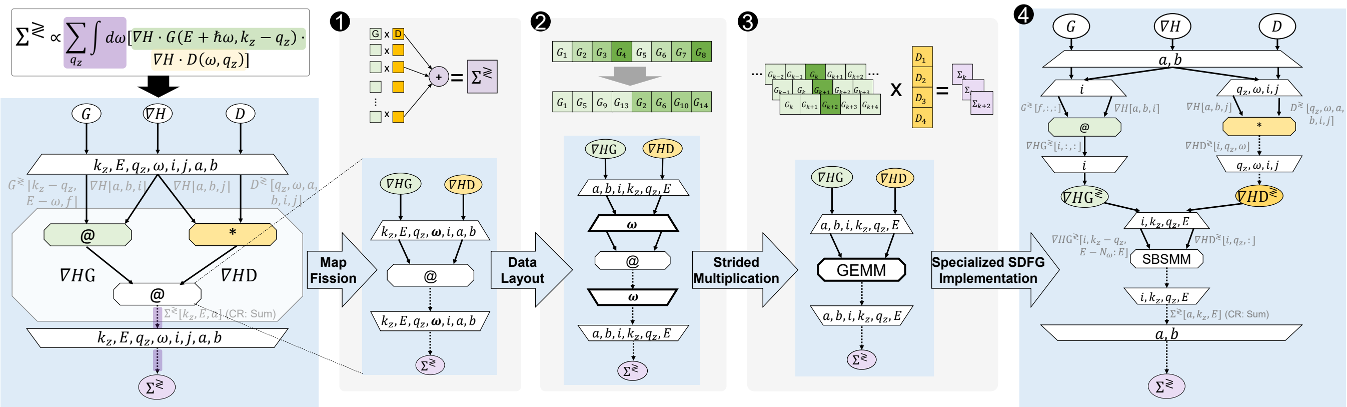

$\Sigma^{\gtrless} \propto \sum_{q_z} \int d\omega [\nabla H \cdot G(E+\hbar\omega, k_z - q_z) \cdot \nabla H \cdot D(\omega, q_z)]$
❶
❷
❸
❹
G
∇H
D
$k_z, E, q_z, \omega, i, j, a, b$
@
*
∇HG
∇HD
(CR: Sum)
$\Sigma^{\gtrless}$
Map Fission
Data Layout
Strided Multiplication
Specialized SDFG Implementation
GEMM
SBSMM
a, b

Figure 18: Optimizing Scattering Self-Energies with SDFG Transformations [74]

there are multiple parallel dimensions that compute small matrix multiplications (denoted as @) and Hadamard products (*), reducing the result with a summation.

In step ❶, we split the parallel computation into several maps, creating transient 5D and 6D arrays. Steps ❷ and ❸ transform the data layout by reorganizing the map dimensions and transient arrays such that the multiplications can be implemented with one "batched-strided" optimized CUBLAS GEMM operation. Lastly, step ❹ replaces the CUBLAS call with a *specialized* nested SDFG that performs small-scale batched-strided matrix multiplication (SBSMM), transformed w.r.t. matrix dimensions and cache sizes in hardware.

Table 2: SSE Performance

Variant	Tflop	Time [s]	% Peak	Speedup
OMEN [50]	63.6	965.45	1.3%	1×
Python (numpy)	63.6	30,560.13	0.2%	0.03×
DaCe	31.8	**29.93**	**20.4%**	**32.26×**

Table 3: Strided Matrix Multiplication Performance

	CUBLAS			DaCe (SBSMM)		
GPU	Gflop	Time	% Peak (Useful)	Gflop	Time	% Peak
P100	27.42	6.73 ms	86.6% (6.1%)	1.92	**4.03 ms**	10.1%
V100	27.42	4.62 ms	84.8% (5.9%)	1.92	**0.97 ms**	28.3%

Table 2 lists the overall SSE runtime simulating a 4,864 atom nanostructure over OMEN, numpy (using MKL for dense/sparse LA), and DaCe. Using data-centric transformations, the underutilization resulting from the multitude of small matrix multiplications is mitigated, translating into a **32.26× speedup for SDFGs over manually-tuned implementations**, and 1,021× over Python.

Breaking down the speedup, the specialized SDFG implementation of SBSMM (Table 3) outperforms CUBLAS by up to 4.76×. This demonstrates that performance engineers can use the data-centric view to *easily tune dataflow for specific problems that are not considered by existing library function implementations.*

7 RELATED WORK

Multiple previous works locally address issues posed in this paper. We discuss those papers below, and summarize them in Fig. 19.

Performance Portability Performance-portable programming models consist of high-performance libraries and programming languages. Kokkos [26] and RAJA [48] are task-based HPC libraries that provide parallel primitives (e.g., forall, reduce) and offer execution policies/spaces for CPUs and GPUs. The Legion [9] programming model adds hierarchical parallelism and asynchronous tasking by inferring data dependencies from access sets, called *logical regions*, which are composed of index spaces and accessed fields. Logical regions are similar to subsets in memlets, but differ by implicitly representing data movement. The Legion model was also extended to support numpy as a frontend for distributed GPU computations in Legate [8]. Language and directive-based standards such as OpenCL [35], OpenACC, and OpenMP [22] also provide portability, where some introduce FPGA support through extensions [21, 47]. SYCL [37] is an embedded DSL standard extending OpenCL to enable single-source (C++) heterogeneous computing, basing task scheduling on data dependencies. However, true performance portability cannot be achieved with these standards, as optimized code/directives vastly differ on each platform (especially in the case of FPGAs). Other frameworks mentioned below [7, 18, 31, 45, 58, 61, 63] also support imperative and massively parallel architectures (CPUs, GPUs), where Halide and Tiramisu have been extended [62] to target FPGA kernels. As opposed to SDFGs, none of the above models were designed to natively support both load/store architectures and reconfigurable hardware.

