Jacob, D., Trinder, P. and Singer, J. (2019) Python Programmers Have GPUs Too: Automatic Python Loop Parallelization with Staged Dependence Analysis. In: DLS 2019, Athens, Greece, 20-25 Oct 2019, pp. 42-54. ISBN 9781450369961 (doi:10.1145/3359619.3359743).

There may be differences between this version and the published version. You are advised to consult the publisher's version if you wish to cite from it.

© Association for Computing Machinery 2019. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proceedings of DLS 2019, Athens, Greece, 20-25 Oct 2019, pp. 42-54. ISBN 9781450369961.

http://eprints.gla.ac.uk/193798/

Deposited on: 23 August 2019

Enlighten – Research publications by members of the University of Glasgow
http://eprints.gla.ac.uk

Python Programmers have GPUs too

Automatic Python Loop Parallelization with Staged Dependence Analysis

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

Phil Trinder
School of Computing Science
University of Glasgow
Glasgow, UK
phil.trinder@glasgow.ac.uk

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

Abstract

Python is a popular language for end-user software development in many application domains. End-users want to harness parallel compute resources effectively, by exploiting commodity manycore technology including GPUs. However, existing approaches to parallelism in Python are esoteric, and generally seem too complex for the typical end-user developer. We argue that implicit, or automatic, parallelization is the *best* way to deliver the benefits of manycore to end-users, since it avoids domain-specific languages, specialist libraries, complex annotations or restrictive language subsets. Auto-parallelization fits the Python philosophy, provides effective performance, and is convenient for non-expert developers.

Despite being a dynamic language, we show that Python is a suitable target for auto-parallelization. In an empirical study of 3000+ open-source Python notebooks, we demonstrate that typical loop behaviour 'in the wild' is amenable to auto-parallelization. We show that *staging* the dependence analysis is an effective way to maximize performance. We apply classical dependence analysis techniques, then leverage the Python runtime's rich introspection capabilities to resolve additional loop bounds and variable types in a just-in-time manner. The parallel loop nest code is then converted to CUDA kernels for GPU execution. We achieve orders of magnitude speedup over baseline interpreted execution and some speedup (up to 50x, although not consistently) over CPU JIT-compiled execution, across 12 loop-intensive standard benchmarks.

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

DLS '19, October 20, 2019, Athens, Greece
© 2019 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '19), October 20, 2019, Athens, Greece*, https://doi.org/10.1145/3359619.3359743.

ACM Reference Format:
Dejice Jacob, Phil Trinder, and Jeremy Singer. 2019. Python Programmers have GPUs too: Automatic Python Loop Parallelization with Staged Dependence Analysis. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '19), October 20, 2019, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359619.3359743

1 Introduction

No matter which programming language popularity index you consult [9, 37], Python is one of the most widely used languages at present [3]. Python programmers operate across a broad spectrum of application domains, from archaeology [29] to zoology [12] with particular emphasis on data science [23] and machine learning [26]. One notable strength of Python is its accessibility to end-user developers [7, 15] which in part accounts for its popularity. The growing adoption of the interactive computational notebook as a programming environment also facilitates ease of use [35].

In general, standard Python programs are executed sequentially, in the CPython interpreter. The global interpreter lock [4] means programs run with single-threaded performance characteristics. Parallel resources are now available as commodity hardware, whether in multicore processors or GPUs. Given the ubiquity of GPUs, it is reasonable to assess whether we can take advantage of them when executing Python programs.

1.1 Our Contributions

We show in Section 2.1 that end-user developers write Python code in a style that is amenable to loop dependence analysis. Classical loop parallelism techniques were pioneered by Allen and Kennedy [18] in their optimizing compilers for Fortran.

In this work, we advocate *implicit parallelization*, i.e. auto-parallelization [31], for end-user programmers. Many developers don't use current explicit approaches to parallelism in their code; we argue that implicit parallelism is more in the spirit of Python programming, Section 2.2.

The main novelty of our work springs from applying known dependence analysis and parallelization techniques in a single platform staged across static and dynamic analysis phases. Section 3 demonstrates how the dynamic nature of

Python enables *staging* dependence analysis, with some happening ahead-of-time and the remainder happening just-in-time. We analyze the overheads of this dependence analysis.

Section 4 describes ALPyNA, our parallelization framework which is integrated with the Python runtime; it relies on introspective features of Python and targets CUDA GPU. We compare our new framework with the default CPython interpreter and a state-of-the art CPU JIT code generator across a range of benchmarks, as explained in Section 5.

Section 6 reports significant speedups in a number of cases, although this depends heavily on the benchmark loop structure, its dynamic dependence relationships, and the input data size.

Finally, Section 8 identifies the need for runtime decision making, i.e. a cost model that facilitates intelligent selection of backend code generator based on features of each program and its input data.

1.2 Relation to Previous Work

We have already published a brief description and preliminary study of an earlier version of ALPyNA [17]. Since that report, we have extended the framework to handle a richer set of loop nest semantics. These include support for control flow divergence (via synthesized if conditions, Section 4.1.1) and *pure* (*i.e.* side effect free) *intrinsic* functions (like `exp`, `sin` and `sqrt`, see Section 4.7) which map directly from Python onto the CUDA Math API. Additionally, we describe how staged analysis of variable loop bounds and/or loop-invariant variables in subscripts generates optimal kernels (Section 3.1).

ALPyNA has been extended to perform runtime evaluation and optimized kernel generation on each individual loop nest within a function using closures (Section 4.2). In Section 4.1.2, we describe how ALPyNA partially evaluates and caches the results of dependence analysis for use in the hybrid static/dynamic dependence analysis. We also describe how ALPyNA marshals scalar variables that are targets of writes in GPU kernels (Section 4.5).

The present paper gives detail regarding the motivation and implementation, as well as a more extensive empirical evaluation, using 12 benchmarks (cf. four in [17]); the majority of extra benchmarks require the richer set of Python features now supported in ALPyNA.

2 Motivation for Parallelization

2.1 End-Users like Loops

If end-user developers employ loops (and particularly, nested loops) as common control-flow constructs in their code, then we can treat these as appropriate targets for optimization. On the other hand, if there are few loops in end-user code, then parallel optimizations would not bring noticeable performance benefits.

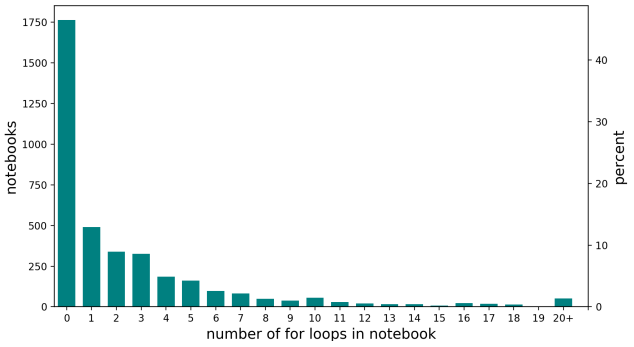

Figure 1. Histogram showing number of `for` loops per notebook, for sample of end-user code

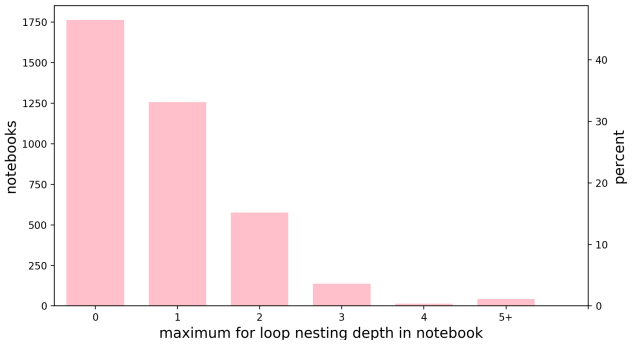

Figure 2. Histogram showing maximum `for` loop nesting depth per notebook, for sample of end-user code

We analyse a large corpus of Jupyter notebooks, downloaded from github in a recent study [35]. We explore the sub-corpus which is 'Component 1 of 13' in the original study; it comprises over 6K notebooks, of which our Python 3 parser could parse 3792 (others were written in Python 2). These code samples are representative of end-user programming style, perhaps skewed to the scientific domain—for more details, refer to the original study [35].

