ALPyNA: Acceleration of Loops in Python for Novel Architectures

Dejice Jacob
School of Computing Science
University of Glasgow
UK
d.jacob.1@research.gla.ac.uk

Jeremy Singer
School of Computing Science
University of Glasgow
UK
jeremy.singer@glasgow.ac.uk

Abstract

We present *ALPyNA*, an automatic loop parallelization framework for Python, which analyzes data dependences within nested loops and dynamically generates CUDA kernels for GPU execution. The ALPyNA system applies classical dependence analysis techniques to discover and exploit potential parallelism. The skeletal structure of the dependence graph is determined statically (if possible) or at runtime; this is combined with type and bounds information discovered at runtime, to auto-generate high-performance kernels for offload to GPU.

We demonstrate speedups of up to 1000x relative to the native CPython interpreter across four array-intensive numerical Python benchmarks. Performance improvement is related to both iteration domain size and dependence graph complexity. Nevertheless, this approach promises to bring the benefits of manycore parallelism to application developers.

CCS Concepts • **Software and its engineering** → **Dynamic compilers**; **Scripting languages**; *Parallel programming languages*; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

Keywords code generation, nested loop parallelization, GPU

ACM Reference Format:
Dejice Jacob and Jeremy Singer. 2019. ALPyNA: Acceleration of Loops in Python for Novel Architectures. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3315454.3329956

ARRAY '19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6717-2/19/06…$15.00
https://doi.org/10.1145/3315454.3329956

1 Introduction

Dynamically typed, high-level scripting languages such as Python, R, Ruby and Javascript are increasingly popular. Python has been in widespread use for many years as shown by various programming language surveys [5, 21]. The Python language is particularly attractive to end-user developers [3, 18] given its simplicity and accessibility. For these reasons, Python has high usage in a broad range of scientific application domains including astronomy [16] bio-informatics [20] and meteorology [14]. Array-intensive numerical code is often prototyped, even deployed, as Python scripts or interactive notebooks.

While Python execution, via the CPython interpreter, is generally slow, many users are reluctant to switch to more optimization-amenable programming languages and systems. For this reason, there are various schemes to improve the runtime performance of Python— this work (along with many others) proposes exploiting manycore parallelism for Python. However we distinctively advocate that:

1. parallelism should be extracted *transparently*, from the point of view of the developer, to maintain maximum *user-friendliness* and
2. at runtime we must generate code tuned to the specific dependence relationships between memory accesses in each instantiation of a loop-nest within a code fragment.

Commodity GPUs offer huge numbers of cores for minimal cost and are often extremely effective for data parallel tasks. Depending on the workload, such accelerators can provide orders of magnitude better performance. Programming GPUs is, however, highly complex as it exposes the programmer to the physical realities of the GPU being used. Each kernel must be written in low-level domain specific languages like CUDA and OpenCL which involves the developer carefully calculating the size of the iteration domain and calculating any dependences between data accesses.

The process of reasoning gets progressively harder for complex code and imposes high cognitive burdens on the developer. Many attempts have been made to make this process easier for end-user developers as detailed in section-2.1.

Moreover, dynamic languages like Python resolve types at runtime. This complicates the generation of GPU code

since such kernels are required to be compiled with type information. While static type annotations can help, some form of templating is normally required to reuse the same kernel code for different types.

1.1 ALPyNA

This paper introduces *ALPyNA*, a dynamic loop parallelization framework for Python. The programmer writes numerical kernels using dense nested *for*-loops with a linear iteration space using the Python `range` function. Restricting the analysis to linear loops allows the analysis engine to reason about dependences carried by the loops, and apply various optimisations.

ALPyNA is extensible to multiple types of accelerators by abstracting the accelerator specific code generation for different classes of hardware. It is envisaged to extend the framework to generate code compiled specifically for CPUs or FPGA devices.

ALPyNA has three key novelties:

1. The dependence analysis is *staged*, with initial analysis occurring ahead-of-time and generating in-memory data structures that are preserved till runtime. As the program executes, information about types and bounds is incrementally added to the in-memory dependence data, allowing safe dynamic parallelization to occur.
2. The automatic parallelization takes *plain old Python* code as input, with no need for user annotations or rewriting.
3. The system generates a set of observationally equivalent *variants* for each method, targeting different compute devices. This allows selection of the optimal backend for each specific loop nest, for which performance may be input-dependent.

The ALPyNA system is implemented in Python and integrated with the standard CPython interpreter, as Section 3 describes. Adoption simply requires the user to import an extra Python library. We have evaluated ALPyNA with a range of array-intensive Python benchmarks (described in Section 4) and demonstrated significant speedup for moderately sized input data sets on stock hardware, as reported in Section 5.

To the best of our knowledge, this is the first framework for general auto-parallelization in Python that does not require invasive user annotations. Further, this is the first parallelization system for Python that stages the analysis.

2 Background

This section reviews concepts and material that are relevant for parallel Python (Section 2.1) and automatic parallelization techniques (Section 2.2).

2.1 Parallelism in Python

The Global Interpreter Lock (GIL) in the vanilla CPython runtime is an obvious impediment to parallelism. It prevents multiple threads from executing Python bytecode concurrently. The GIL is required since memory management is not thread-safe in CPython.

Given the near-ubiquitous availability of manycore processors, there is increasing pressure to support parallel execution in Python. A range of techniques have been applied, as reviewed below.