Separation of Concerns Multiple frameworks explicitly separate the computational algorithm from subsequent optimization schemes. In CHiLL [18], a user may write high-level transformation scripts for existing code, describing sequences of loop transformations. These scripts are then combined with C/C++, FORTRAN or CUDA [59] programs to produce optimized code using the polyhedral model. Image processing pipelines written in the Halide [58] embedded DSL are defined as operations, whose schedule is separately generated in the code by invoking commands such as `tile`, `vectorize`, and `parallel`. Tiramisu [7] operates in a similar manner, enabling loop and data manipulation. In SPIRAL [31], high-level specifications of computational kernels are written in

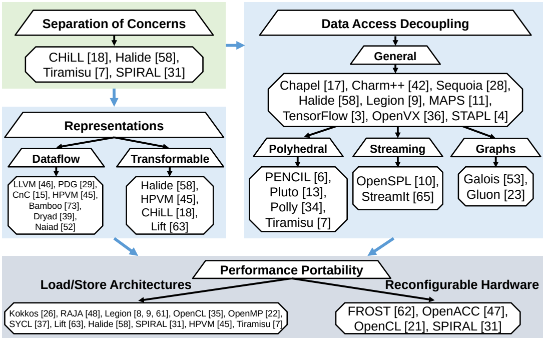

Separation of Concerns
CHiLL [18], Halide [58], Tiramisu [7], SPIRAL [31]
Data Access Decoupling
General
Chapel [17], Charm++ [42], Sequoia [28], Halide [58], Legion [9], MAPS [11], TensorFlow [3], OpenVX [36], STAPL [4]
Polyhedral
Streaming
Graphs
PENCIL [6], Pluto [13], Polly [34], Tiramisu [7]
OpenSPL [10], StreamIt [65]
Galois [53], Gluon [23]
Representations
Dataflow
Transformable
LLVM [46], PDG [29], CnC [15], HPVM [45], Bamboo [73], Dryad [39], Naiad [52]
Halide [58], HPVM [45], CHiLL [18], Lift [63]
Performance Portability
Load/Store Architectures
Reconfigurable Hardware
Kokkos [26], RAJA [48], Legion [8, 9, 61], OpenCL [35], OpenMP [22], SYCL [37], Lift [63], Halide [58], SPIRAL [31], HPVM [45], Tiramisu [7]
FROST [62], OpenACC [47], OpenCL [21], SPIRAL [31]

Figure 19: Related Work

a DSL, followed by using breakdown and rewrite rules to lower them to optimized algorithmic implementations. SDFGs, along with DIODE and the data-centric transformation workflow, streamlines such approaches and promotes a solution that enables knowledge transfer of optimization techniques across applications.

Dataflow Representations Several IRs combine dataflow with control flow in graph form. The LLVM IR [46] is a control-flow graph composed of basic blocks of statements. Each block is given in Single Static Assignment (SSA) form and can be transformed into a dataflow DAG. Program Dependence Graphs (PDGs) [29] represent statements and predicate expressions with nodes, and the edges represent both data dependencies and control flow conditions for execution. The PDG is a control-centric representation in which statements are assumed to execute dynamically, a model that fits instruction-fetching architectures well. SDFGs, on the other hand, define explicit state machines of dataflow execution, which translate natively to reconfigurable hardware. Additionally, in PDGs data-dependent access (e.g., resulting from an indirection) creates the same edge as an access of the full array. This inhibits certain transformations that rely on such accesses, and does not enable inferring the total data movement volume, as opposed to the memlet definition. SDFGs can be trivially converted to the SSA and PDG representations. In the other direction, however, the parametric concurrency that the SDFG scopes offer would be lost (barring specific application classes, e.g., polyhedral). As the representations operate on a low level, with statements as the unit element, they do not encapsulate the multi-level view as in SDFGs. HPVM [45] extends the LLVM IR by introducing hierarchical dataflow graphs for mapping to accelerators, yet still lacks a high-level view and explicit state machines that SDFGs offer. Other representations include Bamboo [73], an object-oriented dataflow model that tracks state locally through data structure mutation over the course of the program; Dryad [39] and Naiad [52], parametric graphs intended for coarse-grained distributed data-parallel applications, where Naiad extends Dryad with definition of loops in a nested context; simplified data dependency graphs for optimization of GPU applications [70]; deterministic producer/consumer graphs [15]; and other combinations of task DAGs with data movement [32]. As the SDFG provides general-purpose state machines with dataflow, all the above models can be fully represented within it, where SDFGs have the added benefit of encapsulating fine-grained data dependencies.

Data-Centric Transformations Several representations [18, 44, 51, 58, 60, 63] provide a fixed set of high-level program transformations, similar to those presented on SDFGs. In particular, Halide’s schedules are by definition data-centric, and the same applies to polyhedral loop transformations in CHiLL. HPVM also applies a number of optimization passes on a higher level than LLVM, such as tiling, node fusion, and mapping of data to GPU constant memory. Lift [63] programs are written in a high-level functional language with predefined primitives (e.g., map, reduce, split), while a set of rewrite rules is used to optimize the expressions and map them to OpenCL constructs. Loop transformations and the other aforementioned coarse-grained optimizations are all contained within our class of data-centric graph-based transformations, which can express arbitrary data movement patterns.

Decoupling Data Access and Computation The Chapel [17] language supports controlling and reasoning about locality by defining object locations and custom iteration spaces. Charm++ [42] is a parallel programming framework, organized around message passing between collections of objects that perform tasks in shared- or distributed-memory environments. In the Sequoia [28] programming model, all communication is made explicit by having tasks exchange data through argument passing and calling subtasks. MAPS [11] separates data accesses from computation by coupling data with their memory access patterns. This category also includes all frameworks that implement the polyhedral model, including CHiLL, PENCIL [6], Pluto [13], Polly [34] and Tiramisu. Furthermore, the concept of decoupling data access can be found in models for graph analytics [23, 53], stream processing [10, 65], machine learning [3], as well as other high-throughput [36] and high-performance [4] libraries. Such models enable automatic optimization by reasoning about accessed regions. However, it is assumed that the middleware will carry most of the burden of optimization, and thus frameworks are tuned for existing memory hierarchies and architectures. This does not suffice for fine-tuning kernels to approach peak performance, nor is it portable to new architectures. In these cases, a performance engineer typically resorts to a full re-implementation of the algorithm, as opposed to the workflow proposed here, where SDFG transformations can be customized or extended.