We examined the number of distinct `for` loops in each notebook, and the maximum `for` loop nesting depth in each notebook. Figure 1 shows a histogram of the number of `for` loops per notebook: over 50% of the notebooks contain at least one loop. Figure 2 shows the maximum `for` loop nest depth per notebook: the median depth is 1, and the maximum depth is 7.

This preliminary study confirms that there is potential for performance gains from loop-based parallelism in end-user code. The next question is: what should such parallelism look like, for Python programmers?

2.2 Pythonic Parallelism

The adjective 'Pythonic' describes elegant Python code, as agreed by the community of practice. This philosophy is encapsulated in the PEP-20 document [27] which is available

inside the Python interpreter as an 'easter egg'—simply enter `import this` to see the complete list of guidelines.

We draw on these principles to specify our parallelization system for Python, which targets commodity GPUs to execute loop nests. This section explains how we interpreted PEP-20 in our context.

Beautiful is better than ugly. Simple is better than complex. Readability counts. We advocate implicit, i.e. automatic, parallelization. Sequential code is analysed and transformed into GPU code without explicit rewriting by the user. This ensures code should never become 'ugly' or 'complex' due to parallel refactoring or library calls.

Explicit is better than implicit. We are doing implicit parallelism, for sure; however, we only parallelize loop nests with integer `range` iteration spaces. In that sense, the user (who should be aware of this semantic restriction) knows which regions of code may be appropriate for parallel execution. Further, the user directly identifies each potential target for parallelization with a simple call to the framework. We treat parallelization like memory management, as a runtime concern that the developer does not need to handle explicitly; instead the developer devolves responsibility to the Python execution engine. This is a case where **Practicality beats purity**.

Errors should never pass silently. If a loop nest cannot be parallelized, then the system reports the error. There may be a dependence violation, or the presence of Python structures we cannot handle (e.g. function calls in loop bodies). Another cause of failure is that the dynamic data structure the loop nest iterates over may be too large to fit into GPU memory. In each case, at the appropriate stage, the framework should report an error to the user.

Now is better than never. This is the rationale for our just-in-time resolution of loop bounds and value types. While there is some runtime overhead to this deferred analysis, it provides more accurate dependence resolution and more efficient GPU code.

Namespaces are one honking great idea. We can use namespaces to manage multiple runtime variants of a single loop nest source fragment, perhaps targeting different backends including GPU.

3 Background

Allen and Kennedy [18] define the existence of a data dependence relationship between two statements in a region of code *iff* both statements access the same memory location and at least one of the operations is a write. When a dependence occurs in a loop nest and the *source* and *sink* of the dependence occur on two different instances of the loop execution, the outermost loop causing the dependence *carries* the dependence. *True* (read-after-write), *anti* (write-after-read) and *output* (write-after-write) dependences are denoted by δ_i, δ_i^{-1} and δ_i^o respectively, where i may denote either the numeric nesting level or the named iterator variable of the loop carrying the dependence. A loop independent dependence is denoted by δ_∞.

3.1 Benefits of Deferring to Runtime

Consider the code in Listing 1. Classical dependence analysis would classify the dependence relationship as a Strong SIV (*Single Index Variable*). Equation 1 shows the influence of the loop-invariant variable k on the distance and direction of cross-iteration dependences. The cross-iteration dependence reverses direction depending on whether k is positive or negative.

$$Dependence = \begin{cases} |k| \geq arr\_len, & no\ dependence \\ |k| = 0, & \delta_\infty \\ |k| < arr\_len \ \wedge \ k > 0, & \delta_i \\ |k| < arr\_len \ \wedge \ k < 0, & \delta_i^{-1} \end{cases} \quad (1)$$

Listing 1. Benefit of runtime parallelization

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

Conventional static analysis has to assume conservatively that a loop-carried dependence exists and execute the loop sequentially. For simple loops, sequential and parallel *variants* of the code could both be generated and executed subject to the guard conditions being satisfied dynamically.

For larger loop nests, speculative generation of such variants becomes increasingly expensive. However, deferring the analysis until runtime can yield an optimized parallel version of a loop nest once loop iteration domains are known.

Listing 2. ALPyNA example

```
def ln_func(arg_a,arg_b,constants,limits) :
 im, jm, km, mm = limits
 p1, p2, p3 = constants
 for i in range(0,im,1):
   for j in range(2,jm,1):
     for k in range(0,km,1):
       for m in range(0,mm,1):
         # Statement - S1
         arg_a[i+p1,j,k,m] = arg_a[i,j,k,m] + 4 +
             arg_b[i]
         # Statement - S2
         arg_a[i,j+p2,k,m] = arg_a[i,j+p3,k,m] +
             43
```

To demonstrate the potential optimizations that can be unlocked due to a dynamic knowledge of loop bounds and/or loop-invariant subscript values, consider the code in Listing 2. The quadruple nested loop has limits (*im,jm,km,mm*) and three loop-invariant values (*p1,p2,p3*) within the subscripts. These unresolved values complicate the discovery of parallelism within the loop structure.

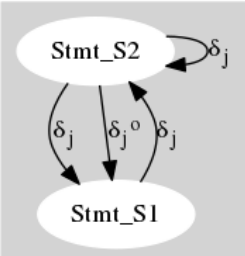

Figure 3. Dependence graph of loop nest in Listing 2 with iteration domain (*i,j,k,m*) → (10,100,100,100) and (*p1,p2,p3*) → (0,1,-2)

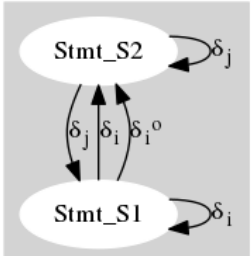

Figure 4. Dependence graph of loop nest in Listing 2 with iteration domain (*i,j,k,m*) → (10,100,100,100) and (*p1,p2,p3*) → (1,1,-1)

At runtime, assuming we resolve *(im,jm,km,mm)* → (10, 100,100,100) and the values within the subscripts are *(p1,p2,p3)* → (0,1,-2), we obtain the dependence graph as shown in Figure 3. The cyclic dependences between statements S1 and S2 require loop *j* to be executed sequentially while the (*i,k,m*) loops may be executed in parallel.

If the iteration domain space remains the same, but the runtime variable values change to *(p1,p2,p3)* → (1,1,-1), the dependence graph for the loop nest becomes as depicted in Figure 4. Although the dependence graph has more edges, we can safely execute statement S1 in parallel along the *j,k,m* axes if we run the *i*-loop sequentially. Statement S2 can be executed in parallel along two (*k,m*) dimensions if we execute the *i,j*-loops in sequential order to maintain dependence constraints. Similarly if the values *(p1,p2,p3)* → (10,99,-1), we find that both statements can be executed in parallel along all four loop dimensions. In summary, there may be significant benefit in deferring loop dependence resolution decisions until runtime, when precise dynamic values are known.

4 ALPyNA Architecture

Our ALPyNA loop parallelization framework is designed to generate performant GPU kernels from Python loop nests. It is not intended as a whole program compiler, but rather targets linear loop nests as the unit of analysis. Dependence analysis of the loops reveals opportunities for parallelism while still maintaining the ordering constraints expressed in the original computation.

The ultimate aim of ALPyNA is to generate code variants targeting various accelerators in heterogeneous systems. While ALPyNA currently targets GPUs and the CPython VM, it has a device abstraction layer that is modular and can easily be extended to JIT compilers for other hardware devices.