Code annotation: Lam *et al* [13] introduce *Numba*, which uses `@decorator` syntactic sugar to selectively compile functions for CPU or GPU. Numba requires code annotations, optionally including type information. It analyzes Python bytecode and compiles methods Just-in-Time (JIT) to native code using the LLVM infrastructure. In particular, the `@cuda.jit` decorator only works for a restricted subset of Python, effectively a one-to-one mapping from Python semantics to GPU kernel operations.

Parallelising higher order functions: Functional patterns involving higher-order functions like `map` and `filter` are attractive candidates for GPU offload since loop iteration independence is guaranteed by construction. Optimization is effectively a syntactic rewriting operation. Fumero *et al* [8] compile and parallelize `map` functions in Ruby and R. For Python, the effectiveness of this method has been demonstrated by Catanzaro *et al* [6] and Rubinsteyn *et al* [17].

Embedded Domain Specific Languages (eDSLs): *Loo.py* [11] is an eDSL that allows a developer to specify loop iteration ranges and sequences of array update operations, inlined in the Python code. These specifications can be transformed to parallel kernels, and invoked programmatically. Loo.py is an elegant code generation library for array-intensive calculations, targeting CPU and GPU devices. However, this explicit parallelism requires careful thinking on the part of the developer. No data dependence analysis or resolution is provided.

Task graphs: This approach allows for scheduling of inter-kernel dependences by modelling computation as nodes in a Directed Acyclic Graph (DAG) with edges representing data dependence. The *Pydron* system [15] is directed by user annotations to build a task graph for decomposing a program into parallel sections for cloud-based concurrent execution. It relies on further annotations to indicate pure (i.e. side-effect free) functions.

Writing GPU kernels: Klöckner *et al* [12] directly target the GPU by binding to CUDA (PyCUDA) and OpenCL (PyOpenCL) libraries. This grants direct access for the developer to program the GPU. The disadvantage is that kernels must be written in low-level C-like syntax and must also contain data types. Information can be patched in by editing the source code just before compiling the kernel. However this is left for the developer to do.

Library parallelism: Many Python libraries support GPU execution, e.g. the TensorFlow framework [1]. In such cases, all parallelism occurs inside a black box; the developer has little understanding and is unable to go 'below the API'. Python simply acts as a coordination language, executed sequentially, with all parallelism devolved to the library code.
Our new approach: Our motivation for ALPyNA is that the user should be able to write code as conventional, undecorated, 'plain old Python' functions and pass these to the analysis tool. The ALPyNA framework will return a specialised object containing a dictionary of callable functions which the programmer can invoke with relevant arguments. When such functions are called at runtime, ALPyNA's dynamic analysis and introspection system intercepts each call; it then generates, compiles, and executes relevant GPU kernels with appropriate data marshalling and transfer.

2.2 Automatic Parallelization

Automatic parallelization has a long and chequered history [2]. Commonly, the key control structure for parallelism is the *loop*, particularly hot loops (where most of the execution time is concentrated). The key data structure for parallelism is often the *array*, where most of the dynamic memory is allocated.

There is a vast compendium of loop parallelization techniques. Much is encapsulated in the work of Allen and Kennedy [10]. For the majority of use-cases, Goff *et al* [9] present simplified fast dependence tests. By computing the cyclic dependences between statements carried by various loops, we can identify which loops can be executed in parallel for the overall set of nested loops, without changing the computation.

These techniques generally apply to imperative, numerical computation. Code is usually written in Fortran; parallelization is also desirable for C/C++ high-performance computing code although the variable aliasing problem is more acute for C-style languages.

The difficulties with general auto-parallelization derive from the following root causes:

1. complexity of analysis (for both aliasing and dependence).
2. conservative nature of static analysis, since runtime values like loop bounds are usually unavailable.
3. difficulty of mapping parallel tasks to available hardware resources to achieve significant speedup.

ALPyNA overcomes the above difficulties using a hybrid analysis technique, combining static and runtime dependence analysis. It benefits from the Python language's relative simplicity, in terms of structured control flow (no C-style `goto`) and loop iterator guarantees provided by the `range` function semantics. These features make analysis much less complex.

Because Python execution is interpreter-based and relatively slow, we can often afford significant analysis overhead. As our results demonstrate (cf. Section 5) the analysis time is commensurate with the interpreter execution time for numerical codes.

Dependence analysis can exploit dynamic information. The rich nature of the Python runtime environment means we can preserve analysis artifacts throughout program execution and refine the knowledge base as information about data types and loop bounds becomes available. This enables more effective parallelization, as explained below.

Since we generate parallel work at runtime, we can do profiling to determine improvements. Further, we have exact knowledge of the nature of the target platform, since we are executing directly on it.

2.3 Benefits of Deferring to Runtime

Consider the code-segment in Listing 1. Dependence analysis tells us that the statement in the *for*-loop can be parallelised as long as the loop iteration domain is within the range $[0, 1024)$ in order to be correct.

Listing 1. Benefit of runtime parallelization

```
def function_foo( arg_a, arg_b, arr_len ) :
 for i in range(0,arr_len,1):
     arg_a[i+1024] = arg_a[i] + arg_b
```

In a static language like Fortran, symbolic resolution of the limits would result in the addition of a guard condition to check if the loop iteration domain would be less than 1024. If so, a parallel version of the loop would be invoked for execution. Otherwise the loop would be executed sequentially. Smarter compilers would do *strip mining* to tile all iterations that can be run in parallel and execute them with SIMD instructions.

