Productivity, Portability, Performance: Data-Centric Python

Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, and Torsten Hoefler
Department of Computer Science, ETH Zurich
Switzerland

ABSTRACT

Python has become the *de facto* language for scientific computing. Programming in Python is highly productive, mainly due to its rich science-oriented software ecosystem built around the NumPy module. As a result, the demand for Python support in High Performance Computing (HPC) has skyrocketed. However, the Python language itself does not necessarily offer high performance. In this work, we present a workflow that retains Python's high productivity while achieving portable performance across different architectures. The workflow's key features are HPC-oriented language extensions and a set of automatic optimizations powered by a data-centric intermediate representation. We show performance results and scaling across CPU, GPU, FPGA, and the Piz Daint supercomputer (up to 23,328 cores), with 2.47x and 3.75x speedups over previous-best solutions, first-ever Xilinx and Intel FPGA results of annotated Python, and up to 93.16% scaling efficiency on 512 nodes.

1 INTRODUCTION

Python is *the* language to write scientific code [30]. The capability to write and maintain Python code with ease, coupled with a vast number of domain-specific frameworks and libraries such as SciPy, Matplotlib, scikit-learn [62], or pandas [74], leads to high productivity. It also promotes collaboration with reproducible scientific workflows shared using Jupyter notebooks [42]. Therefore, numerous scientific fields, ranging from machine learning [2, 61] to climate [72] and quantum transport [75] have already adopted Python as their language of choice for new developments.

As a result, the scientific community now pushes to also make Python *the* language for writing high-performance code. For most scientific applications, NumPy arrays [36] provide the core data structures, interfaces, and BLAS and LAPACK library interoperability. NumPy is optimized to provide efficient data structures and fast library implementations for many common operations. However, the performance benefits of NumPy are tied to optimized
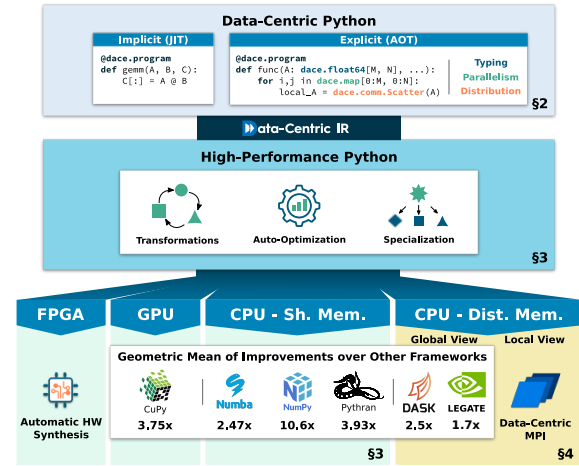
Figure 1: Data-Centric Python overview.

method calls and vectorized array operations, both of which evaporate in larger scientific codes that do not adhere to these constraints. Therefore, there is a significant performance gap between the *numpythonic* code-style and general code written by domain scientists.

In HPC, the three **P**s (**Productivity, Portability, Performance**) are driving recent developments in infrastructure and programming model research to ensure sustainability [14, 53]. In Python, this drive has resulted in many optimized domain-specific libraries and frameworks [2, 17, 46, 61, 72]. Simultaneously, the diversity of the hardware landscape motivated the creation of interface libraries, such as CuPy [56], which provides replacements to NumPy operations for NVIDIA and AMD GPUs, and MPI4PY [22], which offers direct MPI bindings. Lower-level interfaces, such as Cython [12], promise high performance at the cost of writing code that resembles C, and lazy evaluation of array operations [10, 43, 61] that enable high-performance runtime systems. Furthermore, a variety of JIT compilers [17, 34, 45] address the performance degradation resulting from the interpreter. Last but not least, runtime systems [10], distributed tasking (Dask) [23], and remote procedure calls [52] further scale Python to distributed systems. Despite the abundance of choices, Python still struggles: while each approach works towards one or more of the **P**s, none of them supports all at the same time.

We propose a way to bridge the gap between the three **P**s for Python programming using a data-centric paradigm. In particular, we empower Python users with an automatic optimization and specialization toolbox, which spans the entire Python/HPC ecosystem (Fig. 1) — from the code, through communication distribution, to hardware mapping. At the core of the toolbox, we use the Stateful Dataflow multiGraphs (SDFG) [13] data-centric intermediate representation, which enables these optimizations in the form of

multi-level data movement transformations. With a data-centric representation, as opposed to library bindings, all data dependencies and potential overlap are inferred statically from the code, and interpreter overhead is mitigated. Compared with implicit and lazy evaluation approaches, we also provide a set of extensions that give power users complete control over parallelism and partitioning schemes, using *pythonic* principles and syntax (e.g., returning "local view" objects from global data, but allowing users to operate on the "global view" as well).

We demonstrate a wide variety of benchmarks using the automatic toolbox ***over annotated Python code***, both on individual nodes and the Piz Daint supercomputer. For the former, we show that it consistently outperforms other automatic approaches on multicore CPUs and GPUs, and *for the first time* show automatic Python HPC compilation results for both Xilinx and Intel FPGAs, which vastly differ in architecture and programming language. In distributed-memory environments, we show high scaling efficiency and absolute performance compared with distributed tasking. Thus, we realize all three **P**s within a single system.

The paper makes the following contributions:

- (**Productivity**) Definition of high-performance Python, a methodology to translate it to a data-centric IR, and extensions to improve said conversion via explicit annotation.
- (**Portability**) A set of automatic optimizations for CPU, GPU and FPGA, outperforming the best prior approaches by 2.47× on CPU and 3.75× on GPU on average (geometric mean [1]).
- (**Performance**) Automatic implicit MPI transformations and communication optimizations, as well as explicit distribution management, with the former scaling to 512 nodes with up to 93.16% efficiency.

2 DATA-CENTRIC PYTHON

The central tenet of our approach is that understanding and optimizing data movement is the key to portable, high-performance code. In a data-centric programming paradigm, three governing principles guide development and execution:

(1) Data containers must be separate from computations.
(2) Data movement must be explicit, both from data containers to computations and to other data containers.
(3) Control flow dependencies must be minimized, they shall only define execution order if no implicit dataflow is given.

In the context of SDFGs, examples of data containers are arrays and scalar data, which have a NumPy-compatible data type, such as int32 or float64.

Python is an imperative language and, therefore, not designed to express data movement. Its terseness makes the process of understanding dataflow difficult, even when comparing to other languages like C and FORTRAN, as the types of variables in Python code cannot be statically deduced.

Below, we define high-performance Python programs, discuss the decorators that we must add to Python code to make the dataflow analyzable, and then detail how we translate them into the SDFG data-centric intermediate representation.

2.1 High Performance Python

Our approach supports a large subset of the Python language that is important for HPC applications. The focus lies on NumPy arrays [36] and operations on such arrays. In addition to the low-overhead data structures NumPy offers, it is central to many frameworks focused on scientific computing, e.g., SciPy, pandas, Matplotlib. As opposed to lazy evaluation approaches, high-performance Python must take control flow into account to auto-parallelize and avoid interpreter overhead. This tradeoff between performance and productivity is necessary because Python features such as co-routines are not statically analyzable and have to be parsed as "black-boxes". To combat some of these Python quirks, we propose to augment the language with analyzable constructs useful for HPC.

2.2 Annotating Python

The Data-Centric (DaCe) Python frontend parses Python code and converts it to SDFGs on a per-function basis. The frontend will parse only the Python functions that have been annotated explicitly by the user with the `@dace.program` decorator. DaCe programs can then be called like any Python function and perform Just-in-Time (JIT) compilation.

Static symbolic typing. To enable Ahead-of-Time (AOT) compilation, which is of key importance for FPGAs and for reusing programs across different inputs, SDFGs should be statically typed. Therefore, the function argument data types are given as type annotations, providing the required information as shown below:

```
N = dace.symbol()
@dace.program
def jacobi_1d(TSTEPS: dace.int32,
              A: dace.float64[N],
              B: dace.float64[N]):

    for t in range(1, TSTEPS):
        B[1:-1] = 0.33333 * (A[:-2]+A[1:-1]+A[2:])
        A[1:-1] = 0.33333 * (B[:-2]+B[1:-1]+B[2:])
```

The Python method jacobi_1d has three arguments; TSTEPS is a 32-bit integer scalar, while A and B are double precision floating-point vectors of length N. The symbolic size N, defined with **dace.symbol**, indicates that the vector sizes can be dynamic (but equal). All subsets are then symbolically defined (e.g., the subset B[1:-1] becomes B[1:N-1], and symbolic manipulation can then be performed in subsequent data-centric transformations.

