Automatic Parallelization of Python Programs for Distributed Heterogeneous Computing

Jun Shirako[1], Akihiro Hayashi[1], Sri Raj Paul[2],
Alexey Tumanov[1], and Vivek Sarkar[1]

[1] Georgia Institute of Technology, Atlanta, GA 30332, USA
{shirako,ahayashi,atumanov3,vsarkar}@gatech.edu
[2] Intel Corporation, Austin, TX 78746, USA
sriraj.paul@intel.com

Abstract. This paper introduces a new approach to automatic ahead-of-time (AOT) parallelization and optimization of sequential Python programs for execution on distributed heterogeneous platforms. Our approach enables AOT source-to-source transformation of Python programs, driven by the inclusion of type hints for function parameters and return values. These hints can be supplied by the programmer or obtained by dynamic profiler tools; multi-version code generation guarantees the correctness of our AOT transformation in all cases.
Our compilation framework performs automatic parallelization and sophisticated high-level code optimizations for the target distributed heterogeneous hardware platform. It introduces novel extensions to the polyhedral compilation framework that unify user-written loops and implicit loops present in matrix/tensor operators, as well as automated selection of CPU vs. GPU code variants. Finally, output parallelized code generated by our approach is deployed using the Ray runtime for scheduling distributed tasks across multiple heterogeneous nodes in a cluster, thereby enabling both intra-node and inter-node parallelism.
Our empirical evaluation shows significant performance improvements relative to sequential Python in both single-node and multi-node experiments, with a performance improvement of over 20,000× when using 24 nodes and 144 GPUs in the OLCF Summit supercomputer for the Space-Time Adaptive Processing (STAP) radar application.

Keywords: Parallelizing compilers · Python language · Parallel computing · Heterogeneous computing · Distributed computing.

1 Introduction

Multiple simultaneous disruptions are currently under way in both hardware and software, as we consider the implications for future parallel systems. In hardware, extreme heterogeneity has become critical to sustaining cost and performance improvements with the end of Moores Law, but poses significant productivity challenges for developers. In software, the rise of large-scale data science and AI applications is being driven by domain scientists from diverse backgrounds

who demand the programmability that they have come to expect from high-level languages like Python. While this paper focuses on Python as an exemplar of modern high-productivity programming, the approach in this paper is equally applicable to other high-productivity languages such as Julia [3].

A key challenge facing domain scientists is determining how to enable their Python-based applications to use the parallelism inherent in both distributed and heterogeneous computing. A typical workflow for domain scientists is to experiment with new algorithms by starting with smaller datasets and then moving on to larger datasets. A tipping point is reached when there is a need to use intra-node parallelism with multiple cores and accelerators such as GPUs, and another tipping point is reached when the dataset size becomes too large to be processed within a single node.

One approach to dealing with these tipping points is to rely on experienced programmers with a deep "ninja level" expertise in computer architecture and code optimization for accelerators and inter-node communication who use low-level programming languages such as C/C++. However, this approach is a non-starter for many domain scientists due to the complexity and skills required. For example, even though Python bindings for MPI [6] have been available for many years, there has been very little adoption of these bindings by domain scientists. An alternate approach is to augment a high-productivity language with native libraries that include high-performance implementations of commonly used functions, e.g., functions in the NumPy [14] and SciPy [20] libraries for Python. However, fixed library interfaces and implementations do not address the needs of new applications and algorithms. Yet another approach is to develop and use Domain Specific Languages (DSLs); this approach has recently begun showing promise for certain target domains, e.g., PyTorch and TensorFlow for machine learning, Halide for image processing computations, and TACO for tensor kernels. However, the deliberate lack of generality in DSLs poses significant challenges in requiring domain scientists to learn multiple DSLs and to integrate DSL kernels into their overall programming workflow, while also addressing corner cases that may not be supported by any DSLs.

In this paper, we make the case for new advances to enable productivity and programmability of future HPC platforms for domain scientists. The goal of our system, named `AutoMPHC`, is Automation of Massively Parallel and Heterogeneous Computing. It aims to deliver the benefits of distributed heterogeneous hardware platforms to domain scientists without requiring them to undergo any new training. As a first step towards this goal, this paper introduces a novel approach to automatic ahead-of-time (AOT) parallelization and optimization of sequential Python programs for execution on distributed heterogeneous platforms, which supports program multi-versioning for specializing code generation to different input data types and different target processors. The optimized code is deployed using the Python-based Ray runtime [10] for scheduling distributed tasks across multiple heterogeneous nodes in a cluster.