Listing 2. ALPyNA example

```
import numpy as np
import Alpyna as alp

user_code = """
def loopy_kernel( arg_a, arg_b, limits) :
 i_max, j_max, k_max, m_max = limits
 for i in range(0,i_max,1):
   for j in range(0,j_max,1):
     for k in range(0,k_max,1):
       for m in range(0,m_max,1):
         #  loopy_kernel_stmt_01
         arg_a[i][j+10][k][m] =
           arg_a[i][j][k][m] + 4 + arg_b[i]

         #  loopy_kernel_stmt_02
         arg_a[i+1][j][k][m] =
           arg_a[i][j][k][m] + 43
"""
...
...
```

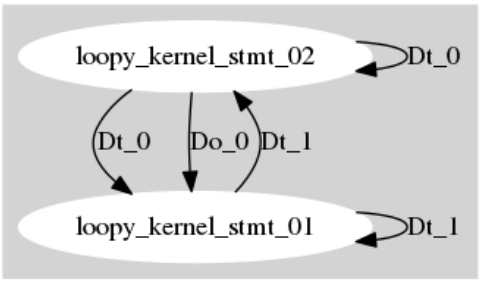

Figure 1. Dependence graph of loop nest in Listing 2 when end of iteration domain of *j*-loop causes a loop-carried dependence (end of *j* iteration domain > 10).

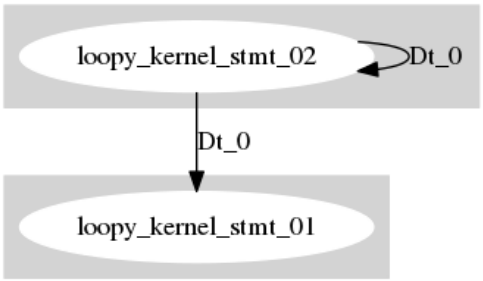

Figure 2. Dependence graph of loop nest in Listing 2 when iteration domain of *j*-loop = [0,10).

```
alpyna_ex_engine = alp.static_analyse(user_code)
alpyna_ex_engine.loopy_kernel(arr_a, arr_2, lims)
```

When the number of data dependences become larger, the dependence relationships make generating all the guard conditions and code variants *NP*-hard. By deferring this analysis to runtime, we can infer how much parallelism can be extracted from the loop-nest depending on the loop size and generate code to satisfy the dependence constraints while still executing in parallel.

To demonstrate the potential gains in parallelization by deferring analysis to runtime, consider the code in Listing 2. We depict dependences between statements in a nested loop as shown in Equation 1. The loop carrying the dependence is shown by its nesting level *n*. and has a range [0, *max-loop-nest-level*).

$$Dependence = \begin{cases} Dt\_n, & True\ dependence\ (\delta_n) \\ D\hat{}\_n, & Anti\ dependence\ (\delta_n^{-1}) \\ Do\_n, & Output\ dependence\ (\delta_n^o) \end{cases} \quad (1)$$

During static analysis, the iteration domain of the loops are unknown. There could potentially be loop-carried dependences from *statement-1* to *statement-2* and vice versa. To be correct, a static optimising compiler would have to be conservative and assume such dependences exist.

For example, if the iteration domain for dimensions *(i,j,k,m)* was $[0-50), [0-20), [0-40), [0-50)$, we would obtain a dependence graph as shown in Figure 1. A *True* dependence is carried by the *j*-loop from *statement-1* to *statement-2*. A *True* and *Output* dependence is carried by the *i*-loop in the opposite direction creating a cycle. Further cyclical dependences in this loop structure exist due to

1. a *True* dependence from *Statement-1* to itself carried by the *j*-loop
2. a *True* dependence from *Statement-2* to itself carried by the *i*-loop

The cyclical dependences between *statement-1* and *statement-2* necessitates running both statements sequentially inside the *i*-loop. *Statement-2* can then be parallelized over the inner *j,k* and *m*-loops. To preserve the dependence relationship, the first statement would have to be run sequentially inside the *i* and *j*-loops while parallelizing the *k* and *m*-loops.

If the iteration domain was changed to $[0-50), [0-4), [0-40), [0-50)$,the dependence graph, as shown in Figure 2, would be applicable. In this case, the only cycle is generated by the *True* dependence from *statement-2* to itself. This would allow us to parallelize the first statement across all four (*i,j,k,m*) loop iteration dimensions while the second statement would have to be run sequentially within the *i*-loop and parallelized across the other three iteration domains.

3 Compiler Implementation

In principle, the ALPyNA analysis and kernel code construction is a *staged* process. In this section, we refer to the *static* (Section 3.2) and *dynamic* (Section 3.3) stages distinctly. The static part occurs during program initialization, described in Section 3.1. This may run in the interpreter or as a setup code block in an interactive notebook. The dynamic part happens as the loop is executed for the first time, when runtime types and loop bounds have been resolved.

3.1 Application Programmer Interface (API)

ALPyNA is intended to be easy to use for a non-expert Python developer, i.e. an end-user programmer. A minimal setup process is required, as shown in Listing 2. A developer calls one function from ALPyNA to initiate static analysis of developer-specified 'plain old Python' kernels; this returns an ALPyNA runtime execution object. ALPyNA assumes loop-nests will always be enclosed within functions. This ensures developers are able to reference parallelized loops via named function abstractions. This workflow is similar to the OpenCL online compiler workflow.