Summary In essence, SDFGs provide the expressiveness of a general-purpose programming language, while enabling performance portability without interfering with the original code. Differing from previous models, the SDFG is not limited to specific application classes or hardware, and the extensible data-centric transformations generalize existing code optimization approaches.

8 CONCLUSION

In this paper, we present a novel data-centric model for producing high-performance computing applications from scientific code. Leveraging dataflow tracking and graph rewriting, we enable the role of a performance engineer — a developer who is well-versed in program optimization, but does not need to comprehend the underlying domain-specific mathematics. We show that by performing transformations on an SDFG alone, i.e., without modifying the input code, it is possible to achieve performance comparable to the state-of-the-art on three fundamentally different platforms.

The IR proposed in this paper can be extended in several directions. Given a collection of transformations, research may be conducted into their systematic application, enabling automatic optimization with reduced human intervention. More information about tasklets, such as arithmetic intensity, can be recovered and added to such automated cost models to augment dataflow captured by memlets. Another direction is the application of SDFGs to distributed systems, where data movement minimization is akin to communication-avoiding formulation.

ACKNOWLEDGMENTS

We thank Hussein Harake, Colin McMurtrie, and the whole CSCS team granting access to the Greina and Daint machines, and for their excellent technical support. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880). T.B.N. is supported by the ETH Zurich Postdoctoral Fellowship and Marie Curie Actions for People COFUND program.

REFERENCES

[1] 2010. 9th DIMACS Implementation Challenge. http://www.dis.uniroma1.it/challenge9/download.shtml

[2] 2014. Karlsruhe Institute of Technology, OSM Europe Graph. http://i11www.iti.uni-karlsruhe.de/resources/roadgraphs.php

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283.

[4] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. 2003. *STAPL: An Adaptive, Generic Parallel C++ Library*. Springer Berlin Heidelberg, Berlin, Heidelberg, 193–208.

[5] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner (Eds.). 2013. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*. Contemporary Mathematics, Vol. 588. American Mathematical Society.

[6] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 138–149. https://doi.org/10.1109/PACT.2015.17

[7] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2018. Tiramisu: A Code Optimization Framework for High Performance Systems. *CoRR* abs/1804.10694 (2018). arXiv:1804.10694 http://arxiv.org/abs/1804.10694

[8] Michael Bauer and Michael Garland. 2019. Legate NumPy: Accelerated and Distributed Array Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. ACM, New York, NY, USA, Article 23, 23 pages. https://doi.org/10.1145/3295500.3356175

[9] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 66, 11 pages.

[10] Tobias Becker, Oskar Mencer, and Georgi Gaydadjiev. 2016. *Spatial Programming with OpenSPL*. Springer International Publishing, Cham, 81–95.

[11] Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri Rubin. 2015. Memory Access Patterns: The Missing Piece of the Multi-GPU Puzzle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, Article 19, 12 pages.

[12] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. ACM, New York, NY, USA, 93–104. https://doi.org/10.1145/3078597.3078616

[13] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[14] Roberto Bruni and Ugo Montanari. 2017. *Operational Semantics of IMP*. Springer International Publishing, Cham, 53–76. https://doi.org/10.1007/978-3-319-42900-7_3

[15] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşirlar. 2010. Concurrent Collections. *Sci. Program.* 18, 3-4 (Aug. 2010), 203–217. https://doi.org/10.1155/2010/521797

[16] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and P Krishna Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. *ICWSM* 10, 10-17 (2010), 30.

[17] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312. https://doi.org/10.1177/1094342007078442

[18] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations*. Technical Report. University of Southern California.

[19] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (Oct 2004), 1367–1372. https://doi.org/10.1109/TPAMI.2004.75

[20] CUB 2019. CUB Library Documentation. http://nvlabs.github.io/cub/.

[21] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. 2012. From OpenCL to high-performance hardware on FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 531–534. https://doi.org/10.1109/FPL.2012.6339272

[22] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313

[23] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 752–768. https://doi.org/10.1145/3192366.3192404

[24] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (2011), 25 pages.

[25] NumPy Developers. 2019. NumPy Scientific Computing Package. http://www.numpy.org

[26] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[27] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag, Berlin, Heidelberg.

[28] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. ACM, New York, NY, USA, Article 83. https://doi.org/10.1145/1188455.1188543

[29] 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 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[30] Python Software Foundation. 2019. The Python Programming Language. https://www.python.org

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

[32] V. Gajinov, S. Stipic, O. S. Unsal, T. Harris, E. Ayguadé, and A. Cristal. 2012. Supporting stateful tasks in a dataflow graph. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 435–436.

[33] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[34] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 04 (2012), 1250010. https://doi.org/10.1142/S0129626412500107

[35] Khronos Group. 2019. OpenCL. https://www.khronos.org/opencl

[36] Khronos Group. 2019. OpenVX. https://www.khronos.org/openvx

[37] Khronos Group. 2019. SYCL. https://www.khronos.org/sycl

[38] Intel. 2019. Math Kernel Library (MKL). https://software.intel.com/en-us/mkl

[39] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. ACM, New York, NY, USA, 59–72. https://doi.org/10.1145/1272996.1273005

[40] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (March 2004), 1–34. https://doi.org/10.1145/1013208.1013209

[41] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14)*. ACM, New York, NY, USA, Article 6, 11 pages. https://doi.org/10.1145/2676870.2676883

[42] Laxmikant V. Kale and Sanjeev Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '93)*. ACM, New York, NY, USA, 91–108. https://doi.org/10.1145/165854.165874

[43] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379

[44] M. Kong, L. N. Pouchet, P. Sadayappan, and V. Sarkar. 2016. PIPES: A Language and Compiler for Task-Based Programming on Distributed-Memory Clusters. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. 456–467. https://doi.org/10.1109/SC.2016.38

[45] 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 (PPoPP '18)*. ACM, New York, NY, USA, 68–80. https://doi.org/10.1145/3178487.3178493

[46] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. San Jose, CA, USA, 75–88.