Instead of speculative generation of variants depending on a combination of iterator, subscript and hardware properties, ALPyNA uses a *staged* approach to parallelization. Figure 5 illustrates ALPyNA's staged compilation architecture. The left hand side outlines the static analysis and compilation while the right hand side outlines the runtime compilation. We describe each phase in the following sections, and discuss the underlying Numba compiler [21] (Section 4.6) and the current restrictions (Section 4.7).

4.1 Static Analysis

The programmer specifically targets a fragment of Python source code for ALPyNA optimization, by invoking a simple API call. The target Python code should be a compute-intensive array-centric nested loop.

Conditional execution of code within the body of a loop nest creates *control* dependences between statements within a loop body. To aid discovery of parallelism in the presence of control-flow dependence, ALPyNA first executes an *if-conversion* pass (Section 4.1.1).

Following common practice, loop iteration domains are normalized to aid dependence analysis. While parsing the code during the static analysis phase, any loop bounds expressions i.e. parameters of the `range` function) are parsed. If they cannot be symbolically evaluated, they are marked for runtime evaluation. If possible, any such expression is hoisted out of the loop nest.

After these initial passes, the Python Abstract Syntax Tree (AST) is parsed to generate a simple flat record structure to aid in loop analysis. This record structure comprises of information required for dependence analysis of the loop. These are further parsed to create 'loop landmarks', as well as group variable and subscript pairs.

If all the loop bounds and data dependences can be determined statically, ALPyNA generates untyped GPU kernels and caches these in memory to reduce dynamic analysis time (Figure 5). In such a scenario, only the type information is required to be patched into the cached untyped kernel, and this is obtained at runtime by using introspection (Section 4.3). Corresponding GPU kernel driver code that respects the loop carried dependence constraints is generated and cached. If any data dependence cannot be determined statically, the partial evaluation of dependences is cached (Section 4.1.2) to use during runtime evaluation of loop nest dependences.

Scalar variable writes in loop nests are a special case, requiring runtime analysis and code generation (Section 4.5).

The final stage of ALPyNA's static compilation phase is to prepare all data structures that are required for runtime evaluation, compilation and execution. ALPyNA replaces each loop nest with a closure (Section 4.2). These closures are given a unique handle for the ALPyNA runtime to dereference on demand.

ALPyNA generates a Python module on the fly and returns a handle to this module as a Python object to any code that calls the ALPyNA static analysis function. Any in-memory data structures generated so far within the static analysis phase and required for runtime evaluation are stored in this newly generated module within their own namespaces. A reference to the module is stored within itself to map the

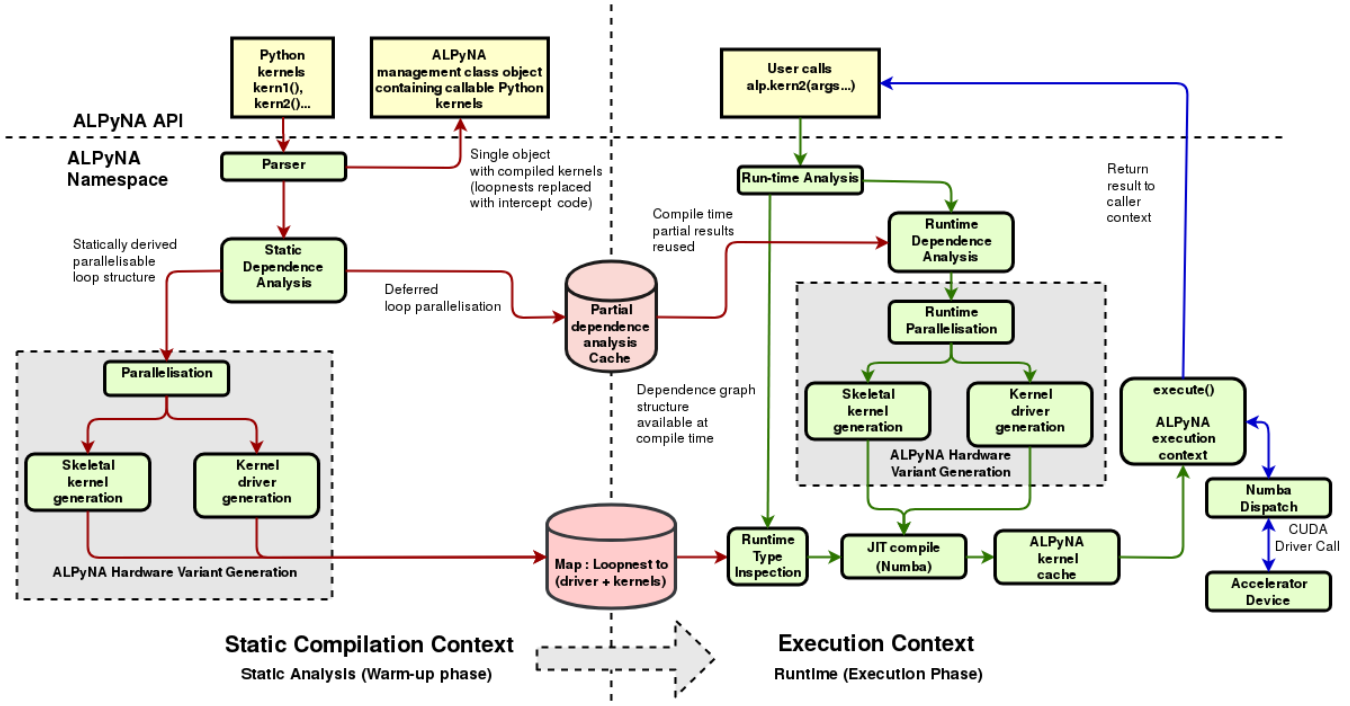

Figure 5. The ALPyNA system architecture is staged, with an ahead-of-time static analysis and a near-identical structure for the lazy dynamic analysis; note some information is preserved in memory from the initial stage.

function to its corresponding in-memory *partial evaluation cache*. At this point control is handed over to the ALPyNA runtime.

4.1.1 Conditional Code Execution

Control flow dependences within a loop nest cannot be modelled using conventional data dependence techniques. To model control flow dependences, all statements within a loop body are transformed to a predicated execution form [2].

Control flow divergence typically occurs due to the presence of `if` statements. A *forward* branch is a branch for which the target of the control flow jumps to a location within the same loop. A branch transferring control to a target outside the loop body is classified as an exit branch. This can typically be mapped onto the Python `break` statement.

Scalar expansion [25] is performed on compiler generated conditional variables to increase opportunities for parallelization. Predicate variables generated for forward branches are expanded to the dimensions of the loops enveloping its definition. As a memory optimization, *exit* branch predicate variables are expanded to a vector with the dimensions of size of the innermost loop that references the exit branch predication variable. This is due to the existence of a dependence between the compiler introduced exit branch predication variable carried by the innermost loop.

Expansion of compiler generated variables is simplified, as every read from such predicate variables is dominated by a single definition for *forward* branches and by two definitions in the case of *exit* branches.

4.1.2 Partial Dependence Analysis

To ensure that dependence analysis is safe, static compilers should make conservative assumptions about dependences if it cannot accurately resolve a dependence. To determine dependences in linear subscripts, loop limits must be available.

Another factor that hinders discovery of parallelism in linear array accesses is the presence of symbolic constants within array subscripts. A static optimizing compiler must conservatively assume loop carried dependences exist in both directions between two references to the array.

ALPyNA overcomes this hurdle by solving and caching statically resolvable dependence relationships in-memory in a *partial evaluation cache*. This cache is transferred to the runtime execution context after static analysis. At runtime, ALPyNA re-executes dependence analysis. Statically resolved dependence results are extracted from the partial evaluation cache. The resulting loop is parallelized at runtime according to the exact evaluation of loop boundaries and constants within subscripts. The dependence relationships that emerge at runtime can be optimized by opportunistically exploiting a loop structure more amenable to parallelization.