3.2 Compiler Frontend

The compiler implementation takes as its input kernel functions written in plain old Python. These functions may have more than one dense linear loop nest specified in Python. It

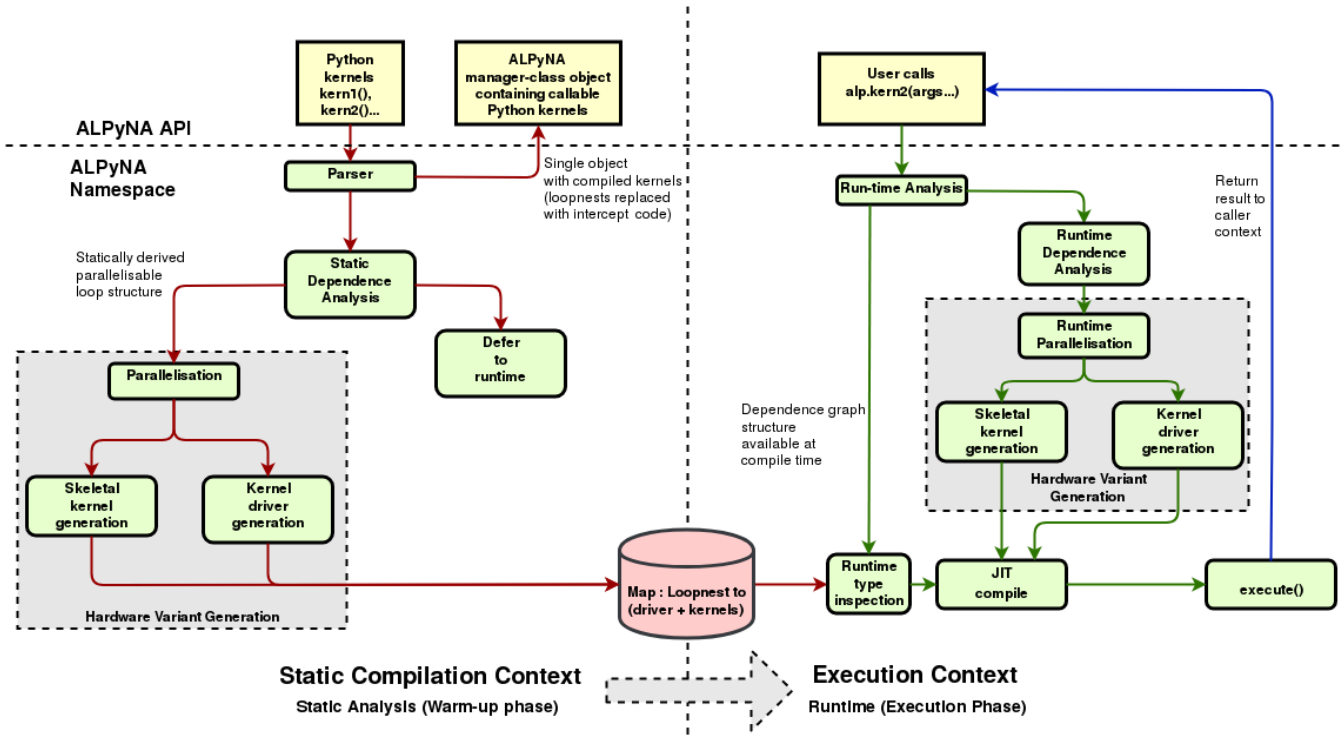

Figure 3. The ALPyNA system architecture is staged, with an ahead-of-time static analysis and a near-identical structure for the lazy dynamic analysis; note some information is preserved in memory from the initial stage.

is assumed that the developer will use Numpy arrays for vectors. Only basic subscripting of single or multi-dimensional arrays is supported, i.e. no slicing or sequence indexing.

Dereferencing the vector subscripts should evaluate to instances of the underlying *Numpy* `dtype` object or to scalar values. We further restrict ALPyNA to loop bodies that contain no control flow divergence.

The compiler takes dense loop nests as its unit of analysis. All other code constructs will be executed in the CPython interpreter as normal. This allows developers to intersperse loop nests with standard Python code, e.g. conditional execution constructs (*if/else* constructs that are not inside loop bodies).

To simplify analysis, loops are automatically rewritten in a normalised form, with an iteration increment of 1. Any loop bounds expressions, i.e. parameters of `range`, that are to be evaluated dynamically are hoisted above the loop and stored in temporary variables.

As shown in Figure 3, a call to the static analysis function parses the AST of all nested loops with a linear iteration domain. The subject functions are parsed using Python's AST library, to create a flat record structure consisting of 'loop landmarks,' i.e. fragments of abstract syntax that determine the looping behaviour. This landmark record structure is processed during the static analysis phase, allowing us to generate the subscript and variable pairings required to perform dependence analysis.

If all the loop bounds and data dependences can be determined statically, ALPyNA can generate the untyped GPU kernels (corresponding to the statements in the loop nest body) at compile time and cache these kernels in memory to reduce dynamic analysis time. In this case, with ahead-of-time generated kernels, we only need to patch the type information into the generated GPU kernels at runtime. Based on the structure of the dependence graph, kernels corresponding to the loop body statements are generated, along with a Python driver function, which invokes the GPU kernels.