As a simple illustration of our approach, consider two versions of the Poly-Bench [1] `correlation` benchmark shown in Figures 1 and 2. The first case

```
1  def kernel(self, float_n: float, data: list, corr: list, mean: list, stddev: list):
2      ...
3      for i in range(0, self.M-1):
4          corr[i][i] = 1.0
5          for j in range(i+1, self.M):
6              corr[i][j] = 0.0
7              for k in range(0, self.N):
8                  corr[i][j] += (data[k][i] * data[k][j])
9              corr[j][i] = corr[i][j]
10     corr[self.M-1][self.M-1] = 1.0
```

Fig. 1. PolyBench-Python `correlation`: List version (default)

```
1  from numpy.core.multiarray import ndarray
2  ...
3  def kernel(self, float_n: float, data: ndarray, corr: ndarray, mean: ndarray, stddev: ndarray):
4      ...
5      corr[np.diag_indices(corr.shape[0])] = 1.0
6      for i in range(0, self.M - 1):
7          corr[i,i+1:self.M] = (data[0:self.N,i] * data[0:self.N,i+1:self.M].T).sum(axis=1)
8      tril_indices = np.tril_indices( n=self.M, m=self.M, k=-1 )
9      corr[tril_indices] = corr[triu_indices]
10     corr[self.M - 1, self.M - 1] = 1.0
```

Fig. 2. PolyBench-Python `correlation`: NumPy version

Table 1. Execution time of `correlation` benchmark (dataset = large on Titan Xp workstation equipped with Intel i5-7600 4-core CPU and NVIDIA Pascal GPU)

List version	NumPy version	`AutoMPHC` (input: List)	`AutoMPHC` (input: NumPy)
152.5 [sec]	2.212 [sec]	0.1760 [sec]	0.07163 [sec]

represents a list-based pattern implemented using three explicit Python loops that access elements of lists (as surrogates for arrays), which might have been written by a domain scientist familiar with classical books on algorithms such as [16]. The second case represents a NumPy-based pattern with one explicit loop and a two-dimensional array statement in line 7 of Figure 2, which might have been written by a domain scientist familiar with matrix operations. A unique feature of our approach is the ability to support both explicit Python loops and implicit loops from NumPy operators and library calls in a unified optimization framework. The performance results for this example in Table 1 show that the NumPy-based version of the `correlation` benchmark performs better than the list version, while our approach (which can be applied to either style of input) performs significantly better than both. Additional performance results are discussed in Section 5.

In summary, this paper makes the following contributions:

– A novel approach to automatic ahead-of-time (AOT) parallelization and optimization of sequential Python programs for execution on distributed heterogeneous platforms. Our approach is driven by the inclusion of type hints for function parameters and return values, which can be supplied by the

programmer or obtained by dynamic profiler tools; multi-version code generation guarantees the correctness of our AOT transformation in all cases.
- Automatic parallelization and high-level code optimizations for the target distributed heterogeneous hardware platform, based on novel extensions to the polyhedral framework that unify user-written loops and implicit loops present in matrix/tensor operators, as well as automated selection of CPU vs. GPU code variants.
- Automatic code generation for targeting the Ray runtime to schedule distributed tasks across multiple heterogeneous nodes in a cluster.
- An empirical evaluation of 15 Python-based benchmarks from the PolyBench suite on a standard GPU-equipped workstation, and multi-node evaluation of the Space-Time Adaptive Processing (STAP) radar application in Python. Both evaluations show significant performance improvements due to the use of `AutoMPHC`. In the case of STAP, the performance improvement relative to the original Python code was over 20,000× when using 24 nodes and 144 GPUs (6 GPUs/node) in the OLCF Summit supercomputer.

2 Background

2.1 Intrepydd Compiler

The Intrepydd programming language [30] introduced a subset of Python that is amenable to ahead-of-time (AOT) compilation into C++. It is intended for writing kernel functions rather than complete or main programs. The C++ code generated from Intrepydd kernels can be imported into a Python application or a C++ application.

A key constraint in the Intrepydd subset of Python is the requirement that Intrepydd function definitions include type annotations for parameters and return values. Given these type annotations, the Intrepydd compiler statically infers the types of local variables and expressions. The Intrepydd tool chain includes a library knowledge base, which specifies type rules for a wide range of standard library functions used by Python programs. As discussed in the following sections, the `AutoMPHC` system extends the Intrepydd tool chain to serve as a Python-to-Python optimization and parallelization system; there is no C++ code generated by the current version of `AutoMPHC`.

It is important to note that Intrepydd also includes extensions to standard Python to enable C++ code generation. These extensions include statements with explicit parallelism (e.g., `pfor` for parallel loops) and special library functions. In contrast, `AutoMPHC` does not rely on any of these extensions. All input code to `AutoMPHC` and all output code generated by `AutoMPHC` can be executed on standard Python implementations.

2.2 Ray Runtime

We use Ray [10] as the base distributed runtime framework. Ray features a number of properties beneficial for `AutoMPHC`. First, the ability to simultaneously support both *stateless* and *stateful* computation—one of its key research

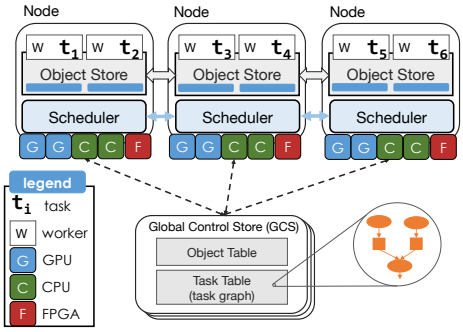

Fig. 3. `AutoMPHC` distributed runtime architecture.