4.2 Closure Generation

ALPyNA replaces loop nests within the body of marked kernels with Python closures. The closure provides a handle object for the ALPyNA runtime to dynamically query loop limit expressions and loop-invariant constants within subscripts. In contrast, a purely static compiler would need to rely on conservative dependence analysis.

The closure marshals all the variables used by the loop nest into a variant selection function along with a reference to the closure. The closure also has access to the flat record structure of the AST and the partial evaluation cache (Section 4.1). ALPyNA uses the closure handle to introspect vectors and variables for their types and vector dimensions. Variable types discovered at runtime are patched into the generated kernels to create typed variants. Each compiled kernel is then cached by Numba, our backend GPU compiler (Section 4.6). We also introspect the binding of loop limit expression evaluations to inform runtime dependence analysis.

In general, a cost model is required to facilitate backend target selection and code variant generation, as discussed in Section 8.

4.3 Runtime Analysis

After ALPyNA has completed the static analysis phase of compilation, a reference to the generated ALPyNA module is returned to the caller. When a programmer dereferences a kernel in the ALPyNA module, the closure corresponding to the loop nest is called. If all the dependence relationships were resolved statically, loop nest variable types are inspected, patched into the statically generated kernel, compiled and executed.

Any loop nest with dependences unresolved at compile time is re-analyzed. All dependence relationships that were statically resolved are retrieved from the partial evaluation cache and augmented with dependence relationships determined at runtime.

This dependence graph is generated for each instantiation of a loop nest. GPU kernels are generated with the newly discovered dependence constraints. Resolution of the expressions which make up the loop domain are used to set the GPU *grid* and *block* sizes with suitable padding (Section 4.4).

4.3.1 Execution Context and Data Management

ALPyNA creates a light-weight kernel for each statement in the body of the loop nest. Dependence analysis evaluates ordering constraints between statements and determines which loops have to be executed sequentially. Kernels representing each statement are called in the topological order in which the dependence graph is parsed. The whole set of kernels are executed together on the GPU to avoid inter-kernel data transfer within the context of a loop nest.

Data transfer to and from the GPU is hoisted outside the loop nest context that represents the overall computation. Data transfer is done within the context of the closure (refer Section 4.2) that represents the loop nest computation.

4.4 GPU Thread Hierarchy

In CUDA terminology, a group of threads on the GPU constitute a *block* and a group of *blocks* constitute a grid. These thread groupings may be spread over *one, two* or *three* dimensions. Each GPU has minimum and maximum *threads-per-block* and in some cases, the maximum number of threads in one dimension may not be the same as others.

ALPyNA determines loop bounds statically by parsing the original code (Section 4.1) or dynamically using introspection within the context of the closure that represents the loop nest (Section 4.2). The bounds for each instantiation of a loop nest is determined by introspecting the evaluation of the bounds expression at runtime (as demonstrated in previous work [17]).

Dependence analyis [18] determines which loops should be executed sequentially for each statement. All other loops that envelope the original statement in the loop nest are executed in parallel within a GPU kernel. Each iteration of a loop executed in parallel is mapped on to a thread that can be calculated using a GPU's $(blockid_{axis} * blocksize_{axis} + threadid)$ semantics.

ALPyNA matches the number of *threads-per-block* and *blocks-per-grid* based on device specific constraints and the iteration domain sizes of a loop nest. Loop bounds for each parallel loop enveloping a statement determine how many threads are required to execute it in parallel. The loops that are determined to be safe to execute in parallel are sorted in descending order of their iteration domain sizes. Each parallel loop is assigned to one of the GPU's *thread axes*, up to the maximum number of axes (three in modern GPUs) supported by the device. Sorting the loops in descending order allows for maximising the parallelizable iteration domain space when the number of parallel loops are greater than the number of *thread axes* supported by the GPU.

If there are loop nesting levels greater than the number of axes allowed by the accelerator, any loops that can be run in parallel and have not been allocated to a GPU axis, are run sequentially within each kernel. This preserves correctness as, if we execute outer loops that carry dependences sequentially, all inner loops can be executed in parallel without causing a dependence violation.

To calculate the CUDA thread hierarchy for each instantiation of a loop nest, an initial block size equal to the iteration domain sizes of each parallel loop is generated. This allocation is chosen if it fits in the maximum block size of the GPU device. If the number of threads is greater than the maximum block size of the GPU, we iteratively halve and pad the number of threads along the largest of the allocated parallel axes in a block until the *block* size is small enough to fit the GPU's maximum *block* size. During each iteration, the *grid* size along the corresponding axes is adjusted to

match the reduction in the *block* size. We converge on the *grid* and *block* sizes in $\left\lceil log_2 \left(\frac{\prod_{i=1}^{n} D_i}{B} \right) \right\rceil$ iterations where D_i is the the domain size of a loop allocated to execute in parallel, B the maximum block size supported by the GPU and n is the number of parallel thread axes allocated; e.g. a $1k \times 1k$ domain size for *gemm* (Section 5.2), would converge to a thread hierarchy of $(grid, block) \rightarrow ((32 \times 32), (32 \times 32))$ within 10 iterations.

4.5 GPU Scalar Marshalling

The effect of a write to a scalar variable within a loop body must be preserved for subsequent reads, which may not be part of the same loop nest structure. However, GPUs do not support call-by-reference semantics to support statements where scalars are the target of a write. Instead, a scalar has to be transferred as a unit-size vector. Marshalling many such scalars as a vector incurs a large cost.

To mitigate this overhead, ALPyNA performs runtime introspection to determine variable types. For scalar values that are to be offloaded to GPU, we dynamically assemble one composite array per type. This composite array can be dereferenced within a runtime generated GPU kernel using *vector + displacement* semantics. The closure representing the loop nest unmarshals each composite array back into the appropriate original scalar variables upon the completion of a loop nest execution.

4.6 Numba

ALPyNA uses the Numba [21] high performance Python compiler 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. Numba also compiles kernels written using a restricted subset of Python which is very similar to CUDA-C.

ALPyNA relies on Numba's CUDA interface to serialise data and to transfer it to/from the GPU. Numba only supports Numpy arrays that are already byte aligned in memory for the array element type. Numba exposes bindings to CUDA intrinsics that map to the GPU *grid*, *block* and *synchronize* programming primitives to the programmer.

Unless the types are provided by the programmer, Numba applies automatic runtime type inference on every invocation of a kernel. If a kernel is executed multiple times, this would cause a compilation overhead which is an order of magnitude higher. To reduce this overhead, ALPyNA patches the runtime inferred type information to the kernels once per loop nest invocation. This allows Numba to cache the compiled kernel for further use in every invocation within an instance of a loop nest.

4.7 Restrictions

As ALPyNA uses Numba for GPU compilation and data transfer between RAM and GPU memory, currently only Numpy arrays and basic scalar types are allowed. Serialization of other Python objects for GPU execution is future work.

ALPyNA considers each loop nest within a function as a single unit for dependence analysis. Currently only intra-procedural dependence analysis is performed on the loop nest. Function calls within a loop nest allow for a richer representation of programs. Although ALPyNA now supports calls to pure intrinsic functions that are supported by Numba, these calls are not subject to inter-procedural analysis and their validity must be guaranteed by the user.

Each variable subscript must be a linear function of loop iterator values. ALPyNA only supports basic indexing so array slices are not allowed. All dereferencing of a Numpy vector read should evaluate to a value which has that vector's '*dtype*' and every write should access the location of a scalar within a Numpy vector. This precludes Numpy array broadcasting semantics. Currently only *Zero Index Variables* (ZIV) and *Single Index Variable* (SIV) dependence tests have been implemented. *Multiple Index Variable* (MIV) dependence solvers are planned to be added in the near future to solve complex linear functions of iterators.