Parametric parallelism. An important feature that has no direct expression in Python is a loop that can run in parallel. Our approach supports explicit parallelism declaration through map scopes, similarly to an N-dimensional parallel for. There are two ways to take advantage of this feature. DaCe provides the **dace.map** iterator, which can be used in Python code as a substitute to the Python built-in **range** iterator and generates a map scope when parsed:

```
for i, j in dace.map[0:M, 0:N]:
    A[i, j] = B[j, i]
```

Alternatively, the DaCe framework provides a LoopToMap transformation that detects for-loops in the IR, whose iterations can be

Python	SDFG Equivalent
Declarations and Types	
Primitive data types	Scalar data container
NumPy array	Array data container
Assignments	
Assignments	Tasklet or map scope with incoming and outgoing memlets for read/written operands
Array subscript	Memlet (dataflow edge)
Statements	
Branching (if)	Branch conditions on state transition edges
Iteration (for)	Conditions, increments on state transition edges
Control-flow (break, continue, return)	Edge to control structure or function exit state
Functions	
Function calls (with source) and decorator for argument types	Nested SDFG for content, memlets reduce shape of inputs and outputs
External/Library calls	Tasklet with callback or Library Node

Table 1: Mapping of Python syntax and constructs to SDFG.

executed safely in parallel (using symbolic affine expression analysis), and converts them to map scopes automatically.

2.3 From Python to DaCe

We turn to present the SDFG intermediate representation (IR) and a novel data-centric Python translation procedure in tandem. While previous work [13] converted a restricted, low-level Python definition of the SDFG IR, here we aim to cover the majority of the Python/NumPy language constructs via static analysis and fallback for unsupported features. We summarize the equivalence between Python constructs and SDFG counterparts in Table 1, and present the generation of an SDFG from a Python program using the gemm kernel as an example:

```
@dace.program
def gemm(alpha, beta, C, A, B):
    C[:] = alpha * A @ B + beta * C
```

Our first pass traverses the Python AST to simplify the expressions into individual steps, similar to static single assignment [7], respecting order of operations:

```
    tmp0 = alpha * A
    tmp1 = tmp0 @ B
    tmp2 = beta * C
    C = tmp1 + tmp2
```

The first step in the above code multiplies each element of A with alpha. SDFGs view data containers separately from the computations the data are part of, as per the first data-centric tenet.
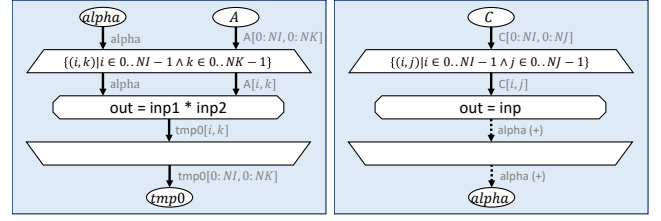
(a) Element-wise array operation tmp0 = alpha * A. **(b) Augmented assignment with WCR.**

Figure 2: SDFG representations

These containers are represented by oval *Access nodes*. In the first statement, these refer to tmp0, alpha, and A (see Fig. 2a).

In SDFGs, connections to data containers are called *memlets*, and they describe the data movement — the edge direction indicates whether it is read or written, and its contents refer to the part of the data container that is accessed. Computations consume/produce memlets and can be divided into multiple types:

(1) Stateless computations (*Tasklets*, shown as octagons), e.g., representing scalar assignments such as a = 1.
(2) Calls to external libraries (*Library Nodes*, folded rectangles), that represent calls to functions that are not in the list of functions decorated with **dace.program**. Matrix-matrix multiply is a common and important operation tmp1 = tmp0 @ B and is contained in a Library Node called *MatMul*.
(3) Calls to other SDFGs (*Nested SDFGs*, rectangles), which represent calls to functions decorated with **dace.program**.
(4) *Maps* are a particular type of Nested SDFGs matching the language augmentation previously discussed in Section 2.2 and express that the content can be processed in parallel.

Element-wise array operations automatically yield Map scopes. Similarly, assignments to arrays yield a Map scope, containing a Tasklet with the element-wise assignment. Augmented assignments such as C += 1 are a special case where the output is also an input when no data races are detected.

Parallel maps can be augmented to express how *Write-Conflict Resolution* (WCR) should determine the value of data when multiple sources write to it concurrently. If data races are found, the outgoing edges are marked as dashed. E.g., the following program requires WCR (SDFG representation is shown in Fig. 2b):

```
for i, j in dace.map[0:NI, 0:NJ]:
    alpha += C[i, j]
```

By connecting the inputs and outputs of computations with the containers explicitly, the data-centric representation of a program can be created. To represent control dependencies, we encapsulate code driven by pure data dependencies in *States* (bright blue regions) in the SDFG. The states are connected with *State Transition Edges* (in blue) that can be annotated with control flow, representing loops, branch conditions, and state machines in general. The following Python program is represented by the SDFG in Fig. 3:

```
for i in range(NI):
    C[i] += 1
```
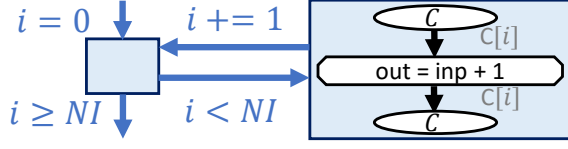
Figure 3: SDFG representation of a Python for-loop. There are two states (left: guard, right: body) connected by control flow (state transition) edges that define the loop range.
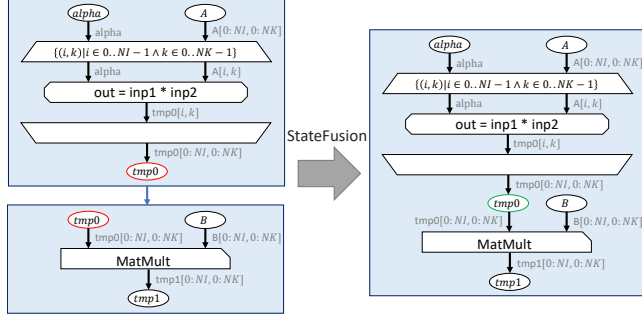

Figure 4: State fusion of `tmp0 = alpha * A` and `tmp1 = tmp0 @ B`.

The above loop also is an excellent use case for the `LoopToMap` transformation (Section 2.2) since its iterations are independent.

In the conversion to the SDFG IR, we also replace calls to library functions (e.g., `np.linalg.solve`) and object methods (`A.view()`) with custom subgraphs or Library Nodes, which users can extend for other libraries and object types via a decorated function. Following this initial conversion, the resulting SDFG contains a state per statement and a nested SDFG per function call.

2.4 Dataflow Optimization

The direct translation to SDFGs creates a control-centric version of the code, which matches the Python semantics but does not contain dataflow beyond a single statement (similarly to compilers' `-O0`). To mitigate this, we run a pass of IR graph transformations that coarsens the dataflow, exposing a true data-centric view of the code (similar to `-O1`). The transformations include redundant copy removals, inlining Nested SDFGs, and others (14 in total), which only modify or remove elements in the graph, such that they cannot be applied indefinitely.

To understand this pass, we showcase one transformation — state fusion — which allows merging two states where the result does not produce data races. For example, the two states containing the assignments below can be merged:

```
tmp0 = alpha * A
tmp1 = tmp0 @ B
```

Internally, the transformation matches a subgraph pattern of two states connected together and compares source and sink Access nodes using symbolic set intersection. If no data dependency constraints are violated, Access nodes are either fused (if they point to the same memory, see Fig. 4) or set side by side, creating multiple connected components that can run in parallel.

All transformations in the DaCe framework follow the same infrastructure, either matching a subgraph pattern or allowing the user to choose an arbitrary subgraph [13]. While the dataflow coarsening pass happens automatically as part of our proposed toolbox, one can also apply transformations manually and separately, without changing the original Python source code (we color such "performance engineering codes" in cyan):

```
sdfg = gemm.to_sdfg()
sdfg.apply(StateFusion)
```

2.5 Python Restrictions

Some features available in Python are incompatible with our definition of high performance Python, and are discussed below. This does not exclude programs using the full feature set of Python from analysis, but calls to functions containing unsupported features will not benefit from our optimization. The restricted features are:

(1) Python containers (lists, sets, dictionaries, etc.) other than NumPy arrays as arguments, as they are represented by linked lists and difficult to analyze from a dataflow view. Note that this does not preclude internal containers (which can be analyzed) and list comprehensions.

(2) Dynamic typing: fixed `structs` are allowed in SDFGs, so fields can be transformed and their class methods into functions decorated with the `dace.program` decorator. However, dynamic changes to classes or dynamic types are unsupported as their structure cannot be statically derived.

(3) Control-dependent variable state (i.e., no scoping rules), e.g., the following valid Python:

```
x = ...
if x > 5:
    y = np.ndarray([5, 6], dtype=np.float32)
# use y (will raise exception if x <= 5)
```

(4) Recursion. This is a limitation of the data-centric programming model [13], as recursion is a control-centric concept that is not portable across platforms.

After performing the full translation and coarsening, the resulting `gemm` kernel can be seen in Figure 5 (left-hand side). This data-centric representation can now use the SDFG IR capabilities to further optimize and map the original Python code to different architectures and distributed systems.