contributions useful for a heterogeneous mix of CPU and GPU compute. Stateless computation, in the form of side-effect free tasks, is best suited for processing large data objects or partitions on numerous CPU resources. Stateful computation is beneficial for GPU tasks. We create tasks for this distributed runtime by *automatically* compiling selected regions of code into Ray tasks. Each Ray task then can be spawned asynchronously. A full directed acyclic graph (DAG) of such task instantiations is dynamically constructed and submitted for execution without waiting for intermediate computation results. It enables `AutoMPHC` to (a) hide the latency of task instantiation and propagation to workers for execution, (b) extract pipeline parallelism, and, (c) extract parallelism from the partial order of the dynamically constructed directed acyclic task graph. As Ray tasks are instantiated, they return immediately with a future-like construct, called an ObjectID — an object handle that refers to a globally addressable object. The object is eventually fulfilled and can be extracted with a blocking `ray.get(object_id)` API. We note that the distributed object store (Fig. 3) used for the lifecycle of these objects is immutable—a property that elides the need for expensive consistency protocols, state coherence protocols, and other synchronization overheads needed for data correctness. Further, DAG parallelism alleviates the need for expensive MPI-style distributed barriers and, therefore, does not suffer from the otherwise common straggler challenges—an important property for heterogeneous compute at scale. Finally, data store and the deterministic nature of the task graph jointly enable fault tolerance, as any missing object in the graph can be recomputed by simply replaying the sub-graph leading up to and including the object's parent vertex. This mechanism can be triggered automatically and comes with minimal overhead on the critical path of a task [29].

3 Overview of Our Approach

Figure 4 summarizes the overall design of our proposed `AutoMPHC` system. User-developed code is a combination of *main program code* and *kernel code*, where

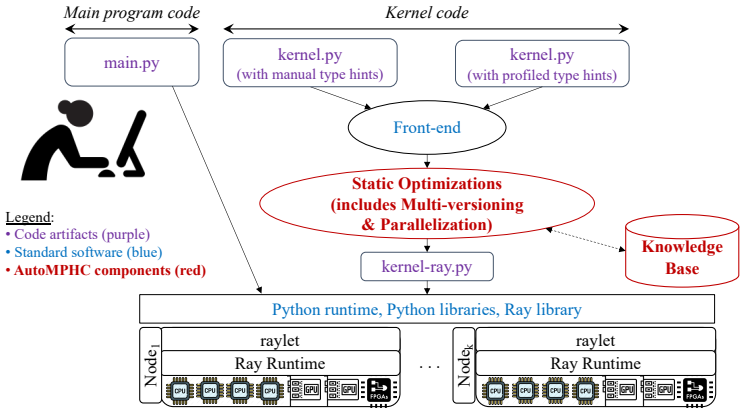

Fig. 4. Overall Design of `AutoMPHC` system

the former is unchanged while the latter is optimized by `AutoMPHC` via automatic ahead-of-time (AOT) source-to-source transformations. Both execute on a standard Python runtime along with Ray and other libraries used by the application. There are two forms of kernel code supported by our system — one in which type annotations are manually provided by the user, and another in which type annotations are obtained by a type profiler such as MonkeyType. In both cases, the type annotations serve as *hints* since multi-version code generation guarantees the correctness of our AOT transformations in all cases (whether or not the actual inputs match the type annotations).

The kernel functions with type annotations (hints) are first translated by the Front-end to an Abstract Syntax Tree (AST) representation implemented using the standard Python Typed AST package [19]. The core optimizations in `AutoMPHC` are then performed on the AST, including multi-version code specialization (Section 4.1), polyhedral optimizations (Section 4.2), and generation of distributed parallel code using Ray tasking APIs along with generation of heterogeneous code using selective NumPy-to-CuPy conversion (Section 4.3). These Static Optimizations benefit from the use of the `AutoMPHC` Knowledge Base, which includes dataflow and type information for many commonly used library functions. The transformed code is then executed on a distributed heterogeneous platform using standard Python libraries in addition to Ray.

4 Optimizations

The `AutoMPHC` compiler is an extension of Intrepydd compiler [30], which supports type inference and basic optimizations including loop invariant code motion, sparsity optimization, and array allocation/slicing optimizations. In the following sections, we present newly developed optimizations for automatic parallelization targeting distributed heterogeneous systems.

```
def kernel(self, float_n: float, data: ndarray, ...):
    if type(float_n) == float and type(data) == ndarray and ...:
        if data.ndim == 2 and ... :
            ...  # Code with type-specific and rank-specific optimizations
        else:
            ...  # Code with type-specific optimizations
    else:
        ...  # Code without type-specific optimizations
```

Fig. 5. Multi-versioning for PolyBench-Python correlation

4.1 Program Multi-versioning for Specialized Code Optimizations

Multi-versioning is an approach to data-aware optimizations, which generates multiple code versions specialized under certain conditions at compile-time and selects a proper code version at runtime. In our framework, we consider two classes of conditions, *legality-based* and *profitability-based*. All the conditions are organized as decision trees, where legality conditions are located at higher levels while profitability conditions are at lower levels in general.

The legality conditions are mainly used to verify the data type annotations attached on function parameters and returns. In our approach, the type annotations are used as hints and the compiler speculatively performs optimizations assuming these hints are correct. For example, the accuracy of array rank/dimensionality inference, which is derived from the type hints, is critical to the polyhedral optimizations (Section 4.2). In contrast, array element types may be less critical in some cases since `AutoMPHC` generates untyped Python code as output (unlike Intrepydd, which generated typed C++ code). In general, since these type annotations can be different from the actual types at runtime, the multi-versioning code generation introduces runtime checks of annotated/inferred types and ranks for specialized code version while ensuring correct behavior for others, as shown in Figure 5.