Loop nest domains are currently expressed only using the `range` function. Other well understood non-mutable linear iterators such as `enumerate` are planned to be added for analysis. While generic Python statements are allowed within a computation kernel, these should not be within a loop nest. Such statements will be executed within the interpreter.

5 Experimental Setup

5.1 System Comparison

To assess the effectiveness of loop parallelization, ALPyNA is evaluated on 12 widely used array-intensive programs. The benchmarks are written as nested Python loops in an array-centric format with variable loop domains, to utilize ALPyNA's runtime analysis execution path. We measure the time taken by ALPyNA for:

- dependence analysis and GPU kernel generation
- GPU kernel compilation
- execution time of generated kernels

We compare total analysis and execution time against the CPython interpreted version of the same program, as well as a JIT-compiled version that targets native CPU execution.

The chosen benchmarks range from simple 1d parallelizable loops (vadd, saxpy) all the way to 7d loops (fbcorr). These benchmarks include those that are are parallelizable along all their loop domain axes (jacobi, mandelbrot, conway) and those which have loop-carried dependences (gemm, gemver, syr2k). The mandelbrot and black-scholes benchmarks use ALPyNA's *if-conversion* pass (Section 4.1.1) and support for

pure intrinsic functions. All the chosen benchmarks can be analyzed with ZIV and SIV subscript tests (Section 4.7).

5.2 Benchmarks

black-scholes implements an option pricing model [5] to calculate prices of derivative financial instruments. It uses various mathematical pure intrinsic functions on each the stock price and exhibits control flow divergence with forward branches (cf. Section 4.1.1).

conv-2d convolves an $N * N$ matrix with an $M * M$ kernel represented as

$$y[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j] \times h[m-i, n-j]$$

This is a quadruple nested loop with an $M * M * N * N$ iteration domain. The loops with dimensions $M * M$ must be executed sequentially, while the loops along the $N * N$ domain can be parallelized.

conway is a zero-player game on a 2d board, representing a cellular automaton. Each element is either alive or dead. At each turn, elements are born, survive, or die, based on neighbour elements' state from 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 2d integer matrix.

gemm is an implementation of the standard $O(n^3)$ dense matrix multiplication algorithm. The loops iterate over the rows and columns of the 2d matrices. The absence of one of the iterators in the access pattern of any subscript of the output matrix generates all three dependence types on the multiply-accumulate statement. This requires the inner *k-loop* to be executed sequentially while allowing the outer *i-* and *j-loops* to be run in parallel.

gemver is a BLAS [6] routine from the Polybench [30] suite. The mathematical calculation is:

$$\hat{A} = A + u1.v1 + u2.v2$$

$$x = \beta \hat{A}^T y + z \ , \ w = \alpha \hat{A} x$$

where inputs are A ($N * N$ matrix), α, β (scalars) and *u1,u2,v1, v2,y,z* (vectors each of size N). The benchmark consists of four separate loop nests, each with a single statement. All the statements have loop-independent dependences between them. This benchmark has a 1d loop and three 2d loops. One 2d loop can be parallelized across both dimensions. The other two 2d loops act as a reduction along one axis inducing all three dependence types along one axis. These loops can be parallelized across the remaining axis.

hilbert computes a 2d matrix used in linear algebra approximation problems. It is calculated as:

$$H_{i,j} = \frac{1}{i+j-1}$$

jacobi is an iterative algorithm to solve a set of linear equations, expressed as a vector product equation. Initial guesses for the solution are plugged into a vector representation. The terms are solved iteratively until the algorithm converges to 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 2d matrix.

mandelbrot is a kernel that computes $z_{n+1} = z_n^2 + c$, where z, c are complex numbers with the initial value $z = 0$. The conditional execution of a statement within the loop body requires an *if-conversion* pass.

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

syr2k is a BLAS [6] routine from the Polybench [30] suite. It computes

$$C_{out} = \alpha AB^T + \alpha BA^T + \beta C$$

where *A,B,C* are $N * N$ matrices, and α, β are scalars. Dependence analysis generates one statement that can be parallelized across an $N * N$ domain and a second statement with a loop-carried dependence along one axis (size N) which runs sequentially and parallelizable across two axes of size $N \times N$.

vadd performs element-wise addition of a pair of 1d vectors. The vector sizes are varied between 8MB to 128MB. The iteration domain size is proportional to the vector length.

fbcorr is the filterbank correlation, used in signal and image processing to classify features. The implementation has seven nested loops of which three must be run in sequential order. The other four loops can be executed in parallel. Current GPU hardware only supports three hardware axes. Since all dependences are carried by the sequentially executing loops, our optimizations will sort the parallel loops in descending order of iteration domain size at runtime. The loop with the smallest loop domain size is executed sequentially within each parallel thread.

5.3 Hardware Platform

All the benchmarks described in Section 5.2 are evaluated on a commodity desktop computer. The CPU is an Intel Core i7-6700 quad-core system, with Simultaneous Multithreading (SMT) enabled, L3 cache size 8MB, and a maximum clock frequency of 3.4 GHz. The machine has 16GB ($2 * 8$ GB) DRAM with a bus clock speed of 2133 MHz.

The GPU is an Nvidia GeForce GTX-1060 with 3GB of GDDR5 RAM, nine Streaming Multiprocessors, each with four partitioned *Single Instruction Multiple Threads* (SIMT) cores. A SIMT parallel core simultaneously executes a *warp* comprising 32 threads. Each SIMT core on the GTX-1060 has its own warp scheduler. The GPU runs at 1.5 GHz and can be boosted to a maximum 1.7 GHz. The GPU is connected to the CPU on a PCI-Express (PCI-E 3.0) bus.

All experiments are conducted on a native x86_64 Linux kernel (version 4.9). ALPyNA is evaluated on the CPython