On the other hand, if the static analysis cannot determine loop bounds at compile time, then it will mark the loop nest for dependence analysis at runtime.

The landmark record structure of the Python AST along with loop nests marked by ALPyNA for deferred analysis are preserved as in-memory data structures, carried over to the runtime execution context to aid dynamic dependence analysis. At runtime, parallelization is again performed on loop nests marked for deferred analysis. This happens lazily, upon invocation of a particular loop nest within an ALPyNA target function.

3.2.1 Runtime Type Determination

We determine the types of the vectors relevant to each GPU kernel at runtime, using Python's introspective facilities. The types are patched into the structure of the GPU kernels. These kernels may have been generated statically, or lazily when the function is invoked at runtime.

While our compiler restricts its analysis to dense loops with linear numeric iteration domains in a kernel, other expressions and statements within the kernel function that are not enclosed in a loop-nest are left to execute in the interpreter.

3.3 Compiler Backend

3.3.1 Runtime GPU Thread Organization

Most modern GPUs use a Single-Instruction-Multiple-Threads (SIMT) approach to execute large numbers of threads in parallel. They have a thread organizational hierarchy reflecting the underlying architecture of the GPU. The CUDA paradigm partitions threads into *blocks* and *grids*. ALPyNA exploits the dynamic introspection capabilities of the CPython interpreter to extract loop bound values immediately prior to loop nest invocation and execution. This is done by inspecting the binding of the result of the loop bounds expression, just prior to the execution of the loop. All the loops that can be run in parallel are mapped to each of the GPU axes and all other parallel loops are executed sequentially within each kernel.

3.3.2 Numba

ALPyNA uses Numba [13] to finalize and compile its automatically generated GPU kernels. Numba is an *LLVM* based compiler for Python functions. It is invoked by applying the *@jit* decorator to specific functions. To write code targeting the GPU, Numba uses the decorator syntax to compile kernels written in a tightly restricted subset of Python. These kernels have specific intrinsics that map to the GPU *grid*, *block* and *synchronize* programming primitives in the CUDA paradigm. These mappings directly refer to the CUDA primitives used to identify the thread hierarchy within the kernel.

While Numba allows runtime type inference, it does this on every invocation to the kernel. In the context of ALPyNA, consequent upon the dependences discovered in a loop nest structure, a kernel might have to be executed sequentially. This happens in the case of a loop-carried dependence. When a kernel is invoked multiple times sequentially, Numba's auto-typing feature re-compiles the code upon each kernel invocation. This slows down execution time by an order of magnitude[1]. To prevent this, ALPyNA applies discovered types to the kernel *once only* at the loop nest level. This allows Numba to cache the compiled kernel for further use over every invocation within a loop.

[1] measured on the Desktop platform specified in evaluation, Section 5

4 Benchmarks

ALPyNA is evaluated using four well-known array-intensive benchmarks expressed in 'plain old Python' as nested linear loops. The benchmark kernel is executed with a range of inputs. The execution time is directly related to iteration domain size. We measure time taken for:

1. dependence analysis and kernel generation
2. GPU kernel compilation
3. execution time for generated code on GPU

This total time is compared with the time taken by equivalent code executed within the CPython interpreter.

Naïve Matrix Multiplication operates on two dense matrices, represented as two-dimensional arrays of floating-point values. The naïve approach to matrix multiplication involves a triple nested *for*-loop, to iterate over rows and columns and compute a dot-product for each result element. While other algorithmically efficient variations of matrix multiplication exist, they are used in the context of overloaded operators for specific Matrix representations. The absence of the *k-loop* iterator in any of the subscript pairs that have to be checked for dependence generates all three dependence types namely *(i) True dependence(RAW [2]), (ii) Anti dependence (WAR [3])* and *(iii) Output dependence (WAW [4]).* Hence the *k*-loop has to be run sequentially (Listing 3).

Saxpy is Single precision AX plus Y. This benchmark combines scalar multiplication and vector addition on two equally sized linear arrays of 32-bit floating point values. Mathematically, the computation is represented by $\alpha\vec{x} + \vec{y}$ (Listing 4).

Jacobi Relaxation Core is an iterative solution for a set of linear equations, expressed as a matrix/vector product equation. Approximate trial values are used initially, and refined to reduce the error terms. The algorithm is iterated until it converges on a solution. This benchmark is the core of the iteration step, a doubly nested loop to compute the next value and the error value for each element in the two-dimensional matrix (Listing 5).

Conway's Game of Life is a zero-player game on a two-dimensional board, representing a cellular automaton. Each element is either alive or dead. At each turn of the game, elements are born, survive, or die, based on their neighbouring element's state at the previous turn. This benchmark is the core of the survival calculation, representing a single turn in the game. Effectively, it is a stencil computation across a two-dimensional integer matrix (Listing 6).