The profitability conditions can cover a broad range of conditions/scenarios related to runtime performance rather than correctness. As described later, the `AutoMPHC` compiler can generate two versions of optimized kernels, one for CPUs and the other for GPUs. The runtime condition used to select between these two versions is a typical example of a profitability condition (Section 4.3).

4.2 Polyhedral Optimizations

Polyhedral compilation has provided significant advances in the unification of affine loop transformations combined with powerful code generation techniques [4, 27, 31]. However, despite these strengths in program transformation, the polyhedral frameworks lack support for: 1) dynamic control flow and non-affine access patterns; and 2) library function calls in general. To address the first limitation, we have extended the polyhedral representation of Static Control Parts (SCoPs) to represent unanalyzable expressions as a compound "black-box" statement with approximated input/output relations. To address the second limitation, we

Table 2. NumPy examples in library knowledge base

Library function	Domain	Semantics and dataflow
transpose_{2D}	(i_0, i_1)	$R[i_0, i_1] := A_1[i_1, i_0]$
$\text{mult}_{1D,2D}$	(i_0, i_1)	$R[i_0, i_1] := A_1[i_1] \times A_2[i_0, i_1]$
sum_{1D}	(0)	$R := \Sigma_k A_1[k]$
$\text{sum}_{2D,axis=1}$	(i_0)	$R[i_0] := sum_{1D}(A_1[i_0, :])$
$\text{dot}_{2D,2D}$	(i_0, i_1)	$R[i_0, i_1] := sum_{1D}(mult_{1D,1D}(A_1[i_0, :], A_2[:, i_1]))$
$\text{fft}_{2D,axis=1}$	(i_0)	$R[i_0, :] := fft_{1D}(A_1[i_0, :])$

took advantage of our library knowledge base to obtain element-wise dataflow relations among function arguments and return values (see examples in Table 2). These unique features enable the co-optimization of both explicit loops and implicit loops from operators and library calls in a unified optimization framework, as detailed in the rest of this section.

Given SCoP representation extracted from the Python IR, the `AutoMPHC` polyhedral optimizer, which is built on the PolyAST [23, 22] framework, computes dependence constraints and performs program transformations. Finally, the optimized SCoP representation is converted back to Python IR with the help of the library knowledge base for efficient library mapping.

Intra-node parallelization: The optimization goal for the intra-node level is to generate sufficient parallelism to fully utilize efficient multithreaded libraries such as BLAS-based NumPy and CuPy. Our modified PolyAST [23] algorithm applies loop distribution to split different library calls into different loops while maximizing the iteration domain (i.e., amount of computation) that can be mapped to a single library function call. The SCoP-to-Python-IR generation stage leverages the library knowledge base to select the most efficient combination of available library functions for each statement when available.

Figure 6a shows the computationally dominant code region in the PolyBench-Python `correlation` NumPy benchmark, which has a `for` loop enclosing a sequence of NumPy function calls: 2-D array transpose overlapping `T` operator; 1-D×2-D array multiply overlapping `*` operator, and 2-D array summation `sum` to produce 1-D result. Based on the type inference results, the polyhedral phase first identifies these library functions with specific types and array ranks. As shown in Table 2, the library knowledge base provides the element-wise dataflow information and operation semantics of these functions, which are used to extract the SCoP information and semantics of each statement (Figure 6b). Note that: both explicit and implicit loops are unified in a triangular iteration domain; and the element-wise dataflow and semantics are summarized as the statement body, `corr[i0, i1] = sum(mult(data[:, i0], data[:, i1]))`. Given the statement body, the SCoP-to-Python-IR generation stage selects the combination of matrix-matrix multiplication `numpy.dot` and 2-D transpose `T` as the best mapping, followed by `numpy.triu` to update only the triangular iteration domain (Figure 6c). As discussed in Section 5.2, this transformation

```
1       for i in range(0, M-1):
2           corr[i, i+1:M] = (data[0:N, i] * data[0:N, i+1:M].T).sum(axis=1)
```

(a) Original code fragment

```
Domain: {S[i0, i1] : 0 <= i0 < M - 1 and i0 + 1 <= i1 < M}
Read:   {S[i0, i1] -> data[any1, i0], data[any2, i1] : 0 <= any1, any2 < N}
Write:  {S[i0, i1] -> corr[i0, i1]}
Body:    S[i0, i1] :: corr[i0, i1] = sum(mult(data[:, i0], data[:, i1]))
```

(b) Extracted polyhedral information (SCoP)

```
1       tmp1 = np.dot(data[0:N, 0:M].T, data[0:N, 0:M])
2       corr[0:M-1, 0:M] = np.triu(tmp1[0:M-1, 0:M], k=1)
```

(c) Transformed code fragment by intra-node polyhedral optimization

Fig. 6. Kernel from the PolyBench-Python correlation

sufficiently increases the intra-node parallelism per library call and contributes to significant improvements for several benchmarks.

When the input program is written only with explicit loops, e.g., the List version in Figure 2, our approach extracts similar SCoP and generates similar code, but with necessary conversions between List and NumPy array data types. **Inter-node parallelization:** The optimization policy for inter-node level is equivalent to the original PolyAST [23] algorithm that maximizes outermost level parallelism, while incorporated with our data layout transformation approach [24, 25] to reduce the total allocated array sizes and data movement across Ray tasks. Analogous to two-level parallelization for GPUs [22], our polyhedral optimizer selects different *schedules* – i.e., compositions of loop transformations and parallelization – for inter-node and intra-node levels individually; and integrates them into the final schedule via loop tiling.