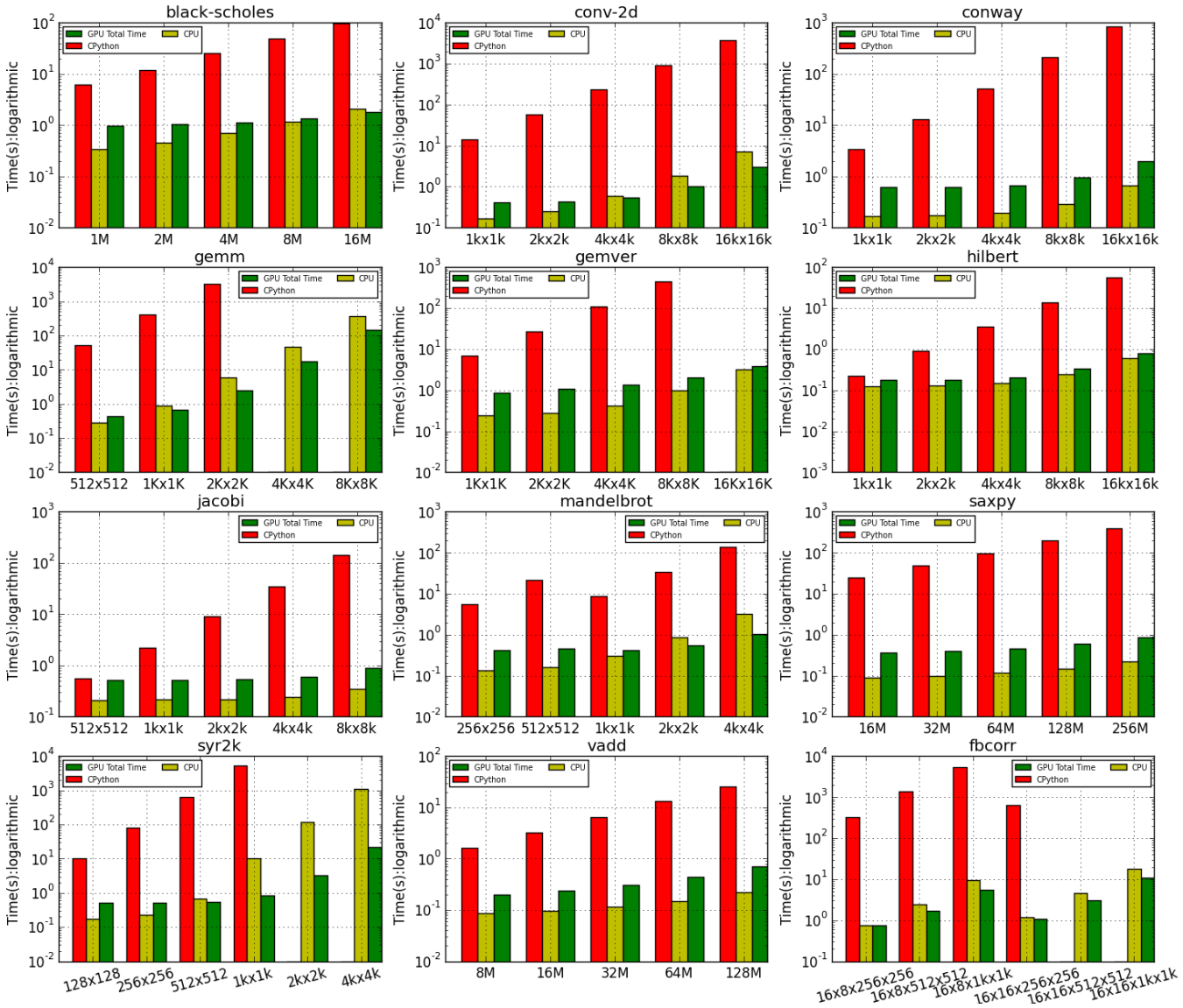

Figure 6. Execution times (*lower is better*) with CPython Interpreter, NUMBA JIT-compiler and ALPyNA for Python benchmarks at different input sizes. Logarithmic y-scale is used due to the orders of magnitude difference in execution times. When time for code analysis and kernel compilation are factored into consideration, GPU offload becomes profitable vs CPython at relatively small iteration sizes. In comparison with the CPU, GPU execution is profitable only at larger iteration domain spaces of kernels where the computation workload is heavy (towards right hand side on each graph).

interpreter (version 3.5.3) along with Numpy version 1.13.3. Numba (version 0.33.0) is used as the JIT compiler for both CPU and GPU variants of each benchmark. Numba uses a CUDA compiler to convert typed GPU kernels into binary format. Our setup has Nvidia CUDA compute version 8.0.44.

5.4 Methodology

Python generates a new object from a sub-slice of the original vector for every subscript dereference. This additional overhead skews the results in favour of both CPU and GPU compiled versions. To prevent this, all multi-dimensional subscripts use tuple notation in a single subscript; tuples in for Numpy arrays prevent temporary object creation.

Each reported execution time is the arithmetic mean of 10 runs. Each benchmark is executed with (i) CPython, (ii) a Numba generated CPU variant of the same code, and (iii) code generated by ALPyNA for the GPU. All benchmark executions are set to timeout after 1.5 hours; hence the interpreted versions of gemm, gemver, syr2k and fbcorr could not be measured for larger problem sizes (see Figure 6).

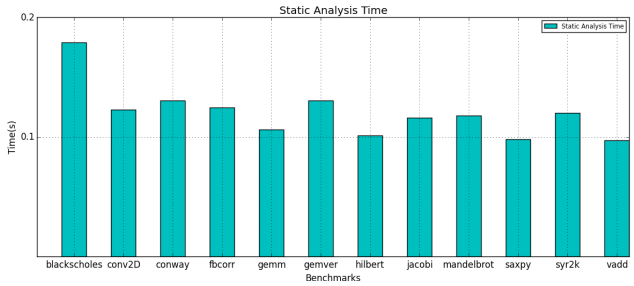

Figure 7. ALPyNA static analysis times for each benchmark.

ALPyNA's warm-up phase for each benchmark is measured to demonstrate the low overhead of static analysis. Figure 7 shows an overhead of around 0.1s for all benchmarks. The *black-scholes* analysis takes longest—there are a total of 66 subscript pairs (49 cross and 17 output variable subscript pairs) to analyze amongst 13 lines of code in the target loop nest. Conditional code within the loop body creates compiler generated predicate variables due to *if-conversion* which increase the time to do static analysis.

6 Evaluation

Figure 6 shows ALPyNA, CPython interpreted, and NUMBA JIT-compiled execution times for different input sizes over the 12 benchmarks. The plots use a logarithmic time scale due to the orders of magnitude difference between the CPython interpreter and the other versions of the code. Reported GPU performance with ALPyNA takes into account the overhead of analysis, JIT compilation and execution. This execution time includes the overhead of transferring data back and forth between host and GPU. A breakdown of *analysis*, *compilation* and *execution* times for four benchmarks is given in prior work [17] and shows that execution time for large domain sizes quickly dominates.

6.1 Comparison with CPython on CPU

Columns 2–4 of Table 1 report minimum, maximum, and geometric mean speedup achieved by ALPyNA over the CPython interpreter. The lowest speedups (1.25x min, 71x max) are achieved for *hilbert*. The largest maximum speedups (8955x) are obtained for *syr2k*. We have not included results for the two largest iteration domains due to our experimental set-up (Section 5.4) timing out. These speedups will keep increasing exponentially, until performance levels off when all GPU SIMT execution units are saturated.

6.2 Comparison with JIT compiled CPU code

Columns 5–7 of Table 1 report minimum, maximum and geometric mean speedup achieved by ALPyNA over sequential JIT-compiled code produced by the NUMBA compiler. The results are classified into three groups, based on relative execution speedups observed.

Table 1. Relative performance of ALPyNA (higher is better) sorted by maximum speedup over Numba CPU across all iteration sizes. Iteration sizes corresponding to each execution time shown in brackets.

Benchmark	Relative speedup					
	ALPyNA vs CPython			ALPyNA vs Numba CPU		
	Min	Max	Mean	Min	Max	Mean
vadd	8.26 (8M)	36.15 (128M)	19.13	0.31 (128M)	0.43 (16M)	0.37
saxpy	121.09 (16M)	564.39 (256M)	293.59	0.32 (256M)	0.44 (16M)	0.38
conway	10.89 (1kx1k)	488.69 (16kx16k)	99.74	0.4 (16kx16k)	0.55 (1kx1k)	0.49
hilbert	1.25 (1kx1k)	70.89 (16kx16k)	12.62	0.7 (1kx1k)	0.77 (16kx16k)	0.72
jacobi	2.12 (512x512)	217.35 (8kx8k)	26.13	0.53 (8kx8k)	0.82 (512x512)	0.72
gemver	13.94 (1kx1k)	260.60 (8kx8k)	63.61	0.41 (2kx2k)	0.95 (16kx16k)	0.54
black-scholes	6.38 (1M)	54.05 (16M)	20.02	0.35 (1M)	1.15 (16M)	0.62
fbcorr	605.23 (16x8x 256x256)	1052.35 (16x8x 1kx1k)	810.72	1.37 (16x8x 256x256)	1.85 (16x8x 1kx1k)	1.59
conv-2d	67.39 (1kx1k)	1329.74 (16kx16k)	438.45	0.8 (1kx1k)	2.54 (16kx16k)	1.52
gemm	209.23 (512x512)	1398.95 (2kx2k)	630.21	1.15 (512x512)	2.64 (4kx4k)	2.05
mandelbrot	13.22 (256x256)	134.44 (4kx4k)	40.56	0.33 (256x256)	3.08 (4kx4k)	0.85
syr2k	39.69 (128x128)	8955.88 (1kx1k)	697.24	0.68 (128x128)	51.48 (4kx4k)	6.05

Light parallel workloads are kernels with small amounts of work to be done within the body of the parallelizable loop. Benchmarks such as *saxpy, conway , hilbert, jacobi* and *vadd* fall into this category. The analysis, compilation and GPU data transfer overhead incurred by ALPyNA to execute on the GPU is greater than compiling and executing the benchmark on the CPU. Figure 6 shows that the relative speedup remains constant for increasing iteration sizes. The analysis time was between $0.84ms$–$3.7ms$ and ranged from 0.12–1.2% of total execution time. Kernel compilation took 0.16s–0.3s and 17.5–99.4% respectively.