[47] S. Lee, J. Kim, and J. S. Vetter. 2016. OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 544–554. https://doi.org/10.1109/IPDPS.2016.28

[48] LLNL. 2019. RAJA Performance Portability Layer. https://github.com/LLNL/RAJA

[49] Michael Löwe. 1993. Algebraic Approach to Single-pushout Graph Transformation. *Theor. Comput. Sci.* 109, 1-2 (March 1993), 181–224. https://doi.org/10.1016/0304-3975(93)90068-5

[50] Mathieu Luisier, Andreas Schenk, Wolfgang Fichtner, and Gerhard Klimeck. 2006. Atomistic simulation of nanowires in the $sp^3d^5s^*$ tight-binding formalism: From boundary conditions to strain calculations. *Phys. Rev. B* 74 (Nov 2006), 205323. Issue 20.

[51] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 429–443. https://doi.org/10.1145/2786763.2694364

[52] 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 (SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[53] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[54] NVIDIA. 2019. CUBLAS Library Documentation. http://docs.nvidia.com/cuda/cublas.

[55] NVIDIA. 2019. CUSPARSE Library Documentation. http://docs.nvidia.com/cuda/cusparse.

[56] Patrick McCormick. 2019. Yin & Yang: Hardware Heterogeneity & Software Productivity . Talk at SOS23 meeting, Asheville, NC.

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

[58] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[59] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. 2011. A Programming Language Interface to Describe Transformations and Code Generation. In *Languages and Compilers for Parallel Computing*, Keith Cooper, John Mellor-Crummey, and Vivek Sarkar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 136–150.

[60] Alina Sbîrlea, Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. 2016. Polyhedral Optimizations for a Data-Flow Graph Language. In *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519 (LCPC 2015)*. Springer-Verlag, Berlin, Heidelberg, 57–72. https://doi.org/10.1007/978-3-319-29778-1_4

[61] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: A High-productivity Programming Language for HPC with Logical Regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 81, 12 pages. https://doi.org/10.1145/2807591.2807629

[62] Emanuele Del Sozzo, Riyadh Baghdadi, Saman P. Amarasinghe, and Marco D. Santambrogio. 2018. A Unified Backend for Targeting FPGAs from DSLs. *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2018), 1–8.

[63] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 205–217. https://doi.org/10.1145/2784731.2784754

[64] SymPy Development Team. 2019. SymPy Symbolic Math Library. http://www.sympy.org

[65] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, UK, 179–196.

[66] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericàs. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (Oct 2017), 3007–3020. https://doi.org/10.1109/TPDS.2017.2703149

[67] US Department of Energy. 2019. Definition - Performance Portability. https://performanceportability.org/perfport/definition/.

[68] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713

[69] Jeffrey S. Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, Brian Van Essen, Shinjae Yoo, Alex Aiken, David Bernholdt, Suren Byna, Kirk Cameron, Frank Cappello, Barbara Chapman, Andrew Chien, Mary Hall, Rebecca Hartman-Baker, Zhiling Lan, Michael Lang, John Leidel, Sherry Li, Robert Lucas, John Mellor-Crummey, Paul Peltz Jr., Thomas Peterka, Michelle Strout, and Jeremiah Wilke. 2018. Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity. (12 2018). https://doi.org/10.2172/1473756

[70] Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-bound GPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 191–202. https://doi.org/10.1109/SC.2014.21

[71] Xilinx. 2019. SDAccel. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[72] Xilinx. 2019. Vivado HLS. https://www.xilinx.com/products/design-tools/vivado

[73] Jin Zhou and Brian Demsky. 2010. Bamboo: A Data-centric, Object-oriented Approach to Many-core Software. *SIGPLAN Not.* 45, 6 (June 2010), 388–399. https://doi.org/10.1145/1809028.1806640

[74] A. N. Ziogas, T. Ben-Nun, G. Indalecio Fernández, T. Schneider, M. Luisier, and T. 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 (SC '19)*.

A SDFG OPERATIONAL SEMANTICS

We begin by defining the different elements of our IR, which is a graph of graphs. We follow by defining how a function expressed in our IR must be called and then give semantic rules of how an SDFG is evaluated. We precede each operational semantic rule by a less formal description of the intent of the rule.

A.1 Elements of an SDFG

An SDFG is a directed multigraph defined by the tuple (S, T, s_0), whose vertex set S represent *states*, its edges T represent *transitions* between them, and has one *start-state* s_0. It is a multigraph since there can be multiple transitions between a pair of states.

An SDFG state $s \in S$ is a named, directed acyclic multigraph defined by the tuple $(V, E, name)$. Each node $v_i \in V$ is of one of the following types, as shown in Table 1 of the paper: *data*, *tasklet*, *mapentry*, *mapexit*, *stream*, *reduce*, *consume-entry*, *consume-exit*, *invoke*. Each node type defines *connectors* — attachment points for edges defining the nature of the connection, and the edges E indicate dependencies which constrain the execution order. Each edge carries a *memlet*, an annotation that specifies dataflow, as well as the connectors on its source and destination nodes. Below, we describe the node types in SDFG states and the exact definition of memlets.

As opposed to inherently sequential representations (cf. [14, Rule 3.15]), in SDFGs the execution order is mainly constrained by explicit dependencies.

To parameterize some aspects of the graph, there exists a global namespace for **symbols**. Symbols can be used in all *symbolic expressions* mentioned below, and at runtime they evaluate to scalar values. (§ A.2).

A **data** node represents a typed location in memory. The memory itself can be allocated outside of the SDFG and passed as a pointer upon invocation, or allocated at runtime if transient. A data node is a tuple (*name*, *basetype*, *dimensions*, *transient*). The name is an identifier, the basetype a basic type (e.g., `int`, `float`, `double`), and dimensions a list of symbolic integer expressions. Memory pointed to by differently named data nodes must not alias. A data node has an implicit connector for each adjacent edge.

A **tasklet** node represents computation. It is defined by the tuple *(inputs, outputs, code)*, where inputs and outputs are sets of elements of the form *(name, basetype, dimensions)* that define connectors with the same name, representing the only external memory the code can read or write. Code is a string, which can be written in Python or languages supported by the target architecture, such as C++.

A **stream** node represents an array of typed locations in memory, accessed using queue semantics, i.e., using `push` and `pop` operations. A stream has the input connectors *data* and *push*, as well as the output connectors *data* and *pop*. The *data* connectors allow initialization/copying of the data stored in the queues from/to an array, *push* and *pop* enqueue and dequeue elements, respectively.