[2] Read-after-Write
[3] Write-after-Read
[4] Write-after-Write

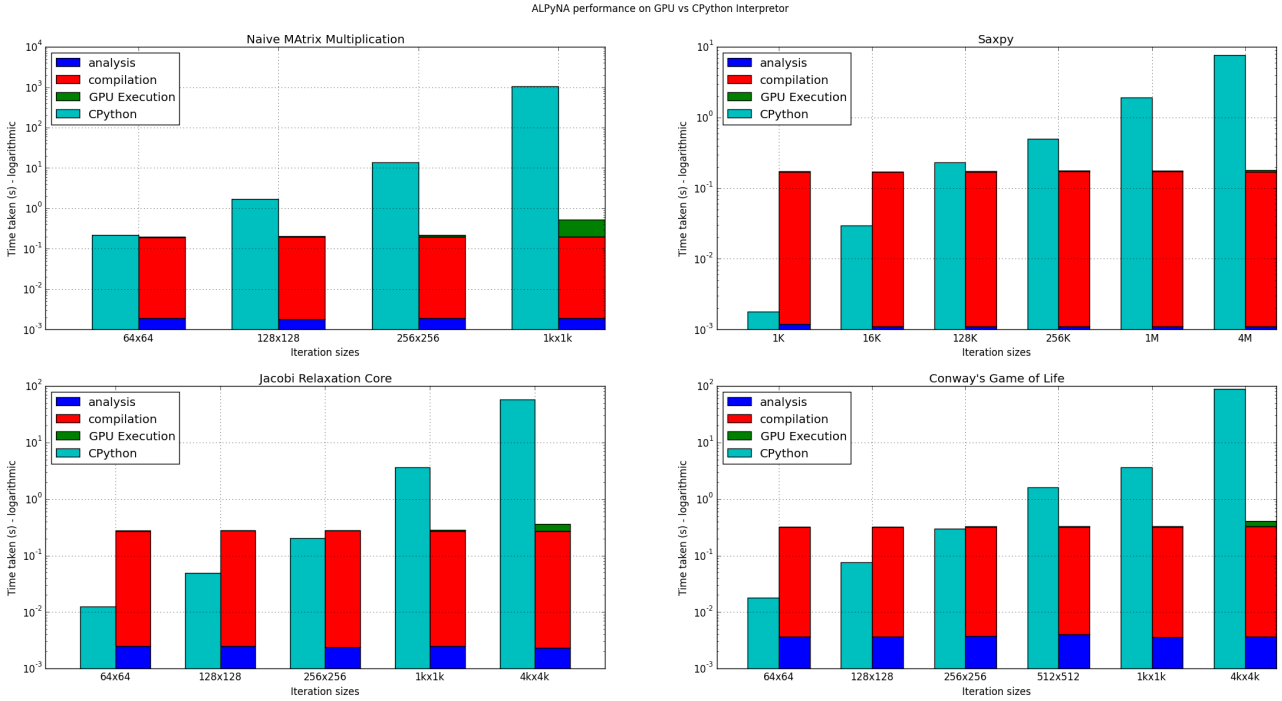

Figure 4. ALPyNA benchmark execution times (*lower is better*). Note logarithmic y-scale is used due to the magnitude of difference in execution time across different input sizes. GPU execution times are much lower in general. When time for code analysis and kernel compilation are factored into consideration, GPU offload become profitable for larger iteration domain spaces (towards right hand side on each graph).

5 Evaluation

5.1 Experimental Setup

The experimental platform used for evaluating the benchmarks is a typical commodity desktop computer. It has an Intel Core i7-6700 quad-core CPU with Simultaneous Multithreading (SMT) enabled and a L3 Cache of size 8MB. The maximum clock frequency is 3.4GHz. The memory fitted to this machine is 16GB ($2 * 8$GB) of 2133MHz DDR-4 RAM.

The GPU used to perform the benchmarking is an Nvidia GeForce GTX-1060. It has a maximum frequency of 1.7GHz with 3GB on-board GDDR-5 RAM. The SIMT compute hardware of the GPU is laid out as nine Streaming Multiprocessors (SM), each holding four partitioned SIMT units. Each SIMT unit has 32 cores that are scheduled simultaneously as well as its own individual *warp* [5] scheduler. This provides for a total of 1152 cores across nine SMs. The graphics card is connected via PCI-Express (PCI-E 3.0).

All experiments are run using native x86_64 Linux, kernel version 4.9. ALPyNA is evaluated with CPython version 3.5.3, with Numpy version 1.13.3. The runtime code generation uses Numba version 0.33.0. Numba itself hooks into an underlying GPU code generator—we use Nvidia CUDA compute version 8.0.44.

5.2 Performance

The benchmarks are executed over a range of iteration domain sizes. The *Saxpy* benchmark has a one-dimensional iteration space. The *Jacobi Relaxation Core* and *Conway's Game of life* benchmarks both have a two-dimensional loop iteration space while *Matrix Multiplication* has a three-dimensional loop iteration space.

Figure 4 plots CPython and ALPyNA runtimes for different iteration domains. To account for the large differences in execution times between CPython and ALPyNA, the time axis is logarithmic. To compare the effectiveness that adoption of ALPyNA might have in the real world, we compared actual time taken by the CPython interpreter against total execution time taken by ALPyNA to execute the benchmark. This includes the overhead of *analysis* and *compilation*, as well as and actual *execution* of GPU kernels. The execution times shown are inclusive of data transfer time.