3 PORTABILITY AND PERFORMANCE

The translated data-centric Python programs can now be optimized for performance on different hardware architectures. In this section, we propose a novel set of data-centric passes to auto-optimize and specialize SDFGs to run at state-of-the-art performance on CPUs, GPUs, and FPGAs, all from the same source IR. The programming portability is very high since our approach starts from the same Python codes parsed to the same SDFGs. Although the final optimized IRs may differ, the process can be automatic, and the user needs only to select the architecture to specialize for. In our evaluation, we only discuss results produced in an **automated** fashion.
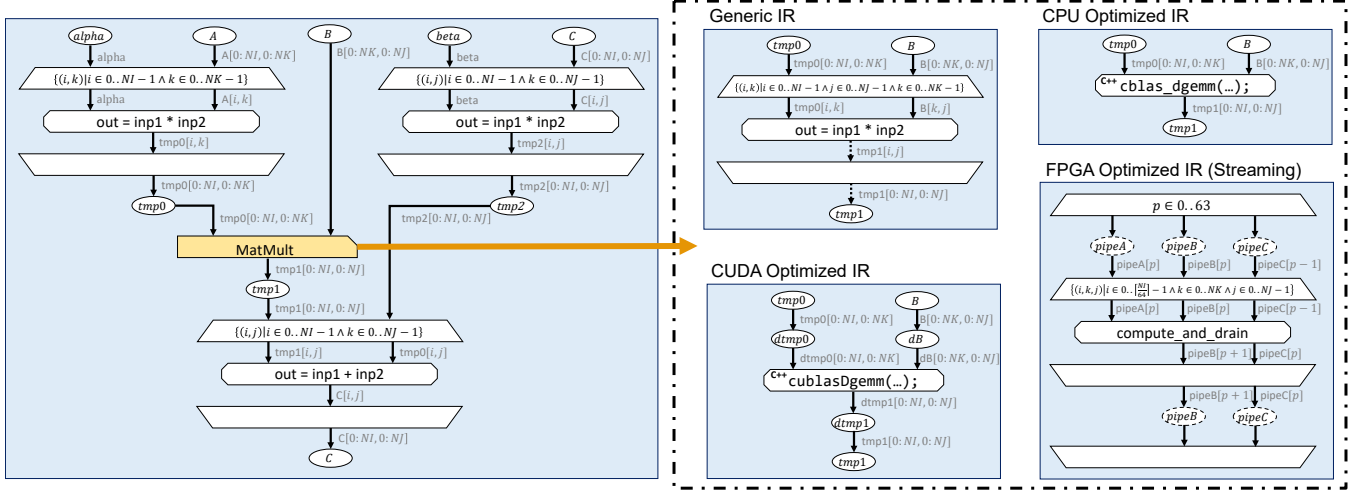
Figure 5: Dataflow-coarsened GEMM SDFG and specializations for different architectures.

3.1 Automatic Optimization Heuristics

As mentioned above, DaCe provides a user-extensible set of graph transformations. Yet, the framework does not perform them automatically [13], to endow performance engineers with fine-grained control and promote separation of concerns. Furthermore, DaCe includes tools and graphical interfaces to assist users with manual optimization without the explicit need for an expert. For productivity purposes, however, we believe that prototyping fast data-centric Python programs *should* be possible with minor code modifications.

By observing the common pitfalls in generated code from SDFGs vs. what a performance engineer would write, we propose a set of transformation heuristics for SDFGs that yield reasonable performance in most cases (-O3 compiler equivalent). This pass can be performed automatically (configurable) or using the following decorator:

```python
@dace.program(auto_optimize=True, device=...)
```

where `device` can be DeviceType.{CPU,GPU,FPGA}.

Our auto-optimizer performs the following passes in order:

(1) **Map scope cleanup**: Remove "degenerate" maps of size 1, repeatedly apply the *LoopToMap* transformation (Section 2.2), and collapse nested maps together to form multidimensional maps. The latter also increases the parallelism of GPU kernels as a by-product.

(2) **Greedy subgraph fusion**: Collect all the maps in each state, fusing together the largest contiguous subgraphs that share the same (or permuted) iteration space or the largest subset thereof (e.g., fusing the common three dimensions out of four). We use symbolic set checks on memlets to ensure that the data consumed is a subset of the data produced.

(3) **Tile WCR maps**: Tile (configurable size) parallel maps with write-conflicts that result in atomics, in order to drastically reduce such operations.

(4) **Transient allocation mitigation**: Move constant-sized and small arrays to the stack, and make temporary data containers persistent (i.e., allocated upon SDFG initialization) if their

size only depends on input parameters. This nearly eliminates dynamic memory allocation overhead.

Beyond the above general-purpose heuristics, we apply more transformations depending on the chosen device: For CPUs, we try to increase parallelism by introducing the OpenMP `collapse` clause. For GPU and FPGA, we perform the `{GPU,FPGA}TransformSDFG` automatic transformations [13], which introduce copies to/from the accelerator and convert maps to accelerated procedures.

On the FPGA, we perform a few further transformations that diverge from the "traditional" fused codes in HPC: we create separate connected components (regions on the circuit) to stream off-chip (DRAM) memory in bursts to the program. Between computations, we try to modify the graph's structure to be composed of separate pipelined units that stream memory through FIFO queue Access nodes (we call this transformation `StreamingComposition`). This also enables further transformations to the graph to create systolic arrays during hardware specialization.

From this point, the only remaining step to lower the SDFG is to specialize the Library Nodes to their respective fastest implementations based on the target platform.

3.2 Library Specialization

Library Nodes, such as the `MatMul` operation in `gemm`, can be expanded to a wide variety of implementations. An *expansion* is defined similarly to a transformation, as a replacement subgraph to a single node, and can specialize its behavior.

We demonstrate specializations of matrix multiplication in Fig. 5. In particular, one can expand a Library Node into a C++ tasklet, calling an external function from MKL or CUBLAS (CPU, CUDA optimized IR in the figure); into an optimized subgraph (e.g., FPGA optimized IR, here using the Streamed expansion); or into a "native" SDFG subgraph (Generic IR).

In the automatic heuristics, we employ a priority list of implementations per platform, starting from the fast library calls, through optimized versions, and if all fail to expand (e.g., if the multiplication occurs within a kernel), we expand to the "native" SDFG. This
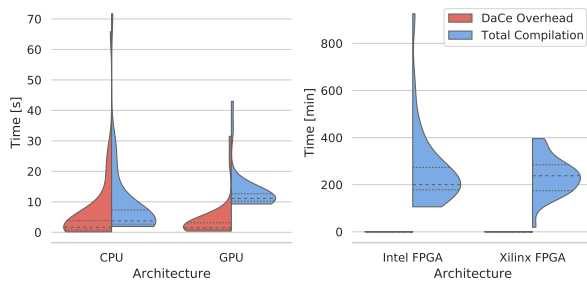
Figure 6: Distributions of DaCe's total compilation times.

yields an SDFG that can both be further optimized by performance engineers, and execute at reasonable performance out-of-the-box.

Users can employ the DaCe API to create their own libraries and Library Nodes [25]. This process is comparable to creating bindings.

3.3 Ahead-of-Time Compilation

DaCe provides extensive support for AOT compilation, due to the risk of overheads JIT compilation introduces in an HPC environment. Namely, DaCe provides the capability of AOT compilation if the decorated function's arguments are type-annotated as described in Section 2.2. A decorated function or an SDFG can be directly compiled to a shared library through a Python script. Alternatively, the user can employ the *sdfgcc* command-line tool to compile SDFGs.

The compilation process utilizes specialized backends to generate optimized code for each supported architecture [13]. For example, C++ code is generated for CPUs, while the Nvidia GPU backend outputs C++ and CUDA programs. For FPGAs we utilize Vitis HLS and OpenCL for Xilinx and Intel architectures, respectively.

In Fig. 6 we present the distributions of DaCe's total compilation times on CPU, GPU, and FPGA for the set of benchmarks presented in Section 3.4. This time includes parsing the Python code, auto-optimizing and compiling with GCC, NVCC, Intel OpenCL SDK, or the Xilinx Vitis compiler. 90% of the CPU and GPU codes compile in less than 15s, while there is a single outlier above one minute. On FPGA architectures, synthesis, placement and routing typically takes hours, rendering DaCe's overhead negligible.

3.4 Evaluation

In the following, we show results for single node shared memory parallel programs created using data-centric Python for CPU, GPU and FPGA, and compare these with other frameworks: NumPy over the CPython interpreter, Numba, Pythran, and CuPy. We collect a set of existing Python codes from different scientific and HPC domains [3, 5, 8, 9, 15, 20, 37, 41, 49, 51, 60, 67, 70–72, 75], as well as a NumPy version of Polybench [63] ported from the C benchmark. In this adaptation, we strive to express the algorithms of the original benchmark in a way that is natural to a Python programmer. E.g., in gemm, 2mm, and 3mm, the matrix-matrix product is implemented with @, Python's dedicated operator for matrix multiplication [29]. All the data used are double-precision floating point numbers or 64-bit integers for CPU and GPU, while the FPGA tests use single-precision floating point numbers and 32-bit integers.

3.4.1 Experimental Setup. The CPU and GPU evaluations are performed on a machine running CentOS 8, with 1.5 TB of main memory, two Intel Xeon Gold 6130 CPUs (2x16 cores), and an NVIDIA V100 GPU (CUDA version 11.1) with 32 GB of RAM. We use CPython 3.8.5 as part of an Anaconda 3 environment. We test NumPy 1.19.2 with Intel MKL support, Numba 0.51.2 with Intel SVML support, the latest Pythran version from their GitHub repository [33] (commit ID 09349c5), and CuPy 8.3.0. For all frameworks that need a separate backend compiler, we use GCC 10.2.0, with all the performance flags suggested by the developers. To put high-performance Python into the perspective of low-level C implementations, we also compare the applications adapted from Polybench with the original Polybench/C [63] benchmark, compiled with GCC and the Intel C Compiler with automatic parallelization enabled (`icc -O3 -march=native -mtune=native -parallel`).