The **memlet** annotation represents dataflow, defined by the tuple *(src_conn, dst_conn, subset, reindex, accesses, wcr)*. The *subset* function selects which subset of elements visible at the source connector will flow to the destination connector. The *reindex* function specifies at which indices the data will be visible at the destination node. We express *subset* and *reindex* as functions on integer sets, and implement them as lists of exclusive ranges, where each range refers to one data dimension and defined by *start:end:stride:tilesize*. *tilesize* is used to propagate multiple elements at a time, e.g., vector types. Fig. 20 extends Fig. 5b from the paper by showing the same memlets in subset/reindex notation rather than array index labels.

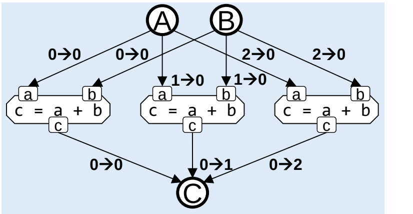


Figure 20: Expanded Map with *subset*→*reindex* Notation

The **wcr** optional attribute (write-conflict resolution) is a function $S \times S \rightarrow S$ for a data type S. The function receives the old value currently present at the destination and the new value, and produces the value that will be written. This attribute is normally defined when data flows concurrently from multiple sources, e.g., in a parallel dot product.

Map entry/exit nodes are defined by a tuple *(range, inputs, outputs)*, creating a scope. The subgraph within the scope of a map is defined by the nodes that are dominated by the map-entry and post-dominated by the map-exit. This subgraph is expanded at runtime and executed concurrently according to the symbolic *range* of the mapentry, which takes the form *identifier=begin:end:step*. Input and output connectors of map nodes are either defined as `IN_*`/`OUT_*` (`*` is any string) to map memlets outside the scope to memlets within it; or other identifiers, which can be used to create data-dependent map ranges. The range identifier becomes a scope-wide symbol, and can used to define memlet attributes (*subset*, *reindex*).

The **invoke** node allows to call an SDFG within an SDFG state. Its semantics are equivalent to a tasklet that has input connectors for each data-node and undefined symbol used in the invoked SDFG, and an output node for each data node.

The **reduce** alias node is used to implement target-optimized reduction procedures. It is defined by axes and a *wcr* function, consisting of an input and an output connector (of the same type). It is semantically equivalent to a map whose range spans the incoming and outgoing memlet data, containing an identity tasklet (`output=input`) and an output memlet annotated by the given *wcr* for the reduced axes.

Consume entry/exit nodes complement maps, enabling producer/consumer relationships via dynamic parallel processing of streams. The two nodes form a scope, similarly to a map, but also accept a *stream_in* connector (connected to a stream), and a quiescence condition after which processing stops. Thus, a consume entry node is a tuple *(range, cond, inputs, outputs)*. Semantically, a consume scope is equivalent to a map using the same *range*, with an internal *invoke*d SDFG containing the scope. This state connects back to itself without any assignments and the condition *cond*.

A **state transition** is defined as a tuple *(source, destination, condition, assignments)*. *Condition* can depend on symbols and data from data nodes in the source state, whereas *assignments* take the form *symbol = expression.* Once a state finishes execution, all outgoing state transitions of that state are evaluated in an arbitrary order, and the destination of the first transition whose condition is true is the next state which will be executed. If no transition evaluates to true, the program terminates. Before starting the execution of the next state, all assignments are performed, and the left-hand side of assignments become symbols.

A.2 Operational Semantics

A.2.1 Initialization. ***Notation*** We denote collections (sets/lists) as capital letters and their members with the corresponding lowercase letter and a subscript, i.e., in an SDFG $G = (S, T, s_0)$ the set of states S as s_i, with $0 \leq i < |S|$. Without loss of generality we assume s_0 to be the start state. We denote the value stored at memory location a as $M[a]$, and assume all basic types are size-one elements to simplify address calculations.

The state of execution is denoted by ρ. Within the state we carry several sets: *loc*, which maps names of data nodes and transients to memory addresses; *sym*, which maps symbol names (identifiers) to their current value; and *vis*, which maps connectors to the data visible at that connector in the current state of execution.

We define a helper function *size*(), which returns the product of all dimensions of the data node or element given as argument (using ρ to resolve symbolic values). Furthermore, *id*() returns the *name* property of a data or transient node, and *offs()* the offset of a data element relative to the start of the memory region it is stored in. The function *copy*() creates a copy of the object given as argument, i.e., when we modify the copy, the original object remains the same.

Invocation When an SDFG G is called with the data arguments $A \equiv [a_i = p_i]$ (a_i is an identifier, p_i is an address/pointer) and symbol arguments $Z \equiv [z_i = v_i]$ (z_i is an identifier, v_i an integer) we initialize the configuration ρ:

1. For all symbols z_i in Z: $sym[z_i] \leftarrow v_i$.
2. For all data and stream nodes $d_i \in G$ without incoming edges s.t. $id(d_i) = a_i$: $loc(d_i) \leftarrow p_i, vis(d_i.data) \leftarrow M[p_i, ..., p_i + size(d_i)]$.
3. Set $current$ to a copy of the start state of G, s_0.
4. Set $state$ to $id(s_0)$.
5. Set $qsize[f_i]$ to zero for all stream nodes $f_i \in G$.

This can be expressed as the following rule:

$$\frac{\begin{array}{c} G = (S, T), \\ start\_state(G) \rightarrow s_0, \\ D : \text{data nodes in } s_0, \\ F : \text{stream nodes in } s_0 \end{array}}{\begin{array}{ll} (call(G, A, Z), \rho) \rightarrow \rho[& state \mapsto id(s_0), \\ & current \mapsto copy(s_0), \\ & \forall a_i = p_i \in A : loc[a_i \mapsto p_i], \\ & \forall z_i = v_i \in Z : sym[z_i] \mapsto v_i, \\ & \forall d_i \in D : vis[d_i.data] \mapsto M[p_i, .., p_i + size(d_i)], \\ & \forall f_i \in G.S : qsize[id(f_i)] \mapsto 0] \end{array}}$$

A.2.2 Propagating Data in a State. Execution of a state entails propagating data along edges, governed by the rules defined below.