Medium parallel workloads are workloads that have a higher amount of work per parallel instance of a loop nest than a *light* parallel workload. For these workloads (such as *black-scholes* and *gemver*), it becomes profitable to offload to the GPU at higher iteration domain sizes. In the case of *gemver*, while there is enough work to be done in parallel, its four separate loops have loop independent dependences between statements. This forces them to be run as four separate kernels with repeated data transfer between them. ALPyNA does not analyze dependences between loop nests to allow for standard Python statements between loop nests which are potentially side-effecting. The analysis time was between $4.7ms$–$10.7ms$ and ranged from 0.13–1.1% of total execution time. Kernel compilation took 0.37s–0.92s and 10.7–94.7% respectively.

Heavy parallel workloads are workloads where the per-loop work is substantial. The performance boost that can

be obtained from these kernels is substantial despite the data communication overhead of the GPU. Kernels such as *conv-2d, gemm, mandelbrot, fbcorr* and *syr2k* show significant speedup at much lower iteration domain sizes. *fbcorr* is classified as a heavy parallel workload, due to the consistent speedups obtained over the JIT-compiled CPU variant. This is due to the kernel being able to run four out of seven loops in parallel and execute these faster than the CPU for a range of data sizes. For these benchmarks, speedups continue to increase with iteration domain size until GPU memory is exhausted. The analysis time was between $1.5ms$–$22ms$ and ranged from 0.001–10.5% of total execution time). Kernel compilation took 0.2s–0.37s and 1.3–93% respectively.

6.3 Future Performance Optimizations

ALPyNA helps developers to accelerate Python code written in a nested loop style. We show performance improvements of *1.25*x–*8955*x over the CPython interpreter on the CPU and *0.3*x–*50*x over Numba JIT on CPU. Below we identify future optimizations to improve ALPyNA's performance.

Cross-iteration dependences Many benchmarks (*gemm, gemver, conv-2d, syr2k, fbcorr*) have cross-iteration dependences when represented as loop nests. This forces sequential execution of kernels with reduced parallel work, incurring a kernel invocation penalty. Kernel fusion may help to mitigate this issue, when safe to do so.

Sub-optimal GPU axes mapping ALPyNA currently distributes parallel loops along each GPU axis to maximise parallelism (Section 4.4). Depending on memory access patterns, this may result in poor memory bandwidth utilization. For instance, *gemm* performance improved by *13x* by reversing the nesting of its parallel loops. Improving performance through better axis allocation is application-specific. We intend to use heuristics for this problem.

Data transfer overheads ALPyNA has higher data transfer costs in *gemver* and *syr2k* due to repeated data transfer between separate loop nests (Section 4.2). and scalar expansion of compiler generated variables during *if-conversion* (Section 4.1.1). This impacts benchmarks like *black-scholes* and *mandelbrot*. To reduce data transfer, we propose dataflow analysis to enable on-device data caching. Data transfer overheads can be eliminated on System-on-Chip devices with integrated GPUs (such as AMD APUs). We intend to investigate performance improvements on such devices by adding an OpenCL compiler backend target.

7 Related Work

Accelerators like GPUs and FPGAs are now widely available. Many techniques to leverage performance for these systems have been proposed. In this section we review various approaches that are integrated with high-level languages.

GPU kernel bindings: Klöckner *et al* [20] initially developed Python bindings to CUDA (PyCUDA) and OpenCL (PyOpenCL). While these provide direct access to the underlying GPU hardware, code must be written in low-level C-like syntax and must contain data-types. Type information can be patched in manually into the C-like code strings. However, this is left for the programmer to do.

Parallelizing higher order functions: Computational patterns expressed in functional idioms like `map` and `filter` are attractive candidates for acceleration as the semantics of these operations guarantee safe parallel execution. Copperhead [10] and Parakeet [34] compile and accelerate higher order functions in Python code on GPUs. Fumero *et al* [14] use the same idea to accelerate Ruby and R.

Code annotation: Tornado [11, 13] uses a set of code annotations for Java. Loops are specifically annotated with primitives such as `@parallel` and `@reduce` along with the number of dimensions. A task graph is created by the developer to generate a valid schedule for code execution. ALPyNA discovers scheduling constraints by doing loop dependence analysis rather than requiring the programmer to describe the schedule.

Numba [21] uses `@decorator` syntactic sugar to selectively compile functions for CPU and GPU. Python bytecode for decorated functions is analyzed and compiled using the LLVM infrastructure. Numba also exposes GPU compiler primitives to the developer. This allows programmers to compile a limited subset of Python code written in the CUDA paradigm to target Nvidia GPUs. We rely on Numba to compile ALPyNA's auto-generated GPU kernels. However, ALPyNA's loop dependence analysis maintains scheduling constraints that occur due to loop carried dependences.

Pydron [24] uses annotations to build a task graph that decomposes a program into parallel sections for cloud-based parallel execution. Pure functions are annotated to inform the compiler about their suitability for parallelization.

Speculative code variants: Apollo [8] is a runtime optimizing polyhedral compiler. Polyhedral transformations are performed on small windows of *LLVM*-IR and variants for each transformation are generated with guard conditions. These variants are speculatively executed. When a mis-prediction occurs, execution falls back to a known correct point.

Embedded Domain Specific Languages: Some embedded languages (eDSLs) can succinctly express parallelism within a host language. *Loo.py* [19], allows a developer to specify loop iteration sequences. The eDSL is embedded into Python code and is transformed into parallel kernels. These kernels are invoked programmatically from within the host VM. While this approach gives maximal control to the programmer, it also requires careful consideration of all the underlying hardware parameters, like thread counts and dimensions. However, no dependence analysis is done.

Library parallelism: Many Python libraries support GPU execution, e.g. the TensorFlow framework [1]. In such cases, all parallelism occurs inside a black box; 'below the API'. The dependences are however constrained by the definition of the API itself and hence can be specialized. Dandelion [33] generates parallel code for C# by parallelizing overloaded functions of specialized array-like objects for each hardware type. The hardware devices may be selected by the developer or left to be decided by the framework. The same approach is used by Ikra [22] for Ruby, River Trail [16] for JavaScript and ASDP [28] for ActionScript. Library parallelism is also implemented by overloading operators calling into specialized accelerator APIs. Python acts as a coordination language, with all parallelism devolved to the library code.

Loop parallelism: Three Fingered Jack (*TFJ*) [36] uses loop dependence analysis to parallelize linear Python `for` loops and generate code for FPGAs. TFJ builds on *Copperhead* [10] to statically compile nested loops with known loop bounds and fixed types. ALPyNA uses its hybrid static/dynamic dependence solvers to discover an optimized parallel execution order and also uses runtime type inference to dynamically create parallel kernels.

MEGAGUARDS [32] uses a polyhedral framework to solve dependences and executes loop bodies in parallel. Loop dependences are computed dynamically and guard conditions are generated at runtime for a speculatively generated parallel OpenCL kernel. The speculative kernel is optimized by hoisting guard conditions outside the loop body.