We evaluate FPGA performance on two different boards from either major FPGA vendor; a Bittware 520N accelerator with an Intel Stratix 10 2800 GX FPGA and the Xilinx Alveo U250 accelerator board. Intel FPGA kernels are built with the Intel OpenCL SDK for FPGA and Quartus 20.3 targeting the `p520_max_sg280h` shell, and Xilinx kernels are built with the Vitis 2020.2 compiler targeting the `xilinx_u250_xdma_201830_2` shell.

For the DaCe Python versions, we annotate types and symbolic shapes on the decorated functions to enable AOT compilation and work with FPGAs. We **do not** annotate loops as `dace.maps` and keep them in their original form, leaving parallelization for the automatic heuristics (Section 3.1).

We compare the performance of the different frameworks and compilers using runtime as our primary metric of execution. Unless otherwise mentioned, we run each benchmark ten times and report the median runtime and 95% nonparametric confidence interval (CI) [39].

3.4.2 Benchmarking Results. CPU results are presented in Fig. 7. The right-most column contains NumPy's execution runtime for each of the benchmarks annotated on the y-axis. Each of the other columns contains the speedup (green tint and upward arrow) or slowdown (red tint and downward arrow) of execution compared to NumPy for each of the competing Python frameworks and Polybench C versions (compiled with ICC or GCC). Furthermore, we compute the 95% CI using bootstrapping [27] and annotate its size (as superscript in brackets) as a percentage of the median; values less than 1% are omitted (Fig. 7 uses vector graphics, and readers can zoom in with a PDF viewer if any values are not readable due to their size). The upper part of the chart aggregates the benchmarks using the geometric mean of the speedups over NumPy. Numba and Pythran have fallback modes for Python code that fails to parse. In such cases, we measure the runtime of the fallback mode.

The figure shows an overall improvement in DaCe Python's performance, both over the Python compilers and the optimizing C compilers. Specifically, the subgraph fusion transformation capabilities surpass those of Numba. This is especially apparent in stencils, where the difference can be in orders of magnitude. In applications such as crc16, all compiled implementations successfully eliminate interpreter overhead. Shorter kernels benefit from the C versions due to runtime (and timing) overhead mitigation. It also appears that with control-flow heavy codes (e.g., nussinov), simple C code can be better optimized by GCC and ICC over generated code. It is worth noting that on some applications, NumPy is faster than the C

Benchmarks	DaCe	GCC	ICC	Numba	Pythran	NumPy
Total	↑10.6	↑1.1	↑4.5	↑4.3	↑2.7	
azimhist	↑5.2	no C version	no C version	↑3.3[1]	↑3.1[1]	22.88 ms
vadv	↑29.1[15]	no C version	no C version	↑6.2[1]	↓1.1	2.67 s
sthamfft	↑1.2	no C version	no C version	↑2.2[3]	1.0[5]	0.42 s[5]
sselfeng	↑6.0	no C version	no C version	↑12.8[38]	1.0	3.74 s
spmv	↑32.7[12]	no C version	no C version	↑26.1	↑153	0.55 s[1]
resnet	↑24.7	no C version	no C version	↑3.8[2]	1.0[1]	3.05 s[1]
npgofast	↑14.8[1]	no C version	no C version	↑8.5[3]	1.0	6.51 s
nbody	↑11.0[2]	no C version	no C version	↑4.4	1.0	1.31 s
mlp	↑1.0	no C version	no C version	↓1.0[2]	1.0	19.72 ms
mandel2	↑1.9[5]	no C version	no C version	↑1.1	1.0	0.75 s
mandel1	↑2.9	no C version	no C version	↑6.6[3]	↓1.1	1.36 s
softmax	↑21.4	no C version	no C version	↑1.0	↓1.2	1.57 s
hdiff	↑39.3	no C version	no C version	↑2.9[2]	↑1.7	0.48 s
crc16	↑948[1]	no C version	no C version	↑970	↑932	9.27 s[2]
conv2d	↑29.5	no C version	no C version	↑17.5[5]	↑3.4	21.08 s
coninteg	↑1.5	no C version	no C version	↑1.1[3]	1.0[14]	0.9 s[14]
clipping	↑33.1[14]	no C version	no C version	↑14.1[3]	↑2.6	10.64 s
cholesky2	↑1.7	no C version	no C version	↓1.1	1.0[4]	64.01 ms[4]
chanflow	↑6.2	no C version	no C version	↑1.4	↑1.4	5.64 s
cavtflow	↑3.0	no C version	no C version	↑1.2	↑1.1	3.78 s[3]
azimnaiv	↑17.0[4]	no C version	no C version	↓1.0	↓3.6	1.15 s
lenet	↑141[2]	no C version	no C version	↑10.8[6]	↑2.8	3.66 s
gramschm	↑9.1[2]	↑9.3	↑7.2[54]	↑3.0	↑8.1	0.15 s
heat3d	↑454[42]	↑24.0[9]	↑179[7]	↑50.1	↑2.3	50.35 s
jacobi1d	↑11.4[1]	↑3.7[8]	↑3.9[13]	↓1.2	↑3.1	0.45 s
jacobi2d	↑56.2	↑7.1[26]	↑58.6[7]	↑18.2	↑21.8	174.61 s
lu	↑3.6	↑2.9	↑4.5	↑2.2	↑4.8	13.02 s
ludcmp	↑3.7	↑5.4	↑5.4	↑2.3	↑4.9	13.7 s[1]
syr2k	↑10.4	↑10.5	↑287[14]	↑5.3	↑5.9	13.94 s
nussinov	↑612	↑1.1k	↑909[18]	↑420	↑871	20.37 s[2]
seidel2d	↑185	↑158	↑78.1[2]	↑95.8	↑141	15.87 s[1]
symm	↑4.2	↑6.4	↑74.9[19]	↑11.5	↑4.2	9.88 s
syrk	↑10.0	↑9.3	↑233[26]	↑3.0	↑5.5	6.58 s
gesummv	↑8.6[2]	↑1.8[6]	↑1.8[3]	↑7.5[5]	↑1.0	0.77 s
mvt	↑1.0	↓52.6[23]	↑13.9[4]	↑1.0[1]	↓1.0[1]	45.22 ms[1]
gemver	↑19.1[13]	↑1.5[5]	↑2.5[1]	↑2.1[3]	↑1.3[5]	0.79 s[5]
atax	↓1.2[7]	↓12.0[2]	↑6.2[6]	↑1.0[5]	↓1.1[8]	73.37 ms[5]
floydwar	↑116[2]	↑3.9[5]	↑2.0	↑57.3	↑7.3	84.0 s
fdtd_2d	↑170[98]	↑3.7[11]	↑141.3[24]	↑4.1	↑1.3	7.4 s
durbin	↑2.2	↑5.0[1]	↑5.9[4]	↓1.1[2]	↑3.2	0.65 s
doitgen	↑39.7[46]	↓11.8	↓2.3[3]	↑1.1	1.0[3]	0.47 s[3]
deriche	↑17.3	↑2.7[6]	↑55.1[9]	↑1.5	↑1.6	2.83 s
covarian	↑1.6[16]	↓21.4	↑5.0[6]	↑1.3[2]	↓20.4[3]	80.11 ms
correlat	↑1.6	↓27.8[2]	↓6.6[4]	↑1.1[9]	1.0[19]	61.55 ms[19]
cholesky	↑12.1	↑3.7	↑3.4	↑10.3	↑15.2	7.05 s
bicg	↓1.1[8]	↓17.2[2]	↓17.5[1]	↑1.1[7]	↓1.1[8]	75.92 ms[9]
trisolv	↑2.3[14]	↓1.6[7]	↓1.8[4]	↑1.9[23]	↑1.3	0.13 s[14]
adi	↑16.5	↑9.1	↑13.7[6]	↑8.2	↑9.4	0.9 s
3mm	↑1.7[3]	↓1.7k[11]	↓1.1[2]	↑1.6[6]	↑1.2[33]	0.46 s[21]
2mm	↑2.4[2]	↓942[22]	↑17.1	↑1.5[10]	↑1.3[10]	0.42 s[12]
gemm	↑2.3[5]	↓133[22]	↓4.8[11]	↑1.4[7]	↑1.4	78.72 ms
trmm	↑36.2	↑4.1	↑50.9[18]	↑15.3[4]	↑4.2	4.12 s[1]

Figure 7: CPU runtime and speedup over NumPy.

Figure 8: CuPy and DaCe GPU runtime (lower is better).

versions: this is because of the performance benefits of vectorized NumPy code compared with explicit, sequential loops. An exceptional case is 3mm, which consists of three matrix multiplications. ICC pattern matches the matrix-matrix product and links to MKL, achieving similar performance to the Python frameworks. On the other hand, GCC does not compile the unoptimized C code to an executable that uses MKL, leading to lower performance.