Data is propagated within a state in an arbitrary order, constrained only by dependencies expressed by edges. We begin by considering all nodes in the current state, which we add to *current*. We gradually remove nodes as they are processed, until *current* is empty and we can proceed to the next state or termination.

To keep the description concise, we omit details of address translation and indexing. We also use the *subset* and *reindex* properties as functions that resolve symbols and return data elements directly.

Element Processing In each step, we take one element q (either a memlet or a node) of *current*, for which all input connectors have visible data, then:

If q is a **memlet** *(src, dst, subset, reindex, wcr)*, update $vis[dst]$ to $wcr(reindex(subset(vis[src])))$:

$$\frac{\begin{array}{c} q = memlet(src, dst, subset, reindex, wcr), \\ (vis[src], \rho) \neq \emptyset, \\ (wcr(reindex(subset(vis[src]))), \rho) \rightarrow [d_0, .., d_n] \end{array}}{(q, \rho) \rightarrow \rho[vis[dst] \mapsto [d_0, .., d_n]]}$$

If q is a **data** node, update its referenced memory for an input connector c_i,
$M[loc(id(q)), .., loc(id(q)) + size(vis[q.data])] = vis[q.data]$:

$$\frac{\begin{array}{c} q = data(id, dims, bt, transient), \\ (\forall x, q.c_i \in current : vis[q.c_i], \rho) \neq \emptyset, \\ (\forall x, q.c_i \in current : vis[q.c_i], \rho) \rightarrow [d_0^i, d_1^i, .., d_{k_i}^i], \\ \forall d_j^i : loc(id(q)) + offs(d_j^i) = l_j^i \end{array}}{(q, \rho) \rightarrow \rho[\forall i, k_i : M[l_j^i, .., l_j^i + size(d_j^i)] = d_j^i]}$$

If q is a **tasklet**, generate a prologue that allocates local variables for all input connectors c_i of q, initialized to $vis[c_i]$ (P_1), as well as output connectors (P_2). Generate an epilogue Ep which updates $\rho[vis[c_i] \mapsto v_i]$ for each output connector c_i of q with the contents of the appropriate variable (declared in P_2). Execute the concatenation of $(P_1; P_2; code; Ep;)$.

$$\frac{\begin{array}{c} q = tasklet(Cin, Cout, code), \\ (\forall x, q.c_i \in current : vis[q.c_i], \rho) \neq \emptyset, \\ (P_1 = [\forall c_i \in Cin : type(c_i) id(c_i) = vis[c_i]];, \rho), \\ P_2 = [\forall c_i \in Cout : type(c_i) id(c_i);], \\ (Ep = [\forall c_i \in Cout : \&vis(c_i) = id(c_i);], \rho), \end{array}}{(q, \rho) \rightarrow \rho[exec(P_1; P_2; code; Ep;)]}$$

If q is a **mapentry** node with range $y = R$ (y is the identifier) and scope $o \subset V$: Remove o from *current*. Remove q and the corresponding map exit node from o. For each element in $r_i \in R$, replicate o, resolve any occurrence of y to r_i, connect incoming connectors of q and p in $state$.

$$\frac{\begin{array}{l} q = mapentry(Cin, Cout, R), \\ \forall cin_i : vis[cin_i] \neq \emptyset, \\ o = scope(q), o' = scope(q) \setminus \{q, mexit(q)\}, \\ NewSyms = [cin_i : \nexists(cin_i, y)] \end{array}}{\begin{array}{ll} (q, \rho) \rightarrow \rho[& current \mapsto o' \cup [\forall r_i : ressym(copy(o'), r_i)], \\ & \forall n_i \in NewSym : sym[n_i] \mapsto vis[cin_i]] \end{array}}$$

If q is a **consume-entry** node, defined by *(range, cond, cin, cout)*, replace q with a mapentry and do the same for the corresponding consume exit node. Then we create a new SDFG *new*, which contains the contents of the original consume scope *scope*(q). *new* consists of one state s_0, and a single state transition to the same state with the condition *cond*, defined by $(s_0, s_0, cond, [])$. Finally, we replace *scope*(q) in *current* with an invoke node for *new* and reconnect the appropriate edges between the entry and exit nodes.

$$\frac{\begin{array}{l} q = consume - entry(range, cond, cin, cout) \\ newsdfg = SDFG(scope(q) \setminus \{q, cexit(q)\}, (s_0, s_0, cond, [])) \\ iv = invoke(newsdfg) \\ men = mapentry(range, cin, cout) \\ mex = mapexit(cexit(q).cin, cexit(q).cout) \end{array}}{(q, \rho) \rightarrow \rho[current \mapsto (current \setminus q, cexit(q)) \cup \{iv, men, mex\}]}$$

If q is a **reduce** node defined by the tuple *(cin, cout, range)*, we create a mapentry node *men* with the same range, a mapexit node *mex*, and a tasklet `o = i`. We add these nodes to the node set of *current, nd(current)*. We connect them by adding edges to the edge set of *current*.

$$\frac{\begin{array}{c} q = reduce(cin, cout, range, wcr), \\ vis[cin] \neq \emptyset, \\ men = mapentry(\{IN\_1\}, \{OUT\_1\}, range), \\ mex = mapexit(\{IN\_1\}, \{OUT\_1\}), \\ t = tasklet(\{i\}, \{o\}, \text{“}o = i;\text{”}) \end{array}}{\begin{array}{ll} (q, \rho) \rightarrow \rho[& nd(current) \mapsto nd(current) \cup \{men, mex, t\} \\ & ed(current) \mapsto ed(current) \cup \{(men.OUT\_1, t.i), (t.o, mex.IN\_1)\} \\ & vis[men.IN\_1] \mapsto vis[q.cin], \\ & vis[q.IN] \mapsto \emptyset] \end{array}}$$

If q is a **stream**, we add the data visible at the *push* connector to the appropriate memory location (indicated by $qsize[id(q)]$), and increment it. A stream node with an edge connected to its pop connector can only be evaluated if $qsize[id(q)] \geq 1$. When propagating through streams, there are three cases for one step:

❶ Data is both pushed into and popped from the stream:

$$\frac{\begin{array}{l} q = \mathit{stream}(\mathit{push}, \mathit{pop}, \mathit{size}),\\ vis[push] \neq emptyset,\\ \exists(q.pop, dst) \in current,\\ \rho[qsize(id(q)) \geq 1] \end{array}}{\begin{array}{ll} (q, \rho) \to \rho[& vis[q.pop] \mapsto M[\mathit{offs}(q), .., \mathit{offs}(q) + size(q)],\\ & M[\mathit{offs}(q) + size(q) \times qsize[q], ..,\\ & \quad \mathit{offs}(q) + size(q) \times (qsize[q] + 1)] \mapsto vis[push],] \end{array}}$$

❷ Data is only pushed to the stream node:

$$\frac{\begin{array}{l} q = \mathit{stream}(\mathit{push}, \mathit{pop}, \mathit{size}),\\ vis[push] \neq emptyset,\\ \nexists(q.pop, dst) \in current \end{array}}{\begin{array}{ll} (q, \rho) \to \rho[& M[\mathit{offs}(q) + size(q)qsize[q], ..,\\ & \quad \mathit{offs}(q) + size(q)(qsize[q] + 1)] \mapsto vis[push],\\ & qsize[q] \mapsto qsize[q] + 1] \end{array}}$$

❸ Data is popped from the stream but not pushed into it:

$$\frac{\begin{array}{l} q = \mathit{stream}(\mathit{push}, \mathit{pop}, \mathit{size}),\\ \nexists(src, q.push) \in S_{state},\\ \exists(q.pop, dst) \in current \end{array}}{\begin{array}{ll} (q, \rho) \to \rho[& vis[q.pop] \mapsto M[\mathit{offs}(q), .., \mathit{offs}(q) + size(q)],\\ & qsize[q] \mapsto qsize[q] - 1] \end{array}}$$

Following the element processing step, we remove q from *current*, repeating the above step until *current* is empty.

A.2.3 Evaluating State Transitions. Once *current* is empty, we evaluate all outgoing state transitions of the current state: $(state, next, cond, assigs) \in T$. For each transition, we resolve all symbols in *cond* and the right-hand sides of *assigs* using ρ, then evaluate arithmetic and boolean expressions using standard semantic rules, which we omit here. If no condition evaluates to true, signal the completion of G to the caller and stop the evaluation of G:

$$\frac{\begin{array}{c} \mathit{current} = \emptyset,\\ \nexists(\rho[state], next, cond, assigs) \in T : (cond, \rho) \to True \end{array}}{\rho[state \mapsto \emptyset]}$$

Otherwise, we choose an arbitrary transition for which $cond \to true$ and update ρ: Set $state$ to $next$, set *curr* to a copy of $S[next]$. For each left-hand side of an assignment z_i, update $sym[z_i]$ with the value of the corresponding right-hand side v_i. Data propagation then follows Section A.2.2:

$$\frac{\begin{array}{c} \text{current} = \emptyset,\\ \exists(\rho[state], next, cond, assigs) \in T : (cond, \rho) \to True,\\ (assigs, \rho) \to [z_i = v_i] \end{array}}{\begin{array}{ll} \rho[& state \mapsto next,\\ & current \mapsto copy(next),\\ & sym[z_i] \mapsto v_i] \end{array}}$$

Table 4: Supported Transformations

Name	Description
Map transformations	
`MapCollapse`	Collapses two nested maps into one. The new map has the union of the dimensions of the original maps.
`MapExpansion`	Expands a multi-dimensional map to two nested ones. The dimensions are split to two disjoint subsets, one for each new map.
`MapFusion`	Fuses two consecutive maps that have the same dimensions and range.
`MapInterchange`	Interchanges the position of two nested maps.
`MapReduceFusion`	Fuses a map and a reduction node with the same dimensions, using conflict resolution.
`MapTiling`	Applies orthogonal tiling to a map.
Data transformations	
`DoubleBuffering`	Pipelines writing to and processing from a transient using two buffers.
`LocalStorage`	Introduces a transient for caching data.
`LocalStream`	Accumulates data to a local transient stream.
`Vectorization`	Alters the data accesses to use vectors.
Control-flow transformations	
`MapToForLoop`	Converts a map to a for-loop.
`StateFusion`	Fuses two states into one.
`InlineSDFG`	Inlines a single-state nested SDFG into a state.
Hardware mapping transformations	
`FPGATransform`	Converts a CPU SDFG to be fully invoked on an FPGA, copying memory to the device.
`GPUTransform`	Converts a CPU SDFG to run on a GPU, copying memory to it and executing kernels.
`MPITransform`	Converts a CPU Map to run using MPI, assigning work to ranks.

B DATA-CENTRIC GRAPH TRANSFORMATIONS

Table 4 lists the transformations used in the Performance Evaluation section of the paper.

$$\begin{tikzcd} L \arrow[r, "r"] \arrow[d, "m"'] & R \arrow[d, "m'"] \\ G \arrow[r, "r'"'] & H \end{tikzcd}$$

Figure 21: Single-Pushout Approach

The implementation of graph transformations is based on *algebraic graph rewriting* [27]. Each transformation is implemented by a *rule* $p : L \xrightarrow{r} R$, which consists of the two sub-graphs L and R, while r is a relation that maps the vertices and edges of L to elements of the same kind in R. Moreover, a specific *matching* of L in the SDFG G is represented by a relation $m : L \to G$. Applying the optimization on G produces the transformed graph H, which can be constructed as part of the *pushout* $< H, m', r' >$ of p, G and m. A visualization of the above method, also known as the *single-pushout approach* [49], is shown in Fig. 21.

C POLYBENCH FLAGS

We compile Polybench code for all compilers using the following base flags: `-O3 -march=native -mtune=native`. Each compiler was also tested with different variants of flags, where we report the best-performing result in the paper. The flags are (variants separated by semicolons):

- **gcc 8.2.0:** `-O2`; Base flags
- **clang 6.0:** Base flags
- **icc 18.0.3:** Base flags; `-mkl=parallel -parallel`
- **Polly (over clang 6.0):** `-mllvm -polly`; `-mllvm -polly -mllvm -polly-parallel -lgomp`
- **Pluto 0.11.4**: `-ftree-vectorize -fopenmp` and among best of `--tile`; `--tile --parallel`; `--tile --parallel --partlbtile`; `--tile --parallel --lbtile --multipar`
- **PPCG 0.8:** Base flags

All variants were also tested with compile-time size specialization.