MEGAGUARDS and ALPyNA both attempt to parallelize nested loops and execute them on parallel hardware. However there are some differences in the approaches used. (i) MEGAGUARDS generates kernels only for perfectly nested loops[1] while ALPyNA can create parallel kernels for imperfectly nested loops. (ii) MEGAGUARDS generates a parallel kernel for perfect loop nests and uses complex guard conditions to evaluate the safety of executing this kernel in parallel. If these guard conditions are violated, execution falls back to the interpreter. ALPyNA computes dependences dynamically (if required) and generates kernels at runtime tailored to the dependence relationships that emerge for each instantiation of a loop nest. This allows for a richer set of optimizations to be discovered. (iii) MEGAGUARDS does not parallelize loop nests with cross-iteration *true*, *anti* or *output* dependences. MEGAGUARDS falls back to the interpreter in the presence of cross-iteration dependences. ALPyNA on the other hand dynamically determines which loops carry cross-iteration dependences and executes these loops sequentially to maintain ordering constraints. This allows us to safely execute all other inner loops in parallel. (iv) MEGAGUARDS is designed to be a whole program compiler while ALPyNA is designed to be an optional accelerating compiler that can be used with subsets of the overall program.

[1] A *perfectly nested loop* is where each loop body either contains or is contained by all other loop bodies in the nest; all assignment statements are located in the innermost loop [18].

8 Conclusions

This paper describes the ALPyNA framework, which enables end-user developers to exploit the performance capabilities of manycore accelerators such as GPUs. The framework allows a programmer to optimize Python nested loops, achieving good performance on modern parallel hardware. ALPyNA kernel code can co-exist with standard Python code allowing interoperability. We show that ALPyNA overcomes the limitations of conservative assumptions made by ahead-of-time parallelizing compilers, since it augments statically extracted loop nest features with runtime introspection of loop limits and subscript values. This means exact scheduling constraints can be extracted for parallel execution. ALPyNA demonstrates that the loop nest is a viable syntactic target for a runtime parallelizing JIT compiler to optimize.

Experimental evaluation of our framework shows dramatic performance increases, compared to the CPython interpreter. These speedups are observed for workloads large enough to amortize the costs arising from ALPyNA's analysis, kernel compilation and data transfer.

Analysis of the relative performance of the JIT-compiled GPU kernels *vis-à-vis* a state-of-the-art JIT-compiled CPU version shows a speedup ranging from 0.3x for *light* workloads (i.e. a slowdown) up to 50x for *heavy* workloads. We cannot naïvely always expect a speedup from a GPU JIT compiler, even when all the loops may be run in parallel. Therefore, we identify the need for a *dynamic cost model* to guide the selection of the most appropriate generated code variant of a loop nest in a heterogeneous manycore environment. This remains as future work. We also intend to expand the capabilities of ALPyNA by adding alternative target backends (e.g. FPGA, system-on-chips with an integrated GPU) to the compiler, which will require a more sophisticated cost model for code variant selection.

Acknowledgments

The authors would like to thank Juan Fumero for his careful and constructive shepherding of this work. We would also like to thank the anonymous reviewers for their help in improving this paper significantly.

This material is based upon work supported by the Engineering and Physical Sciences Research Council under Grants EP/M508056/1 and EP/P004024/1. 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 *Proc. OSDI*. 265–283.

[2] John R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of control dependence to data dependence. In *Proc. POPL*. 177–189.

[3] Anon. 2018. Python has brought computer programming to a vast new audience. The Economist, https://www.economist.com/science-and-technology/2018/07/19/python-has-brought-computer-programming-to-a-vast-new-audience.

[4] David Beazley. 2010. Understanding the Python GIL. In *Proc. PyCON*. https://www.dabeaz.com/python/UnderstandingGIL.pdf.

[5] Fischer Black. 1976. The pricing of commodity contracts. *Journal of Financial Economics* 3, 1 (1976), 167 – 179. https://doi.org/10.1016/0304-405X(76)90024-6

[6] L Susan Blackford et al. 2002. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (2002), 135–151. https://doi.org/10.1145/567806.567807

[7] 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

[8] 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

[9] 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.

[10] 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

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

[12] Berk Ekmekci, Charles E McAnany, and Cameron Mura. 2016. An introduction to programming for bioscientists: a Python-based primer. *PLoS Computational Biology* 12, 6 (2016), e1004867.

[13] Juan Fumero and Christos Kotselidis. 2018. Using Compiler Snippets to Exploit Parallelism on Heterogeneous Hardware: A Java Reduction Case Study. In *Proc. Virtual Machines and Intermediate Languages*. 16–25. https://doi.org/10.1145/3281287.3281292

[14] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In *Proc. VEE*. 60–73. https://doi.org/10.1145/3050748.3050761

[15] W. Harrison. 2004. From the Editor: The Dangers of End-User Programming. *IEEE Software* 21, 4 (2004), 5–7. https://doi.org/10.1109/MS.2004.13

[16] Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. 2013. River Trail: A Path to Parallelism in JavaScript. In *Proc. OOPSLA*. 729–744. https://doi.org/10.1145/2509136.2509516

[17] Dejice Jacob and Jeremy Singer. 2019. ALPyNA: Acceleration of Loops in Python for Novel Architectures. In *Proc. ARRAY*. 25–34. https://doi.org/10.1145/3315454.3329956

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

[19] Andreas Klöckner. 2014. Loo. py: transformation-based code generation for GPUs and CPUs. In *Proc. ARRAY*. 82–87. https://doi.org/10.1145/2627373.2627387

[20] 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

[21] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-based Python JIT compiler. In *Proc. Second Workshop on the LLVM Compiler Infrastructure in HPC*. 7. https://doi.org/10.1145/2833157.2833162

[22] Hidehiko Masuhara and Yusuke Nishiguchi. 2012. A Data-parallel Extension to Ruby for GPGPU: Toward a Framework for Implementing Domain-specific Optimizations. In *Proc. 9th ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*. 3–6. https://doi.org/10.1145/2237887.2237888

[23] Wes McKinney. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing* 14 (2011).

[24] 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

[25] David A. Padua and Michael J. Wolfe. 1986. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM* (1986). https://doi.org/10.1145/7902.7904

[26] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.

[27] Tim Peters. 2004. PEP 20 – The Zen of Python. https://www.python.org/dev/peps/pep-0020/.

[28] Alessandro Pignotti, Adam Welc, and Bernd Mathiske. 2012. Adaptive Data Parallelism for Internet Clients on Heterogeneous Platforms. In *Proc. DLS*. 53–62. https://doi.org/10.1145/2384577.2384585

[29] Finnegan Pope-Carter, Chrys Harris, Thomas Sparrow, and Chris Gaffney. 2015. ArchaeoPY: Developing Open Source Software for Archaeological Geophysics. In *Proc. 43rd Computer Applications and Quantitative Methods in Archaeology*. https://sites.caa-international.org/caa2015/wp-content/uploads/sites/14/2015/04/Book-of-Abstracts_CAA20151.pdf.

[30] Louis-Noël Pouchet, Uday Bondhugula, et al. 2019. The polybench benchmarks. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench.

[31] S. Prema, Rupesh Nasre, R. Jehadeesan, and B.K. Panigrahi. 2019. A study on popular auto-parallelization frameworks. *Concurrency and Computation: Practice and Experience* 31, 17 (2019), e5168. https://doi.org/10.1002/cpe.5168

[32] Mohaned Qunaibit, Stefan Brunthaler, Yeoul Na, Stijn Volckaert, and Michael Franz. 2018. Accelerating Dynamically-Typed Languages on Heterogeneous Platforms Using Guards Optimization. In *Proc. ECOOP*. 16:1–16:29. https://doi.org/10.4230/LIPIcs.ECOOP.2018.16

[33] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proc. SOSP*. 49–68. https://doi.org/10.1145/2517349.2522715

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

[35] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proc. CHI*. 32:1–32:12. https://doi.org/10.1145/3173574.3173606

[36] 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

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