Fig. 8 presents the runtime of applications that were successfully transformed to run on GPU. As with CPU, auto-optimizing DaCe consistently outperforms or is equivalent to CuPy, 3.75x (geomean) faster. The auto-optimization passes contribute to these results, mainly attri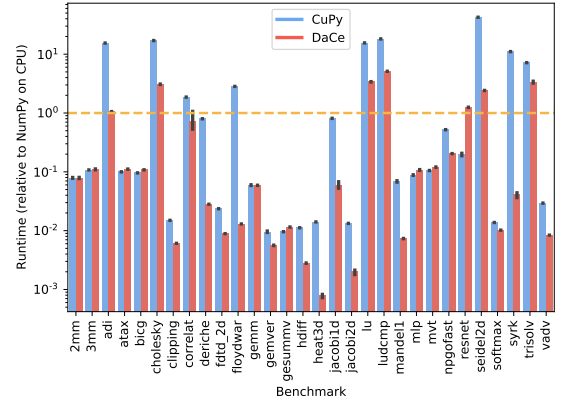buting to subgraph fusion and avoiding intermediate allocations on shorter applications. Due to redundant copy removal and view semantics being native to the SDFG, we see a particular improvement on stencils, e.g., heat3d. Although CuPy-optimized code could potentially employ similar transformations [64], as far as we know, this cannot be performed out-of-the-box. The user must explicitly define element-wise or reduction based kernels, significantly changing the code. There is one instance where CuPy outperforms DaCe — resnet. This is due to a suboptimal vectorized representation of convolution in the Python source code, which translates to a loop of summations. In our generated code, this automatically results in many unnecessary atomic operations, even if tiled. The issue can be easily mitigated with further manual transformations (changing the maps' schedules) after the fact.

FPGA results can be seen in Fig. 9, where there is no comparison point as no other framework compiles high-performance Python directly. Although both platforms use different languages and features (e.g., accumulators), applications can be synthesized for both from the same annotated Python code. There is a noticeable difference in performance, especially on stencil-like applications, likely resulting from Intel FPGA's compiler toolchain superior stencil pattern detection. However, this can also be mitigated with subsequent manual transformations on the SDFG or augmenting the automatic heuristics decision-making process to transform stencils explicitly. Library Node expansions take device-specific features into account. For example, when there are accumulations (e.g., GEMV), we take advantage of hardened 32-bit floating-point accumulation on Intel FPGAs, which allows single-precision numbers to be directly summed into an output register. On Xilinx FPGAs, and, in general, for 64-bit floating-point accumulation, such native support does not exist. Therefore, we perform accumulation interleaving [24] across multiple registers to avoid loop-carried dependencies.

3.4.3 Discussion. While DaCe already outperforms existing libraries, this is not the end of the optimization process. Instead, the generated SDFGs can be *starting* points for optimization by performance engineers. The provided transformations and API can be used to eliminate sources of slowdown in the above applications or extended to use new, domain-specific transformations [25, 40]. Furthermore, the applications can also be adapted to distributed

memory environments, where the productivity and performance benefits can be even greater.

4 SCALABLE DISTRIBUTED PYTHON

The data-centric representation of Python programs can serve as a starting point for creating distributed versions. These distributed SDFGs abandon the global view of the data movement in favor of a local one. Like Message Passing, the flow of data in distributed memory is explicitly defined through Library Nodes. This approach allows for fine-grained control of the communication scheme and better mapping of SDFGs to code using optimized communication libraries.

In this Section, We show how to design transformations that specialize parallel map scopes to support distributed memory systems. We then show how to optimize such distributed data-centric programs. Finally, we show how developers can take control of the distribution entirely by expanding the original Python code with distributed communication while still allowing data-centric optimization to occur.

4.1 Transforming for Scale

Leveraging the data-centric representation, we can create transformations that convert specific shared-memory parallel kernels into distributed memory. The advantage of this approach is that once such a transformation is available, we can apply it to any subgraph in any SDFG that matches the same pattern. Furthermore, transformations can be compounded, building on each other to achieve complex results. We focus again on the gemm kernel for illustrating the transformations. As we shall show, the transformations extend beyond and automatically distribute other kernels as well.

Distributing global view element-wise operations. We distribute these by following a scatter-gather pattern, broadcasting (scalars) or scattering (arrays) input data containers from the root rank to the machine nodes, performing local computation, and gathering or reducing the outputs. By the nature of element-wise operations, careful selection of the array distribution parameters is not always necessary. The primary constraint is that each rank receives matching subsets of data, allowing it to perform local computation without further communication. Therefore, the most efficient distribution for contiguous arrays is to treat them as uni-dimensional and scatter them with MPI_Scatter (1-D block distribution). However, in cases where the result of an element-wise operation is consumed by, e.g., a matrix-matrix product or a stencil computation, it is beneficial to preserve the dimensionality of the arrays. For this reason, the transformation has optional parameters for the block sizes per dimension, leading to *block* or *block-cyclic* distributions. We offer implementations that use PBLAS [55] methods, such as p?gemr2d and p?tran, and MPI derived data types, which have previously demonstrated performance benefits [68]. Applying the above transformation with *block* distributions on the operation tmp0 = alpha * A transforms the SDFG subgraph as shown in Fig. 10. We emphasize that these transformations are applied to an SDFG **without** changing the data-centric program code:
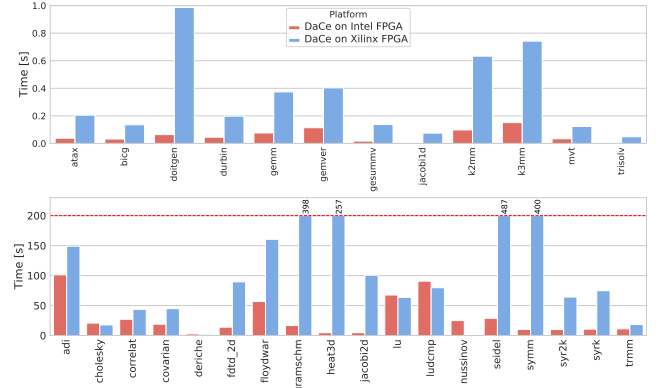
```
sdfg.apply(DistributeElementWiseArrayOp)
```


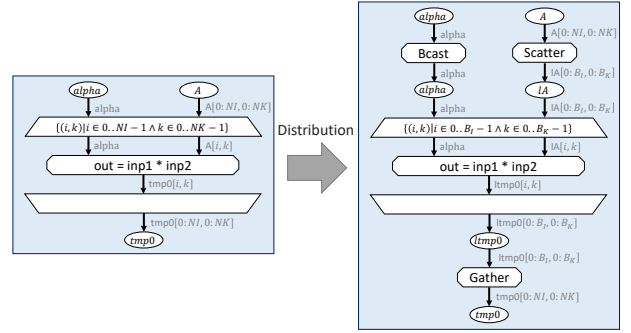**Figure 9: FPGA runtime, Large instance, single precision.**

Figure 10: Distribution of element-wise array operation.

Distributing Library Nodes. We also create expansions for Library Nodes to distributed SDFG subgraphs. For example, the matrix-matrix and matrix-vector products expand to the aforementioned PBLAS library calls, along with the corresponding distribution of inputs and gathering outputs. Using PBLAS requires the definition of a process grid. The DaCe PBLAS library environment handles this automatically using BLACS [54]. The grid's parameters are free symbols that can be chosen by the user or take default values.

4.2 Optimizing Communication

Creating distributed versions of the operations separately from each other will perform correctly but poorly on real applications. Thus, we can use the data-centric aspect of the SDFG IR, which can track access sets through memlets to remove such communication bottlenecks automatically. Doing so separately from the distribution transformations allows users to write more pattern-matching distribution transformations without worrying about inter-operation communication, on the one hand; and on the other hand, allows the system to find such optimizations in any input code (e.g., if manually-written with communication redundancy).

One example of such a transformation is redundant gather-scatter removal. Consider the SDFG representation of gemm, shown in Fig. 5. We distribute all three element-wise array operations and we expand the MatMult node to a call to pdgemm. Due to the produced scatter and gather operations, the outputs tmp1 of pdgemm and

tmp2 of beta * C will be connected to the last element-wise array operation tmp1 + tmp2 as shown in the following **pseudo-code**:

```
pdgemm(..., ltmp1, ...)
tmp1 = gather(ltmp1)
ltmp1 = scatter(tmp1)
ltmp2 = beta * lC
tmp2 = gather(ltmp2)
ltmp2 = scatter(tmp2)
lC = ltmp1 + ltmp2
...
```

The above sequence of operations yields redundant communication on tmp1 and tmp2 and can therefore be omitted (the transformation for tmp1 is shown in Fig. 11). By following the data movement and inspecting other Access nodes in the state, data dependencies of the global array can be inferred. Furthermore, since we know the Scatter and Gather node semantics, we can check whether the data distributions match. We note that users can use the DaCe API to define transformations to, e.g., optimize the re-distribution of data when the distributions do not match. If the distributions are 2D block-cyclic, such a transformation could, among other solutions, utilize PBLAS and a p?gemr2d Library Node to efficiently bypass the Scatter and Gather operations.

