UNIVERSITY OF COPENHAGEN

Just-in-Time Compiled Loops in Python

Master thesis in Computer Science

Sebastian Skjønberg Andersen
sebastian@skjoenberg.com
XTF221

Supervisor:
Mads Ruben Burgdorff Kristensen

Department of Computer Science,
University of Copenhagen

Abstract

High-productivity and high-performance within programming have historically been two separated concepts. Due to the error-prone and time-consuming process of manually constructing programs within high-performance languages, multiple frameworks seek to close the gap between high-performance and high-productivity languages. The Bohrium runtime system targets accelerating array operations within high-productivity languages especially focused on Python. In this thesis, we explore reduction of overhead in regards to using loops with dynamic loop bodies within Bohrium.

We present an extension to the internal loop, allowing dynamic linear modification to views within loop bodies, referred to as *sliding views*. The implementation of sliding views is available through the Numpy bridge and works with all current vector engines. Sliding views introduces an asymptotic decrease in loop overhead in contrast to using native Python loops. The decrease in overhead is documented empirically through benchmarks from Benchpress and VEROS. The Numpy bridge features a user-friendly syntax for sliding views in Python and only requires slight modifications of source code using native Python loops. The official release of the Bohrium runtime system has adopted this implementation of sliding views.

We have also explored the possibility of constructing custom non-linear modifications to views within loop bodies, referred to as *array indexing*. Array indexing uses arrays as placeholders for dynamic modifications to views within the loop body. By utilizing arrays to store metadata, it pushes operations on metadata from the internals of Borhium to the vector engine, which supports all operations in the intermediate bytecode. The implementation showcases both strengths and limitations. Due to limitations regarding user-friendliness, Bohrium has not adopted array indexing.

Acknowledgements

First of all, I want to thank my supervisor Mads R. B. Kristensen for great counseling and support throughout the making of this thesis. And since I have been knocking on his office door time and time again for clearance, discussions and an uncountable amount of questions about Bohrium, I must also thank him for his patience.

Secondly, I would like to thank the eScience group at the Niels Bohr Institute for providing me with a table at an office space together with other master students and inviting me to their strategic retreat. I want to thank all employees and students within this working environment for the warm welcoming and friendly attitude. And a big thanks for a large amount of free coffee, keeping me quick throughout the process.

Lastly, I would like to thank my lunch group at DIKU for the great times and good friendships throughout our studies of computer science. It has been great having close friends, who are also writing their thesis, for motivating and supporting each other. A big thanks to Henrik, Jacob, Mathias, Michael, Rasmus, Tobias and Troels.

Contents

Abbreviations

API Application Programming Interface

CPU Central Processing Unit

GPGPU General Purpose Graphic Component Unit

HPC High Performance Computing

MIMD Multiple instruction streams, Multiple data stream

SIMD Single instruction stream, Multiple data streams

SIL Sub-instruction list

VEROS The versatile ocean simulator

VE Vector Engine

VEM Vector Engine Manager

Chapter 1

Introduction

Constructing complex solutions, for scientific and engineering computational problems, often requires a certain amount of experimenting and prototyping. It is not always clear how well different solutions achieve specific properties, such as performance and simplicity, before there is a functioning prototype. Benefits of comparing different algorithms, tuning parameters and receiving feedback on the parameters are all critical aspects of verifying and optimizing a solution. For this approach to be feasible, prototyping must somewhat quick and easy. The cost in time of prototyping within a low-level architecture-specific programming language is too high to experiment with multiple solutions.

The scientific community satisfies the demand for prototyping by relying on high-productivity languages and libraries, such as Python with Numpy and Matlab. The characteristics of these languages are, they support high-level abstractions and does not require fiddling with low-level constructs. Their support for the declarative array programming model allows a more mathematical notation, while specific properties of the model can lead to computational benefits. As for the programming experience, programs are more comfortable to debug because errors appear high-level. The programs can also be easily visualized, making the effects of changing parameters more intuitive and noticeable.

High productivity comes at the expense of performance. Python's native high-level data structures, such as lists and dictionaries, are not suitable for producing programs within the domain of high-performance computing [31]. The programs required by scientific research operates on too large inputs and uses algorithms with a high degree of complexity. Different solutions to a scientific problem are often prototyped in a high-productivity language, while the final solution is reimplemented in a high-performance language, requiring further time and work. The structure and code of the reimplementation is more tedious, requires a deeper understanding of hardware specifics and is handcrafted to achieve good performance on a particular hardware setup. This process is both time-consuming and error-prone, due to the increased complexity of low-level programming. The great benefit is, the handcrafted solution performs much better than the high-productivity prototype. The faster execution makes it feasible to run the program with large inputs.

Multiple frameworks target the issue of bypassing the reimplementation

phase by enhancing the performance of high-productivity languages. These frameworks seek to tighten the gap between high-performance and high-productivity. In the case of Python, the framework Numpy optimizes the performance through utilizing the array programming model and external libraries. Numpy has shown that great performance gain is indeed possible. The *dot-product* is an example where using Numpy is $70\times$ faster than using native Python for loops [31].

1.1 The Bohrium Runtime System

The Bohrium runtime system (Bohrium) is a framework seeking to enhance the performance of array programming within high-productivity languages and related frameworks. The manual process of reimplementing solutions is replaced by transparent code generation, utilizing both parallelism and hardware specifics, purely based on interpreting source code from a high-productivity language or framework. The goal of Bohrium is not to surpass the performance of a highly-optimized handwritten solution but seeks to achieve 80% of the performance [26].

Bohrium functions as a Just-in-Time (JIT) compiler, taking advantage of established programming languages and compilers for producing efficient code. As part of this process, Bohrium performs internal code optimization through specialized fusers. Bohrium provides a backend for multicore-CPUs, GPUs and cluster setups. The efficiency of Bohrium is heavily relying on the inherent parallelism within array programming. Bohrium has an extensive Python library, translating array operations in Python into an intermediate bytecode, handled internally in Bohrium.

1.1.1 Loops within Bohrium

Bohrium is targeted towards array programming and lacks support for efficiently handling explicit loops. The intermediate bytecode has no notion of loops, forcing the Python interpreter to unroll loops. Interpreting unrolled loops can lead to a considerable amount of overhead within Python, slowing down the performance of the Numpy bridge. The overhead increases linearly with the number of iterations, making Bohrium slower than pure Numpy in specific scenarios.

Bohrium has a specialized *flush-and-repeat* loop structure with the specific purpose of reducing loop overhead. The structure is accessible in the Numpy bridge through a Python function called `do_while`. The function executes a *do while* loop, while the Python interpreter only adds a constant amount of overhead. The downside of `do_while` is limited expressiveness in the loop body. One of the substantial limitations is, all views in the loop body are static. In other words, the views in the loop body cannot change shape or offset. An example, showcasing the limitation of static views within loops, can be seen in Figure 1.1 (*note that the source code in this example is pseudo-Python, `do_while` has a different syntax*).

This thesis explores the subject of extending the flush-and-repeat loop structure with dynamic views and it accessible through `do_while` in the Numpy bridge of Bohrium.

```
a = np.arange(5)
for i in range(4):
  a[i+1] += a[i]
# resulting a with unrolled loop: [1 3 6 10 15]
# resulting a with do_while:      [1 6 0 0  0 ]
```

Figure 1.1: Results of executing a loop with a sliding view using unrolled loops and `do_while` with Bohrium

1.2 Limitations

The loop extensions of this thesis have not been incorporated with the filters `bccon` and `bcexp`. We have not experienced any problems with using these filters, but their might be some cases where they modify the views and their metadata.

1.3 Contributions

The main contributions of this thesis is summarized to the following:

- We present an extension of Bohrium's flush-and-repeat loop structure, allowing linear slides of views within loop bodies. Sliding views is accessible through using `do_while` in the Numpy bridge. The linear sliding is handled internally in Bohrium. Using `do_while` with sliding views reduce the overhead of the Python interpreter, from a linear to a constant factor, in contrast to using native Python loops. The official release of Bohrium has adopted this extension in its master branch on GitHub.
- We provide an intuitive syntax for utilizing sliding views within in `do_while` in Python, with a syntax similar to Numpy's.
- We provide empirical documentation of overhead reduction through concrete benchmarks, showcasing the performance gain by using `do_while` with sliding views in contrast to native Python loops and pure Numpy.
- We present an additional extension to the flush-and-repeat loop structure, allowing views to be manipulated through information within arrays at runtime. This extension allows non-linear custom modifications to views within a loop body. The vector engine handles the manipulation of the arrays and thereby supports the operations of the intermediate bytecode. This implementation enforces limitations to the user-experience within Python. Due to these limitations, the official Bohrium release has not adopted this implementation.

1.4 Thesis Structure

This thesis is structured into chapters, which all serves a special purpose. A brief overview of the content of the chapters is listed below.

***Chapter 2* Parallelism** This chapter revolves around parallel computation. The chapter begins with a theoretical foundation of parallelism and then describes concrete tools for achieving it. The chapter serves as a fundamental base for the thesis.

***Chapter 3* The Bohrium Runtime System** This thesis is heavily based on the Bohrium runtime system. This chapter first motivates Bohrium and then describes its overall structure. The following sections dig into details of internal components in Bohrium. The chapter focuses on the details of components relevant for the loop extensions of this thesis.

***Chapter 4* Loop extensions** This chapter begins by motivating the need of loop extensions and then presents the two loop extensions implemented during this thesis: *Sliding views* and *array indexing*.

***Chapter 5* Performance** This chapter measures the performance of the *sliding view* implementation by benchmarking multiple programs. The chapter also describes the benchmarking environment, the computations the benchmarks perform and discusses the results. The chapter also contains a guide for reproducing the results.

***Chapter 6* Related work** This chapter explores approaches of other frameworks seeking to tighten the gap between high-productivity and high-performance.

***Chapter 7* Future work** This chapter describes future work related to the implemented loop extensions and loops in Bohrium in general.

***Chapter 8* Conclusion** This chapter concludes the work of the thesis.

Chapter 2

Parallelism

Parallelism emerges as a mechanism for gaining performance by executing computations on multiple compute units simultaneously. As of today, most computer systems utilize parallelism. General-purpose computers often have two or four cores, while high-performance supercomputers have millions of cores [7].

There are generally two high-level abstractions of parallelism:

Task parallelism The simultaneous execution of functions using multiple cores on the same or different datasets

Data parallelism The simultaneous execution of the same function using multiple cores on the same dataset

Generally, using task parallelism on the same or multiple datasets differ in structure. Task parallelism operating on the same dataset often uses a pipeline structure. Each phase of the pipeline is a function, that must be applied to all data chunks sequentially. The parallelism appears when all phases of the pipeline apply their function on a different data chunk simultaneously. When the execution of the functions ends, each phase passes the result onto the next phase. The sequential overhead only consists of filling and emptying the pipeline, while all phases execute their functions in parallel the rest of the time. Functions within the pipeline must have similar execution time to avoid the overhead of phases waiting on each other. Task parallelism operating on different datasets often uses processes. A process occupies a single core and executes sequentially. Processes are then scheduled and distributed among cores. Parallelism occurs by executing multiple processes simultaneously. The processes do not necessarily need to have similar execution-time unless they somehow depend upon each other. In task parallelism, the parallelization is proportional to the number of tasks.

Data parallelism executes the same function over a whole dataset. The data is split into subsets and distributed among cores to achieve simultaneous execution. In data parallelism, the parallelization is proportional to the amount of data. The GPGPU is a hardware component tailored towards data parallelism, by using SIMD, and a popular hardware choice for high-performance computing. Data parallelism is in close connection with the array programming model (described in Section 2.4) which is an essential part of Bohrium.

This chapter focuses on the fundamentals of parallelism, which will lay the groundwork for this thesis. This thesis mostly focuses on data parallelism, reflected in the chosen topics.

2.1 Amdahl's Law

Amdahl's law is a formula describing a theoretical upper bound of the achievable speedup of an execution task by optimizing certain parts of the task. Gene Amdahl presented the idea in 1967 concerning optimization through parallelism [10]. The formula uses two variables f and n:

f : An infinitely parallelizable part (without overhead) of the execution task

n : The number of parallel processors

The original formula of Amdahl's law, regarding multicore-parallelism, is defined as:

$$Speedup_{parallel}(f, n) = \frac{1}{(1 - f) + \frac{f}{n}}$$

The law only provides an upper bound, due to the overly simplified variable f. The law indicates the possible speedup related the number of cores used for the computation. A plot of the optimization related to f and n can be seen in Figure 2.1. Amdahl's law points out a fundamental limiting property of parallelization: A program often contains parts that require sequential execution. Thus, parts of the program cannot gain a speedup by adding further processors. As shown in Figure 2.1, a program that requires 5% sequential execution has a limited speedup-factor of 20 through parallelism. Amdahl's law implies that adding more processors contributes less and less speedup as the number of processors rises.

Amdahl argued in 1967 that the sequential part $1 - f$ was often enough to favor a fast single-core processor over a multicore-processor. This statement also took the overhead involved with using multiple cores into account, which is not a factor in Amdahl's law.

2.2 Flynn's Taxonomy

Flynn's taxonomy is a classification model for computer systems. It originated in 1966 with the four classifications: SISD, SIMD, MISD, and MIMD [22]. Despite its age, the precise distinction between the two parallel models SIMD and MIMD is still widely used, while the relevance of SISD computer system has decreased heavily during the last ten years (described in Section 2.3.1). MISD is rarely seen in practice and not covered in this section.

Flynn's taxonomy has a notation containing four essential components:

Instruction stream The sequence of instructions to be performed by the machine.

Data stream The sequence of data needed by the instructions, including the input data and temporary results.

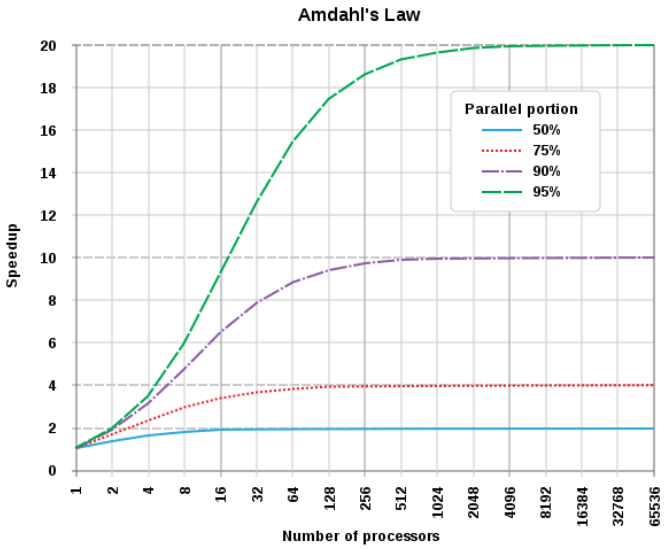


Figure 2.1: Illustration of possible speedup in regard to the parallel portion of a program based on Amdahls law. Illustration is borrowed from [8].

Control Unit A decoder that translates the instruction stream into control signals.

Processing Element A computation unit that operates on data from the data stream, given the control signals from the control unit.

In all of the four models, the first two letters indicate the type of the instruction stream, while the latter two indicate the data stream.

Multiple data stream systems are categorized into shared memory and distributed memory. A shared memory system uses a globally accessible piece of memory with a connection to each processor. Since each processor can access the global piece memory, maintaining memory coherence is necessary. In a distributed memory system, each processor has a dedicated piece of memory, and there is no shared memory. The processors must request data from one another. Unless the processors are fully interconnected, such requests introduce routing. Maintaining memory coherency between processors requires communication. The number of processors scales the overhead of communication. Depending on the concrete models and setup, the overhead scales differently.

2.2.1 SISD

Single instruction stream, Single data stream

This classification expresses a sequential computer system, exploiting no parallelism at any level. The computer system has a single control unit, processing element, instruction stream, and data stream. The flow of computations on a SISD computer system is the following:

1. The control unit fetches instructions from the instruction stream, one at a time.
2. The instruction is translated into control signals and forwarded to the processing element.
3. The processing element performs computations, one instruction at a time, on the data stream.

The most common example of a SISD computer system is a uniprocessor machine, such as an older personal single-core computer.

2.2.2 SIMD

Single instruction stream, Multiple data streams

A SIMD computer system introduces parallelism at the execution step. The computer system has a single control unit and instruction stream, but multiple processing elements and data streams. All processing elements receive the same instruction signal from the instruction stream. Thus, SIMD is unable to execute different instructions simultaneously.

The multiple processing elements lead to several different designs of the architecture. Two general architectures are the array processor and the pipelined processor. The pipelined processor was later considered an independent architecture. Thus, SIMD often refers to the array processor [27].

The array processor has all processing elements connected to a central control unit. The process elements are independent, having dedicated registers and memory. All processing elements perform the same operation at a given time step. The operation all processing elements performs may change at each time step.

The pipelined processor uses processing elements tailored towards a particular function. Each processing element has a pipeline of data to compute the function on a new pair of data for each time step. The control unit issues a vector operation to memory, which places the data in the pipeline of the given function. In the pipelined processor, each processing element performs an operation that might be different from the other processing elements at a given time step. The operation that a single processing element performs does not change over time.

The SIMD organization leads to certain difficulties. The communication between processing elements can lead to poor performance. Sequential code can be difficult to fit into SIMD. Using high-level operations with internal parallelism sequentially can solve this problem. The execution of the high-level operations is still sequential, but the internal execution of the operation is in parallel (described further in Section 2.4). Instruction streams with branching can lead to the processing elements being in different states. Due to the single central control unit, only one state can be managed at a time, leading to less parallelism [23].

2.2.3 MIMD

Multiple instruction streams, Multiple data stream

A MIMD computer system introduces parallelism at both the instruction stream and execution step. Each processing element has a dedicated data stream and a dedicated instruction stream, allowing asynchronous execution. The processing elements are allowed to execute different operations on different data at any given time step.

The difference between a MIMD computer system and a multi-core SISD computer system is MIMD executes subprograms that represent different parts of the same task. Thus, MIMD can shorten the execution time of a single program, while multi-core SISD can only execute multiple programs simultaneously.

MIMD can also lead to several problems. Especially the overhead of communication between the subprograms.

2.3 Brief History

This section focuses on the evolution of hardware trends and how performance gain shifted towards parallelism. Flynn's taxonomy points out that orchestrating parallelism comes with a cost of communication overhead, memory/locality concerns and work occupation of processors. Despite these disadvantages, parallelism has become a dominating factor for gaining performance in all from personal computers to supercomputers.

2.3.1 Increasing the Clock Frequency

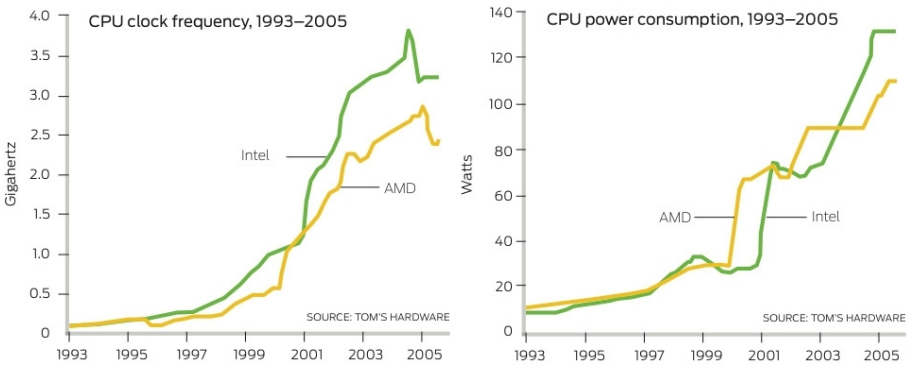


Figure 2.2: The evolution of CPU clock frequency and power usage. Illustrations borrowed from [28].

Up until about 2004, a powerful approach for gaining performance was increasing the clock frequency of processors. Increasing the clock frequency ultimately led to more hardware operations in the same amount of time, making programs faster without changing them. The hardware structure itself did not change but just became faster. The problem with this approach was that

increasing the clock frequency has the disadvantage of increasing power consumption and heat generation [30]. Figure 2.2 illustrates the relation between clock frequency and power consumption. A noticeable breakpoint was when Intel canceled its Pentium 4 processor with 4 GHz [24].

Hereafter, general-purpose processors shifted from increasing the clock frequency to adding multiple cores onto the same chip. Increasing performance became reliant on occupying multiple cores with computations simultaneously. The multicore-approach was utterly different from the prior. Having multiple cores occupied simultaneously lead to new challenges that could not be solved efficiently without changing programs. These challenges involved cache and memory coherency, communication between cores and load balancing the computations. A mechanism for avoiding these challenges is executing independent sequential programs on different cores. However, this requires that the programs do not communicate or access the same memory, which are hard restrictions to follow. It does not speed up the runtime of a single program but executes multiple programs simultaneously. A program must be written to utilize the available parallelism to gain performance. Utilizing parallelism introduces overhead, as mentioned earlier. Therefore, the general tendency is that adding cores will lead to less than linear speedup [30]. If the program contains inherently sequential parts, the speedup is also upper bounded by Amdahl's law.

2.3.2 High Performance Computing

HPC is concerned with solving complex computational problems by using massively parallel supercomputers. The HPC community has a close relation to the scientific community since scientific models and simulations often require extensive computation power. The problem sizes and complexity of algorithms make the problems infeasible to run on a SISD computer system. Since the high-performance computer systems are massively parallel, the scientific problems must utilize a high degree of parallelism. If not, Amdahl's law bounds the speedup.

Achieving a high degree of parallelism is not a trivial matter. It must both utilize the parallelism within the multicore-nodes and connect the nodes into a cluster. Cluster-parallelism introduces even more difficulties, as the physical distance becomes larger leading to more complicated and time-consuming communication.

Supercomputer uses SIMD, MIMD or a combination of both to achieve parallelism.

2.4 Array Programming

This section covers the programming paradigm called *the array programming model*. Array programming is inherited from the more general *collection-oriented* model. A concise definition of collection-oriented programming, given by Jay Sipelstein and Guy E. Blelloch in 1990, is:

"... their basic data type is an aggregate of other data types and their functional primitives operate on these aggregates" [29].

The essential idea of a collection is to encapsulate a group of data objects. The collection is then viewed and manipulated as a whole. The particular type of collection is called *basic-datatype* of the language. Naturally, there exist many different types of collection-oriented programming languages. A common basic-datatype is the (multi-dimensional) array, which names array programming. The typical functional primitives for array programming is heavily reliant on high-level linear algebra operations.