Figures 7a and 7b respectively show the computational kernel of the STAP radar application and the extracted SCoP information. The explicit loop with statement S and the `fft` call of statement T are handled as 1-D iteration domains while 2-D×2-D array multiply of statement U is handled as a 2-D iteration domain. The polyhedral optimizer identifies the outermost level parallelism and computes the inter-node schedule that fuses these statements into a single parallel loop. The transformed code after integrating the inter-node and intra-node schedules is shown in Figure 7c, where `pfor` is an intermediate parallel loop construct that is implemented using Ray tasks.

4.3 NumPy-to-CuPy Conversion and Parallelized Code Generation

After the polyhedral phase, the program multi-versioning pass (Section 4.1) is applied to the *pfor* parallel loops and generates both sequential and parallel versions. The profitability condition, which makes the decision on whether the loop should be distributed across nodes via the Ray runtime, is generated by a

```
for idx in range(numPulses):
    ...
    beamforming[idx,:] = np.squeeze(np.matmul(steerVector11, dataCube)) # S
d_X = np.fft.fft(beamforming, fftSize, axis=1) # T
d_Y = d_X * d_matchFilterMultiply # U
```

(a) Original code fragment

```
Domain: {S[i0]     : 0 <= i0 < numPulses}
        {T[i0]     : 0 <= i0 < numPulses}
        {U[i0, i1] : 0 <= i0 < numPulses and 0 <= i1 < fftSize}
Body:    S[i0]     :: beamforming[i0, :] = ...
         T[i0]     :: d_X[i0, :] = fft(beamforming[i0, :])
         U[i0, i1] :: d_Y[i0, i1] = d_X[i0, i1] * d_matchFilterMultiply[i0, i1]
```

(b) Extracted polyhedral information (read/write omitted due to space)

```
pfor t1 in range(0, numPulses, __tile_size):  # Parallel loop across nodes
    up1 = min(t1 + __tile_size, numPulses)
    for c1 in range(t1, up1):
        ...
        beamforming[c1,:] = np.squeeze(np.matmul(steerVector11, dataCube)) # S
    d_X = np.fft.fft(beamforming[t1:up1, :], fftSize, axis=1) # T
    d_Y = d_X * d_matchFilterMultiply # U
```

(c) Transformed code fragment by inter-node polyhedral parallelization

Fig. 7. Kernel from the STAP Signal Processing Application

simple cost-based analysis and summarized as a threshold expression using loop counts. This analysis also includes the profitability check of the CuPy conversion for a given sequence of NumPy library calls. The current implementation takes an all-or-nothing approach for NumPy-to-CuPy conversion in a code region, and more fine-grained control, e.g., per-array decisions, is a topic for future work.

To generate Ray-based distributed code from a high-level *pfor* loop, the polyhedral phase provides the following supplemental information related to data access and NumPy-to-CuPy conversion.

`pfor (output =` $\{var_{out_1} : type_{out_1}, var_{out_2} : type_{out_2}, ...\}$`,`
`input =` $\{var_{in_1} : type_{in_1}, var_{in_2} : type_{in_2}, ...\}$`,`
`transfer =` $module\_name$`)`

The `output` and `input` clauses respectively specify the produced and referenced variables by the *pfor* loop and their corresponding types, while the `transfer` clause indicates the possibility of NumPy-to-CuPy conversion based on the polyhedral dataflow analysis and library compatibility.

4.4 Important Packages Used in `AutoMPHC` Tool Chain

Our `AutoMPHC` compilation flow is built on top of the Python Typed AST package [19], which serves as the baseline IR to perform fundamental program analyses and transformations such as type inference, loop invariant code motions,

Table 3. Hardware Platform Information (per node) and software versions

Per node	Cori-GPU	Summit	Titan Xp (workstation)
CPU	2 × Intel Xeon Gold 6148 @ 2.40 GHz (40 cores/node)	2 × IBM POWER9 @ 3.1 GHz (44 cores/node)	1 × Intel i5-7600 CPU @ 3.50GHz (4 cores)
GPU	8 × NVIDIA Tesla V100	6 × NVIDIA Tesla V100	1 × NVIDIA Pascal
Memory	384GB	512GB	15GB
Interconnect	InfiniBand + PCIe (CPUs-GPUs) + NVLink (GPUs)	InfiniBand + NVLink (CPUs-GPUs, GPUs)	PCIe (CPU-GPU)
Python / NumPy / CuPy	3.7.3 / 1.16.4 / 7.4.0	3.7.3 / 1.16.0 / 7.4.0	3.6.9 / 1.19.5 / 7.2.0
Ray	0.8.4	0.7.7	0.8.4

Table 4. PolyBench-Python baselines: Execution time in second (dataset = large)