Combing the above transformations, the shared-memory gemm program from Section 2.3 can be converted to distributed-memory as follows, again without altering the code of the dace-centric Python program (type annotations omitted for brevity):

```
@dace.program
def gemm(alpha, beta, C, A, B):
    C[:] = alpha * A @ B + beta * C
```

```
dist_sdfg = gemm.to_sdfg()
dist_sdfg.apply(DistributeElementWiseArrayOp)
dist_sdfg.expand_library_nodes('PBLAS')
dist_sdfg.apply(RemoveRedundantComm)
```

4.3 Assuming Direct Control via Local Views

The implicit, global view approach works well for a plethora of Python programs that make heavy use of high-level array operations. However, as highly-tuned HPC applications often use specific partitioning schemes, our data-centric toolbox also provides explicit control via Python annotations. As opposed to the existing tools that manage communication implicitly, the aim of the interface is to use *(num)pythonic* concepts to retain productivity while maximizing performance. E.g., the jacobi_2d stencil below would yield unnecessary Scatter and Gather collectives at every timestep:

```
@dace.program
def jacobi_2d(TSTEPS: dace.int32, A: dace.float64[N,N],
              B: dace.float64[N,N]):
    for t in range(1, TSTEPS):
        B[1:-1,1:-1] = 0.2 * (A[1:-1,1:-1] + A[1:-1,:-2]
            + A[1:-1,2:] + A[2:,1:-1] + A[:-2,1:-1])
        A[1:-1,1:-1] = 0.2 * (B[1:-1,1:-1] + B[1:-1,:-2]
            + B[1:-1,2:] + B[2:,1:-1] + B[:-2,1:-1])
```

For this reason, our data-centric approach allows the user to express arbitrary communication patterns by integrating explicit communication directly into Python. The above shared-memory data-centric Python program can be *modified* to be distributed:
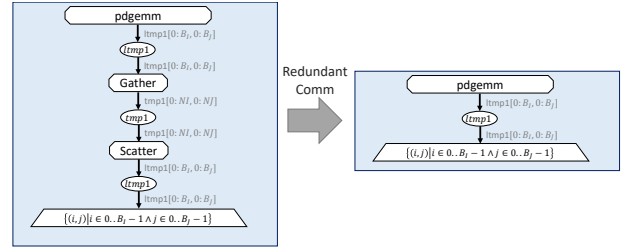

Figure 11: Redundant communication elimination.

```
@dace.program
def half_step(inpbuf: dace.float64[lNx+2, lNy+2],
              outbuf: dace.float64[lNx+2, lNy+2]):
    req = np.empty((8,), dtype=MPI_Request)
    dace.comm.Isend(inpbuf[1, 1:-1], nn, 0, req[0])
    # ...
    dace.comm.Irecv(inpbuf[1:-1, -1], ne, 2, req[7])
    dace.comm.Waitall(req)
    outbuf[1+noff:-1-soff, 1+woff:-1-eoff] = 0.2*(
        inpbuf[1+noff:-1-soff, 1+woff:-1-eoff] +
        # ...
        inpbuf[noff:-2-soff, 1+woff:-1-eoff])

@dace.program
def j2d_dist(TSTEPS: dace.int32, A: dace.float64[N, N],
             B: dace.float64[N, N]):
    lA = np.zeros((lNx+2, lNy+2), dtype=A.dtype)
    lB = np.zeros((lNx+2, lNy+2), dtype=B.dtype)
    lA[1:-1, 1:-1] = dace.comm.BlockScatter(A)
    lB[1:-1, 1:-1] = dace.comm.BlockScatter(B)
    for t in range(1, TSTEPS):
        half_step(lA, lB)
        half_step(lB, lA)
    A[:] = dace.comm.BlockGather(lA[1:-1, 1:-1])
    B[:] = dace.comm.BlockGather(lB[1:-1, 1:-1])
```

In the above program, we distribute the arrays A and B into 2D blocks at the beginning. The local views lA, lB are then computed and communicated via explicit halo exchange, using Isend, Irecv, and Waitall MPI calls in every time-step.

While this approach is similar to the *mpi4py* bindings, there are two distinct advantages to the data-centric approach with an explicit local view. First, the MPI calls are integrated into the program's dataflow with Library Nodes, enabling the above transformations and other automatic code generation features such as overlapping. Second, explicit *Isend* and *Irecv* calls communicate strided data using the MPI vector datatype, avoiding extraneous copies. The latter is also an example of using symbolic information on the graph to assert that high performance is attained — our MPI derived data type creation code, which is created once for each data type, relies on the symbol values not changing over the run. E.g., the initialization of the inpbuf[1:-1, 1] data type:

```
MPI_Datatype ntype;
MPI_Type_vector(lNx, 1, lNy+2, MPI_DOUBLE, &ntype);
MPI_Type_commit(&ntype);
```

would raise a performance warning in DaCe Python if the sizes may change at runtime. This avoids potential mistakes that even experienced performance engineers can make in large HPC codes.

4.4 Evaluation

To measure the performance of distributed data-centric programs, we conduct scaling experiments on the multi-core partition of
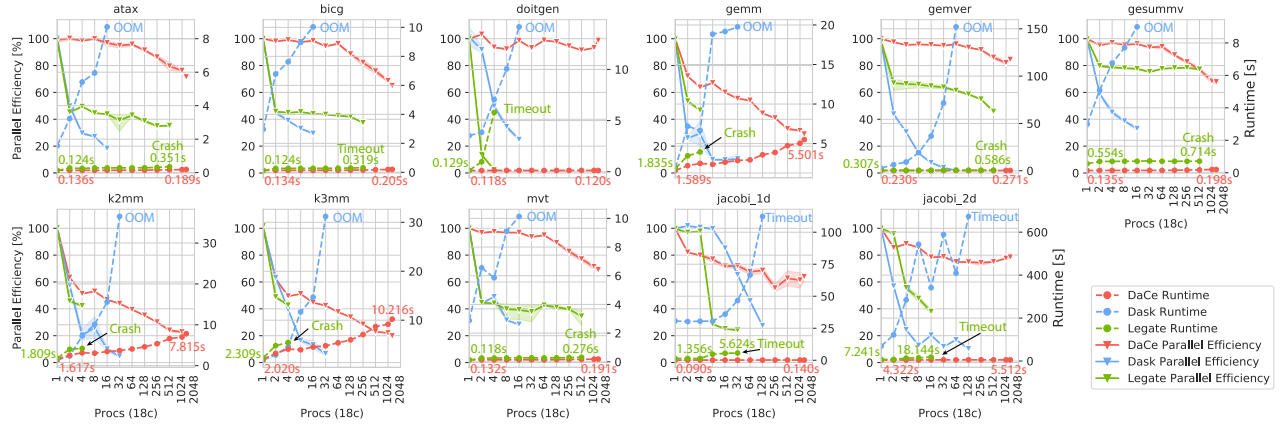
Figure 12: Distributed runtime (dashed lines) and scaling efficiency (solid lines) on 23,328 cores of Piz Daint.

Benchmark	F	Initial Problem Size	S.F.
atax (M, N)	DaCe/Legate Dask	20000, 25000 10000, 12500	all $\sqrt{S}$
bicg (M, N)	DaCe/Legate Dask	25000, 20000 12500, 10000	all $\sqrt{S}$
doitgen (NR, NQ, NP)	DaCe/Legate Dask	128, 512, 512 128, 512, 512	$(S, -, -)$
gemm (NI, NJ, NK)	DaCe/Legate Dask	8000, 9200, 5200 4000, 4600, 2600	all $\sqrt[3]{S}$
gemver (N)	DaCe/Legate Dask	10000 5000	$\sqrt{S}$
gesummv (N)	DaCe/Legate Dask	22400 11400	$\sqrt{S}$
jacobi_1d (T, N)	DaCe/Legate Dask	1000, 24000 1000, 24000	$(-, S)$
jacobi_2d (T, N)	DaCe/Legate Dask	1000, 1300 1000, 1300	$(-, \sqrt{S})$
k2mm (NI, NJ, NK, NM)	DaCe/Legate Dask	6400, 7200, 4400, 4800 3200, 3600, 2200, 2400	all $\sqrt[3]{S}$
k3mm (NI, NJ, NK, NL, NM)	DaCe/Legate Dask	6400, 7200, 4000, 4400, 4800 3200, 3600, 2000, 2200, 2400	all $\sqrt[3]{S}$
mvt (N)	DaCe/Legate Dask	22000 11000	$\sqrt{S}$

Table 2: Distributed benchmarks, initial problem sizes for the different frameworks (F), and scaling factors (S.F.) as a function of the number of processes S.

the Piz Daint supercomputer. Each node has two 18-core Intel E5-2698v3 CPUs and 64GB of memory. The nodes are connected through a Cray Aries network using a Dragonfly topology. We benchmark a subset of the Polybench kernels from Section 3.4, which could be automatically transformed to use distributed memory (Section 4.1); adi, bicg, doitgen, gemm, gemver, gesummv, k2mm, k3mm, mvt, jacobi_1d, and jacobi_2d. We compare this work with Dask [23] v2.31 and Legate [10] (commit ID febd3bf [57]), two state-of-the-art distributed tasking Python frameworks, in weak scaling from 1 to 1,296 processes (648 nodes). Dask Array [4] scales a variety of NumPy workflows, including element-wise array operations, reductions, matrix-matrix products, and linear algebra solvers, among others. Legate is providing a drop-in replacement for the NumPy API to accelerate and distribute Python codes.