Two important properties that motivate collection-oriented programming are:

1. Operations are high-level, easily expressible (in a notation close to mathematics) and are all based upon manipulating collections
2. The operations are inherently massively parallel since the operations are naturally applied to multiple data-objects simultaneously

These properties hold for all collection-oriented programming models, but throughout this section, the focus will remain on the array programming model.

2.4.1 High-level Operations

The notion *high-level* refers to an abstraction that allows programmers to get around low-level details of computer systems. Such details involve manual memory-management, parallelization, data-dependencies and general knowledge of hardware. The high-level abstraction allows expressing programs with general operations, without worrying about the underlying implementation of these. Array programming is inspired by linear algebra, due to the similarity between multidimensional-arrays and tensors.

High-level programming often leads to higher productivity of the programmer, in contrast to low-level programming. Low-level programming is closely related to explicitly utilizing underlying architecture, while high-level abstractions push this responsibility to either a framework or compiler. Thus, high-level programming has the benefit of general and architecture-independent programs but comes with the price of adding complexity to the underlying analysis and compilation of the programs. The flexibility of a high-level program is dependent upon the compilers (or multiple compilers) ability to boil the program down to architecture-specific executables. High-level source code is often easier maintainable and supports multiple hardware platforms. Thus, high-level programming has a trade-off between explicitness and the complexity of compilation.

2.4.2 Massively Parallel

The array programming model is inherently data parallel in its operations. The possible amount of parallelism is therefore dependent upon the size of these arrays. The array-oriented operations are in most cases suitable for massively parallel hardware [29]. As an example of parallelism within operations, component-wise addition on arrays is embarrassingly parallel and computed in the following fashion:

1. The corresponding data elements of the two arrays are paired
2. These pairs are each assigned to a given processor
3. The given processor performs addition on the two numbers in a SIMD fashion.

By viewing data chunks as individual objects and performing bulk operations on these, the result is a naturally parallel paradigm. Seemingly sequential operations are, sometimes, able to be performed in parallel. An example is the *scan*-operation using an associative operator. A scan is an accumulative reduction, which may seem inherently sequential due to the accumulation. However, the complexity of scan is $\mathcal{O}(\log(n))$ in the pram model, with n being the length of the input array [16].

2.4.3 Expressiveness

Naturally, limiting the operations to be array-based has certain consequences. One limitation is that array programming has limited expressiveness. The core of array programming does not include common programming structures, such as control flow which includes conditionals and loops. Array programming is therefore often wrapped within a programming language that supports these structures but enforces the programmer to use them with care. The downside of using these structures is that they do not carry the properties of array programming and can lead to a decrease in parallelism. The descriptive power of array programming is at a point between domain-specific and general programming.

2.5 Low-level APIs

In contrast to parallel programming paradigms, it can be useful to have hand-tailored low-level APIs for specific purposes. Low-level APIs seeks to gain performance by using highly-optimized implementations that target specific hardware platforms. Low-level APIs often have multiple underlying implementations by different vendors.

This section covers four low-level APIs, which are all influential in their application field. MPI is concerned with communication, BLAS with solving linear algebra and CUDA and OpenCL with utilizing GPGPUs.

2.5.1 MPI

Message Passing Interface (MPI) is a communication protocol for parallel computer systems, developed in the early 1990's. The benefits of MPI is portability and a straightforward API, powerful enough for achieving high performance in complex setups. As of today, MPI remains the dominant and defacto communication model for high-performance computer systems. The MPI protocol has multiple efficient implementations, where many are open source. The MPI protocol is accessible through language-specific APIs that communicate with the underlying MPI implementation. The MPI protocol has an API in many popular programming languages such as C, C++, Fortran, Java, and Python.

The essential component in MPI is a process. Commonly, a compute node only runs a single process based on the Single Program Multiple Data (SPMD) model. The processes communicate by rendezvous-style point-to-point messages. The MPI protocol comes in three main versions:

Version 1 Contains core communication tools for processes. It focuses on message passing and has a static runtime environment.

Version 2 Extends the protocol with features such as parallel I/O, dynamic process management and remote memory operations. These features extend the original focus on message passing by allowing explicit memory and process management.

Version 3 Further extends the protocol with features, such as non-blocking communication and one-sided operations.

Due to the success of the MPI protocol, it has slowly extended itself beyond being a message passing interface. Version 2 and 3 provides an extensive API, supporting over 500 operations.

Despite having a straightforward communication interface, correctly utilizing the MPI protocol requires relatively low-level knowledge. The usage of the MPI protocol is directly related to the particular program and hardware setup.

2.5.2 BLAS: Basic Linear Algebra Subprograms

Linear algebra is a mathematical discipline concerning linear equations and functions, represented by vectors and matrices. Many problems are naturally expressed within this domain and can be solved using its operations.

BLAS [9] are the de facto library for general operations within the field of linear algebra. It is used under the surface by many high-performance libraries such as LAPACK and LINPACK, and also in array programming frameworks such as Python's Numpy. BLAS started as a library for the Fortran programming language but has since added support for C as well.

There are three levels of BLAS routines categorized by the structure of the inputs and the complexity of the operations. In the following description, matrices are represented using capital letters, vectors by lowercase letters and constants by Greek letters. The levels are:

Level 1 Vector-vector operations. Includes dot-products, vector norms and generalized vector addition of the form $y \leftarrow \alpha x + y$ (axpy). The operations typically have a complexity of $\mathcal{O}(n)$.

Level 2 Matrix-vector operations. This level introduces many matrix operations, along with a generalized matrix-vector multiplication of the form $y \leftarrow \alpha Ax + \beta y$ (gemv) and a solver for $Tx = y$ with T being triangular. The level 2 operations can be implemented using level 1 operations, but are introduced for performance on array processors [21]. The operations typically have complexity of $\mathcal{O}(n^2)$.

Level 3 Matrix-matrix operations. This level primarily targets generalized matrix-matrix multiplication of the form $C \leftarrow \alpha AB + \beta C$ (gemm) and a solver for $B = \alpha T^{-1}B$. The operations typically have complexity of $\mathcal{O}(n^3)$.

There are multiple BLAS implementations optimized for different hardware platforms. Both vendors and open-source projects provide implementations of BLAS.

2.5.3 CUDA

Compute Unified Device Architecture (CUDA) is a platform for parallel computing and API created by NVIDIA. It allows using NVIDIA GPUs for general purpose computations. Many popular programming languages support the CUDA API, such as C, C++, and Fortran.

The programming model of CUDA is called Single Instruction Multiple Threads (SIMT) which is closely related to SIMD. In the SIMT model, threads have unique ids instead of being assigned to specific data. The threads can decide the data to operate on based on their id.

2.5.4 OpenCL

Open Computing Language (OpenCL) is a low-level framework for writing parallel programs. OpenCL supports multiple hardware platforms. It is generally known for targeting GPUs (generally AMD) but also targets CPUs, digital signal processors and field-programmable gate arrays. The framework offers support for writing both task and data parallel programs.

In the OpenCL terminology, *kernels* are functions executed on compute devices. OpenCL both contains a programming language (based on C and C++) for expressing kernels and an API for controlling the devices, copying and allocating data and executing kernels. A compute device often consists of multiple compute units that can each execute kernels in parallel.

2.6 Programming Languages

Almost every programming language supports parallelism to a certain extent. Some programming languages take it a step further by having parallelism as their main focus. This section covers four programming languages. The three first languages (NESL, HPF, and Chapel) are all based on parallelism. The fourth (Python) is known for its simplicity but has limited native support for efficient parallelism. The Numpy framework extends Python by introducing array programming, which achieves more efficient parallelization. Bohrium (Described in Chapter 3) has a bridge based on Numpy.

The parallel languages all use the *global-view* model. In the global-view model, the programmer expresses an algorithm as a whole, while introducing parallelism through language constructs. This model is in contrast to the *fragmented* model, which defines algorithms on a data chunk basis. In the fragmented model, an algorithm is executed each data chunks simultaneously. The algorithms are responsible for decomposing data structures, handling control-flow and communication/synchronization between processes. Global-view languages are primarily used within academia and do typically not have a solid foundation in other areas [20].

2.6.1 NESL

NESL is a parallel programming language based on parallel algorithms, functional programming, and array programming. It is known for nested data parallelism and an internal performance model related to parallel programs. The SCandAL project developed NESL and released it in 1993.

Nested data parallelism are based on using higher-order bulk operators, such as `scan`, `reduce`, `map`, `filter` etc. The bulk operators apply a function to a collection of data in a structured fashion. The structure of these operators is all inherently parallel. Using nested bulk operators leads to nested parallelism. Nested parallelism is more difficult to handle since the parallel function cannot be split among processors as easily. An example of nested parallelism is using a `map` over the rows in a 2D-array with a `reduce`, summing the elements of the given row, as the argument. The concrete source code for this example can be seen in listing 2.1. Nested parallelism is non-trivial to transform into a single level of outer parallelism. NESL handles this problem by the process of flattening arrays, described in detail in [17]. The flattening and parallelization NESL performs can lead to excessive memory usage and poor cache performance.

The performance model of NESL is split into two complexities:

Work complexity This represents the total amount of work of the program execution as if executed a serial machine.

Depth complexity This represents the parallel depth of the program execution as if executed on a machine with unbounded parallel processors.

Each built-in operation has an associated work and depth cost. Therefore, it is possible to derive the complexity of the program by adding and multiplying these complexities.

Even though some concrete implementations of NESL exists, it mostly remains as a theoretical programming language. NESL has inspired many data parallel programming languages and frameworks, such as Data Parallel Haskell [18].

Listing 2.1: An example of row summation in NESL

```
    {sum(row) : row in [[1,2,3], [4,5,6], [7,8,9]]};
 => [6, 15, 24] : [int]
```

2.6.2 High Performance Fortran

High Performance Fortran (HPF) is a high-level data-parallel extension of Fortran 90 supporting various constructs for parallel computation. It provides a Fortran-based interface for array programming, focusing on portability to different parallel hardware platforms. The programming model of HPF was in part inspired by NESL. High Performance Fortran Forum developed HPF in the early 1990's.

HPF is regarded as a front-runner for high-performance languages, due to its philosophy and ideas. However, it never gained a large user base for multiple reasons. The *HPF timeline and discussion* [25] profoundly inspire the following reasons:

Missing features HPF 1.0 lacked multiple wanted features since their implementation would sacrifice performance. Especially, a more advanced form of data distribution and more powerful strategies regarding task parallelism. Many of these features were fixed in HPF 2.0 but was not enough to revive HPF.

No reference implementation HPF did not offer an open source implementation. Each compiler project had to implement a unique version of the language, which was a complex and time-consuming task.

Divided compiler focus The lack of a reference implementation forced vendors to create a custom implementation of HPF. The vendor-specific implementations often focused on the hardware the vendor produced. Because of this, HPF programs could have a significant difference in performance based on the specific HPF implementation. It ultimately forced programmers to reprogram their software to perform in a given HPF implementation, which defeated the purpose of supporting multiple platforms.

Difficulties regarding performance tuning HPF compilers dramatically transformed the inputted source code to achieve performance. The resulting code was a mix of Fortran 90 and MPI. Therefore, it became a very complex process to tune the performance of a program since it was not clear which parts of the source code led to slow procedures in the compiled program.

2.6.3 Chapel

Chapel (Cascading High Productivity Language) is a programming language based on a multithreaded parallel programming model. It supports abstractions for data parallelism, task parallelism, and nested parallelism. Both NESL and HPF has inspired Chapel. Cray develops Chapel, and it was released in 2009.

The philosophy of Chapel is making high-performance computing easier for programmers who are not experts in this domain. Even though it supports high-level structures and implicit parallelism, it also allows the programmer to use low-level constructs to gain full control. The low-level constructs involve locality management of data and concurrency control. The programmer is thereby able to decide how architecture-specific the source code should be. The source code can also contain a mixture, where parts of a program can be fine-tuned by an expert, while the rest can stay high-level.

Chapel's approach is different from HPF in the sense that it does not extend an existing programming language. Extending a programming language comes with the price that programmers carry knowledge and experience, which does not necessarily fit with the paradigm of scalable computing [19]. Chapel supports interoperability with other languages, which forces programmers to select which parts to port actively and which parts that can be left as they are.

2.6.4 Python

Python is an interpreted high-level programming language for general-purpose programming. It is especially known for its philosophy of producing user-friendly and readable code. Python uses a purely dynamic type system, meaning that it verifies the types during execution and functions are inherently polymorphic. It also provides automatic memory management by a mix of a reference counting and a cycle-detecting garbage collector. Python is multi-paradigm since it supports object-oriented, imperative, functional, reflective, and procedural programming paradigms [6].

The reference interpreter implementation of Python is written in the programming language C and is, therefore, called *CPython*. A fundamental limitation of CPython is the *Global Interpreter Lock* (GIL), which enforces the interpreter to disallow any parallelism within a process. A process can have multiple threads, which is executed sequentially through some form of concurrency. Parallelism within CPython is achievable by executing each parallel thread within a dedicated interpreter-process or calling external methods which are not dependent on the Python interpreter. The Python interpreter can start an external method, continue execution and receive the result of the external method later.

Numpy

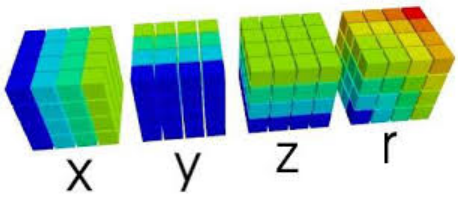
The framework Numpy (Numerical Python) extends CPython with the array programming model. Numpy uses a homogeneously-typed multi-dimensional array object called *nd-array*. Numpy serves the dual purpose of providing a high-level interface for array programming and producing efficient code by utilizing the properties of this interface.

Numpy supports many operations on nd-arrays. A large part of the operations is inspired by linear algebra, due to the similarity between the nd-arrays and mathematical tensors. Numpy utilizes fine-tuned external libraries, such as LAPACK and BLAS, for executing numerical operations on nd-arrays. Nd-arrays are allowed to be views into memory buffers allocated by C/C++, Fortran or CPython, which avoids extensive copying when utilizing external libraries. Numpy is somewhat able to achieve parallelism despite the GIL by calling external operations. The external methods are also utilizing internal parallelism. Since Numpy uses underlying APIs for executing operations, it does not perform any fusion of these operations.

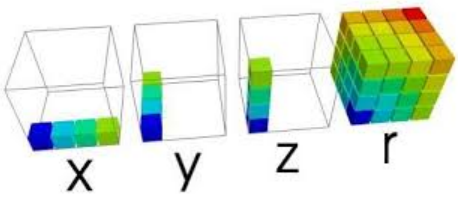
Views Apart from numerical operations, Numpy also supports many measures of generating and modifying nd-arrays. An essential feature of nd-arrays is *views*. A view is a subset of an nd-array, which itself behaves as an nd-array. As the name suggests, it is a view into the nd-array. When a view is modified, it directly modifies the corresponding elements in the nd-array. Views are often created by indexing an nd-array with slices. A slice is consists of 3 optional colon-separated integers, which corresponds to the start, the end and the stride. The default start is 0, the default end is the last element, and the default stride is 1. The user can provide a slice for each dimension and thereby get a view into nd-array. Views can also be created by providing direct indices or elements that satisfy a predicate.

A view also supports changing the shape and stride of the elements of an nd-array, known as reshaping. As an example, it is possible to reshape a view of a 1D-array with nine elements into a 3×3 2D-array.

A view also supports blowing up the shape of an nd-array by broadcasting, which replicates the elements in a structured fashion. An example of broadcasting can be seen in Figure 2.3. Array operations automatically broadcast views (when possible) if they do not match in shape. Otherwise, it results in a runtime error.



(a) Illustration of addition of dense arrays with Numpy



(b) Illustration of addition of vectors that are broadcasted by Numpy

Figure 2.3: An illustration of addition with dense and broadcasted nd-arrays. Illustration borrowed from [31]

Chapter 3

The Bohrium Runtime System

This chapter focuses on *The Bohrium Runtime System* (Bohrium) developed at the Niels Bohr Institute [26]. The loop extensions of this thesis are based upon Bohrium, making its structure and implementation a core subject.

Bohrium is an extensive system with many features and structured components for supporting these. The first part of this chapter motivates the creation of Bohrium and presents a general overview. The subsequent parts dig deeper into details of each component. Many of the components in Bohrium are highly complicated. This chapter mainly focuses on the relevant components for the implemented loop extensions (described in Chapter 4), while briefly describing the others.

3.1 Overview

The essence of Bohrium is transforming high-level code from a high-productivity language or framework to low-level code in a high-performance language. This process from high-level code to efficient architecture-specific executables involves multiple stages, referred to as the *Bohrium stack*. This section describes the stages of the stack and their responsibility briefly.

The distinct layers within the Bohrium stack can be boiled down to the following:

bridge The bridge maps array operations from the external high-level programming language or framework to Bohrium's intermediate representation called vector bytecode. The bridge is written in the external programming language.

vector-engine manager (VEM) The VEM manages the ownership and location of arrays. When using a distributed architecture, the VEM also distributes jobs between the vector engines.

filter The filters analyze and apply transformations to vector bytecode. The purpose of the transformations is often optimization. Filters support usage within or between any component that operates on an instruction list.

vector-engine (VE) The VE constructs an architecture-specific implementation based on the manipulated vector bytecode. The produced source code files are then compiled using an existing compiler and executed on the relevant piece of hardware.

The flow from high-level array programming to low-level executables goes through the following steps:

1. The bridge parses the high-level array programming of the external programming language or framework to vector bytecode. The vector bytecode is gathered in an instruction list until an operation forces a *flush*. A flush transfers the instruction list to the Bohrium core.
2. The instruction list is manipulated and transformed during runtime for optimization purposes, with no pre-execution analysis.
3. The instruction list is translated into source code in the architecture-specific language.
4. The source code is compiled *just-in-time* using an external compiler.
5. The executables are executed on the relevant hardware.

The complete structure of the Bohrium stack is visualized in Figure 3.1.

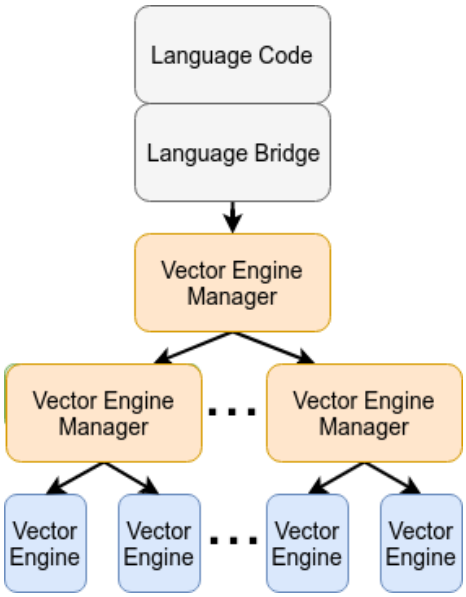


Figure 3.1: An illustration of the flow between components in Bohrium

3.2 Vector Bytecode

Bohrium has an intermediate representation based on array programming, referred to as *vector bytecode*. The bytecode has an instruction set inspired by

RISC, where instructions are bound to memory, arithmetic or logical operations. Many of the instructions use *three-address code* with an assignment array, two arguments arrays combined with a binary operator.

The vector bytecode is based on array programming operations and does not support control flow, which simplifies the bytecode. The simplification allows more optimization but limits the expressiveness of the bytecode (described in Section 2.4.3). The array programming model is represented directly in the bytecode, where memory instructions operate on arrays, or views into arrays, instead of registers. The bytecode instructions have no further control over the arrays besides reading and writing. The bridge is responsible for creating arrays and views unless the array is created using a generate bytecode. Arrays and views within an operation must have correct structure. The vector bytecode does not include any support for broadcasting or reshaping.

The bridge gathers bytecode instructions in an ordered instruction list called *Bohrium Intermediate Representation*. The instruction list can be manipulated several times by filters. In the end, the vector engine translates the final instruction list into executables. All runtime components within Bohrium communicates through vector bytecode and are fully bridge-independent.

3.2.1 Operation Classes

Bohrium has 83 different bytecodes for expressing operations on arrays. Many of these operations are conceptually the same in structure but uses different operators on the data. The operations structures can be boiled down to the eight conceptual classes:

Generate bytecodes create data in a structured fashion. The supported operations are scalar replication, range $[0..N]$ and random numbers.

Element-wise These bytecodes apply an operator to all elements within arrays. Unary operators operate on a single array, and binary operators operate on two arrays. The output array can only overlap with input arrays if they access data in the same pattern. There are 53 supported element-wise operations including addition, multiplication, sinus, and logarithm. Due to the operations being stateless, combined with the limitations on data overlap, they allow parallel computation of each data point or paired data points.

Reduction These bytecodes reduce the dimensionality of the input array by using a binary operator, where one of the arguments is an accumulator. The reduction-operator must be associative, and the input must not overlap with the output. Due to these properties, reductions supports parallel execution. There are ten supported reductions, which include addition, multiplication, and logical-and.

Scan These bytecodes accumulate the results of using a binary operator on an input array. As with reductions, the operator must be associative and data overlap between input and output is disallowed. By utilizing these properties, the scan operation can be parallelized as well [16]. The supported scan operations are addition and multiplication.

Gather The bytecode perform an indexed read from an array. Given an input array, an output array, and an array of indices, the gather operation writes the indexed data elements in the input to the of the output. The output and indices must have of the same length.

Scatter The bytecode perform an indexed write to an array. It is very similar to gather and takes the same arguments. The difference is, scatter writes the data elements from the input to the indices of the output. The input and indices must have the same length.

Data management These bytecodes handle data management. It consists of a free operation that deallocates arrays and related metadata.

Extension methods All the bytecodes above represent the core of Bohrium and are within the array programming model. However, for some programs or algorithms, these operations are not efficient or expressive enough. Introducing a special `extension method` bytecode solves this issue. Bohrium does not impose any restrictions on extension methods whatsoever. The consequence of this freedom is that Bohrium cannot guarantee correct execution. A common limitation is extension methods only targets a specific hardware platform. The user registers extension methods with paths for execution and receives a handle. This handle can be used like any other operation and is present in the instruction list through the extension method bytecode. Some examples of extension methods are operations from highly optimized libraries, such as BLAS (described in Section 2.5.2) for CPUs or CUBLAS for NVIDIA GPUs.