ALPyNA generates the skeleton of the GPU kernels at compile time, if all the loop bounds can be resolved at compile-time. In such a scenario, only the type-inspection is done at

[5] Nvidia's term for smallest single schedulable unit of threads

runtime. To reflect more real-world scenarios and to exercise ALPyNA's runtime dependence analysis and subsequent GPU kernel generation, the benchmarks were written with dynamic loop-limits

Each benchmark is an average of five runs. For the smaller iteration domains (*i.e.* 64x64 for Matrix-Multiplication, 256x256 for Jacobi Relaxation Core and Conway's Game of Life and 256K for Saxpy), we found a +/- 10% variability amongst the run-time analysis and execution times for both the execution on the CPython interpreter and the GPU. For all the other iteration domain sizes that we tested, variability reduced from +/- 5% to almost 0%, inversely proportional to increasing iteration domain size. The variability of the GPU compilation times was always constant at +/- 10%. This can be explained by observing that the number of kernels to be compiled for these benchmarks remain constant irrespective of iteration domain sizes.

All benchmarks executed on the CPython interpreter were observed to run at the maximum CPU frequency of 3.4 GHz. This is due to the CPython interpreter running in single-threaded mode which caused only one CPU core to be fully utilized, leaving the others idle. Thus no throttling of the CPU due to potential thermal issues was noticed.

For GPU execution of the kernels, compiling all the relevant kernels for each loop nest dominated total execution time. However, the compilation time is relatively constant for all iteration domain sizes. This is due to the fact that these are all light-weight kernels that are equal to the number of statements in the loop body. Only the structure of each kernel would change if the dependence graph varied due to runtime factors.

All the benchmarks show that there is a crossover input size threshold, above which GPU execution becomes profitable. Since:

1. kernel compilation overhead is constant *and*
2. execution times on the GPU (for parallelizable code) are orders of magnitude lower than time taken to execute on the interpreter,

we can surmise that code executed on the GPU will be faster for loop iteration domains that are large enough to amortize the cost of GPU kernel compilation.

6 Related Work

Sheffield *et al* [19] describe *Three fingered Jack(TFJ)*, a system that uses loop dependence analysis to parallelize linear Python *for*-loops and generate code for FPGAs. They build on the *Copperhead* compiler system [6] to do static compilation of nested loops for which loop-bounds are known ahead of time. It can also only compile for known fixed types.

ALPyNA can also vectorize loop nests with known loop-bounds at compile time. However, unlike *Three Fingered Jack*, we may also defer this dynamically to runtime when the loop-bounds are not known at compile time. This allows us to potentially run more work in parallel. Additionally our system can also discover the types of the vectors at run-time through using the introspection capabilities of the CPython virtual machine.

Caamaño *et al* [4] describe the working of a runtime optimising polyhedral compiler called *Apollo*. To reduce complexity, they analyze small windows of *LLVM*-IR and statically generate variants of code for which dependences cannot be known until runtime. These are speculatively executed, with relevant guard conditions checking the accuracy of the speculation. It falls back onto a known correct point when a mis-prediction is detected.

Apollo targets CPU based loop parallelization; on the other hand, ALPyNA targets heterogeneous architectures. Unlike Apollo, ALPyNA does not *speculate* and *recover*. We introspect the values of the loop-domains and the types of the data from the runtime environment to analyze the structure of the loop and concomitantly, we generate the structure of the kernels based on structure of the dependence graph discovered at runtime.

Tornado, described by Clarkson *et al* [7], uses a system of annotations in code for Java. The developer annotates functions containing loop-nests with annotations like *@parallel* or *@reduce*. To generate parallel code, the number of dimensions to parallelize are specified by the developer. A Directed Acyclic Graph representation is used to describe the dependences between kernels. Tornado does not attempt to discover any parallelism in the developer code, but relies on guarantees provided by the developer to parallelize loops while ALPyNA discovers dependences in the structure of the code itself.

Loo.py [11] is an eDSL that can perform dense loop-nest array manipulation. It relies on the developer specifying the loop-bounds and operations to be executed in a descriptive way. Various loop-level transformations in the polyhedral mould are provided as a library of optimizations for the developer to use on the specified operations. Code is generated for backends once all the specified transformations are applied to the operations described in the DSL. *Loo.Py* is a code-generator aimed at the expert developer while the aim of ALPyNA is to allow the expression of code in regular Python, while still providing significant speed-ups.

7 Conclusions

With *ALPyNA*, we have evaluated the possibility of parallelizing standard Python loops and make a case for its

- **productivity**: by showing that end-user developers need not be aware of the precise low-level programming paradigm that is required to extract performance from the GPU.

- **portability**: by maintaining the original valid Python code provided by the programmer which we may execute in the interpreter if this code cannot be transferred to the GPU.
- **dynamic analysis capability**: by using the introspection capabilities of the interpreter during JIT analysis. This enables us to vectorize to the maximum possible extent that is allowed by dependence relationships identified by runtime analysis.

We have shown up to 1000x performance increase relative to the time taken by the CPython interpreter for workloads where the iteration domain space is large enough to amortize the (*i*) analysis of the loop structure, (*ii*) generation of requisite kernels and corresponding driver code and (*iii*) compilation of the required accelerator code.