We select initial problem sizes that fit typical HPC workloads without having excessive runtime. The kernels' scaling factors and the initial problem sizes for each framework are presented in Tab. 2. For kernels with computational complexity ranging from $O(n)$ to $O(n^2)$, we target a runtime in the order of 100ms. For kernels with higher complexity, we target a runtime in the order of 1s. We note that with the problem sizes selected for benchmarking data-centric Python and Legate, Dask either runs out of memory or exhibits unstable performance. Thus, we halved the problem sizes for Dask but still encountered out-of-memory errors at larger node counts. Furthermore, several issues rendered testing Legate at scale difficult. With the assistance of the developers, many of those problems were solved; however, others remained. We could only run each benchmark up to a fraction of the total nodes available, either due to runtime errors or because a single execution did not finish within 10 minutes of allocated time. We annotate Fig. 12 with these errors.

We ignore the time needed for initializing and distributing data and only measure the main computation and communication time. We run all frameworks using default parameters where possible, i.e., "auto" as chunk size in Dask and block distributions (not block-cyclic, which would allow the user to fine-tune the block-sizes) on a 2D process grid for DaCe. Furthermore, we spawn one process per socket (2 processes per node) and 18 threads per process (equal to the number of physical cores in a socket). Legate is executed with parameters suggested by the developers: two NUMA domains per node, 28GB of memory and one CPU with 16 threads per domain.

Fig. 12 presents the distributed runtime and scaling efficiency of DaCe, Dask, and Legate. DaCe exhibits four different efficiency patterns. In doitgen, the workload is distributed in an embarrassingly parallel manner, and no communication is needed. Therefore, the efficiency is close to perfect. The kernels atax, bicg, gemver, gesummv, and mvt compute matrix-vector products and scale very well until 64 processes, where the drop in efficiency becomes more pronounced, but remains above 60% for all data points. The matrix-matrix product kernels, gemm, k2mm, and k3mm, exhibit lower efficiency, which is consistent with the expected behavior of MKL-ScaLAPACK [44]. Finally, the stencil kernels' efficiency (jacobi_1d and jacobi_2d) falls between the last two categories of kernels. On the other hand,

Dask and Legate exhibit a sharp drop in efficiency in almost all kernels immediately from the second process. An exception is the `jacobi_1d` kernel, where Dask has higher efficiency than DaCe up to 16 processes. However, this is possible due to Dask being much slower than DaCe (over 30x on the same problem size), allowing for much higher communication-computation overlap. On BLAS-heavy benchmarks, Legate matches the runtime of DaCe on a single CPU, whereas in others we observe slowdowns of 1.7–15×. In the benchmarks that scale to a large number of nodes (`atax`, `bicg`, `gemver`, `gesummv`, and `mvt`), Legate's efficiency, after the initial drop, remains constant.

DaCe uses MPI for communication and links to the optimized Cray implementation. Legate is built on top of the Legion runtime [11], which employs the GASNet library [16], a networking middleware implementing global-address space. Finally, Dask uses TCP for inter-worker communication. Although the different communication approaches cannot explain all performance discrepancies, there are the most significant factors in the overall picture.

5 PRODUCTIVITY

Python is already a very productive language, especially for domain scientists, due to its rich ecosystem described in Section 1. Data-Centric Python is Python with extensions that themselves are valid Python syntax. For example, the **@dace.program** decorator follows the PEP 318 standard [28]. Therefore, Data-Centric Python essentially inherits the Python language's programming productivity. Since performance is the most important metric in HPC, scientific applications must eventually be lowered to a representation that is amenable to low-level optimizations for the underlying architectures. Traditionally, this translates to writing these applications in C, C++, and Fortran, among other device-specific languages. Therefore, an HPC project must force the domain scientists to sacrifice productivity and work directly on the lower-level languages or maintain two different code-bases. DaCe, and other frameworks that accelerate Python, increase HPC productivity by bridging the code that domain scientists want to write with the code that achieves high performance.

6 RELATED WORK

Approaches similar to our own targeting Python code have already been introduced and compared with in Sections 1, 3, and 4. In this section, we further discuss relevant frameworks, libraries, and approaches towards the three **P**s.

Productivity. The complexity of optimizing applications, combined with the repetitive nature of performance engineering for specific domains, has given rise to a wide variety of Domain-Specific Languages (DSLs) [19, 38, 59, 69] and embedded DSLs, particularly in Python [46, 72]. In the latter category, a notable example is deep learning frameworks, which use Python's various capabilities to construct readable code that performs well. PyTorch [61] uses object-oriented programming to construct deep neural networks as modules, relying on reflection to detect parameters and nonblocking calls for asynchronous execution to avoid interpreter overhead. TensorFlow [2] used Python's weak typing system to

construct graphs from Python functions but has recently transitioned to "eager" execution to improve productivity, making codes more readable and simplifying debugging.

Portability. In the past three decades, compilers have undergone a transition from all-pairs solutions (between source languages and hardware platforms) to funneling through Intermediate Representations (IR), on which they can perform language- and platform-agnostic optimization passes. Although DaCe is currently developed in Python, many other research compilers are based on the LLVM [47] infrastructure and IR. There is an ongoing movement in the compiler community towards Multi-Level IRs [48], in which a multitude of IR *dialects* can retain domain- and platform-specific information, in turn enabling domain-specific optimizations [35]. MLIR performs optimization passes on each dialect to compile programs, followed by lowering passes to subsequent dialects, down to hardware mapping. This feature could be utilized to implement DaCe Library Nodes. The data-centric transformation API is also shared by languages such as Halide [65], which enables users to invoke schedule optimizations separately from program definition.

Performance portability. Aimed at keeping a consistent ratio of performance to peak performance across hardware [73], it is the core premise of several standards [6, 21, 31, 32, 58]. In directive-based frameworks [21, 58], *pragma* statements are added to C/C++ and FORTRAN programs to introduce parallelism, similarly to our proposed annotations. In kernel-based frameworks [6, 31, 32], *kernels* are constructed as functions with a limited interface and offloaded to target devices, such as CPUs, GPUs, or FPGAs. As each platform requires its own set of directives, kernel parameters, or sometimes kernel implementations, programs often contain multiple codes for each target. To resolve such issues, HPC languages such as Chapel [18] and HPF [66] define high-level implicit abstractions used as parallel primitives. Also popular in the HPC world are performance-portable libraries, embedded within C++, notably Kokkos [26], RAJA [50], and Legion [11] (which powers Legate [10]). These allow integrating heterogeneous and distributed systems through task-based abstractions, data dependency analysis (e.g., Legion's logical regions), and common parallel patterns. Such patterns can also be found in NumPy, and the SDFG can be seen as a generalization of these graphs with symbolic data dependencies.

7 CONCLUSION

Discussion. While distributed SDFGs forego the global view of data movement to facilitate the design of custom communication schemes, future work could explore the trade-offs between a more *pythonic* approach to communication and extracting the best performance. Moreover, improvements to existing transformations, e.g., Vectorization, and implementation of new ones could increase the out-of-the-box performance, reducing the need for manual optimization.

We present Data-Centric Python — a high-performance subset of Python with annotations that produces supercomputer-grade HPC codes. Based on the SDFG intermediate representation, we show how a pipeline of static code analysis and dataflow transformations can take input NumPy code, leverage its vectorized nature, and map

it efficiently to CPUs, GPUs, and FPGAs, outperforming current state-of-the-art approaches on each platform by at least 2.4x.

The resulting data-centric programs effectively eliminate the performance pitfalls of Python, including interpreter overheads and lack of dataflow semantics for library calls, the latter being crucial for running at scale. Many even outperform baselines written in C code. A key feature of the data-centric toolbox is giving users explicit control when necessary, rather than making assumptions at the framework level. We evaluate Data-Centric Python on a distributed environment and show that the parallel efficiency remains above 90%, even on hundreds of nodes. These promising results indicate that productive coding with Python can scale *and* map to heterogeneous compute architectures, setting the once-scripting language at the same level as FORTRAN, C, and other HPC giants.