	2mm	3mm	atax	bicg	correlation	covariance	doitgen	gemm
List Default [sec]	224.4	356.2	0.6578	0.6730	152.5	305.7	54.46	147.4
List Pluto [sec]	205.2	337.9	0.8381	0.8304	152.1	153.8	54.45	191.5
NumPy [sec]	0.0214	0.03252	0.002516	0.002447	2.212	3.813	0.1250	0.01789

	gemver	gesummv	mvt	symm	syr2k	syrk	trmm	
List Default [sec]	1.510	0.3068	0.8710	140.4	171.4	96.66	91.10	
List Pluto [sec]	1.453	0.3154	0.8714	140.5	137.9	81.73	93.27	
NumPy [sec]	0.04676	0.001074	0.002537	1.656	2.667	0.7839	0.8499	

and constant propagations. For the polyhedral optimizations presented in Section 4.2, we employ the `islpy` package, the Python interface to the Integer Set Library (ISL) [28] for manipulating sets and relations of integer points bounded by linear constraints. Beside the polyhedral representations using `islpy`, we employ `sympy` [26] to analyze symbolic expressions observed in the Typed AST.

5 Experimental Results

5.1 Experimental Setup

We use a standard GPU-equipped workstation, Titan Xp, for single-node experiments (Section 5.2) and two leading HPC platforms, NERSC Cori [11] and OLCF Summit [8] supercomputers, for multi-node experiments (Section 5.3). The single-node specification of these platforms is summarized in Table 3. For Summit, we manually built Ray and its dependencies from scratch because there is currently no out-of-the-box Python Ray package for the POWER processor.

5.2 Single-node Results (PolyBench)

We first evaluate the impact of our polyhedral optimizations using PolyBench-Python [1], which is the Python implementation of PolyBench [15], a widely used benchmark kernels for compiler evaluations. We use a total of 15 benchmarks shown in Table 4, which are well suited to our current library-oriented optimization strategy. The evaluation of other 15 benchmarks is a topic for future work that supports hybrid Python/C++ code generation.

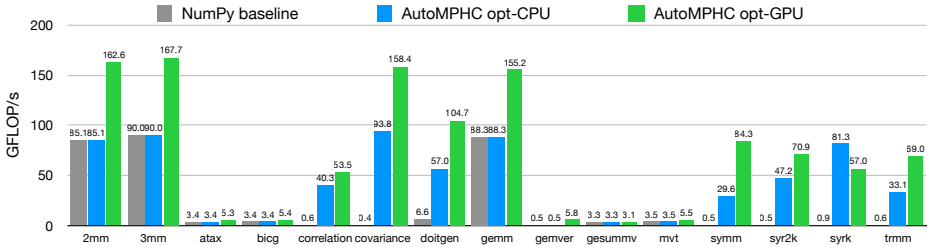

Fig. 8. PolyBench-Python performance on NVIDIA Titan Xp (dataset = extra large)

PolyBench Python provides a variety of benchmark implementations, including the default List version, the optimized List version by the Pluto polyhedral compiler [4], and the NumPy version. Table 4 shows the execution time of these versions using the "large" datasets. While the Pluto optimization improves the performance, NumPy version largely outperforms List versions for all cases.

In the following experiments, we use NumPy version as the baseline of our comparison, and "extra large" dataset to ensure sufficient execution time. Note that "extra large" and "large" respectively refer to the first and second largest datasets. Figure 8 shows the GFLOP/s of three experimental variants:

- NumPy baseline: the original NumPy implementation from PolyBench.
- `AutoMPHC` opt-CPU: the CPU optimized version by `AutoMPHC` framework.
- `AutoMPHC` opt-GPU: the GPU optimized version by `AutoMPHC` framework enabling NumPy-to-CuPy conversion.

Comparing the NumPy baseline and `AutoMPHC` opt-CPU versions, our polyhedral optimization gives 8.7× – 212.4× performance improvements for `correlation`, `covariance`, `doitgen`, `symm`, `syr2k`, `syrk`, and `trmm`, while showing comparable performance for other benchmarks. Enabling NumPy-to-CuPy conversion, i.e., `AutoMPHC` opt-GPU version, further improves the performance for most benchmarks with the exception of `gesummv` and `syrk`. In this evaluation, our profitability conditions always selected GPU variants. The improvement of CPU/GPU selection based on offline profiling is an important topic for future work.

5.3 Multi-node Results (STAP)

We demonstrate the multi-node performance of our `AutoMPHC` compiler framework using one of our target applications in the signal processing domain, namely the Space-Time Adaptive Processing (STAP) application for radar systems [9]. The problem size used for STAP is to evaluate the analysis of 144 data cubes for the CPU case; and 2304 data cubes for the GPU case, where each data cube has # pulses per cube = 100, # channels = 1000, and # samples per pulse = 30000. The throughput performance required for real-time execution is 33.3 [cubes/sec]. We compare four experimental variants as listed below:

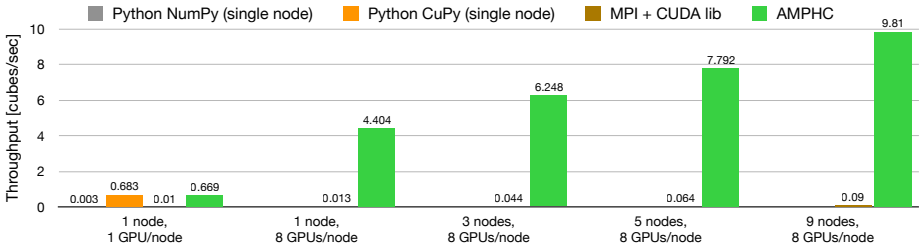

Fig. 9. STAP radar application performance on NERSC Cori supercomputer

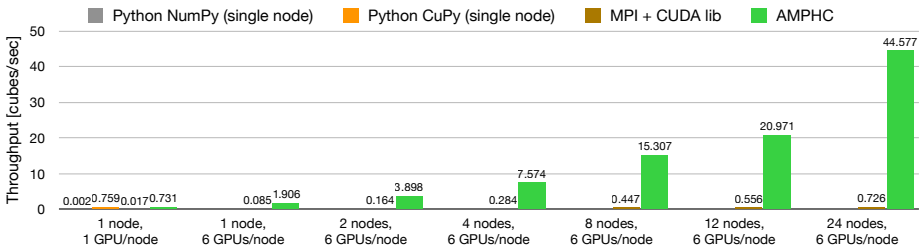

Fig. 10. STAP radar application performance on OLCF Summit supercomputer

- Python NumPy: The original single-node CPU implementation.
- Python CuPy: CuPy-based single-node GPU implementation, manually ported from the original Python NumPy version.
- MPI+CUDA lib: MPI C/C++ and CUDA library-based multi-node GPU implementation, manually ported from the Python CuPy version by a domain expert with past experience in MPI and C/C++ programming.
- `AutoMPHC`: Automatic parallelization by the `AutoMPHC` compiler of the original Python NumPy version, running on the Ray distributed runtime.

Figures 9 and 10 show the throughput performance, i.e., number of data cubes processed per second, respectively on Cori and Summit clusters. Given the Python NumPy version as input, the `AutoMPHC` compiler automatically parallelized the major computation kernel and mapped it to GPUs via NumPy-to-CuPy conversions. This significantly improves the throughput performance, resulting in comparable single-GPU performance with the manually ported CuPy implementation on both clusters. The MPI+CUDA lib version shows significantly worse performance compared to the `AutoMPHC` version. The major performance bottleneck of this variant lies in the unoptimized data transfers among CPUs and GPUs. Although the majority of computation in the MPI+CUDA lib version is covered by CUDA-based libraries such as `cufft`, the interleaving of C-based CPU code with the CUDA-based GPU library calls resulted in a large amount of CPU/GPU data transfers. Optimizing these data transfers would have required deep (ninja-level) experience with CUDA programming as well as sufficient time

for performance tuning. In contrast, the `AutoMPHC` version shows good multi-node scalability with performance of up to 44.58 [cubes/sec] using 24 nodes on Summit, which exceeds 33.3 [cubes/sec] of the domain-specific throughput requirement for the real-time radar systems. The `AutoMPHC` version also achieves 4.40 [cubes/sec] of single-node performance on Cori while the multi-node scalability is more limited than that on Summit. This stems from the difference in network, i.e., Summit's NVLink (50GB/s) vs. Cori's PCIe 3.0 (16GB/s).

Breakdown on CPU-GPU interconnect performance: In the parallelized STAP code by `AutoMPHC`, each parallel task performs the computation on the GPU-side and returns a few gigabytes of the result via device-to-host (D2H) data transfers. We developed a synthetic benchmark that mimics the behavior of the D2H transfers in our `AutoMPHC`-parallelized STAP application and evaluated its impact on the overall performance. Because `nvprof` could not profile the GPU part invoked from the Ray distributed runtime, our D2H benchmark is implemented using OpenMP+CUDA and spawns parallel threads to simultaneously perform the D2H transfers. The benchmarking results, i.e., timings to complete D2H data transfer using all GPUs of a single node, are summarized as:

- Cori: 8 GPUs per node, PCIe 3.0 (16GB/s)
  - 6 cubes (2.4GB) per GPU – 39.563sec
  - 16 cubes (6.4GB) per GPU – 150.224sec
- Summit: 6 GPUs per node, NVLink (50GB/s)
  - 6 cubes (2.4GB) per GPU – 3.251sec
  - 16 cubes (6.4GB) per GPU – 8.791sec

The results clearly show that simultaneously transferring back large amounts of data significantly degrades the D2H bandwidth of PCIe 3.0 on Cori. As expected, NVLink on Summit largely outperforms PCIe 3.0 in terms of D2H data transfers, which is why NVLink contributed to the good scalability of the `AutoMPHC`-generated code on the Summit system.

6 Related Work

There are a number of compilation frameworks for enhancing Python performance, most notably Numba [13], PyPy [17] and Pyston [18] which use just-in-time compilation; and Cython [5], Nuitka [12], and Shed Skin [21] which use source-to-source translation and native compilation. As an example of loop-aware optimizations, PyPy's Tracing JIT [2] enables common subexpression elimination and memory allocation removal within loops, based on the interpreter execution traces. Numba, implemented as a Python library, dynamically translates a subset of Python code into machine code via LLVM-based JIT compilation. Despite the rich set of optimization passes in LLVM framework, the low-level translated code is not amenable to a large segment of the LLVM optimizations including the polyhedral optimizations by Polly [7]. While all these efforts aim to improve performance through generating native code, to the best of our knowledge, none of them has leveraged high-level abstractions of Python source program for AOT compilation as in our approach.