3.2.2 Bytecode Example

This section gives a concrete example of the vector bytecode of a Python program using Numpy, parsed by the Numpy bridge. Figure 3.2a shows an example of multiple operations on two nd-arrays. The `arange` function generates one of the nd-arrays, while the other nd-array receives its data elements from a list of floats. The operations include addition, multiplication, squaring and sinus. A pretty print of the bytecode produced from this example is shown seen in Figure 3.2b. The bytecode shows that array b uses a generator bytecode and the power-of-2 function translates into a multiplication. The views are also shown directly in the bytecode. The rest is of the bytecode is straightforward.

3.3 Bridge

The bridge is a component that *bridges* programs expressed in predefined external programming languages to vector bytecode. The choice of programming language is not biased towards any particular paradigm, as long as the language is compatible with the array programming model [26]. The array programming paradigm can also be available through a framework within the programming language.

Different external languages are all integrated into Bohrium through a language-specific bridge. Bohrium currently contains bridges for Numpy (Python), C++,

```
a = np.array([5.0, 7.0, 2.0, 1.0])
b = np.arange(4)
c = a * b
c = c ** 2
d = np.sin(c[0:2] + a[2:4])
```

(a) A code example of performing multiple operations to array using Numpy.

```
BH_RANGE a2[0:4:1]
BH_IDENTITY a3[0:4:1] a2[0:4:1]
BH_FREE a2[0:4:1]
BH_IDENTITY a4[0:4:1] a3[0:4:1]
BH_MULTIPLY a5[0:4:1] a1[0:4:1] a4[0:4:1]
BH_FREE a4[0:4:1]
BH_MULTIPLY a6[0:4:1] a5[0:4:1] a5[0:4:1]
BH_FREE a5[0:4:1]
BH_ADD a7[0:2:1] a6[0:2:1] a1[2:4:1]
BH_SIN a8[0:2:1] a7[0:2:1]
BH_FREE a7[0:2:1]
BH_FREE a1[0:4:1]
BH_FREE a6[0:4:1]
BH_FREE a3[0:4:1]
BH_FREE a8[0:2:1]
```

(b) The vector bytecode corresponding to the example above, parsed by the Numpy bridge.

Figure 3.2: An example of a Python program using Numpy and the corresponding vector bytecode, parsed by the Numpy bridge.

C, and CIL. The Numpy bridge is the primary focus of Bohrium, which the content of this chapter reflects.

3.3.1 Responsibility

Concretely, the bridge maps external source code into vector bytecode. The runtime components then handle this bytecode. The essential responsibility of the bridge is ensuring this mapping returns valid and semantically correct vector bytecode. General tasks that apply to multiple bridges should, as widely as possible, be handled by the runtime components to avoid rewriting identical functionality in multiple bridges.

A bridge is typically implemented within the external language it supports and is naturally dependent on its characteristics. It is encouraged to utilize the available tools for simple optimization or validating input. As an example, the type checker of the external language can be used to detect illegal operations and return runtime errors in user-written high-level code.

It is also of great importance that the bridge makes it as intuitive as possible to write code, which can be translated to vector bytecode. Therefore, the Numpy bridge aims to implement functionalities through an interface which is almost identical to the Numpy interface. In many cases, the user only has to import Bohrium instead of Numpy and the program is automatically executed with Bohrium, without modifications to the source code.

As mentioned briefly in Section 3.2, the bridge is responsible for providing

the means of manipulation and reshaping arrays. This translates to 4 general operations that allow convenient interaction with multiple subsets of the same array. The distinction between a view and an array is that a view imposes structure on a subset of the array, while the array contains the data. The term array can also refer to a view that contains the whole array in its original form.

Aliases/views The bridge must support creating a view based on another view. Since views are not allowed to be nested (see Section 3.4), it involves creating a new view with metadata, pointing to the same array.

Slicing The bridge must also support creating a view based on a subset of an existing view. The user should be able to modify stride, shape and offset of the view. These changes should then be adjusted, such that the metadata represents the slice expressed in the external language. The bridge should provide a user-friendly syntax for slicing.

Reshaping Views should be able to change shape. This involves both adding/removing dimensions which affects the strides. The bridge should handle the general reshaping of any change allowed in the external language.

Broadcasting Broadcasting a view refers to enlarging a view by repeating elements (described in Section 2.6.4). An example could be broadcasting a view with a single element into a vector, or a vector into a matrix.

3.3.2 `CXX`: Bohrium for C++

The Bohrium bridge for C++ serves a dual purpose. It is both a gateway for all other bridges, as well as a self-contained domain-specific embedded language for array programming in C++. All bridges must go through the C++ bridge, which interacts with the runtime components through a C-interface.

3.3.3 `npbackend`: Bohrium for Numpy

Numpy (Described in 2.6.4) is an array programming framework for Python. The Numpy bridge is based on Python as a host language, but completely reliant on the Numpy framework for providing the array programming features.

Numpy resolves around the multi-dimensional array (*nd-array*) as the central data object. Numpy has an extensive interface supporting many operations on nd-arrays. The approach of the Numpy bridge is to implement as large a part of the features in Numpy interface as possible. Due to the extensiveness of this task, Bohrium can fall back to Numpy, if a feature is not implemented. Fallbacks to Numpy introduces overhead while the execution is only as fast as Numpy.

The Numpy bridge offers many features, such as all bytecode operations on arrays, random number generation, primitive loops, disk I/O and more complex linear algebra operations, including a linear matrix solver and LU decomposition.

The Numpy bridge consists of a mix between Python and the C-API of *CPython* (the standard Python interpreter). The C-API supports defining Python functions within C++, which is handy for interacting with the C++ bridge.

Memory spaces

The Numpy bridge operates with two memory spaces: The Bohrium Memory Space (BMS) and the Numpy Memory Space (NMS). Arrays are in the NMS as default but can be set to use the BMS explicitly by setting a parameter. An array is transferred from one memory space to another in two circumstances:

Numpy to Bohrium When an array in the NMS occurs in an operation that can be translated to a Bohrium operation, the array is transferred to the BMS.

Bohrium to Numpy When an array in the BMS occurs in an operation that cannot be translated to a Bohrium operation, the array is transferred to the NMS.

It should be noted, that the arrays are not copied between the memory spaces if Bohrium uses the node VEM. The relevant memory pages are remapped between the memory spaces using `mremap`. This does still involve the overhead remapping memory pages. The memory pages are also locked by `mprotect`. Accessing an array in another memory space will trigger a segmentation fault. If Bohrium uses another VEM, the data have to be transferred from another machine. The cluster VEM requires communication over a local network, while the proxy VEM requires communication over the internet.

3.4 Array Representation

Bohrium's internal array representation is split into two structures called *base arrays* and *views*. Base arrays contain a pointer to the continuous segment of data within the array, together with the number of elements and the type of the elements. Elements within an array must all have the same type. Views have the purpose of structuring the data within a base array. A view can expose the full base array or be limited to a structured subset of it. Views contain shape, stride between the data elements in each dimension and an offset from the beginning of the data.

Bohrium instructions operate on views. A view is a simple structure that is not dependent upon other views. Creating a view based on another view is possible by inheriting the reference to the base array while modifying the metadata. Thus, views do not have any unique identification, and only appears as an operand to a single instruction. If a view is used multiple times in the high-level code, it creates a new each time. Using simple views makes it easy to transform and fuse views since it does not have side-effects on other views. A base array can have multiple views enforcing structure upon it, but a view only structures a single base array. The bridge is responsible for creating and modifying views. Therefore, there are no bytecodes that represent such changes to views.

An example of the internal data structures of a view into an array is visualized in Figure 3.3.

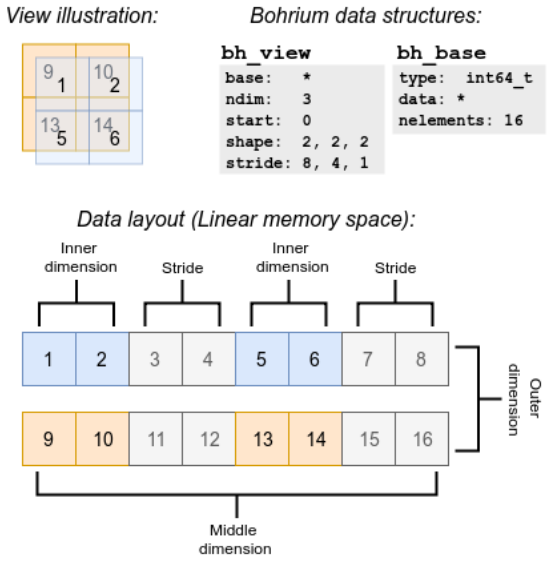


Figure 3.3: Illustration of an array, its data layout and internal Bohrium data structures

3.5 Runtime components

The runtime components are a collection of components handling the transition from the internal vector bytecode to architecture-specific executables. As part of this process, the vector bytecode can be transformed and optimized to gain performance. Compiling source code and executing the executables is also a responsibility of the runtime components.

The runtime components are a collection of four types of components. These components are mutually independent and can be replaced or modified while leaving the others intact. The components communicate through vector bytecode. The categories are filters, VEMs, VEs, and caches. The individual components are described in detail later on in this section.

The runtime components interact in the following fashion:

1. The VEM passes bytecode on to VE as an instruction list.
2. The VE hashes the instruction list and performs a lookup in the codegen cache. A hit results in reusing previously generated code, while a miss results in generating the code.
3. The generated code is hashed and searched for in the compilation cache, which is a check of whether a compiled file exists. If the file does not exist, the sources code is compiled into an executable.
4. A filter modifies the instruction list. It can be applied anywhere within or in between components. Filters may also have a dedicated cache.

It is important to notice that each runtime component introduces some form of overhead. The overhead of all runtime components is relative to the length

of the instruction list. However, it is faster for the cache to hash a longer instruction list than for a fusion-filter to perform sophisticated analysis. Thus, the instruction list should contain operations computation-heavy enough to dominate the overhead.

3.5.1 Filters

A filter is, in essence, a function that takes an instruction list as input, manipulates it in some form, and returns a modified instruction list. Filters are a very general construct that can serve any purpose that does not fit naturally in any of the predefined components. It is straightforward to implement a filter since it only depends on inputted the instruction list. Filters can modify an instruction list within or between any of the runtime components. Currently, Bohrium has three core filters and some extra filters for printing and debugging. The core filters have the following functionalities:

fuser The fuser performs a dependency analysis on the instruction list to contract or modify the composition of array operations. Due to the complexity of this process, the user can choose between 6 algorithms of transformation. The fuser has a great influence on the performance of the generated code.

bccon The `bccon` filter is used for contracting sequences of vector bytecode. The filter detects whether a single or sequence of bytecode is replaceable by a more efficient approach. An example is replacing the bytecode `POW a, a, 2` by the faster `MUL a, a, a`. It is also able to map certain instruction sequences to highly optimized external methods, such as matrix multiplication

bcexp The `bcexp` filter is used for expanding sequences of vector bytecode. It might seem contrary to the filter above, but in some cases, it can lead to faster performance. The filter has a specialized version for both CPU and GPU.

3.5.2 Vector Engine Manager

The vector engine manager handles the vector engine(s). It handles the global address space, which differs depending on the hardware complexity. A single node has a shared address space, allowing the instructions to be passed through to the single vector engine. In cluster setups, a machine is set up to handle the global address space and distribute jobs between child nodes. The cluster setup is highly complex and is not within the scope of this thesis and therefore not discussed further.

3.5.3 Vector Engine

The purpose of the Vector Engine (VE) is to generate source code in an architecture-specific programming language based on an instruction list. Bohrium currently supports three backend languages:

C99 with OpenMP Uses the C programming language (version 99) as the base language and uses the library OpenMP for parallelization within C. This vector engine is targeting shared memory multicore CPUs with non-uniform memory access.

OpenCL Uses the OpenCL programming language to target GPUs in general.

CUDA Uses the CUDA programming language to target NVIDIA GPUs.

Different VEs can have considerable differences in how they produce and execute binaries. The process of generating architecture-specific code within each VE is relatively complicated. The details are not crucial for the loop extensions and not discussed further.

3.5.4 Caches

Caching is an effective mechanism for gaining performance by reusing previous work. Bohrium has support for caching work at the three following stages:

Fuse The fuser is a filter, only reliant on an instruction list as input, allowing caching of the resulting fusion of the instruction list. The input instruction list is hashed based upon important factors in the fuser. Before fusing an instruction list, the fuser checks whether the hashed instruction list is in the cache. If not, the fusion is performed and stored in the cache.

Codegen The code generation produces a string based on an instruction list. A hashmap stores the string with instruction list as key. Before generating the code, the VE performs a look-up in the codegen cache.

Compilation The compilation cache hashes the source code string and produces a file with the hash as a name. If this file already exists, it is used directly without the need for compiling the file.

Note that the compilation cache is maintained between executions, as long as the files exist in the cache folder. The other caches are not temporary and therefore not reusable between executions.

3.6 Loops

Bohrium supports native loops within Python. A native loop refers to using the loop-keywords within Python, which is either `for` or `while`. Using such loops within the Numpy bridge results in parsing the loop as a continuous instruction stream, just as an unrolled loop. The parsing can lead to poor performance in two scenarios:

Many iterations A loop that goes on for many iterations leads to several issues. The instruction stream going into Bohrium will become large and make internal optimizations, such as fusion, much slower to perform. Since there is no natural flush between the iterations, the caches do not contain an entry for the loop body itself.

Iterative loops An iterative loop that contains a control flow, handling when to break out of the loop, leads to a natural flush between Bohrium and Numpy in each iteration. It allows efficient use of cache but comes with the price of many flushes, linear to the number of iterations.

An alternative to the native loops within Python is the internal flush-and-repeat loop feature of Bohrium. Since Bohrium as based upon array programming, it might seem odd to support loops explicitly. This topic of mixing loops and array programming is discussed in both Section 2.4.3 and 4.1.1.

3.6.1 `do_while` Loop

Bohrium has a limited internal flush-and-repeat loop, which bypasses the need for parsing the loop body multiple times and flushing it to the runtime components. The functionality is offered through a *do while* loop in Python. A do while loop requires a loop body and a condition. The loop executes the body repeatedly until the condition is no longer satisfied. The loop checks the condition after each iteration. The loop executes the body at least once. A *for loop* can be derived from a do while loop by having the number of iterations as the condition.

Bohrium's Numpy bridge offers the flush-and-repeat loop through the function `do_while`, taking the following arguments:

func A function with the body of the loop. The function can take any number of arguments (and keyword arguments) and may return a condition array with one boolean element, indicating whether the condition is satisfied. If the function returns `None`, the function is called `niters` times.

niters The maximum number of iterations the loop is allowed to continue. If the argument is `None`, then the maximum iterations are determined by a default maximum in Bohrium. This argument corresponds to adding `and i < niters` to the condition and increasing `i` by one at the end of the loop body.

***args, **kwargs** The arguments for `func`.

The Numpy bridge gathers the array operations in the loop body and passes them onto the C++ bridge as an instruction list. Before parsing the loop body, `do_while` performs a flush to ensure that the next instruction list only contains the operations in the loop body. After the interpretation, it performs two checks:

1. The loop body must not contain any flushes.
2. The condition, if there is one, must have the correct format.

If these checks pass, `do_while` flushes the instruction list to the C++ bridge. The instruction list contains the condition array and maximum amount of iterations as metadata.

The runtime components handle the execution of the instruction list. The instruction list is kept within Bohrium and is repeatedly passed through the VE until either the condition is false or the loop reaches the maximum iterations.

The condition must be evaluated in Python and is synced to avoid automatically freeing the array.

The amount of overhead involved in the execution of an iteration in a loop bounds performance difference between native Python loops and `do_while`. If the loop body contains relatively light work, the overhead becomes a factor and `do_while` will give a noticeable speedup. On the other hand, if a loop body contains very heavy computations, the speedup of using `do_while` will be minimal.

The `do_while` loop is very dependent upon the caches since the same instruction list repeated in VE each iteration. During the loop, there is no overhead in regards to the Numpy bridge.

The fact that `do_while` uses the flush-and-repeat structure has pros and cons. Naturally, executing the same instruction list leads to good cache performance. It also fits well with the structure of Bohrium, since it only requires features, which are already implemented. On the other hand, the loop bodies have a forced static structure. The instruction list is the same in each iteration, which limits the expressiveness of loop bodies. The only dynamic factor is the data within arrays.

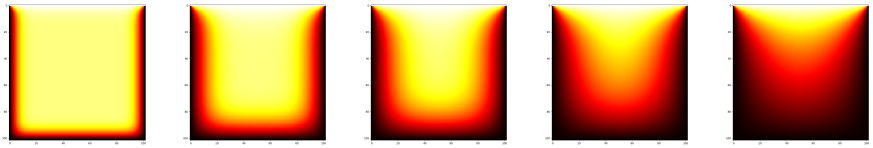
Performance

The motivation of Bohrium's `do_while` is reducing overhead. The loop body is only parsed once, which reduces the amounts of Python object allocations and flushes. Each time the Python interpreter encounters a view, it allocates space and creates a Python Object. The instruction list of the loop body stays within Bohrium and thus only needs to be parsed once, instead of once per iteration. It gives a noticeable difference concerning overhead (shown empirically later in this section).

The overhead involved in executing a loop is not limited to the Numpy bridge. The vector engine still needs to perform specific tasks before executing the architecture-specific executable. It involves

1. Performing look-ups in the code generation, compilation, and fusion caches. If any of these does not result in a hit, the task must be executed and put into the corresponding cache.
2. Preparing the arguments to serve to the executable.
3. Cleaning up and deallocating temporary arrays.

The Heat Equation benchmark is used to illustrate the difference in performance between using `do_while` and a native Python loop. Heat Equation is a benchmark that describes the distribution of heat in a region over a given period. The input matrix represents an area, where the value of each point corresponds to its temperature. For each iteration, the heat of each point becomes the average of its heat and the heat of its four neighbors. It is a mix between a simulation and an iterative benchmark (described in Section 5.3.1). The benchmark simulates a distribution of heat, but will at some point reach a fix-point. An illustration of the Heat Equation, after a variable amount of iterations, can be seen in Figure 3.4.

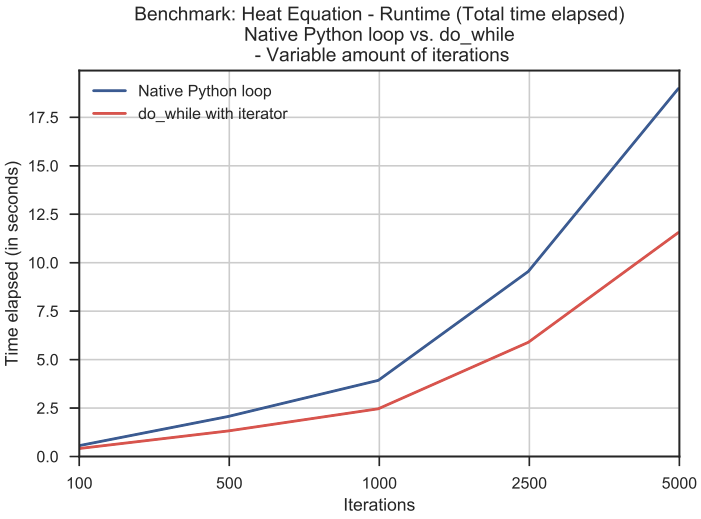


(a) 100 it **(b)** 500 it **(c)** 1000 it **(d)** 2500 it **(e)** 5000 it

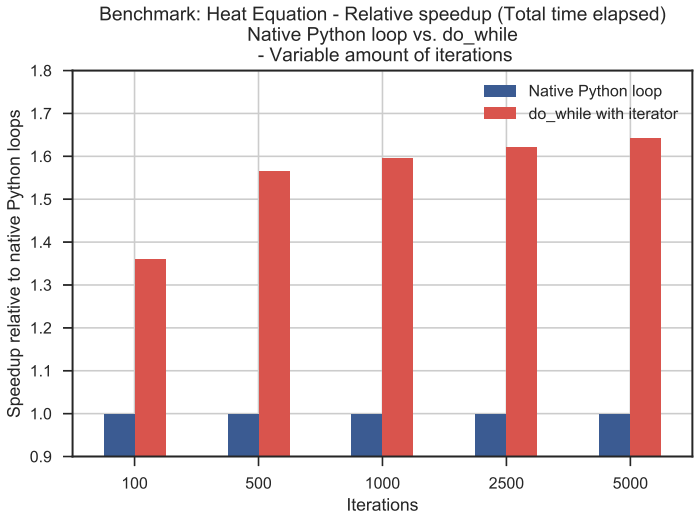
Figure 3.4: Illustration of the heat distribution in the region after 100, 500, 1000, 2500 and 5000 iterations of the Heat Equation on a 100×100 region.

The Heat Equation benchmarks use a variable number of *max iterations* to illustrate the performance difference related loop iterations. The Heat Equation has a condition, which is synced between each iteration and evaluated in Python. The benchmarks use a 1000×1000 input region.

The difference in the total time spent is shown in Figure 3.5 and the difference in time spent outside the runtime components is shown in 3.6. As these figures illustrate, the overhead decreases significantly by using the `do_while` as opposed to native Python loops. The decrease in overhead relates to the fact that the Heat Equation fits inside a single `do_while` loop body, reducing the overhead of Python interpretation to a constant amount. The performance gain by using `do_while` goes towards $\approx 1.7\times$ speedup in total time elapsed. It shows the relation between the overhead and execution time in each iteration. If the size of the input matrix increases, the performance gain of `do_while` decreases since the overhead becomes less of a factor in each iteration. On the other hand, decreasing the input matrix leads to a greater performance gain.