8 ACKNOWLEDGEMENTS

REFERENCES

[1] Standard Performance Evaluation Corporation. 2020. CPU 2017 Metrics. Retrieved 2021-04-09 from https://www.spec.org/cpu2017/Docs/overview.html#metrics
[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.
[3] Anaconda Inc. [n.d.]. A 5 minute guide to Numba. Retrieved 2021-02-04 from https://numba.readthedocs.io/en/stable/user/5minguide.html
[4] Anaconda Inc. [n.d.]. Dask Array. https://docs.dask.org/en/latest/array.html
[5] Anaconda Inc. [n.d.]. Example: Histogram. Retrieved 2021-02-04 from https://numba.pydata.org/numba-examples/examples/density_estimation/histogram/results.html
[6] Ben Ashbaugh, Alexey Bader, James Brodman, Jeff Hammond, Michael Kinsner, John Pennycook, Roland Schulz, and Jason Sewall. 2020. Data Parallel C++: Enhancing SYCL Through Extensions for Productivity and Performance. In *Proceedings of the International Workshop on OpenCL* (Munich, Germany) *(IWOCL '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 2 pages. https://doi.org/10.1145/3388333.3388653
[7] John Aycock and Nigel Horspool. 2000. Simple Generation of Static Single-Assignment Form. In *Compiler Construction*, David A. Watt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 110–125.
[8] M. Baldauf, A. Seifert, J. Förstner, D. Majewski, and M. Raschendorfer. 2011. Operational convective-scale numerical weather prediction with the COSMO model: Description and sensitivities. *Monthly Weather Review, 139:3387–3905* (2011).
[9] Lorena Barba and Gilbert Forsyth. 2019. CFD Python: the 12 steps to Navier-Stokes equations. *Journal of Open Source Education* 2, 16 (2019), 21. https://doi.org/10.21105/jose.00021
[10] 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* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 23, 23 pages. https://doi.org/10.1145/3295500.3356175
[11] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.
[12] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The best of both worlds. *Computing in Science & Engineering* 13, 2 (2011), 31–39.
[13] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*.
[14] T. Ben-Nun, T. Gamblin, D. S. Hollman, H. Krishnan, and C. J. Newburn. 2020. Workflows are the New Applications: Challenges in Performance, Portability, and Productivity. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 57–69. https://doi.org/10.1109/P3HPC51967.2020.00011
[15] Gabriel Bengtsson. [n.d.]. *Development of Stockham Fast Fourier Transform using Data-Centric Parallel Programming*. Ph.D. Dissertation. http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-287731
[16] Dan Bonachea and Paul H. Hargrove. 2018. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18) (Lecture Notes in Computer Science, Vol. 11882)*. Springer International Publishing. https://doi.org/10.25344/S4QP4W Lawrence Berkeley National Laboratory Technical Report (LBNL-2001174).
[17] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax
[18] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (Aug. 2007), 291–312. https://doi.org/10.1177/1094342007078442
[19] Valentin Clement, Sylvaine Ferrachat, Oliver Fuhrer, Xavier Lapillonne, Carlos E. Osuna, Robert Pincus, Jon Rood, and William Sawyer. 2018. The CLAW DSL: Abstractions for Performance Portable Weather and Climate Models. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (Basel, Switzerland) *(PASC '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 10 pages. https://doi.org/10.1145/3218176.3218226
[20] COSMO. 1998. Consortium for Small-scale Modeling. Retrieved 2021-02-04 from http://www.cosmo-model.org
[21] 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
[22] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. 2011. Parallel distributed computing using Python. *Advances in Water Resources* 34, 9 (2011), 1124–1139. https://doi.org/10.1016/j.advwatres.2011.04.013 New Computational Methods and Software Tools.
[23] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling*. https://dask.org
[24] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. 2021. Transformations of High-Level Synthesis Codes for High-Performance Computing. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2021), 1014–1029. https://doi.org/10.1109/TPDS.2020.3039409
[25] J. de Fine Licht, A. Kuster, T. De Matteis, T. Ben-Nun, D. Hofer, and T. Hoefler. 2021. StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 315–326. https://doi.org/10.1109/CGO51591.2021.9370315
[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] Bradley Efron. 1992. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*. Springer, 569–593.
[28] Python Software Foundation. [n.d.]. PEP 318 – Decorators for Functions and Methods. https://www.python.org/dev/peps/pep-0318/
[29] Python Software Foundation. [n.d.]. PEP 465 – A dedicated infix operator for matrix multiplication. https://www.python.org/dev/peps/pep-0465/
[30] GitHub. 2020. The 2020 State of the Octoverse. https://octoverse.github.com/
[31] Khronos Group. 2021. OpenCL. https://www.khronos.org/opencl
[32] Khronos Group. 2021. SYCL. https://www.khronos.org/sycl
[33] Serge Guelton. [n.d.]. Pythran. https://github.com/serge-sans-paille/pythran.
[34] Serge Guelton, Pierrick Brunet, Mehdi Amini, Adrien Merlini, Xavier Corbillon, and Alan Raynaud. 2015. Pythran: Enabling static optimization of scientific python programs. *Computational Science & Discovery* 8, 1 (2015), 014001.

[35] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2020. Domain-Specific Multi-Level IR Rewriting for GPU. arXiv:2005.13014 [cs.PL]

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

[37] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[38] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-level Image Processing Code into Hardware Pipelines. *ACM Trans. Graph.* 33, 4, Article 144 (July 2014), 11 pages. https://doi.org/10.1145/2601097.2601174

[39] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. https://doi.org/10.1145/2807591.2807644

[40] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data Movement Is All You Need: A Case Study on Optimizing Transformers. In *Proceedings of the Fourth Conference on Machine Learning and Systems (MLSys'21)*. mlsys.org.

[41] Jérôme Kieffer and Giannis Ashiotis. 2014. PyFAI: a Python library for high performance azimuthal integration on GPU. In *Proceedings of the 7th European Conference on Python in Science (EuroSciPy 2014)*. arXiv:1412.6367 [astro-ph.IM]

[42] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, Fernando Loizides and Birgit Scmidt (Eds.). IOS Press, Netherlands, 87–90. https://eprints.soton.ac.uk/403913/

[43] Mads RB Kristensen, Simon AF Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. 2013. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. In *Workshop on Python for High Performance and Scientific Computing (PyHPC 2013)*.

[44] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–22.

[45] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

[46] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman. 2016. Devito: Towards a Generic Finite Difference DSL Using Symbolic Python. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. 67–75. https://doi.org/10.1109/PyHPC.2016.013

[47] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[48] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[49] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1, 4 (1989), 541–551. https://doi.org/10.1162/neco.1989.1.4.541 arXiv:https://doi.org/10.1162/neco.1989.1.4.541

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

[51] Philip Mocz. 2020. nbody-python: Create Your Own N-body Simulation (With Python). https://github.com/pmocz/nbody-python.

[52] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 561–577.

[53] Sebastian Nanz and Carlo A. Furia. 2015. A Comparative Study of Programming Languages in Rosetta Code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 778–788.

[54] NetLib. [n.d.]. BLACS. https://www.netlib.org/blacs/

[55] NetLib. [n.d.]. PBLAS. http://www.netlib.org/scalapack/pblas_qref.html

[56] Preferred Networks. 2021. CuPy. https://cupy.dev/

[57] Nvidia. [n.d.]. Legate: High Productivity Performance Computing. Retrieved 2021-05-26 from https://github.com/nv-legate

[58] OpenACC Organization. 2021. OpenACC. https://www.openacc.org/

[59] Carlos Osuna, Tobias Wicky, Fabian Thuering, Torsten Hoefler, and Oliver Fuhrer. 2020. Dawn: a High-level Domain-Specific Language Compiler Toolchain for Weather and Climate Applications. *Supercomputing Frontiers and Innovations* 7, 2 (2020). https://www.superfri.org/superfri/article/view/314

[60] Øystein Sture. [n.d.]. Implementation of crc16 (CRC-16-CCITT) in python. Retrieved 2021-02-04 from https://gist.github.com/oysstu/68072c44c02879a2abf94ef350d1c7c6

[61] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[62] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[63] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012).

[64] Preferred Networks, inc. and Preferred Infrastructure, inc. [n.d.]. CuPy: User-Defined Kernels. https://docs.cupy.dev/en/stable/user_guide/kernel.html

[65] 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* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[66] CORPORATE Rice University. 1993. High Performance Fortran Language Specification. *SIGPLAN Fortran Forum* 12, 4 (Dec. 1993), 1–86. https://doi.org/10.1145/174223.158909

[67] Nicolas P. Rougier. 2016. *rougier/from-python-to-numpy: Version 1.1*. Zenodo. https://doi.org/10.5281/zenodo.225783

[68] Timo Schneider, Robert Gerstenberger, and Torsten Hoefler. 2014. Application-oriented ping-pong benchmarking: how to assess the real communication overheads. *Computing* 96, 4 (01 Apr 2014), 279–292. https://doi.org/10.1007/s00607-013-0330-4

[69] 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* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 81, 12 pages. https://doi.org/10.1145/2807591.2807629

[70] Stefan Behnel, Robert Bradshaw, Dag Sverre Seljebotn, Greg Ewing, William Stein, Gabriel Gellner, et al. [n.d.]. Cython for NumPy users. Retrieved 2021-02-04 from https://cython.readthedocs.io/en/latest/src/userguide/numpy_tutorial.html

[71] Christian Stieger, Aron Szabo, Teutë Bunjaku, and Mathieu Luisier. 2017. Ab-initio quantum transport simulation of self-heating in single-layer 2-D materials. *Journal of Applied Physics* 122, 4 (2017), 045708. https://doi.org/10.1063/1.4990384 arXiv:https://doi.org/10.1063/1.4990384

[72] Swiss National Supercomputing Centre (CSCS). [n.d.]. GT4Py. Retrieved 2021-04-02 from https://github.com/GridTools/gt4py

[73] US Department of Energy. 2016. Definition - Performance Portability. https://performanceportability.org/perfport/definition/.

[74] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 56 – 61. https://doi.org/10.25080/Majora-92bf1922-00a

[75] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Guillermo Indalecio Fernández, Timo Schneider, Mathieu Luisier, and Torsten Hoefler. 2019. A Data-Centric Approach to Extreme-Scale Ab Initio Dissipative Quantum Transport Simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*.