7 Conclusions

This paper describes AutoMPHC —a programming system designed to deliver the benefits of distributed heterogeneous hardware platforms to domain scientists who naturally use high-productivity languages like Python. In our approach, the parameters and return values of kernel Python functions are annotated with type hints, manually by users or automatically by profiling tools. Based on these type hints, the AutoMPHC compiler performs automatic AOT parallelization, based on advanced polyhedral optimizations, CuPy-driven GPU code generation, and Ray-targeted heterogeneous distributed code generation and execution. The correctness of our AOT parallelization is guaranteed by multi-version code generation, since code versions with type-specific optimizations are executed only when the actual runtime types match the type hints. Our empirical evaluations using PolyBench-Python for workstation performance and the STAP radar application for heterogeneous distributed performance show significant performance improvements, e.g., up to 358× improvement for PolyBench and up to 20,000× improvement for the STAP radar application, relative to baseline NumPy-based implementations. Opportunities for future work include hybrid Python/C++ code generation, fine-grained NumPy-to-CuPy conversion, and profile-based CPU/GPU runtime selection.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR0011-20-9-0020. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. Also, this research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. Abella-González, M.A., Carollo-Fernández, P., Pouchet, L.N., Rastello, F., Rodríguez, G.: Polybench/python: Benchmarking python environments with polyhedral optimizations. In: Proc. of CC 2021 (2021). https://doi.org/10.1145/3446804.3446842
2. Ardö, H., Bolz, C.F., FijaBkowski, M.: Loop-aware optimizations in pypy's tracing jit. SIGPLAN Not. **48**(2), 63–72 (Oct 2012). https://doi.org/10.1145/2480360.2384586
3. Bezanson, J., Karpinski, S., Shah, V.B., Edelman, A.: Julia: A fast dynamic language for technical computing. CoRR **abs/1209.5145** (2012)
4. Bondhugula, U., Acharya, A., Cohen, A.: The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests. ACM Trans. Program. Lang. Syst. **38**(3) (Apr 2016). https://doi.org/10.1145/2896389
5. Cython. https://cython.org/ (2007)

6. Dalcin, L., Fang, Y.L.L.: mpi4py: Status update after 12 years of development. Computing in Science Engineering (2021). https://doi.org/10.1109/MCSE.2021.3083216
7. Grosser, T., Größlinger, A., Lengauer, C.: Polly - performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters **22**(4) (2012)
8. LCF Summit supercomputer. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/ (2019)
9. Melvin, W.L.: Chapter 12: Space-time adaptive processing for radar. Academic Press Library in Signal Processing: Volume 2 Comm. and Radar Signal Proc. (2014)
10. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M.I., Stoica, I.: Ray: A distributed framework for emerging ai applications. In: Proc. of OSDI'18 (2018)
11. NERSC Cori supercomputer. https://docs.nersc.gov/systems/cori/ (2016)
12. Nuitka. https://nuitka.net/pages/overview.html (2012)
13. Numba. https://numba.pydata.org/ (2012)
14. NumPy. https://numpy.org/ (2006)
15. PolyBench: The polyhedral benchmark suite, http://www.cse.ohio-state.edu/ pouchet/software/polybench/
16. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes 3rd Edition: The Art of Scientific Computing. 3 edn. (2007)
17. PyPy. https://pypy.org/ (2019)
18. Pyston. https://blog.pyston.org/ (2014)
19. Python typed AST package. https://pypi.org/project/typed-ast/ (2019)
20. SciPy. https://www.scipy.org/ (2001)
21. Shed Skin. https://shedskin.github.io/ (2012)
22. Shirako, J., Hayashi, A., Sarkar, V.: Optimized two-level parallelization for gpu accelerators using the polyhedral model. In: Proc. of CC 2017 (2017). https://doi.org/10.1145/3033019.3033022
23. Shirako, J., Pouchet, L.N., Sarkar, V.: Oil and water can mix: An integration of polyhedral and ast-based transformations. In: Proc. of SC'14 (2014). https://doi.org/10.1109/SC.2014.29
24. Shirako, J., Sarkar, V.: Integrating data layout transformations with the polyhedral model. In: Proc. of IMPACT 2019 (2019)
25. Shirako, J., Sarkar, V.: An affine scheduling framework for integrating data layout and loop transformations. In: Proc. of LCPC 2020 (2020). https://doi.org/10.1007/978-3-030-95953-1˙1
26. SymPy. https://www.sympy.org (2017)
27. Verdoolaege, et al.: Polyhedral parallel code generation for CUDA. ACM Trans. Archit. Code Optim. **9**(4), 54:1–54:23 (2013). https://doi.org/10.1145/2400682.2400713
28. Verdoolaege, S.: isl: An integer set library for the polyhedral model. In: Mathematical Software ICMS 2010 (2010). https://doi.org/10.1007/978-3-642-15582-6˙49
29. Wang, S., et al.: Lineage stash: Fault tolerance off the critical path. In: Proc. of the ACM Symposium on Operating System Principles (SOSP'19). SOSP '19 (2019)
30. Zhou, T., et al.: Intrepydd: Performance, productivity and portability for data science application kernels. In: Proc. of Onward! '20 (2020). https://doi.org/https://doi.org/10.1145/3426428.3426915
31. Zinenko, O., et al.: Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In: Proc. of CC 2018 (2018). https://doi.org/10.1145/3178372.3179507