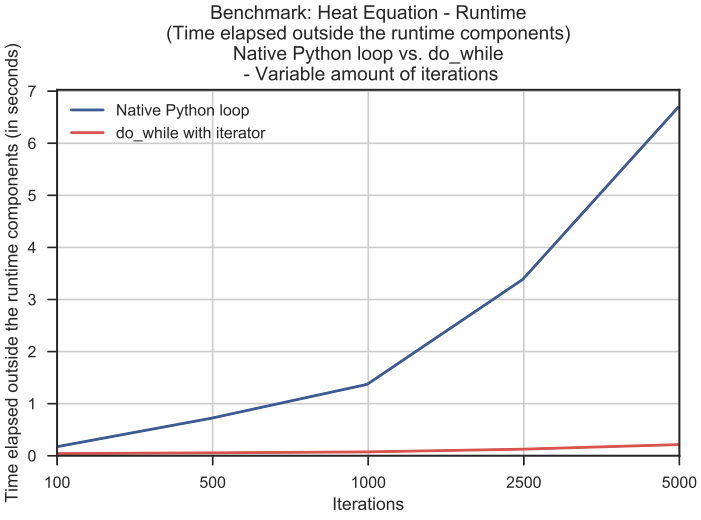


(a) Total time elapsed without compilation

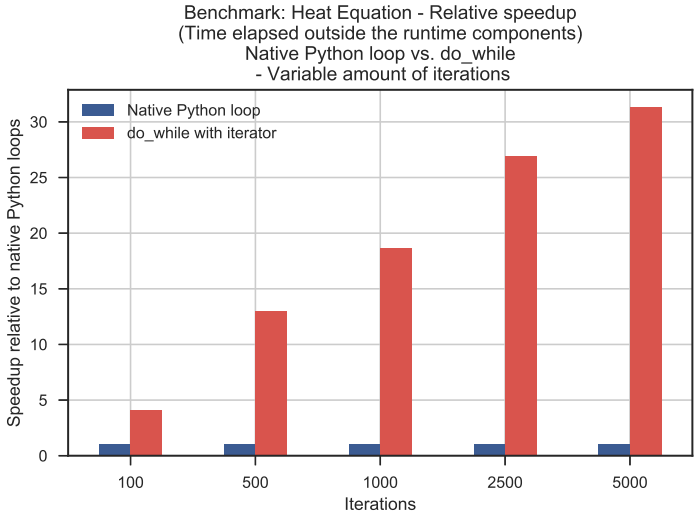


(b) Relative speedup

Figure 3.5: Absolute time difference and relative speedup in total time elapsed between using internal loop and native loops. The input matrix is fixed to 1000×1000, while the amount of iterations is variable.



(a) Total time elapsed without compilation



(b) Relative speedup

Figure 3.6: Absolute time difference and relative speedup in time elapsed outside the runtime components between using internal loop and native loops. The input matrix is fixed to 1000×1000, while the amount of iterations is variable.

Chapter 4

Loop extensions

This chapter gives an overview of the loop extensions implemented during this thesis. The contributions are split into two categories, referred to as *sliding views* and *array indexing*. Both of these contributions are based the flush-and-repeat loop structure and `do_while` within the Numpy bridge (described in Section 3.6).

The official release of Bohrium has adopted the sliding view loop extension. Master branch of the official Bohrium GitHub contains the source code of the sliding view implementation, accessible at [2]. The source code additions and modifications of the sliding view loop extension can be seen through two pull requests at [13].

The array indexing loop extension is not a part of the official Bohrium release. The source code for this implementation can be found in the `ArrayIndexing` branch of in fork of Bohrium at [12].

4.1 Motivation

Bohrium targets accelerating array operations in external programming languages and frameworks. Efficiency is, therefore, in the heart of Bohrium. Even though Bohrium focuses on handling pure array programming, it be would be a great benefit to support loops without introducing a vast amount of overhead. Reducing loop overhead is the main purpose of Bohrium's internal flush-and-repeat structure, which it achieves to a certain extent. The essential motivation for the implementations presented in this chapter can be boiled down to the two following subjects:

1. Combining array programming and efficient high-level loops in Bohrium.
2. Extending the existing flush-and-repeat loop with dynamic features.

4.1.1 Array Programming and Loops

Bohrium accelerates array programming through *just-in-time* optimization and compilation of low-level source code. As discussed in Section 2.4, array programming does not encourage the explicit usage of loops within programs.

Optimization by operating on data-collections with bulk operators is a fundamental aspect of array programming, which using explicit loops compromise. The natural question that arises is then: *Why is it interesting to focus on loops within a framework based on array programming?*

Answering this question requires a precise definition of how loops operate on different levels within a program. Thus, the notion of *high-level* and *low-level* loops is introduced.

Low-level loops handle the underlying implementation of high-level operations with repetitive structures. An example is *nd*-array addition, where corresponding indices of the input arrays are added and stored in a result array. The low-level loop approach is applying a nested amount of loops, corresponding to the shape of the arrays. However, this requires that the program explicitly utilize the inherent parallelism of the operation. Array programming frameworks have specialized implementations that achieve parallel execution under the surface. The source code Bohrium produces in C also uses loops, but with OpenMP for parallelization.

High-level loops handle structures that are inherently sequentially. However, the operations within the loop body might contain parallelism. An example is a solver for an iterative numerical problem. For each iteration, the result becomes more precise. In such a problem, the internals of the loop can be expressed through array programming (or other parallel paradigms) to exploit parallelism. The loop, therefore, utilizes inner parallelism instead of outer parallelism.

With these definitions in mind, it is correct that array programming does not encourage the usage of low-level loops, which undermines the fundamentals of the model itself. However, the usage of high-level loops can significantly expand the expressiveness of array-programming without compromising its fundamentals. Thus, it is meaningful to introduce an efficient high-level loop within Bohrium, as long as it utilizes the internal parallelism of the body.

4.1.2 Bohrium's Internal `do_while` Loop

Bohrium supports an internal flush-and-repeat loop structure. It is accessible in the Numpy bridge through by the function `do_while`. Reducing the overhead of the repetitive Python interpretation of loops was the primary motivation for introducing `do_while`. The current implementation of `do_while` successfully bypasses this overhead but has certain limitations.

Limitations

Loop bodies in the flush-and-repeat loop are very static in the current implementation. The loop body is only parsed once in Python and repeatedly executed within Bohrium. Therefore, the loop body never changes between the iterations.

Flush-and-repeat and `do_while` has the following limitations:

No flushes `do_while` does not allow flushes within the loop body. A flush passes the current instruction list to Bohrium and forces an execution.

Thus, flushes split the loop body into multiple instruction lists, making it more complicated to reassemble within the runtime components. Flushes often occur when Python is dependent upon a value within Bohrium. Since the goal of `do_while` is avoiding Python evaluation in each loop iteration, it disallows flushes.

Dynamic branching The first iteration of the loop body is parsed by the Python interpreter and the Numpy bridge. The Numpy bridge gathers the array operations in an instruction list, which is handled by Bohrium when a flush occurs. The Python interpreter handles the rest. Thus, the instruction list only contains array operations from the first iteration. Since Bohrium repeatedly executes this instruction list, the execution will only reflect the branches taken during the first iteration. `do_while` does, therefore, not support dynamic branching within the loop body.

No nesting of `do_while` `do_while` is a high-level function defined within Python as part of the Numpy bridge. It is not a part of the vector bytecode. As explained earlier, `do_while` forces a flush at the beginning and one at the end. Since flushes within `do_while` is disallowed, the function cannot be nested.

Dynamic views The creation of views is a general responsibility of the bridge unless it uses a generate bytecode (described in Section 3.2.1). Since the Numpy bridge only creates the views as they appear in the first iteration, the views never change.

It is durable to reassemble the multiple instruction lists that an internal flush causes. However, handling the evaluation Python perform requires extensive additions to the bytecode. Extending the bytecode raises the complexity of internal analyzation and optimization and is not an option. Thus, allowing flushes is not a focus as a loop extension.

Supporting dynamic branches leads to similar problems. The lack of control flow within the bytecode is a design choice. Without supporting branches within the bytecode, having varying conditionals does not fit within `do_while`.

Nested `do_while` cannot be implemented without having multiple instruction lists as part of the loop body. Therefore, it depends upon having multiple flushes.

The lacking support of dynamic views is a limitation, which is not directly in conflict with Bohrium. It can be achieved with reasonable additions and modifications to Bohrium while preserving the vector bytecode. The focus of the loop extensions is, therefore, to support dynamic views within flush-and-repeat and `do_while`.

4.2 Sliding Views

The notion *sliding view* refers to a view sliding with a fixed amount between each loop iteration. Sliding views are a common approach to change views based on the iteration counter. An example of a sliding view within Python using Numpy is showed in Figure 4.1. In this example, the sliding view has a static shape of 5 and slides by 1 in the first dimension in each iteration.

```
a = np.zeros(10)
for i in range(6):
  a[i:i+5] += 1
# Resulting a: [ 1, 2, 3, 4, 5, 5, 4, 3, 2, 1 ]
```

Figure 4.1: Example of a sliding view within a loop written in Python with Numpy

As described in the motivation, Bohrium's internal flush-and-repeat loop does not natively support sliding views due to the static loop body.

This section covers the implementation of sliding views as part of the flush-and-repeat structure. It describes features added to the Bohrium core, as well as extensions to `do_while` in Python for handles sliding views using a syntax close Numpy's.

4.2.1 Overview

In Python, a view into an array is often created by indexing with a *slice*. A slice contains a *start*, *stop* and *stride*. The indexing can have a slice for each dimension of the array. The view is constructed by using the three attributes in the slice in the following fashion:

Start The start defines offset in the given dimension. It also contributes to the shape in the given dimension.

Stop The stop contributes to the shape in the given dimension.

Stride The strides defines the spacing between elements. If the stride is 2 in a dimension, then the view contains every second element. For example, a 10×20 2D-array has a stride of 20 in the first dimension and a stride of 1 in the second. If it is indexed with a slice with stride 2 in the first dimension, then the stride of the view becomes 40.

The shape of the view (in the sliced dimension) is determined by

$$\frac{slice_{stop} - slice_{start}}{slice_{stride}}.$$

The same goes for Bohrium's view structure. It contains a data pointer to a base array, the dimensionality, offset, shape and stride (described in Section 3.4). The shape and stride contain a value for each dimension.

Sliding views are constructed based on indexing views with slices containing at least one iterator. Since iterators may change between each iteration, the views may also change based on the iterator.

In the example in Figure 4.1, the iterator is the variable i. The slice that indexes a has the start i, end $i + 5$ and stride 1 (the default stride). The slice forces the view to slide, since i increases by one between each iteration. The sliding view into a does not change shape but increases its offset in the first dimension by one between each iteration.

Sliding Views in Flush-and-repeat

The idea behind introducing sliding views in the flush-and-repeat loop structure is to create metadata within views that define the changes between each iteration. The concrete changes to a view are added through using a custom *iterator object* (described later in Section 4.2.4). As for now, let's assume the function $slide$ returns the slide that an iterator produces in an expression. Then, it is possible to decide the changes to a view between each iteration in the following manner:

Offset The change to offset is purely based on $slide(slice_{start})$. If the slide is 0, the offset is static. Otherwise, the slide is added to the offset after each iteration.

Shape The change in shape (in the given dimension) is determined by $slide(slice_{stop}) - slide(slice_{start})$.

Figure 4.2 shows a view that both slides and changes shape. The offset is slid by 1, and the shape is changed by $2 - 1 = 1$.

```
a = np.zeros(9)
for i in range(5):
  a[i:2*i+1] += 1
# resuling a: [ 1, 1, 2, 2, 3, 2, 2, 1, 1]
```

Figure 4.2: Example of a sliding view with changing shape

The following sections cover extending metadata of Bohrium's view structures to support sliding and shape change, updating the sliding views between each iteration within the flush-and-repeat structure and usage of sliding views within Python through the Numpy bridge.

4.2.2 Extending Metadata of Views

A view is a structure for organizing a subset of a base array, which points to a continuous data segment. It does so through metadata, which imposes structure on the array. Supporting sliding views within the flush-and-repeat structure requires that views carry information concerning their change between each iteration.

The Bohrium stack contains multiple stages, which make views appear in various structures. Thus, extending the metadata requires additional metadata in all these view structures. When using the Numpy bridge, the different view structures are:

Numpy bridge A view appears as a Bohrium nd-array object.

C++ bridge A view appears as a `BhArray` object.

Bohrium core A view appears as a `bh_view` struct.

Firstly, the Numpy bridge parses the views and creates the corresponding Bohrium nd-array objects. Views become a part of the instruction list through

array operations, which use views as operands. The Numpy bridge transfers the instruction list to the C++ bridge when a flush occurs. During this shift, the Numpy bridge translates the operand views into `BhArray` objects, which can be handled by the C++ bridge. The C++ bridge then translates the operand views into `bh_view` structures and passes them on to the Bohrium core.

The slide information must go through all these view structures to finally reach the Bohrium core. Therefore, it is necessary to extend the metadata of all these view structures. Each component must also correctly forward the sliding information during these shifts, such that it is not modified or lost.

Extending Metadata in the Numpy Bridge

At the Python stage, the sliding information is stored within the Bohrium ndarray as an attribute called `bhc_dyn_view`. This attribute either contains an object called `dynamic_view_info` or `None` indicating that the view does not contain any sliding information. The `dynamic_view_info` object has the following attributes:

dynamic_changes A dictionary containing all slides to the view. The key is the relevant dimension, while the value is a 5-tuple of integers. These integers indicate the change to offset, change to shape, step delay, shape of the indexed view, and stride of the indexed view.

shape A default value for the shape of the indexed view.

stride A default value for the stride of the indexed view.

resets Indicates after how many steps, the changes to the current dimension should be reset.

Note that step delay and resets is used for supporting simple nested loops. An outer loop must wait for the inner loops to finish before modifying its iterator. When an inner loop finishes, it must reset its iterator and its changes to the views.

Extending Metadata in the C++ Bridge and Bohrium Core

In the C++ bridge, `BhArray` is extended by an attribute named `slides` that contains an `bh_slide` struct. `bh_view` is also extended by a `bh_slide`. The `bh_slide` structure contains the following attributes:

offset_change A vector of changes to the offset

shape_change A vector of changes to the shape

dim A vector of the corresponding dimensions

dim_stride A vector of the stride in the given dimension

dim_shape A vector of the shape of the indexed view in the given dimension

step_delay A vector that contains the numbers of iterations to wait before applying changes in the given dimension

iteration_counter An integer that counts the amount of executed iterations

resets A map of dimension and the amount iterations before a reset

changes_since_reset A map of dimension and the changes in the offset since the last reset

Many of these attributes directly correspond to the attributes within `dynamic_view_info`. The iteration counter is used together with the step delay and resets to check whether the view should slide during the specific iteration, or whether to reset the changes to views. Subtracting the accumulative change from the offset reverts the slides. Since the shape changes linearly, it can be reverted by calculating the shape change and subtracting it from the shape.

Transferring Metadata Between Stages

The sliding information is transferred from the Numpy bridge to the C++ bridge by calling a function named `slide_view`. This function takes a view and the values corresponding to each vector within the `bh_slide` structure. A function called `add_slide_info` basically unpacks the `dynamic_view_info` and translate the information into a `slide_view` call for each change to a dimension. Since `bh_slide` is used by both C++ and the runtime system, it can be copied in shift between these components.

The `bh_slide` structure is initiated with empty vectors and maps, indicating that there is no sliding information attached to the view.

4.2.3 Updating Views Between Loop Iterations

The flush-and-repeat structure must update all sliding views between each iteration. By utilizing the slide information in the metadata, it is possible to modify the views within the specific VE.

A filter called `slide_views` modifies the sliding views after each execution of the instruction list. Using this filter only requires a single line of code and is independent of VE specifics. It is, therefore, used within all vector engines currently supported by Bohrium.

The `slide_views` Filter

The `slide_views` filter inspects all views in the instruction list, which requires inspecting each operation in the instruction since they store the views as operands.

A view is updated, if these criteria are met:

1. The view contains non-empty slide metadata.
2. The step delay is 1, indicating that view requires an update between each iteration. Alternatively, the step delay is different from 1, and the iteration counter is a non-zero multiple of the step-delay.
3. The stride in the given dimension is different from 0 (used for avoiding sliding in broadcasted dimensions).

If these criteria are met, the view is updated using the information in `bh_slide`. The dimensions containing a slide is updated in the following fashion:

1. The offset change is multiplied with the dimension stride. If this value goes beyond the boundaries of the of the indexed view, it is wrapped around, such that it starts from the other end. The change is then added to the offset of the view. Out of bounds errors are handled within the Numpy bridge and will result in a runtime error in Python.
2. The change in shape is added to the shape of the dimension
3. If there is an entry for the dimension in the reset map, a check is performed. If the iteration-counter is a non-zero multiple of the reset value, then the accumulated changes are subtracted from the shape and offset.

The views can slide in any directions by using either a positive or negative slide.

It is important to notice that the operations in the instruction list are not modified, only the operands. If no views change shape, the instruction list hits both the codegen cache and the compilation cache, since the executables take offsets as arguments. Unfortunately, the source code Bohrium produces does not have the size of the loops as arguments, resulting in creating new source code for each change in shape. The instruction list never hits the fuser cache because it is dependent upon both offset and shape.

Hitting the cache is not dependent on using flush-and-repeat. It will also happen if there is a flush at the end of a native Python loop.

Negative Indices

In Numpy, it is possible to traverse a view in reverse by using negative indices. Figure 4.3 shows an example of two programs, where one uses positive indices, and the other uses negative indices. Both programs perform the same. Supporting this behavior with iterators requires views to loop around the given dimension when the index goes below zero. The sliding information must, therefore, carry the shape of the indexed view. The `dim_shape` attribute in `bh_slide` reflects this.

```
a = np.arange(10)
for i in range(1, 11):
    print(a[-i])
```

(a) Printing numbers using negative indices

```
a = np.arange(10)
for i in range(1,11):
    print(a[len(a)-i])
```

(b) Printing numbers using positive indices

Figure 4.3: Two code snippets that prints the numbers from 9 to 0 by traversing an array in reverse order.

Dynamic Shapes

Handling changes to the shape of views are generally more complex than offset becuase they can lead to further changes of implicitly allocated arrays. Thus, the dynamic change in views must be copied to implicitly allocated arrays based on them.

Bohrium generally uses three-address bytecode with two input views, a binary operator, and an output view. If a calculation in Numpy consists of multiple instructions, the results of sub-calculations are stored within a temporary array, unless fusion can avoid it. The Numpy bridge allocates the temporary arrays. If the operand views change shape, the output view must also change shape accordingly. It is essential in this context that the runtime system relies on the bridge for detecting illegal shapes in the operands. If not, it can lead to undefined behavior and lead to incorrect results or crashes during runtime.

Figure 4.4a shows an example of using implicit temporary arrays, while Figure 4.4b shows an example of explicitly defining temporary arrays.

```
def loop_body(a,b,c,d):
  i = get_iterator()
  d[i:] += a[i:] + b[i:] - c[i:]
```

(a) Example using implicit temporary arrays

```
def loop_body(a,b,c,d,t1,t2):
  i = get_iterator()
  t1[i:] = a[i:] + b[i:]
  t2[i:] = t1[i:] - c[i:]
  d[i:]  = d[i:] + t2[i:]
```

(b) Example using explicit temporary arrays

Figure 4.4: An example of using sliding views with and without explicit temporary arrays

Broadcasting of views leads to a similar situation. A broadcasted view is expanded by repeating elements and having a stride of 0 in the broadcasted dimensions. Array operations in Numpy implicitly broadcast operand views if the shapes do not match (described in Section 2.6.4). Therefore, the broadcasted views must inherit the slides of the view it is based on.

4.2.4 Python Interface

The philosophy of the Numpy bridge is following the original Numpy interface as much as possible. The usability of sliding views within Python is therefore of great importance. Since sliding views are dependent on `do_while`, where the loop body must be within a function and passed to `do_while` with the relevant arguments. Using `do_while` is not naturally a part of the Numpy interface but is required by Bohrium. Apart from calling `do_while`, the code within the loop body should be as close to Numpy syntax as possible.

Iterator Object

An iterator is a variable that changes between each iteration. When used in a slice for indexing, the iterator indicates that the view is changing between iterations. To get an iterator in the `do_while`, the user must call the function `get_iterator`. It returns a special iterator variable, which is only usable for indexing into the views within a loop body. At the end of a loop iteration, all iterators increase by one (only semantically, since the update is within the Bohrium). It is possible to get a different slide than one by using multiplication to get the wanted increase or decrease. Slides become decreasing by multiplying the iterator with a negative number. In the Numpy bridge, an iterator is a Python object with the following attributes:

step The change to the offset between each iteration (default 1).

offset The starting offset (default 0).

max_iter The maximum amount of iterations (default set in `do_while`).

step_delay The number of iterations to wait before applying an update in the given dimension (default 1).

reset The number of steps before the changes made by the iterator is reset (default `None`).

The iterator supports multiplication, addition, and subtraction with integers. Addition and subtraction changes the `offset` attribute, while multiplication changes the `step` attribute. These operations support a syntax such as **`a[2*i-2:2*i+2]`**. `get_iterator` can be called with an integer that defines the offset.

The iterators can also be used to create a simple nested loop. The function `get_grid` takes a variable amount of integers as input and returns as many iterators. It sets the correct step delays and resets based on the number of nested loops. Figure 4.5 shows an example of a nested loop in Python with Numpy and `get_grid` in `do_while`.

```
a = np.zeros((10,5))
for i in range(10):
  for j in range(5):
    a[i, j] += 1
```

(a) An example of a nested loop using pure Numpy

```
a = bh.zeros((10,5))
def loop_body(a)
  i, j = get_grid(10, 5)
  a[i, j] += 1
bh.do_while(loop_body, 10*5, a)
```

(b) An example of a nested loop using `get_grid` within `do_while`

Figure 4.5: An example of a nested loop using pure Numpy and `do_while`

Overwriting Bohrium nd-array Views

In Python terminology, *container objects* are objects that contain data in a structured way. It involves sequences in general, such as lists and tuples, and is also inheritable by custom Python objects. The Numpy nd-array classifies as a container object and follows its interface. Concerning indexing into the object, the interface requires that the object should implement the two functions called `__getitem__` and `__setitem__`. When indexing an nd-array, it executes one of these functions. The two functions have the following interface:

__getitem__ Takes two arguments: A container object and indices. The indices can either be an integer or slice, or a tuple of variable length with a mix of these. The function must return an element or subset from the container object, based on the given indices.

__setitem__ Takes three arguments: A container object, indices, and a value. The two first arguments are the same as above. The container subset, given by the indices, is set the third argument. This function only has side effects on the container object and does not return anything (or more formally `None`).

By overwriting these two functions, it is possible to achieve custom behavior when indexing with iterator objects. If any of the indices are an iterator or a slice that contains at least one iterator, the functionality is specialized to handle them. It involves calculating the changes within the sliding view and adding/merging the changes to/with the `dynamic_view_info`. The procedure of translating indices to changes is described in Section 4.2.1.

`__getitem__` should return a view as a Bohrium nd-array, but with modified sliding metadata.

Translating Views to `slide_view` Calls

As briefly mentioned earlier, the sliding metadata stays within the nd-array object until an instruction containing the view is flushed. The sliding metadata is transferred to the C++ bridge by the function `add_slide_info`. It loops over each key-value pair in the `dynamic_changes` dictionary and for each of these, the information is translated into a `slide_view` call. The `resets` dictionary follows the same procedure, but calls the function `add_reset` instead. `add_reset` takes the dimension and the amount of iterations before the dimension is reset.