A Appendix

A.1 Code Listings

Listing 3. Naïve Matrix Multiply

```
def matmul( mat_a , mat_b , mat_c ):
  ma_rmax, ma_cmax = np.shape( mat_a )
  b_rmax, mb_cmax = np.shape( mat_b )
  for k in range(ma_cmax):
    for i in range(ma_rmax):
      for j in range(mb_cmax):
        mat_c[i][j] = mat_c[i][j] + mat_a[i][k] *
            mat_b[k][j]
```

Listing 4. Saxpy

```
def saxpy( arr_y , arr_x , constval ):
  for idx_i in range(len(arr_y)):
    arr_y[idx_i] = arr_y[idx_i]
                   + constval * arr_x[idx_i]
```

Listing 5. Jacobi Relaxation Core

```
def jacobi_relax_core( next_x , curr_x , err):
  i_max , j_max = np.shape( curr_x)
  for i in range(1,i_max-1):
    for j in range(1,j_max-1):
      next_x[i][j] = 0.25 * ( curr_x[i][j+1]
                              + curr_x[i][j-1]
                              + curr_x[i-1][j]
                              + curr_x[i+1][j])
      err[i][j] = next_x[i][j] - curr_x[i][j]
```

Listing 6. Conway's Game of Life

```
def conway( curr , nxt , size ):
 for i in range(1,size-1):
   for j in range(1,size-1):
     # first count number of live neighbours (
         between 0 and 8)
     nxt[i][j] = curr[i-1][j-1] + curr[i-1][j] +
                 curr[i-1][j+1] + curr[i][j-1] +
                 curr[i][j+1]   + curr[i+1][j-1] +
                 curr[i+1][j] + curr[i+1][j+1]
     # next cell is live if curr is dead
     # but has 3 live neighbours,or if curr
     # is alive and has 2 or 3 live neighbours
     nxt[i][j] = ((~(-curr[i][j]) & nxt[i][j])
                 | ((curr[i][j]*nxt[i][j])|curr[i
                     ][j]))==3
```

Acknowledgments

This material is based upon work supported by the Engineering and Physical Sciences Research Council under Grant No. EP/L000725/1. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the EPSRC.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*. 265–283.

[2] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A Padua. 1993. Automatic program parallelization. *Proc. IEEE* 81, 2 (1993), 211–243.

[3] Margaret Burnett, Curtis Cook, and Gregg Rothermel. 2004. End-user software engineering. *Commun. ACM* 47, 9 (2004), 53–58. https://doi.org/10.1145/1015864.1015889

[4] Martinez Caamaño, Juan Manuel, Manuel Selva, Philippe Clauss, Artyom Baloian, and Willy Wolff. 2017. Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience* 29, 15 (2017), e4192. https://doi.org/10.1002/cpe.4192

[5] Stephen Cass and Parthasaradhi Bulusu. 2018. IEEE Spectrum Top Programming Languages Survey. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018. Accessed: 2019-04-03.

[6] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2011. Copperhead: Compiling an Embedded Data Parallel Language. *SIGPLAN Not.* 46, 8 (2011), 47–56. https://doi.org/10.1145/2038037.1941562

[7] James Clarkson, Christos Kotselidis, Gavin Brown, and Mikel Luján. 2017. Boosting Java performance using GPGPUs. In *International Conference on Architecture of Computing Systems*. 59–70.

[8] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 60–73. https://doi.org/10.1145/3050748.3050761

[9] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. 1991. Practical Dependence Testing. *SIGPLAN Not.* 26, 6 (May 1991), 15–29. https://doi.org/10.1145/113446.113448

[10] Ken Kennedy and John R Allen. 2001. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc.

[11] Andreas Klöckner. 2014. Loo. py: transformation-based code generation for GPUs and CPUs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. 82. https://doi.org/10.1145/2627373.2627387

[12] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.* 38, 3 (2012), 157–174. https://doi.org/10.1016/j.parco.2011.09.001

[13] 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*. 7. https://doi.org/10.1145/2833157.2833162

[14] Johnny Wei-Bing Lin. 2012. Why Python is the next wave in earth sciences computing. *Bulletin of the American Meteorological Society* 93, 12 (2012), 1823–1824. https://doi.org/10.1175/BAMS-D-12-00148.1

[15] Stefan C Müller, Gustavo Alonso, and André Csillaghy. 2014. Scaling Astroinformatics: Python + Automatic Parallelization. *IEEE Computer* 47, 9 (2014), 41–47. https://doi.org/10.1109/MC.2014.262

[16] Thomas P Robitaille, Erik J Tollerud, Perry Greenfield, Michael Droettboom, Erik Bray, Tom Aldcroft, Matt Davis, Adam Ginsburg, Adrian M Price-Whelan, Wolfgang E Kerzendorf, et al. 2013. Astropy: A community Python package for astronomy. *Astronomy & Astrophysics* 558 (2013), A33. https://doi.org/10.1051/0004-6361/201322068

[17] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. 2012. Parakeet: A Just-in-time Parallel Accelerator for Python. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*. 14–14.

[18] Christopher Scaffidi, Mary Shaw, and Brad Myers. 2005. Estimating the numbers of end users and end user programmers. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. 207–214. https://doi.org/10.1109/VLHCC.2005.34

[19] David Sheffield, Michael Anderson, and Kurt Keutzer. 2012. Automatic generation of application-specific accelerators for FPGAs from Python loop nests. In *Proc. 22nd International Conference on Field Programmable Logic and Applications*. 567–570. https://doi.org/10.1109/FPL.2012.6339372

[20] Jeet Sukumaran and Mark T Holder. 2010. DendroPy: a Python library for phylogenetic computing. *Bioinformatics* 26, 12 (2010), 1569–1571. https://doi.org/10.1093/bioinformatics/btq228

[21] TIOBE. 2018. TIOBE index. http://www.tiobe.com/tiobe-index/python/. Accessed: 2019-04-03.