D TRANSFORMATION EXAMPLE: REDUNDANT ARRAY REMOVAL

A simple transformation implemented in DaCe is `RedundantArray`, whose code can be found below. The transformation removes a transient array that is used directly before another array, creating a copy, and not used anywhere else (making the copy redundant). This situation often happens after transformations and due to the strict nature of some language frontends (e.g., TensorFlow). The subgraph expression to match (lines 6–14) is a path graph of size two, connecting the two access nodes. The nodes have no restriction on their content (see lines 6–7). The function `can_be_applied` is called on a matching subgraph for further programmatic checks (lines 16–58), and the function `apply` (lines 60–78) applies the transformation using the SDFG builder API. The checks ensure that the array is indeed transient and not used in other instances of data access nodes. To avoid recomputing subsets (which may not be feasible to compute symbolically), if the transformation operates in strict mode, it only matches two arrays of the same shape (lines 51–56). The transformation then operates in a straightforward manner, renaming the memlets to point to the second (not removed) array (lines 66–70) and redirecting dataflow edges to that data access node (lines 73–74). Lastly, the `in_array` node is removed from the SDFG state (line 78).

In line 81, the transformation is registered with a global registry. This process enables users to import custom transformations and integrate them into DaCe easily. However, because `RedundantArray` is a strict transformation (i.e., can only improve performance), it is also hardcoded in a set of such transformations within the SDFG class. Strict transformations are applied automatically after processing a DaCe program, and other such transformations include `StateFusion` and `InlineSDFG`.

```
 1 class RedundantArray(pm.Transformation):
 2     """ Implements the redundant array removal transformation,
 3         applied when a transient array is copied to and from (to
 4         another array), but never used anywhere else. """
 5 
 6     _in_array = nodes.AccessNode('_')
 7     _out_array = nodes.AccessNode('_')
 8 
 9     @staticmethod
10     def expressions():
11         return [
12             nxutil.node_path_graph(RedundantArray._in_array,
13                                    RedundantArray._out_array),
14         ]
15 
16     @staticmethod
17     def can_be_applied(graph, candidate, expr_index, sdfg,
18                        strict=False):
19         in_array =  \
20           graph.nodes()[candidate[RedundantArray._in_array]]
21         out_array = \
22           graph.nodes()[candidate[RedundantArray._out_array]]
23 
24         # Ensure out degree is one (only one target, out_array)
25         if graph.out_degree(in_array) != 1:
26             return False
27 
28         # Make sure that the candidate is a transient variable
29         if not in_array.desc(sdfg).transient:
30             return False
31 
32         # Make sure both arrays are using the same storage location
33         if (in_array.desc(sdfg).storage !=
34               out_array.desc(sdfg).storage):
35             return False
36 
37         # Find occurrences in this and other states
38         occurrences = []
39         for state in sdfg.nodes():
40             occurrences.extend([
41                 n for n in state.nodes()
42                 if isinstance(n, nodes.AccessNode)
43                 and n.desc(sdfg) == in_array.desc(sdfg)
44             ])
45 
46         if len(occurrences) > 1:
47             return False
48 
49         # If strict, only apply if arrays are of same shape
50         # (no need to modify memlet subset)
51         if (strict and (len(in_array.desc(sdfg).shape) !=
52                 len(out_array.desc(sdfg).shape))
53                 or any(i != o for i, o in zip(
54                     in_array.desc(sdfg).shape,
55                     out_array.desc(sdfg).shape))):
56             return False
57 
58         return True
59 
60     def apply(self, sdfg):
61         graph = sdfg.nodes()[self.state_id]
62         in_array = gnode(RedundantArray._in_array)
63         out_array = gnode(RedundantArray._out_array)
64 
65         for e in graph.in_edges(in_array):
66             # Modify all incoming edges to point to out_array
67             path = graph.memlet_path(e)
68             for pe in path:
69                 if pe.data.data == in_array.data:
70                     pe.data.data = out_array.data
71 
72             # Redirect edge to out_array
73             graph.remove_edge(e)
74             graph.add_edge(e.src, e.src_conn, out_array,
75                            e.dst_conn, e.data)
76 
77         # Finally, remove in_array node
78         graph.remove_node(in_array)
79 
80 # Register transformation in global registry
81 pm.Transformation.register_pattern(RedundantArray)
```

E GRAPH DATASETS

Table 5 lists the graph datasets used in the BFS case study of the paper.

Table 5: Graph Properties

Name	Nodes	Edges	Avg. Degree	Max Degree	Size (GB)
Road Maps					
USA [1]	24M	58M	2.41	9	0.62
OSM-eur-k [2]	174M	348M	2.00	15	3.90
Social Networks					
soc-LiveJournal1 [24]	5M	69M	14.23	20,293	0.56
twitter [16]	51M	1,963M	38.37	779,958	16.00
Synthetic Graphs					
kron21.sym [5]	2M	182M	86.82	213,904	1.40

F INDIRECT MEMORY ACCESS

Indirect memory access, i.e., A[b[i]], is an important characteristic of sparse data structure and pointer jumping algorithms. Indirection cannot be directly represented by a single memlet. As shown in Fig. 22 (an excerpt of Sparse Matrix-Vector Multiplication), such statements are converted to a subgraph, where the internal access is given exactly (A_col[j] in the figure), and the indirect memory is given by a memlet with one access (x(1)[:]). The memory is then accessed dynamically and copied using a tasklet.

```
@dace.map
def spmv(j: _[prow[0]:prow[1]]):
  a << A_val[j]
  x_in << x[A_col[j]]
  out >> b(1, lambda x,y: x+y, 0)[i]

  out = a * x_in
```

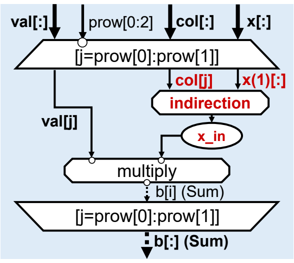


Figure 22: Indirect Memory Access Dataflow