Runtime Exceptions in Python

Having the structure of sliding views within the Numpy bridge allows throwing runtime exceptions directly in Python. When an iterator can run out of the bound of the indexed view, given the maximum iterations, an exception should be thrown. It is a simple calculation to examine, whether the iterator can run out of bounds. The exception tells the user how large the dynamic view is and the index that possibly violates the bounds of the array. The index only possibly violates the bounds, since the condition (if there is one) might be false before it reaches the index. Note that is it legal for an index to go from positive to negative indices (see Section 4.2.3).

4.2.5 Limitations

The implementation of sliding views is made to fit within the Bohrium system without extending the vector bytecode or forcing an extensive rewrite of the core. Using sliding views has certain limitations, which are not easily fixed or essentially requires a different approach. Sliding view is implemented within flush-and-repeat and `do_while`. Therefore, it carries the limitations of these. Besides these limitations, the implementation of sliding views also has the following limitations:

Forced Linearity

A significant limitation to sliding views are that the sliding must be linear. Sliding views can only be non-linear by using grids or negative indices, which has required hardcoding the functionality within Bohrium. Sliding views does not support any other non-linear procedures. As an example of this, it is not possible to square an iterator between each iteration. Supporting squaring of iterators (or any other operation) would first require parsing the operation correctly and then handle it within flush-and-repeat. In the extreme case, this leads to creating a complete DSL for expressing manipulation by iterators. Implementing a DSL leads to several complications and is an extensive task.

Another approach is using the architecture-specific language of the VE to calculate the iterator expressions. This idea is the base of *array indexing* (described in the next section). Array indexing would automatically allow using all operations supported by the vector bytecode.

Reshaping

Reshaping of sliding views can quickly a problem since it is non-trivial to translate the old slides to new slides in the reshaped view. In certain cases, reshaping leads to non-linear slides which are not possible using the described sliding view approach. Reshaping is therefore not supported by the sliding view implementation. It is the user's responsibility to reshape the views before indexing them.

Deciding Changes of Shape to Implicit Arrays

Deciding the change in shape of temporary arrays can also be problematic. Some operations manipulate the structure or reshape views. It is problematic when deciding the change in shape of a temporary allocated array.

As an example, a transposition should force the dynamic changes to swap dimensions. The implementation of sliding views does not support such functionality. To fully support resulting arrays with dynamic shape, each operation that reshapes or changes the view in a non-trivial fashion should be hardcoded.

4.2.6 Discussion

Having sliding views as part of the flush-and-repeat structure and within `do_while` is durable. It does not modify the vector bytecode, and most of the source code for sliding views is within the Numpy bridge, view structures and

`slide_views` filter. The approach fits well within all current vector engines and only requires applying the `slide_views` filter (when using the Numpy bridge, which adds the slide information). Sliding views are only implemented within the Numpy bridge since Python produces much more overhead than the other bridges.

Sliding views was first implemented in a fork of Bohrium, but was turned into pull request and is now adopted within the master branch of Borhium. It is thereby a part of the official release of Bohrium. The pull request contains documentation of iterator usage and multiple tests. The tests can be found at [2] in the file *test/python/tests/test_loop.py*.

The limitation of forced linearity is a design choice of sliding views. Sliding views target common loop use-cases, where the iterator is often linear. Array indexing is an approach that attempts supporting custom non-linear slides. An unexplored approach of changing bridge also tries to tackle this problem (described in Section 7.3.2).

It is durable to allow some reshaping, but it has not been implemented since reshaping not possible in general.

4.3 Array Indexing

The concept of *array indexing* is similar to sliding views but tries to handle the problem regarding expressiveness of iterators. Sliding views the is relatively straightforward in the use case for linear sliding. However, that is as far as the expressiveness goes. Sliding views has limited amount operations on iterators, which are all hardcoded within Bohrium. It has the limitation that the user is bound to the implemented set of operations. As with the example of squaring the iterator, it requires extending the iterator type to handle the operation and implementing the functionality within the `slide_views` filter in Bohrium. Thus, allowing user-defined operations requires extending the expressiveness of iterators.

As mentioned briefly, one approach is to implement a domain specific language that describes iterators. The DSL could then be interpreted within Bohrium and thereby achieve the expressiveness of the DSL itself. The downside to this approach is that it is both complicated and requires implementing a language within Bohrium. Apart from this, it would still have certain limitations, such as utilizing I/O or reading values from Bohrium arrays. It would further require each bridge to parse the expressions with iterators into the DSL. Since the internal loops are a minor part of Bohrium itself, this approach is discarded due to its extensiveness.

4.3.1 Utilizing the Vector Engine for Manipulating Iterators

The overall idea of array indexing is allowing the user to utilize the architecture-specific language to express modifications to iterators. Arrays are, therefore, an essential part of this particular approach. Bohrium is focused around arrays, making it a natural way to let the user define the modifications to iterators. Instead of using the iterator object, an *index-array* is used for indexing and creating dynamic views. Bohrium would then use the data inside the index-array for updating the metadata of the view.

When using sliding views, the changes are calculated by a filter and updated after the VE has executed the instruction list. However, instead of updating the views based on sliding metadata, array indexing takes values from the index-arrays and puts them directly into the metadata. Thus, the metadata is calculated by the language of the specific VE instead of a custom-made filter. It allows the user to utilize the expressiveness of the vector bytecode to create custom modifications without needing to extend a custom filter in Bohrium. Moving the iterator calculations to the vector engine has the natural consequence, that the instruction list must be into sub-instruction lists (SIL).

This approach has lead to two implementations, which are described later in this section.

4.3.2 Array Indexing Basis

Both implementations of array indexing share a basis of extending metadata of views and interpreting indices in Python. This section covers the common ground for array indexing.

Extending the Metadata of Views

By using array indexing, the user should be able to modify the offset, shape, and stride of a view. As with sliding views, this requires extending the metadata of views in the C++ bridge and the Bohrium core. Since the view can only read from a single array, storing the information within an object in Python is skipped. The Numpy bridge directly transfers the metadata to the C++ bridge right after the indexing is performed.

The metadata within the C++ bridge and the Bohrium core is extended with three attributes. Each attribute is a pointer to a Bohrium base array, which must have integer type. The attributes are the following:

start_pointer A pointer to the base array, which is used to overwrite the offset of the view. The overwriting value is the first element in the base array.

shape_pointer A pointer to the base array, which is used to overwrite the shape of the view. The base array must have as many elements as the dimensionality of the view.

stride_pointer A pointer to the base array, which is used to overwrite the stride of the view. The base array must also have as many elements as the dimensionality of the view.

Any of these pointer attributes are allowed to null pointers, indicating that the view does not use an index-array.

The attributes can be set directly by calling the functions `set_start`, `set_shape` and `set_stride`. All these functions takes two arguments: The indexed view and the index-array.

Updating Views Between Loop Iterations

Before the execution of a SIL, the view operands within each instruction is inspected to check whether any of them uses index-arrays. If the view has a

`start_pointer`, the offset of the view is set to the first value of the base array. If the view has a `shape_pointer` or `stride_pointer`, the corresponding list of shapes/strides is set to the first `ndim` values of the base array. It is done by a filter named `update_array_iterators`.

The filter does not perform any checks of whether the values within the base arrays are legal. The user has full control and responsibility for the values written to the metadata. Illegal values can result in unexpected behavior or runtime errors in both Bohrium and the executables.

Python Interface

Even though array indexing gives the user full control of views, it is still essential that it provides intuitive syntax. By using the three functions for using index-arrays above, the usage of these can become more intuitive within Python. A user-friendly syntax is shown in the example in Figure 4.6. The example shows a simple nested loop that divides a view into b by another view into b. Both views are of size 5. After this, the resulting vector is added to an accumulating vector a.

```
a = bh.zeros(5)
b = bh.arrange(10)
for i in range(5):
  for j in range(5):
    a += b[i:i+5]/b[j:j+5]
```

(a) Using native Python syntax

```
a = bh.zeros(5)
b = bh.arrange(10)
it = bh.zeros(1)

def loop_body(a, it):
    i = it / 5 % 5
    j = it % 5
    a += b[i:i+5]/b[j:j+5]
    it += 1

do_while(loop_body, 5*5, a, it)
```

(b) Using array indexing syntax

Figure 4.6: An example of using nested loops with native Python syntax and array indexing syntax

As the example shows, the two index-arrays i and j are used to define the size of the views. All the operation in the vector bytecode can be used to modify both these variables. In this example, the iterators use division and modulus.

To achieve this syntax, `__getitem__` and `__setitem__` must be modified to handle using arrays as indices. The function must check whether any of the indices are an index-array. An index-array only contains a single element and the type integer. The functions handle the view separately if any of the indices/slices is/contains an index-array. The functions then internally create

an offset and shape, which are all initialized to zero. These arrays are later updated based on the indices. For each dimension, the size is $slice_{stop} - slice_{start}$. If the start or stop is an array, the calculation turns into an array operation which will appear inside the instruction list. $slice_{start}$ is added to the offset array.

In the example in Figure 4.6b, the shape is defined as the difference between the two arrays. One array is i, which also defines the offset, while the other is a temporary array that stores the result of $i + 5$. These two arrays are then subtracted and stored in another temporary array, which is used to define the shape. The offset uses the index-array i. The same goes for the second array indexed by j instead.

Delay of Deallocation

Since the shape and offset array is allocated within the `__getitem__` or `__setitem__`, they are considered temporary and is deallocated after the function call. It will lead to a segmentation fault since the arrays are used later for receiving metadata. A filter is implemented to handle this problem and delays these deallocations until after the last usage as index-array. The instruction list is run through once to list every base array used as an array index. After this, the deallocations of these arrays are removed from instruction list and then added again after the last usage of the index-array.

4.3.3 Two-phase Array Indexing

The first implementation is called *two-phase array indexing*. The idea behind this implementation is gathering all instructions that involve index-arrays at the start of the loop body. It leads to a natural split of the instruction list into two SILs: The first SIL functions as a *setup part* and the latter as an *execution part*. Executing an iteration goes through the following steps:

1. The setup SIL is executed to calculate index values and store them within the index-arrays
2. Bohrium updates the metadata within the views of the execution SIL based upon the calculated index-arrays
3. The execution SIL is executed with the updated views

Figure 4.7b shows an example of two using two-phase array indexing. The example shows how to create a nested loop with i as the outer iterator and j as the inner. The `do_while` computes the same as 4.7a, but only requires one flush from the Numpy bridge. The code snippets show that array indexing requires more explicitness, which makes the source code longer.

Splitting the Instruction List

It is easy to split the instruction list of two-phase array indexing due to its limitation. The split occurs right before the first usage of an array iterator. This split is legal because of the assumption that all calculations that modify the index-arrays are at the top of the loop body. All calculations that modify

```
a = bh.zeros(1)
b = bh.arrange(10)
for i in range(10):
  for j in range(10):
    a += b[i]+b[j]
```

(a) Using a native Python loop

```
def loop_body(a, b):
  iterations = bh.zeros(1)
  i = bh.zeros(1)
  j = bh.zeros(1)
  i[0:1] = iterations % 10
  j[0:1] = iterations / 10 % 10
  a += b[i]+b[j]
  iterations += 1
a =  bh.zeros(1)
b = bh.arrange(10)
bh.do_while(loop_body, 10*10, a, b)
```

(b) Using two-phase array indexing

Figure 4.7: An example of a nested loop using a native Python loop and two-phase array indexing

the index-arrays are therefore executed within setup SIL, such that the view metadata can read from the index-arrays before executing the execution SIL.

4.3.4 General Array Indexing

A natural step beyond two-phase array indexing is a generalization to an n-phase indexing. The reason for having a variable amount of phases is to avoid putting any restrictions on the usage. Instead of limiting all updates to index-arrays to appear at the top, the instruction list should automatically split whenever it needs to. It would allow loop bodies that arbitrarily uses index-arrays and allows using an index-array within the calculation of another index-array.

By initiating the index-arrays with values for the first iteration and performing the modifications at the end, the instruction list can also avoid splitting. By having modifications to index-arrays the end, they update automatically at the end of the SIL and does not require splitting. An example of this can be seen in Figure 4.8.

The general array indexing is more expressive than two-phase array indexing. An example of a loop body that requires multiple splits can be seen in Figure 4.9. In this example, an index-array is used to index into a view, which is then used to index into another array. This procedure requires that `it2` and `it3` are set to the correct value before they are used to index into b. Therefore, the kernel must create a split before indexing b.

Splitting the Instruction List

A difficulty of general array indexing is deciding when to split the instruction list. The filter must never read metadata from an index-array it is computed.

```
def loop_body(a, b, it):
    b += bh.sum(a[:it]) / a[it]
    it *= 2

it = bh.ones(1)
a  = bh.arange(1, 2**8+2)
b  = bh.zeros(1)

bh.do_while(loop_body, 8, a, b, it)
# resulting b: [ 255 ]
```

Figure 4.8: An example of a loop body where the updates are at the end of the program.

```
def loop_body(a, b, c, it):
    it2 = a[it]
    it3 = a[it+1]
    c += b[it2] / b[it3]
    it+=1

a = bh.array([ 0, 6, 3, 4, 1, 2, 5, 7, 8 ])
b = bh.repeat(2.0,9) ** bh.arange(9)
c = bh.zeros(1)
it = bh.zeros(1)

bh.do_while(loop_body, 8, a, b, c, it)
# resulting c: [ 17.890625 ]
```

Figure 4.9: An example of a loop body that requires multiple splits and cannot be handled by two-phase array indexing.

A naive approach would be to split the instruction list each time at each usage of an index-array, which guarantees that the index-array is computed. Such an approach would lead to many splits, which are not free from overhead (described in Section 4.3.5).

A more complex approach is to keep track of whether an index-arrays is computed. It requires searching the instruction list searching for all base arrays used as index-arrays. If an index-array is modified, it must force a split before next operation that contains a view that uses it. The split forces the VE to execute of the instruction list that contains the operation on the specific index-array. After this is can be used as view metadata.

The book-keeping is handled through a hashmap that uses base arrays (of the index-arrays) as keys and booleans as values. The boolean indicates whether the index-array requires a split before usage. The procedure is the following. Each time a value is written to a base array within the hashmap, the corresponding boolean is set to `True`. Each time the instruction list is split, all array indexes in the hashmap is set to `False`, since they are all computed. The metadata within a view can be used if its set to `True`, otherwise it causes a flush before it can be used.

The functionality is implemented as a function that takes an instruction list as an argument and returns a vector of instructions lists.

4.3.5 Limitations

The implementation of array indexing has several limitations. Some limitations occur because the changes are handled outside Bohrium, while others occur due the interaction with the Python interpreter. Due to these factors, the implementation did not become a part of the official Bohrium release.

Overhead Regarding Splits

As discussed briefly in Section 3.5, getting from an instruction list to an executable involves a certain amount of overhead. Even in the case where the work is cached, it still takes a certain amount of time to hash the instruction list and performing a look-up. Thus, the loop body must be computational heavy enough that is undermines this overhead.

No Nesting Views

It is not possible to index into a view that uses array indexing. Views are a simple structure that cannot be nested but only transforms metadata of an existing view. Since the array indexing metadata is not apparent in the bridge, it is not durable. Also, the metadata of array-indexing is not relative but absolute.

Python Depends on Values From Arrays within the Bohrium Namespace

When using the Python-like syntax for indexing, a problem can occur concerning the shape of a view in Python. The Python interpreter depends upon the shape of the array for many reasons. Both broadcasting and implicitly allocated arrays require the shape of arrays. Runtime errors regarding illegal shapes in operations also require the shape.

Indexing into a view must return a new view that corresponds to the indexed subset. Array indexing uses the offset and shape array to create the returned view. However, if these arrays are within the Bohrium namespace, they cause a flush. `do_while` does not allows flushes within the loop body which disallows switching between the Bohrium and Numpy namespace. It is therefore not possible to interpret the loop body if an index depends on a value from the Bohrium namespace. Thus, array indexing cannot easily depend on Python for interpreting the loop body.

The user can use explicit hard-coded slices for the views within the loop body and then later use the commands `set_start`, `set_shape` and `set_stride`. It is not intuitive and not at all close to Python-like syntax, which is a large compromise when creating additions to Bohrium.

Different methods can handle some of these scenarios. One solution could be making the views into a static shape of 1 in each dimension and then handle shapes within Bohrium. It would require hard-coding the broadcasting functionality and leads to multiple problems, such as the shape of temporary arrays.

Side-effects of Changing Shape

Also determining the shape of temporary arrays can also become a problem. Bohrium has no control over how the shape of the views is changing, so deter-

mining the shape of temporary arrays dynamically is not durable. The same goes for the broadcasting of views.

Error Handling within Python

Error handling within is the Numpy bridge is not possible since the changes happen within the VE. Errors occur within the source code produced by the VE. It is much more difficult to read since the code might be very different from the high-level array programming.

4.3.6 Discussion

Using array indexing as part of the flush-and-repeat structure is durable but intuitive usage within `do_while` is a problem. Transfering the shape of an array from Bohrium to Python causes a flush, which is disallowed. Thus, it leads to an explicit syntax that is not user-friendly that requires directly setting metadata arrays and hard-coding the shape of the views in the first iteration. Some *hacks* can be applied that might fix some scenarios, but does not solve the general problem. A possible solution is parsing Python bytecode directly to Bohrium and bypass the need shapes in the Numpy bridge (described in 7.3.1).

Array indexing has been implemented with limited features, such as using `set_start`, `set_shape` and `set_stride`. All features in the Bohrium core, such as the delay of deallocation and splitting the instruction list is implemented.

Due to the limitation of user-friendliness, the approach is not satisfying until this is be solved. The implementation of array indexing has therefore not made it into the official release of Bohrium.

The performance section does not include benchmarks with array indexing. A brief discussion of the expected performance of array indexing is described in Section 5.5.1.

Chapter 5

Performance

This section focuses on measuring the performance of the *sliding view* loop extension. The performance is measured through benchmarks from the framework Benchpress and code from VEROS. Benchpress is a benchmark-suite, also containing a set of benchmarks. VEROS (The Versatile Ocean Simulator) is an ocean simulator in pure Python that uses Bohrium for optimizing simulations. The benchmarks used in this chapter can be found at [11].

The benchmarks measures `do_while` with sliding views (Bohrium), native Python loops (Bohrium) and pure Numpy. The purpose of including Numpy is to compare the results with the framework that Bohrium seeks to improve. Each benchmark section shows the code difference between using sliding views and native Python loops.

All results can be seen in Appendix C.

5.1 Hardware Specifications

All the benchmarks have been executed on the *Nelson* machine, made available by the Niels Bohr Institute. Nelson has the following hardware:

CPU: 4 $\times$ Intel(R) Core(TM) i7-3770 (AMD64)

GPU: 2 $\times$ NVidia GTX680

RAM: 32 GB

Disk: 500 GB

5.2 Execution Parameters

The benchmarking-framework Benchpress has been used to execute all benchmarks. Some of the benchmarks are from the Benchpress collection.

The default parameters for the benchmark-executions are:

Runs The results of each benchmark is as an average of 5 executions.

Cache The benchmarks are allowed to utilize cache within each execution but does not utilize compilation cache from previous executions.

VE The benchmarks uses the C/OpenMP VE, which executes on the CPUs.

Input The input matrices are randomly generated by Benchpress, using the same seed.

The description of each benchmark clearly states if any of these parameters are modified.

For measuring the performance between using `do_while` and native Python loops, the native loops have been executed using the Benchpress flag: `--no-do_while`. This flag replaces the `do_while` with a native Python loop and forces a flush after each iteration. The flush ensures that the two loop-versions of Bohrium operate on the same instruction lists. Not flushing the instruction stream after each iteration leads to longer instruction lists, which makes the work of the runtime components different. The fusion takes longer time but might create a more efficient executable. The long instruction lists do not utilize the cache.

5.3 Benchmark Structure

This chapter presents results from executing multiple benchmarks using Numpy and Bohrium with/without `do_while`. The benchmarks have been picked to showcase certain aspects of the differences between these. This section covers different types of loop-based benchmarks and the presentation of the results of the benchmarks.

5.3.1 Loop-based Benchmarks

The benchmarks can be split into three overall categories: Iterative, input-based and simulation.

Iterative These benchmarks are based upon executing a loop body until satisfying a criterion or reaching the maximum number of iterations. Each execution of the loop body brings the method closer to a correct answer. These benchmarks often stop when the change in the result is smaller than some ϵ. Alternatively, they stop after the maximum amount of iterations, indicating that the function is not allowed to spend more time.

Input-based These benchmarks have a fixed amount of iterations, based on some subset of the input. It can be iterating over a single or multiple axes, such as convolving a filter over an input tensor.

Simulation These benchmarks simulate a system that changes over time. Each iteration computes the changes within a given time frame. Thus, the total length of the simulation dictates the number of iterations.

The benchmarks are not handcrafted to showcase the difference between Numpy and Bohrium's two loops. They are general programs that are all picked from Benchpress or VEROS. The benchmarks also showcase that high-level loops with dynamic views commonly appear within scientific programs. Each benchmark has a description of what it performs and how it is computed, and the code modifications needed for using the sliding view extension.

5.3.2 Benchmark Plots

The results of each benchmark is visualized through four plots:

Total elapsed time The total runtime of the execution of the benchmark. This also includes copying the results from Bohrium to Numpy at the end of the benchmark.

Relative performance The speedup related to using native Python loops with Bohrium. The relative speedup is based on the total elapsed time.

Total time elapsed outside the runtime components This measures the overhead by using Bohrium. The overhead includes the Numpy bridge, lookups in caches and the setup of arguments to each executable. The Python interpretation of the loop body contributes a large part of the overhead in the Numpy bridge.

Relative performance in regards to overhead The relative speedup in overhead relative to using native Python loops.

The latter two plots do not include Pure Numpy since it spends all its time outside the runtime components.

Each benchmark also contains a brief breakdown of the results, which seeks to explain the tendencies illustrated by the plots.

5.4 Sliding Views

The motivation for extending Bohrium's `do_while` with sliding views is achieving expressiveness. The extended expressiveness allows reducing the overhead of programs that were forced to use native Python loops before. This section covers four benchmarks that are all able to use sliding views. The expected result of using `do_while` with sliding views is gaining performance by reducing overhead. The performance gain is expected to be similar to the performance gain of using `do_while` without sliding views (shown in Section 3.6.1 with Heat Equation).

The performance of `do_while` with sliding views is measured through the following benchmarks: Gaussian Elimination, Lattice Boltzmann D2Q9, Tridiagonal Matrix Solver and creation of Quasicrystals.

5.4.1 Gaussian Elimination

Gaussian Elimination solves a system of linear equations, also known as row reduction. It takes a $M \times N$ matrix as input, interpreted as M linear equations with $N-1$ variables. This implementation does not use the row with the largest element to reduce the other rows but goes through them from top to bottom. The Gaussian Elimination benchmark is from the Benchpress collection.

The difference between the original benchmark and using `do_while` with iterators can be seen in Figure 5.1. This difference is replacing the iterator c with a function call to `get_iterator`. Gaussian Elimination shows some of the elegance of using Numpy's broadcast functionality, making the loop body is fairly compact. The first part is a division that forces a broadcast of `S[c-1,c-1:c]` from a 1×1 to a $row\_length - c \times 1$. After this, the multiplication forces a broadcast of the two vectors into two matrices, which are multiplied.

```
for c in range(1, S.shape[0]):
    S[c:,c-1:] -= (S[c:,c-1] / S[c-1,c-1:c])[:,None] * S[c - 1, c - 1:]
```

(a) Gaussian elimination using a native Python loop

```
def loop_body(S):
    c = get_iterator(1)
    S[c:,c-1:] -= (S[c:,c-1] / S[c-1,c-1:c]) * S[c-1, c-1:]
B.do_while(loop_body, S.shape[0]-1, S)
```

(b) Gaussian elimination using `do_while` with an iterator

Figure 5.1: The difference in the Gaussian elimination between using native Python loops and `do_while`

Gaussian Elimination is benchmarked with pre-compiled executables. It makes the compilation time is much lower than a fresh execution. The compilation time of Gaussian Elimination is a very dominant factor, making it harder to notice the difference between the different loops (discussed further in the breakdown of results). The benchmark does not compare Numpy to Bohrium because of pre-compiled executables. Numpy is by far faster than a fresh execution using Bohrium.

The total runtime of the benchmarks of running Gaussian elimination with native Python loops and `do_while` can be seen in Figure 5.2. The time spent outside runtime components can be seen in Figure 5.3

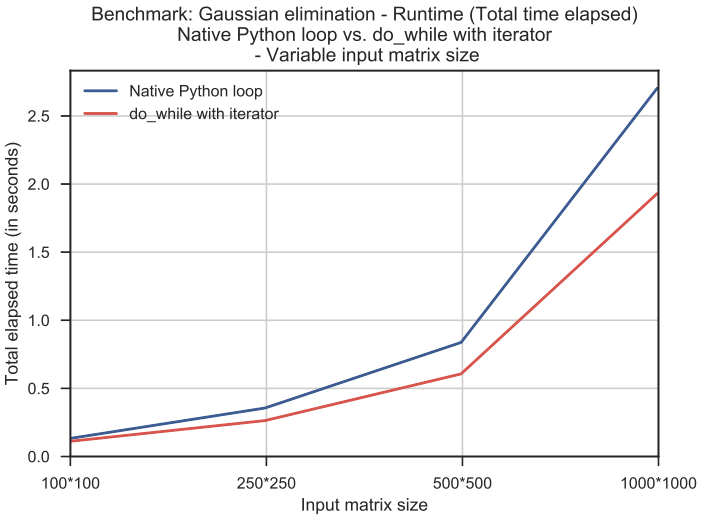
Breakdown of Results

The Gaussian Elimination benchmark is input-based. More specifically, it is based on the size of the first dimension of the input. Thus, the amount of iterations increases as the problem size increases.

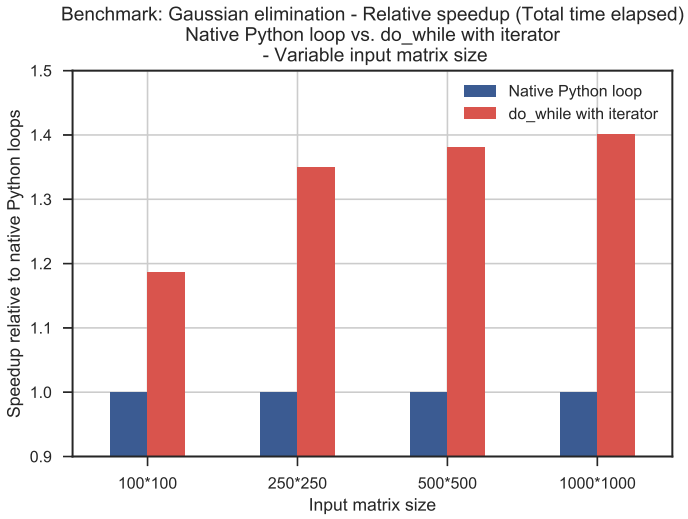
The Gaussian elimination benchmark is not compared to Numpy since it is too compilation-heavy. For each iteration, the vectors within the loop body change size, which forces compilation of a new executable. Since the computations within the loop body are not substantial enough to undermine the

compilation, the resulting execution time is by far slower than Numpy. The benchmarks show some potential if the executables could be reused to avoid the dominating compilation time.

The benchmarks of total elapsed time in Figure 5.2 shows that there is potential to use `do_while` with iterators. The difference is based upon the time spent outside runtime components, shown in Figure 5.3. As we can see, the time spent outside the runtime components of `do_while` grows much slower than native Python loops.

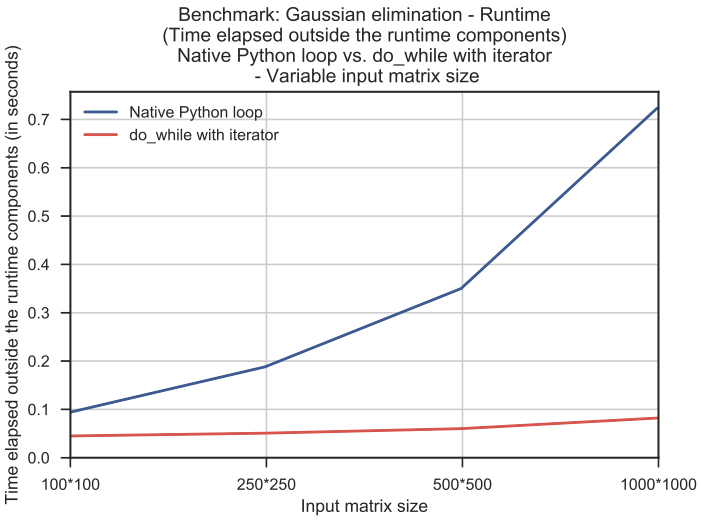


(a) Total time elapsed without compilation

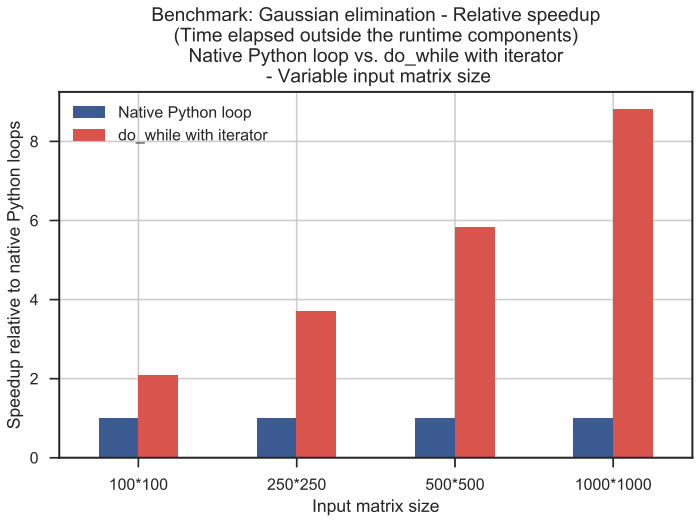


(b) Relative speedup without compilation

Figure 5.2: Absolute time difference and relative speedup in total time elapsed between using `do_while` loops and native Python loops. The size of the input matrix is variable.



(a) Total time elapsed without compilation

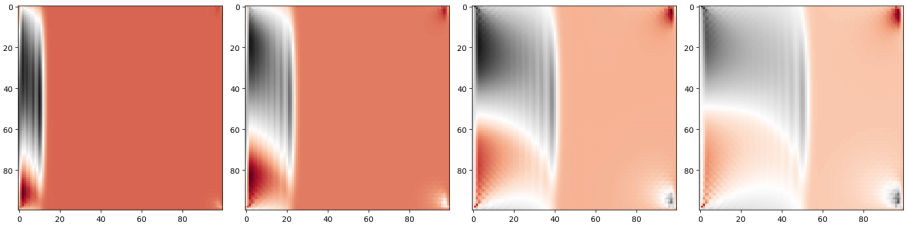


(b) Relative speedup without compilation

Figure 5.3: Absolute time difference and relative speedup in time elapsed outside the runtime components between using `do_while` loops and native Python. The size of the input matrix is variable.

5.4.2 Lattice Boltzmann D2Q9

Lattice Boltzmann (LBM) D2Q9 is a method within the class of computational fluid dynamics, used for fluid simulation. LBM allows a discretization of space to a lattice and the velocity space into a set of microscopic velocities. The LBM D2Q9 benchmark is in the class of *LBM DnQm*, where n denotes the number of dimensions, while m denotes the number of speeds. A visualization of the LBM D2Q9 over time can be seen in the results of an example in Figure 5.4. The LBM D2Q9 benchmark is from the Benchpress collection and is a simulation.



(a) 25 iterations **(b)** 50 iterations **(c)** 75 iterations **(d)** 100 iterations

Figure 5.4: Illustration of the cylinder state after 25, 50, 75 and 100 iterations of the LBM D2Q9 on a 100×100 cylinder.

LBM is split into two steps called the *collision step* and the *streaming step*. These steps both loop over the nine velocities. The streaming step contains an if-statements and cannot use `do_while`. However, the collision step can. In D2Q9 there are nine microscopic velocities, stored in a 3D-array with the shape $velocities \times height \times width$. Note that these loops are inner loops. The overall algorithm is still within a native Python loop.

Since LBM D2Q9 has a large loop body, the code difference between native Python loops and using `do_while` with iterators has been moved to Appendix A.1. LBM D2Q9 with native loops can be seen in Listing A.1, while LBM D2Q9 using `do_while` with iterators can be seen in Listing A.2. The last loop is split into two using `do_while`, since there is a static branch within the loop. The branch is not changed during execution, which makes `do_while` usable in this scenario.

LBM D2Q9 is benchmarked with two different setups. The first is *size-based* with 25 iterations fixed and variable input matrices sizes. This benchmark includes native Python loops, `do_while` with iterators and pure Numpy. The results of this benchmark can be seen in Figure 5.5 (total time elapsed) and 5.6 (time elapsed outside runtime components). The second is *iteration-based* with a variable amount of iterations and a fixed input matrix of size 1500×1500. This benchmark includes native Python loops and `do_while` with iterators. The results of this benchmark can be seen in Figure 5.7 (total time elapsed) and 5.8 (time elapsed outside runtime components).

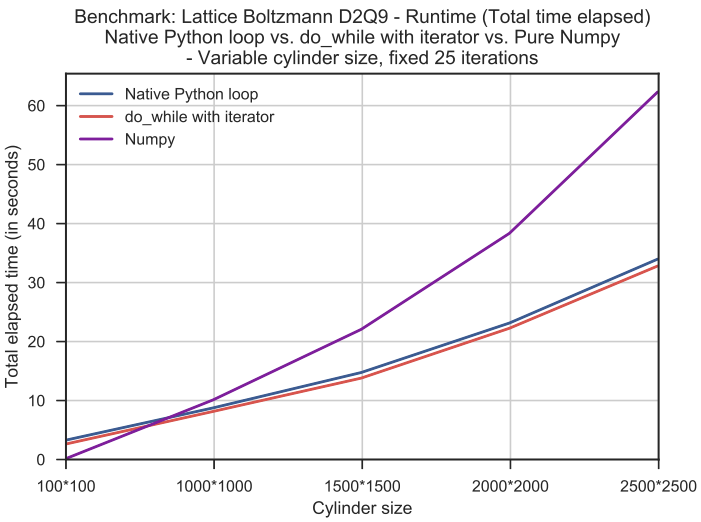
Breakdown of Results

Size-based As it can be seen in Figure 5.5a, both Bohrium with native Python loops and `do_while` is slower than Numpy on the smallest problem size, but becomes faster at 1000×1000. The relative speedup of 100×100 in Figure 5.5b

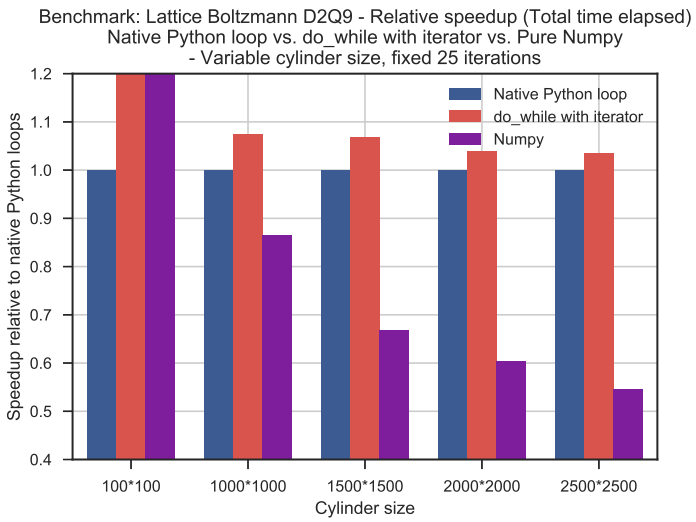
is cut off, since Numpy is $\approx 20\times$ faster than both Bohrium with native Python loops. This is due to the constant overhead factor, such as compilation.

This benchmark has fixed the number of iterations to 25. It has the consequence that using `do_while` is faster than native Python loops by a constant factor. The constant factor is about a second, which is clearly illustrated in Figure 5.7a showing the time spent outside runtime components. It results in the fact that the relative speedup becomes less and less, which is shown in Figure 5.5b.

Iteration-based This benchmark showcases the achievable amount of speedup of using `do_while` as opposed to native Python loops. The input size of this benchmark is fixed to 1500×1500, which fixes the amount of work within the loop body. As it can be seen in Figure 5.7b, the achievable speedup in regard to this problem size is $\approx 7\%$. The `do_while` speedup in regard to the time spent outside the runtime components is illustrated in Figure 5.7b and is $\approx 3\times$ faster.

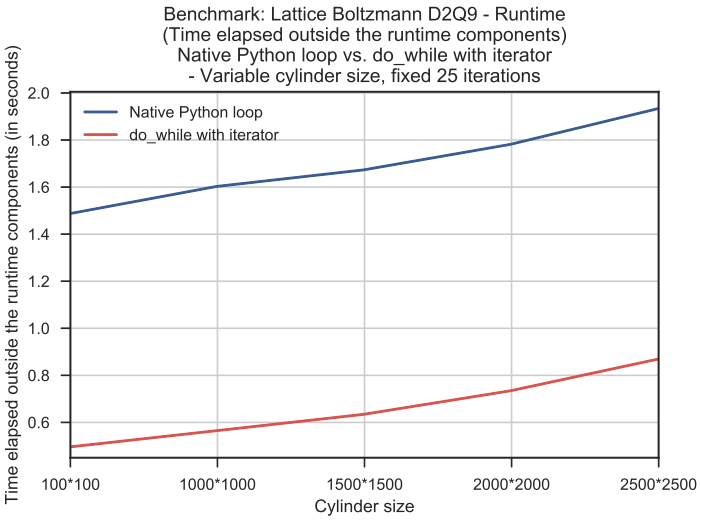


(a) Total time elapsed

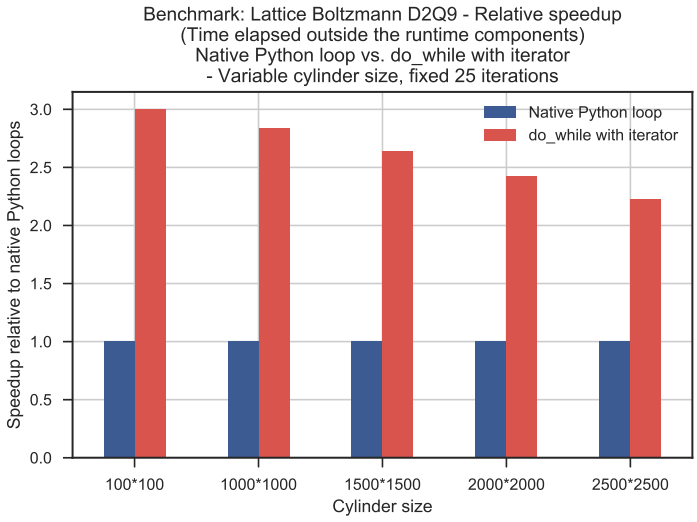


(b) Relative speedup

Figure 5.5: Absolute time difference and relative speedup in total time elapsed between using `do_while` loops, native Python loops and pure Numpy. The size of the input matrix is variable, while the iterations is fixed to 25.

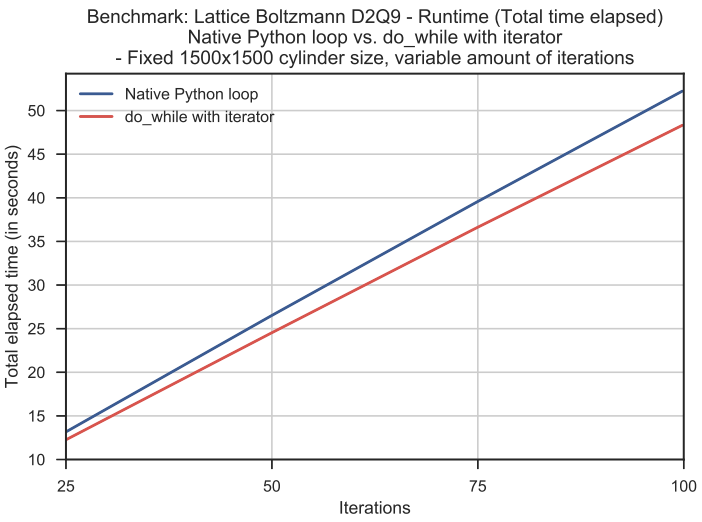


(a) Total time elapsed

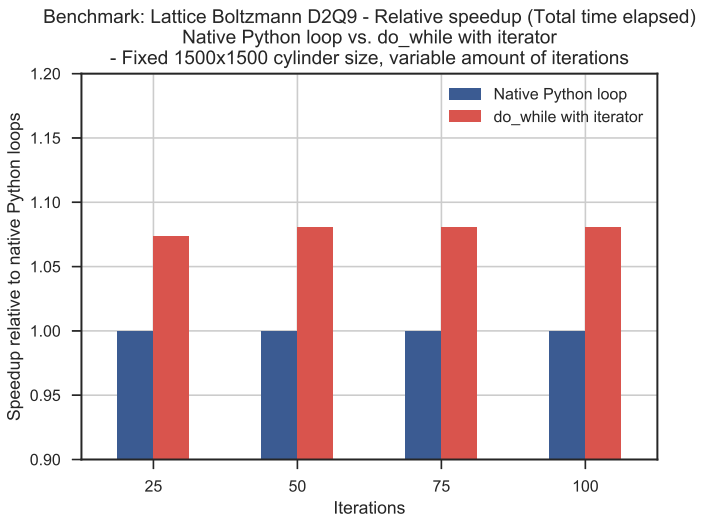


(b) Relative speedup

Figure 5.6: Absolute time difference and relative speedup in time elapsed outside the runtime components between using `do_while` loops and native Python loops. The size of the input matrix is variable, while the iterations is fixed to 25.

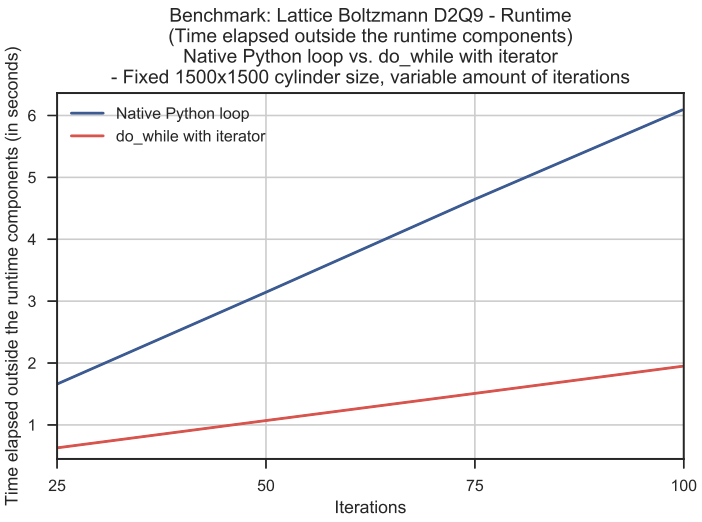


(a) Total time elapsed

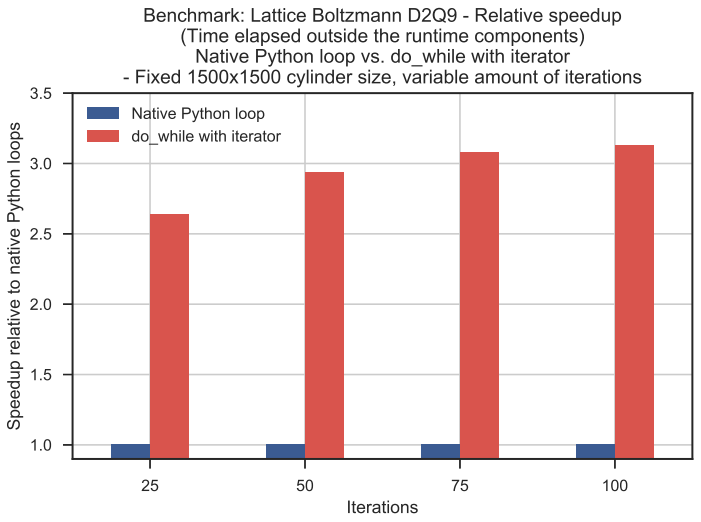


(b) Relative speedup

Figure 5.7: Absolute time difference and relative speedup in total time elapsed between using `do_while` loops and native Python loops. The size of the input matrix is fixed to 1500×1500, while the iterations is variable.



(a) Total time elapsed



(b) Relative speedup

Figure 5.8: Absolute time difference and relative speedup in time elapsed outside the runtime components between using `do_while` loops and native Python loops. The size of the input matrix is fixed to 1500×1500, while the iterations is variable.

5.4.3 Tridiagonal Matrix Solver

The Tridiagonal Matrix Solver (TDMA) is a simplified form of Gaussian elimination that can solve tridiagonal systems of equations. Such a system has the form

$$d_i = \begin{cases} b_1 x_1 + c_1 x_2 & i = 1 \\ a_i x_{i-1} + b_i x_i + c_i x_{i+1} & 1 < i < n \\ a_n x_{n-1} + b_n x_n & i = n \end{cases}$$

where $a_1 = 0$ and $c_n = 0$. TDMA is an input-based benchmark.

The TDMA uses two steps, which is a forward sweep step and a backward substitution step.

Forward sweep The forward sweep step constructs two vectors called c' and d', where each value $\{c'_i \mid 1 < i \leq n\}$ is dependent upon c'_{i-1}. The same goes for d'.

Backwards substitution In the backwards substitution step, each value of $\{x_i \mid 1 \leq i < n\}$ is dependent upon x_{i+1}.

The number of iterations in each loop is based on the width of the input matrix. Both of these steps are inherently sequential and computed within a loop. Both of these loops can be either native Python loops or `do_while` with iterators.

```
for i in xrange(1, n):
    m = b[...,i] - cp[...,i-1] * a[...,i]
    fxa = 1.0 / m
    cp[...,i] = c[...,i] * fxa
    dp[...,i] = (d[...,i]-dp[...,i-1]*a[...,i]) * fxa
x[...,n-1] = dp[...,n-1]
for i in xrange(n-2, -1, -1):
    x[...,i] = dp[...,i] - cp[...,i]*x[...,i+1]
```

(a) TDMA using native Python loops

```
def loop_body(cp, dp):
    i = get_iterator(1)
    m = b[...,i] - cp[...,i-1] * a[...,i]
    fxa = 1.0 / m
    cp[...,i] = c[...,i] * fxa
    dp[...,i] = (d[...,i]-dp[...,i-1]*a[...,i]) * fxa
B.do_while(loop_body, n-1, cp, dp)
x[...,n-1] = dp[...,n-1]
def loop_body(x):
    i = get_iterator()
    x[...,n-2-i] = dp[...,n-2-i] - cp[...,n-2-i]*x[...,n-1-i]
B.do_while(loop_body, n-1, x)
```

(b) TDMA using `do_while` with iterators

Figure 5.9: The difference in the TDMA benchmark using native Python loops and `do_while`

TDMA is benchmarked with two different matrix widths and a variable matrix height. The first benchmark has a fixed matrix width of 100 and can be

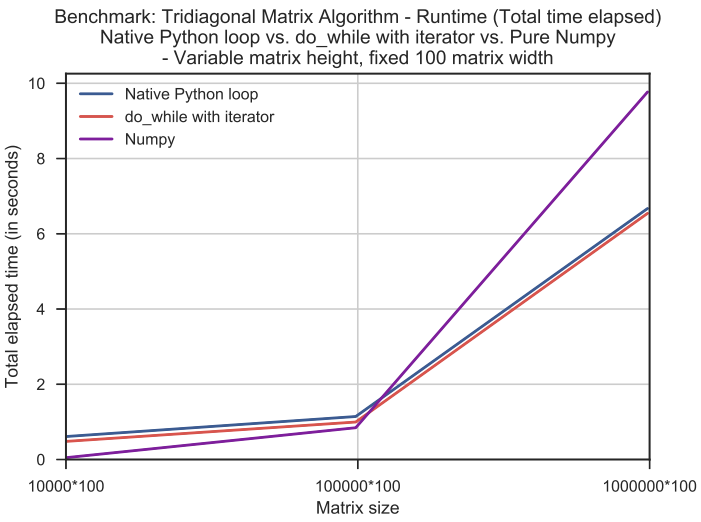
seen in Figure 5.10. The second benchmark has a fixed matrix width of 1000 and can be seen in Figure 5.12. Notice that the matrix height is $10\times$ smaller in the benchmark, where the matrix width is 1000, such that the amount of operations is the same between the two benchmarks.

The TDMA benchmark is from VEROS. The typical input Matrix size is $10^6 \times 100$ in Veros.

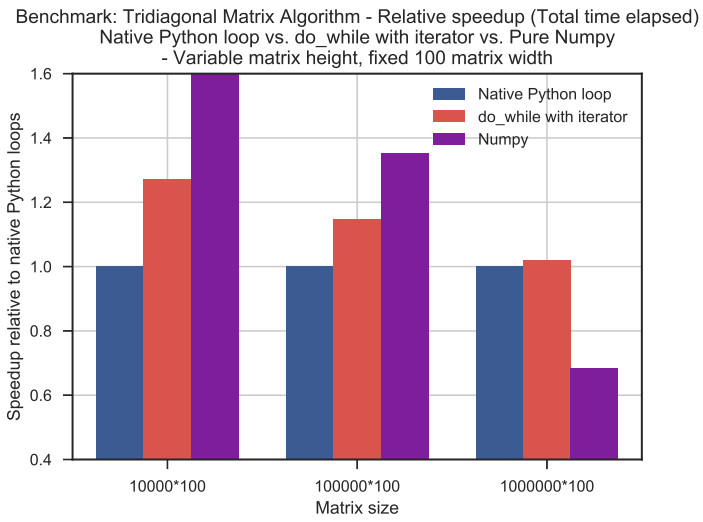
Breakdown of Results

Matrix Width of 100 As it can be seen in Figure 5.10a, pure Numpy is faster on the two first input sizes, but then becomes slower on the last. The difference between using native Python loops and `do_while` is very minimal. It can be seen more clearly in Figure 5.11a, where the overall difference is constant at ≈ 0.125 seconds. It amounts to a very little relative speedup in the case where Bohrium beats pure Numpy. The performance gain is low is because the amount of work within the loop body is heavy enough to undermine the overhead of the 100 loop iterations.

Matrix Width of 1000 The difference between using native Python loops and `do_while` becomes more apparent as the matrix width increases. The results of benchmarking an input matrix with a width of 1000 can be seen in Figure 5.12a. Using `do_while`, the overhead of having a width of 100 is very close to the overhead of having a with a width of 1000. Using native Python loops, the overhead increases linearly with the matrix width, showed in Figure 5.13a. The constant difference is ≈ 1.3 seconds in favor of `do_while`. It showcases that the overhead involved with having a width of 1000 is $\approx 10\times$ more than having a width of 100 using native Python loops.

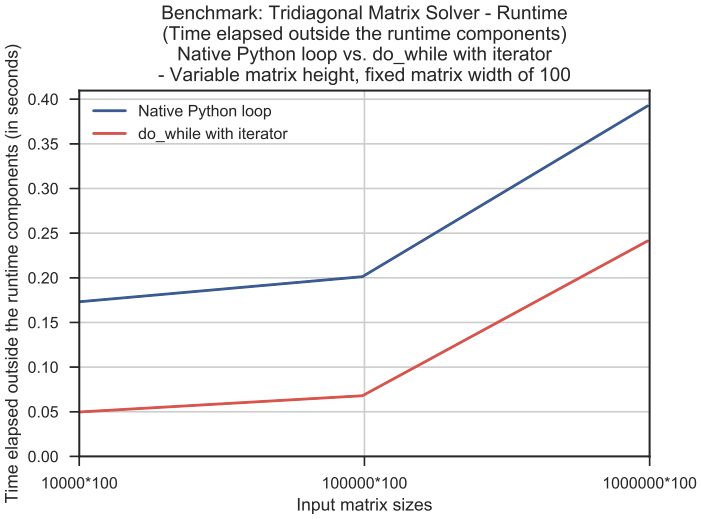


(a) Total time elapsed

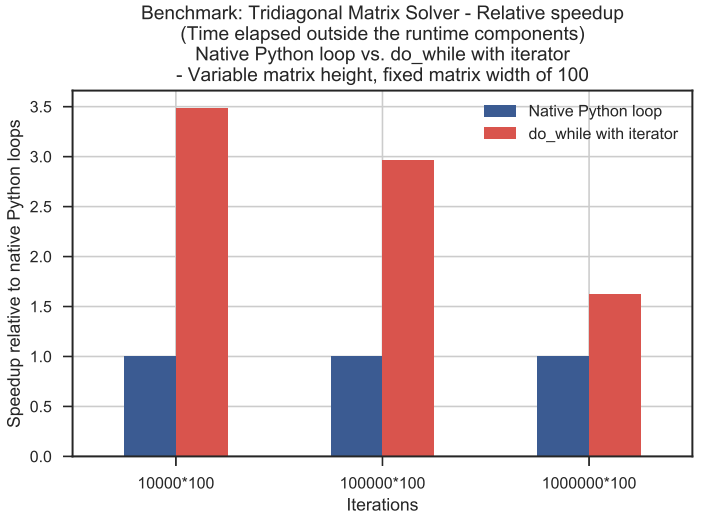


(b) Relative speedup

Figure 5.10: Absolute time difference and relative speedup in total time elapsed between using `do_while` loops, native Python loops and pure Numpy. The amount of iterations fixed to 100, while the matrix input is variable.

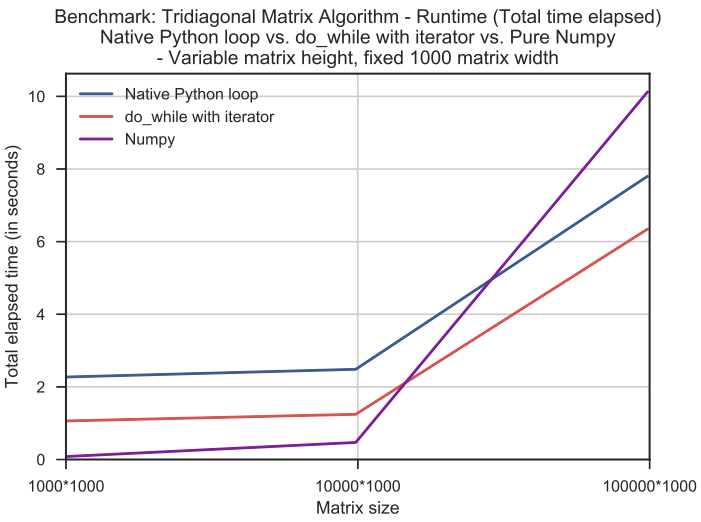


(a) Total time elapsed

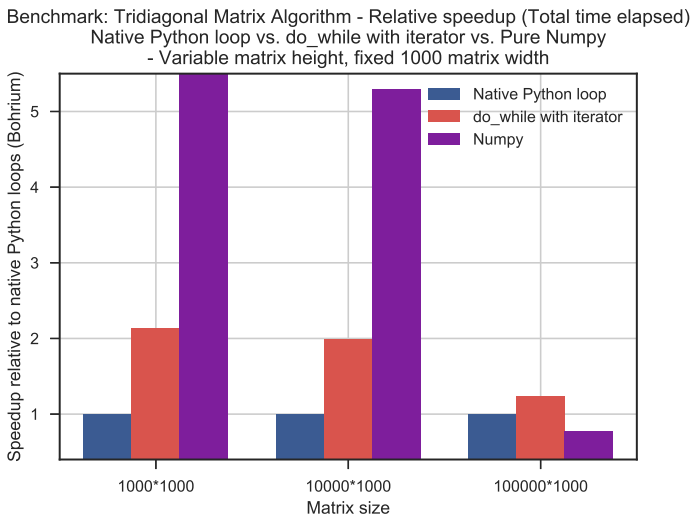


(b) Relative speedup

Figure 5.11: Absolute time difference and relative speedup in time elapsed outside the runtime components between using `do_while` loops and native Python loops. The amount of iterations fixed to 100, while the matrix input is variable.

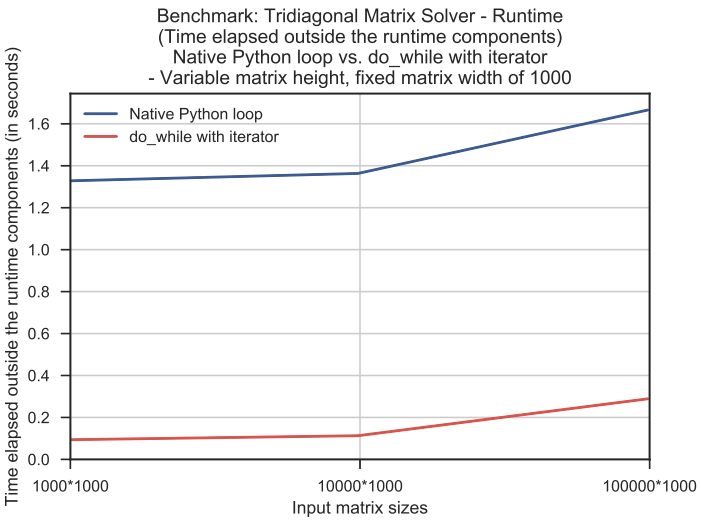


(a) Total time elapsed

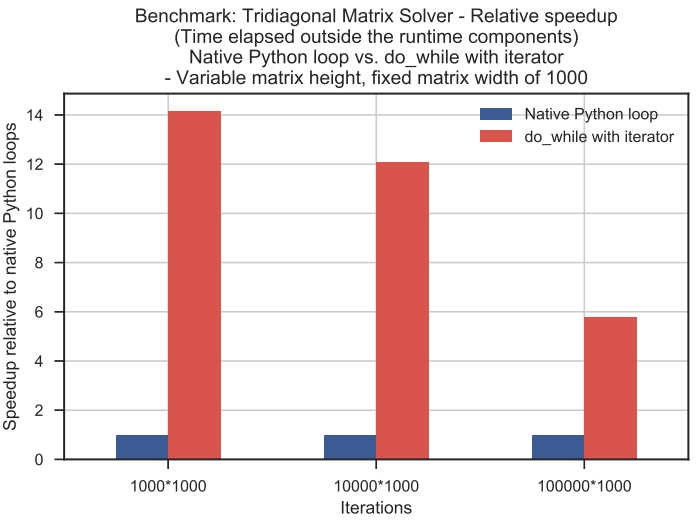


(b) Relative speedup

Figure 5.12: Absolute time difference and relative speedup in total time elapsed between using `do_while` loops, native Python loops and pure Numpy. The amount of iterations fixed to 1000, while the matrix input is variable.



(a) Total time elapsed



(b) Relative speedup

Figure 5.13: Absolute time difference and relative speedup in time elapsed outside the runtime components between using `do_while` loops and native Python loops. The amount of iterations fixed to 1000, while the matrix input is variable.

5.4.4 Quasicrystal

This benchmark produces images of moving quasicrystals, for a given amount of phases. A quasicrystal is an ordered structure, which is not periodic. The quasicrystal benchmark takes a number of plane waves, the number of stripes per wave, the image size in pixels and the number of images to produce. Two examples of quasicrystals can be seen in Figure 5.14.

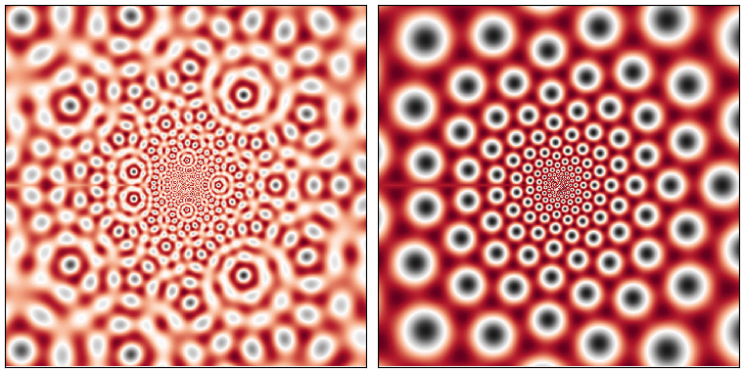


Figure 5.14: Example of two images of Quasicrystals, with a different amount of plane waves and stripes, created by this benchmark.

The loop body within the benchmark produces an image of the given phase. The code difference between native Python loops and `do_while` can be seen in Figure 5.15.

```
for phase in phases:
    image[:] = \
    np.sum(cinner * np.cos(phase) - sinner * np.sin(phase), axis=0) + k
```

(a) Quasicrystal using a native Python loop

```
def loop_body(phases):
    i = get_iterator()
    phase = phases[i]
    image[:] = \
    np.sum(cinner * np.cos(phase) - sinner * np.sin(phase), axis=0) + k
B.do_while(loop_body, len(phases), phases)
```

(b) Quasicrystal using `do_while` with an iterator

Figure 5.15: The difference in the Quasicrystal between using native Python loops and `do_while`

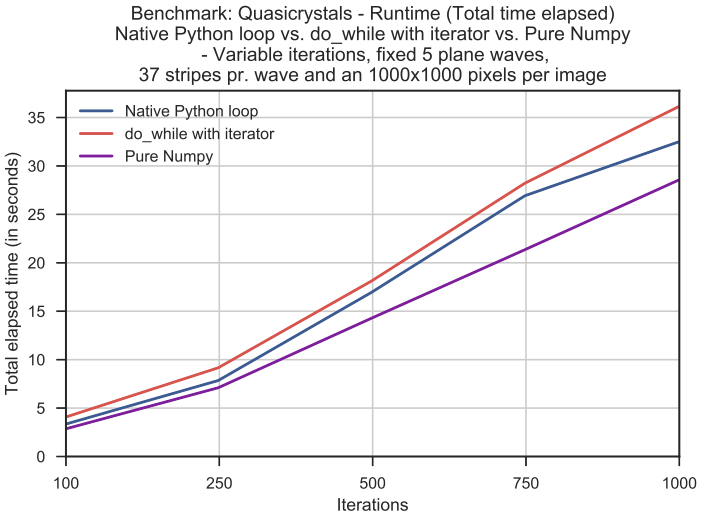
For benchmarking Quasicrystal, the plane waves are fixed to 5, the stripes 37 and image size to 1000, while the amount of iterations is variable. The difference in overall execution time can be seen in Figure 5.16, while the difference in time spent outside the runtime components can be seen in Figure .

The Quasicrystal benchmark is from the Benchpress collection.

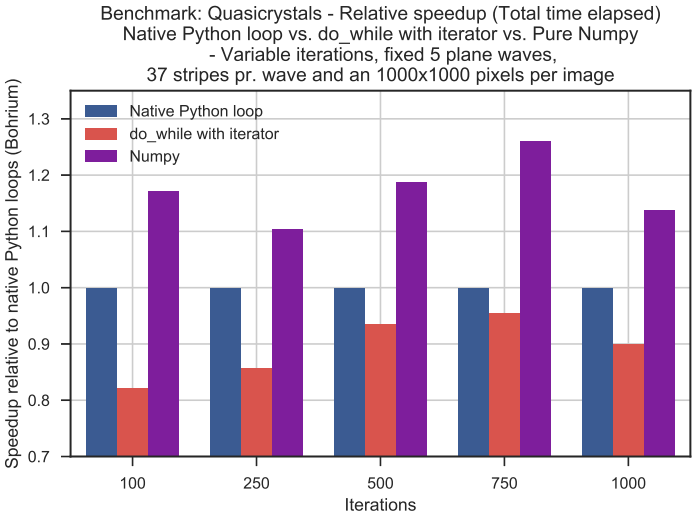
Breakdown of Results

The result of this benchmark is a bit different than the previous. As it can be seen in Figure 5.16, Bohrium never beats pure Numpy in runtime, due to the limited amount of computations within the loop body. It also shows, that using `do_while` with iterators can lead to worse performance than native Python loops in particular scenarios.

The reason why native Python loops beats `do_while` is not related to loop overhead, which is clearly in favor of `do_while` as shown in Figure 5.17a. The reason for the gap concerns the fusion Bohrium performs. In the loop body, the variable `phase` is a constant within the native Python loop, while it is a view of size 1 in the `do_while`. Having `phase` as a constant allows the fuser to perform more aggressive optimization on the loop body. The loop body is therefore executed faster when using a native Python loop instead of a `do_while`.

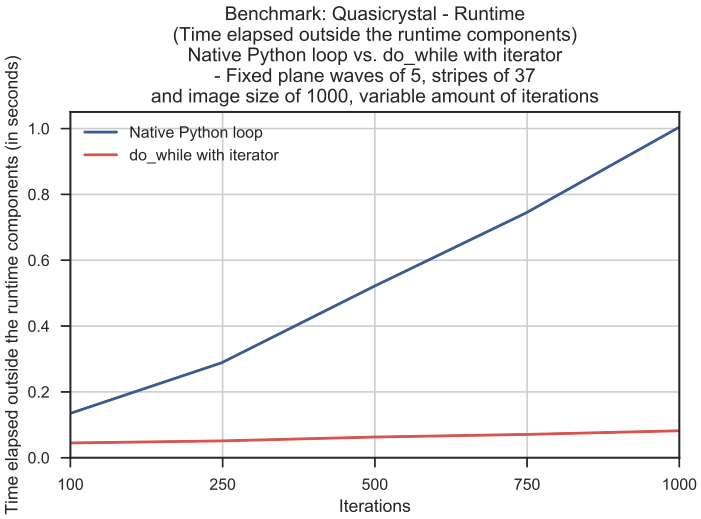


(a) Total time elapsed

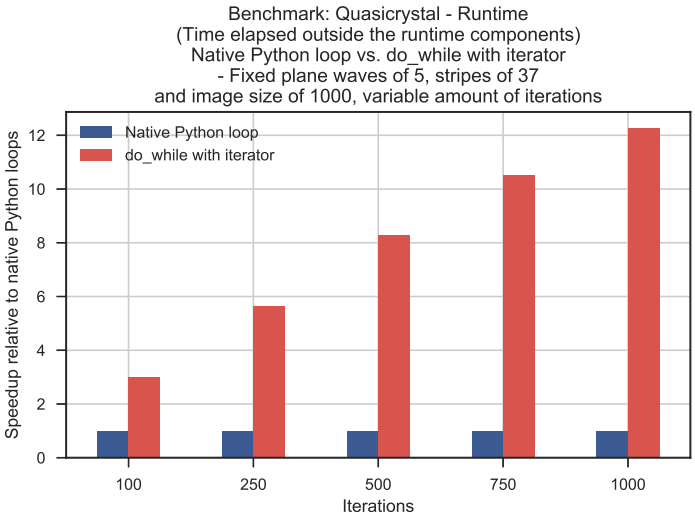


(b) Relative speedup

Figure 5.16: Absolute time difference and relative speedup in total time elapsed between using `do_while` loops, native Python loops and pure Numpy. The amount of iterations is variable.



(a) Total time elapsed



(b) Relative speedup

Figure 5.17: Absolute time difference and relative speedup in time elapsed outside the runtime components between using `do_while` loops and native Python loops. The amount of iterations is variable.

5.5 General Discussion of Performance

The general expectation of `do_while` with sliding views is that it notably reduces the time spent on overhead in the benchmarks. The optimization relies on the sliding handling sliding within Bohrium, instead of within Python. Thus, it is natural that the possible performance gain on a benchmark is proportional to the amount of time spent on an iteration by the Python interpreter and the Numpy bridge. If the loop body is computationally heavy and does not require many iterations, then the speedup of using sliding views and `do_while` is very minimal. It is due to the overhead only accounting for a minimal portion of the overall runtime. On the other hand, a computationally light loop body that goes on for many iterations results in a significant speedup when utilizing `do_while` with sliding views.

Sliding views within `do_while` can generally speed up all the benchmark categories. It primarily depends upon two essential factors:

Iterations The loop should generally contain enough iterations to make the overhead a factor of the total runtime.

Overhead impact The overhead of the Numpy bridge should make an impact on the execution of a loop iteration. If the loop body is very computational heavy, then minimizing the overhead will have a very minimal effect on the total runtime.

If these two factors are satisfied, then the `do_while` with sliding views can speed up the loop by avoiding the overhead.

A concrete example of this can be seen within the Lattice Boltzmann D2Q9 benchmark in Figure 5.7b. Here the bridge overhead of the loops amounts to $\approx 7\%$ which is the upper bound for speedup using `do_while`. Shrinking the size of the input would increase the performance gain, due to the overhead having a more significant impact on the time spent on each iteration.

In these four benchmarks, using the `do_while` with sliding views was faster in 3 out 4 cases. In all these cases, `do_while` was able to make a noticeable difference concerning the overhead. If the iterations are variable, it can easily be seen that the increase in overhead grows much slower using `do_while` instead of native Python loops.

The reason that the Quasicrystal benchmark resulted in a slowdown was that the actual calculations differed from native loops and `do_while`. The native loops were able to use the iterated values as constants instead of a view of shape 1, which allowed the fuser to perform more aggressive optimization. As it can be seen in Figure 5.17b, the time spent on overhead is significantly reduced by using `do_while`. Therefore, it is probably not worthwhile to use `do_while` when iterating over constant, since the aggressive optimization might save more time than reducing the overhead.

5.5.1 Brief Discussion of Array Indexing

Even though array indexing is not benchmarked, it is somewhat similar to sliding views. As with sliding views, it reduces the overhead of interpretation of the loop body to a constant factor. If the loop body requires splits, the result

would be that the overhead of caches and preparing execution would rise linearly with the number of splits. Using sliding views is therefore always as fast as array indexing, and often faster.

The benefit of array indexing is that it allows using `do_while` in some instances where sliding views does not apply. In these scenarios, the benefit of having more `do_while` loops will probably outweigh the increased overhead of executing the instruction(s).

5.6 Running the Benchmarks

As mentioned in the beginning of this chapter, the benchmarks can be found at [11]. Executing these benchmarks requires an installation of both Bohrium and Benchpress. Each benchmark has a dedicated folder that consists of the benchmark using pure Numpy and using `do_while` with iterators and a config generator. The config generator creates a `.json` file that can be executed with benchpress. To view a more detailed profiling of the execution time, set `prof = true` within `$HOME/.bohrium/config.ini` for the used vector engine.

The results can be reproduced by running the following commands within each folder:

1. Execute `python <benchmark_name>_config.py`
2. Execute `bp-run --nruns 5 <config_name>.json`
3. Execute `bp-raw <config_name>.json`

Chapter 6

Related Work

This section covers existing work on optimizing the high-productivity language Python by translating it into a high-performance language.

Bohrium focuses on array programming. Therefore, the intermediate bytecode lacks specific features, such as control flow. However, it allows more straightforward and aggressive optimization. This chapter focuses on frameworks with different approaches for speeding up Python. The goal of these frameworks are not precisely the same as Bohrium but within the same field. The frameworks are compared to Bohrium.

6.1 Cython

Cython is an optimizing static compiler that creates C extensions directly from Python source code. The Python can then call these C extensions, utilizing the efficiency of compiled C programs. Cython requires type-annotation within Python, due to C being statically typed. Cython extensions are thus non-polymorphic, and the notation is more verbose than regular dynamic Python notation.

Adding additional notation and static typing somewhat comprises one of the principles in *The Zen of Python*[5], that *Simple is better than complex*. To avoid changing a Python implementation completely, Cython only encourages using this notation in the computational heavy parts of the source code. The usage of Cython is based on the philosophy of the Pareto Principle[1].

Cython has several motivational factors. Primarily, translating Python to the lower-level language C has the benefit of pre-execution analysis, compiled executables and the potential performance gain during execution. Secondarily, it exposes Python to existing native C libraries such as GNU Scientific Library [3] and C interfaces for libraries such as Fortran's LAPACK [4] and BLAS [1].

In contrast to the vector-based programming model, where the Numpy package is suited, the primary focus of Cython optimization is low-level computational loops. Python tends to be slow at such loops, mostly due to its dynamic nature [14]. Cython and Numpy can be mixed in a fashion where

[1] 80% of the runtime of a program happens in 20% of the source code

Numpy handles the array calculations within the loop body, while Cython takes care of the loop.

An example of rewriting Python code to Cython can be seen in Figure 6.1. The primary difference is the additional type declarations by using the Cython operator `cdef` in Figure 6.1b. The Cython program is 24 times faster than the Python program [15].

```
def f(x):
    return sin(x**2)

def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

(a) Integrate function written in Python

```
def f(double x):
    return sin(x**2)

def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

(b) Integrate function written in Cython with type-declarations

Figure 6.1: An example of writing an integrate function in native Python and in Cython. The example is taken from [15].

One of the limitations of Cython is the lacking support for shared memory parallelism. Cython allows parallelism through message passing with MPI, but not shared memory multiprocessing (such as OpenMP). Thus, it is a bit complicated to achieve data parallelism.

Cython does not provide any fusion but merely executes the operations as described by the user. Therefore, it will require some amount of knowledge and skill to produce highly-optimized Cython programs. Cython's goal is primarily to make Python faster by adding simple annotations and is not a runtime system that optimizes the structure of the code itself. The responsibility is left to the C compiler, compiling the extensions.

In contrast to Bohrium, Cython can be very efficient in programs that do not utilize array programming very much or are inherently sequential. Light computational loops and control flow fits naturally within Cython and beats Bohrium, due to the overhead of the runtime system. However, in large numerical programs that use a significant amount of array programming, Bohrium beats Cython. Cython is also only able to produce C extensions, which are specific to CPUs.

6.2 Numba

Numba is a function-at-a-time just-in-time compiler for CPython that generates optimized machine code using the LLVM compiler. It is a library for Python, that can be loaded without changing the standard interpreter of Python. The initial purpose of Numba was to optimize the inefficient use cases of Numpy, as opposed to writing external programs in high-performance languages such as C or Fortran. Th Numba focuses on using the Numpy nd-array and numeric scalars in loops.

The Numba uses specialized functions, which are declared using a Numba decorator in Python. Such functions will then be compiled just-in-time and executed through the compiled executable. When Python calls the function, it is compiled with the types of the given input arguments. The functions can be annotated by the user to skip this process. The decorated functions are not allowed to be polymorphic. An example of implementing the *axpy* function using a Numba decorator can be seen in Figure 6.2. In this example, Numpy would allocate an output array for both the multiplication and the addition. Numba creates a loop over the arrays, performing these operations on each index and stores the temporary result in a scalar that can fit inside a register. It results in a speed up, due to bypassing the need to allocate another array. This example is a simple, but larger and more complex expressions can also be optimized.

```
@jit
def axpy(a, x, y):
  return a * x + y
```

Figure 6.2: An example of implementing the *axpy* function as a Python function using a Numba decorator.

The compilation of Numba goes through the following steps:

1. The Python function is translated into Numba's immediate representation (IR)
2. Numba inspects the IR to infer the type of the values within the function
3. If Numba can infer all types, the IR can be *lowered* into LLVM, which can be executed as machine code. This process is known as the *nopython mode* since it can be translated directly to machine code without relying on the Python runtime system for any operations. It also means, that the *Global Interpreter Lock* of Python can be avoided, which allows for parallel execution.
4. If Numba cannot infer the type of some values, the type is seen as a Python object. Numba then uses the C-API and is reliant on Python for the runtime execution.

Numba support various backends, such as single-threaded CPU and multithreaded CPU and GPU.

Numba is in contrast to Bohrium in the fact that it relies on creating custom functions. The initial purpose of the two frameworks is also different since Bohrium replaces Numpy (almost) entirely, while Numba extends it in the cases where it is inefficient. Bohrium is generally able to apply more aggressive fusion, but is also more limited and relies on more on Python for interpreting control flow and other operations which are not within the field of array programming. Bohrium therefore often has a more considerable overhead, which makes Numba faster on small problems or problems that lead to high overhead within Bohrium.

Chapter 7

Future work

This thesis has explored two approaches for handling dynamic views within Bohrium. Both of these approaches are based on the `do_while` function and the flush-and-repeat structure. This chapter will cover some improvements that can be made to the concrete implementations, as well as discussing other possible approaches and improvements of loops as a part of Bohrium.

7.1 Improvements to Sliding View within `do_while`

The implementation of sliding views within Bohrium has certain limitations. It carries the limitations of the `do_while`, which is not discussed in this section. However, apart from these limitations, some improvements can be made.

- An improvement to sliding views is handling operations that modify the slide in particular fashion. An example of this could be that a transposition should swap the dynamic changes of the swapped shapes. Unfortunately, this would need to be handled within the bridge and therefore requires re-implementation in each bridge.
- Another improvement is handling more bridges than the Numpy bridge. The overhead should be the essential motivation for the choice of bridge. The C++ bridge does not have nearly as much overhead as the Numpy bridge. Therefore, it is not necessary to implement the functionality of sliding views.
- Supporting some form of reshaping that does not make the slide non-linear.

7.2 Automatic Detection of Loops

An idea that improves the user-experience could be to automatically detect loops within Python and translate them to into using `do_while`. It could make the internal loops easier to use since it could be achieved without changing a single line of code.

Automatically detecting loops includes some immediate difficulties such as

Figuring out when to use `do_while` loops In the current form, the `do_while` loops has certain restrictions due to its limitations (described in Section 4.1.2). The restrictions need to be satisfied before using such a loop.

Finding the loops The loops must be found by inspecting the Python bytecode at runtime or figuring out a clever way to overwrite the internal functionality. Intuitively, this does not seem easily achieved without making some modifications of the CPython interpreter. Changing CPython would make it much more difficult to use Bohrium which directly defeats the purpose of the framework itself. However, there might be a way to make it even easier and intuitive to use `do_while` loops.

7.3 Parsing Python's Function Bytecode

A more complex loop approach is bypassing the dependency of the Python interpreter (CPython) by handling the translation from Python bytecode into vector bytecode in Bohrium with a custom-made parser. The Python bytecode of a function can easily be accessed as a *Python Code Object* through the attribute `.func_code`. The custom-made parser could either translate the Python bytecode directly into vector bytecode, or translate it into another Bohrium bridge which does not force as much overhead. This section motivates that using Bohrium's C++ DSL also leads to a large reduction in overhead.

7.3.1 Python Bytecode Directly to Vector Bytecode

The benefit of translating Python bytecode directly into vector bytecode is that it avoids the overhead of the Python interpreter. Therefore, it naturally limits the expressiveness of the loop body to the set of vector bytecode instructions. Creating a transpiler from Python bytecode to vector bytecode involves implementing multiple extensive features:

Parsing operations Each operation in the loop body must be translatable into a corresponding vector bytecode. It could be implemented as a large look-up table with an operation id and corresponding vector bytecode for that operation.

Symbol table Functions within Python have a symbol table for all variables. It includes the arguments, global variable, and constants. Since the Python bytecode operates by indexing into the symbol tables, the transpiler must also keep track of the variables in the symbol table, such that they can be translated to operands to the vector bytecode.

Parsing functions The loop body function is allowed to call other Python functions. Therefore, the transpiler must be able to execute recursively on these.

Numpy fallback The vector bytecode does not include certain operations, such as control flow. It normally forces a Numpy fallback, where Python handles the operation. It must also be handled by the transpiler, which must force a fallback of certain operations.

The two latter points can be avoided by disallowing certain operations and function calls within the loop body. The first two points are the essence of the transpiler.

The difficulty of parsing expressions goes beyond having a look-up table. It must also handle allocating temporary arrays and view manipulation, such as broadcasting.

This approach achieves something close to using `do_while`. It avoids depending on Python and Numpy for interpreting the loop body but requires a creating a custom-made parser for Python bytecode.

The only possible performance benefit is that the custom-made parser is faster than the Python interpretation of the first iteration. Since the first iteration is generally a small part of the computations within a loop, the possible performance is generally not much. The reason for implementing a custom-made parser is that it allows expressions which cannot be handled by using the Python interpreter.

Array indexing could use this approach, such that is does not depend on shapes within the Python interpreter.

7.3.2 Python Bytecode to C++ DSL

The natural way to achieve extended expressiveness, without extending the vector bytecode, is to use another bridge language for the unsupported operations. The downside of using Python is the overhead it introduces when using loops. An approach to overcome this overhead could be to make a transpiler from Python bytecode into Bohrium's C++ DSL.

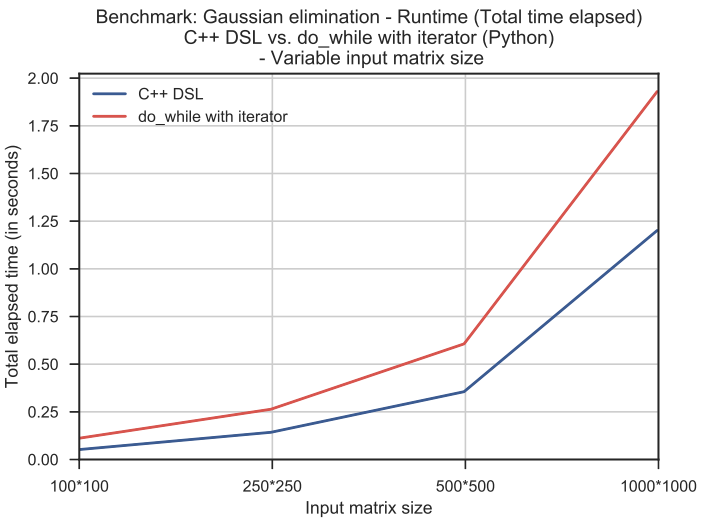
The C++ bridge introduces much less overhead than the Numpy bridge. This point is illustrated through the Gaussian elimination benchmark (described in Section 5.4.1). The results from `do_while` benchmark (also Section 5.4.1) is measured against a handwritten implementation in the C++ DSL for Bohrium. The handwritten C++ DSL implementation of Gaussian elimination can be seen in Appendix B. The results of the benchmark can be seen in Figure 7.1.

These results should not be compared in the sense that transpiling Python to the C++ DSL is as fast as using it natively. The results shows that the overhead, related to the benchmark, is almost identical. It can be seen in Figure 7.1 that Python is ≈ 0.04 seconds slower than C++ in setting up the program, but the growth in overhead is almost equal. It indicates that this approach would be feasible for reducing overhead while improving the expressiveness.

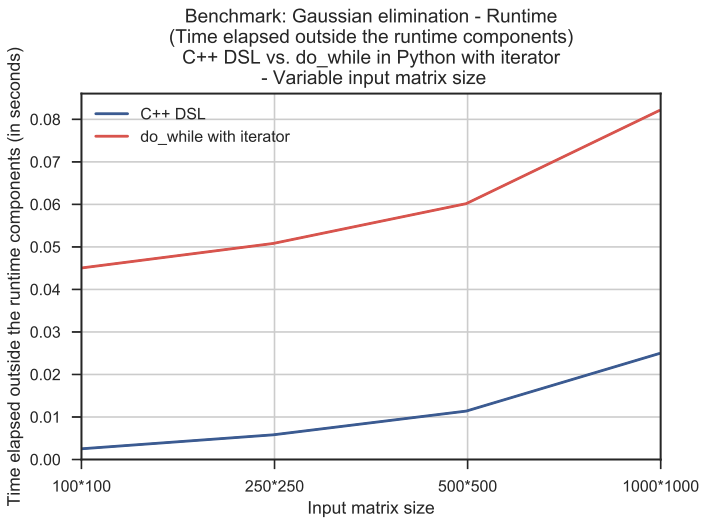
It should be noted, that the source code using the Numpy bridge is four lines and the source code using the C++ DSL is ≈ 80 lines. The difference is because the C++ DSL does not implicit support temporary arrays or implicit broadcasting. The views must also be described explicitly using a base array, shape, stride and offset. Therefore, the operations naturally blow up in sizes. It is a challenge for the transpiler, but it should be durable.

CPython can use functions in C extensions natively, which eases the process of calling the function in the transpiled C++ DSL. Views can also be transferred directly from the Numpy bridge to the C++ bridge. It already happens, since all bridges must go through the C++ bridge as a gateway to Bohrium.

The natural approach of implementing the Python bytecode to C++ DSL is supporting a subset of Python operations. The transpiler can after that be extended. In principle, the transpiled code can handle just the same as using



(a) Total time elapsed without compilation



(b) Time spent outside the runtime components without compilation

Figure 7.1: Absolute time difference between using `do_while` loops in Python and the C++ DSL. The size of the input matrix is variable.

native Python loops, including dynamic if statements, nested loops, and other advanced features.

Chapter 8

Conclusion

Throughout this thesis, we have explored possibilities of efficiently handling loops as a part of the Bohrium runtime system.

We have presented an implementation that extends the internal loop structure of Bohrium with linearly *sliding views* as part of the loop body. The implementation fits well within Bohrium. It does not require modification of the internal bytecode and only requires slight extensions of existing components, which does not break any existing features. The sliding view implementation primarily extends the metadata of views. Views are the updated between each iteration by using a filter that performs the updates based on this metadata. The sliding view implementation works with all current backends in Bohrium and is easily extendable to future backends.

The Numpy bridge fully supports sliding views through the `do_while` function and the iterator object. Using sliding views only requires a slight modification of the Python source code since it has a syntax similar to the one of Numpy. Sliding views support multiple handy features from Numpy, such as implicit broadcasting and nesting views.

The sliding view extension of `do_while` has lead to an asymptotic decrease in overhead in relation to using native Python loops with Bohrium. We have documented this through multiple benchmarks, which all show a decrease in overhead. The results generally show that using `do_while` with sliding views gives a speedup in the total amount of time spent. The ratio between time spent on overhead and time spent on execution of the loop body naturally upper bounds the possible performance gain.

The Bohrium runtime system has adopted the sliding views extension to `do_while` as part of its official release.

The most significant limitation of sliding views is the forced linearity of the slides. To avoid this limitation, we have explored the possibility of using arrays for indexing into views. Array indexing uses arrays as placeholders for user-defined changes to view metadata. It results in great expressiveness for changing views and does not require hardcoding features within Bohrium. The array indexing implementation suffered through some essential limitations, the most substantial being lack of user-friendliness in its syntax within Python. We found that this approach is not a feasible solution, due to these limitations. Bohrium has therefore not included this implementation in its official release.

Bibliography

[1] Blas - basic linear algebra subprograms. http://www.netlib.org/blas/. Accessed 14/02/18.

[2] The github repository of the bohrium runtime system. https://github.com/bh107/bohrium.

[3] Gsl - gnu scientific library. https://www.gnu.org/software/gsl/. Accessed 14/02/18.

[4] Lapack - linear algebra package. http://www.netlib.org/clapack/. Accessed 14/02/18.

[5] Pep 20 – the zen of python. https://www.python.org/dev/peps/pep-0020/. Accessed 21/05/18.

[6] Python (programming language). http://colenak.ptkpt.net/IT/108-1/reflective_3809_colenak-ptkpt.html. Accessed 28/04/18.

[7] The top500 supercomputer list. https://www.top500.org/statistics/list/. Accessed 04/06/18.

[8] Wikipedia page for amdahl's law. https://en.wikipedia.org/wiki/Amdahl%27s_law. Accessed 04/06/18.

[9] An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002.

[10] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[11] S. S. Andersen. Benchmarks for jit compiled loops in python. https://github.com/skjoenberg/JITCompiledLoopsInPython-Benchmarks.

[12] S. S. Andersen. A branch that contains the implementation of array indexing in a fork of bohrium. https://github.com/skjoenberg/bohrium/tree/ArrayIndexing.

[13] S. S. Andersen. The two pull requsts that implements sliding views within the master branch of bohrium. https://github.com/bh107/bohrium/pull/543 and https://github.com/bh107/bohrium/pull/512.

[14] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[15] S. Behnel, R. W. Bradshaw, and D. S. Seljebotn. Cython tutorial. pages 4 – 14, 2009.

[16] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38(11):1526–1538, Nov. 1989.

[17] G. E. Blelloch. Nesl: A nested data-parallel language. Technical report, Pittsburgh, PA, USA, 1992.

[18] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data parallel haskell: A status report. pages 10–18, 2007.

[19] B. Chamberlain. Why chapel? (part 3). https://www.cray.com/blog/why-chapel-part-3/. Accessed 06/06/18.

[20] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[21] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, Mar. 1988.

[22] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec 1966.

[23] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[24] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[25] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of high performance fortran. *Commun. ACM*, 54(11):74–82, Nov. 2011.

[26] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter. Bohrium: A virtual machine approach to portable parallelism. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 312–321, May 2014.

[27] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[28] P. E. Ross. Why cpu frequency stalled - the data (ieee). https://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled. Accessed 04/06/18.

[29] J. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, Apr. 1991.

[30] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[31] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523, 2011.

Appendix A

Appendix

A.1 Code Difference Between Using `do_while` and Native Python Loops In LBM D2Q9

```
    for i in range(0, 9):
        cu = 3 * (cx[i] * ux + cy[i] * uy)
        fEq[i] = rho * t[i] * (1 + cu + 0.5 * cu ** 2 - \
                         1.5 * (ux ** 2 + uy ** 2))
        fOut[i] = fIn[i] - omega * (fIn[i] - fEq[i])

    # Microscopic boundary conditions
    for i in range(0, 9):
        # Left boundary:
        fOut[i, 0, 1:ly-1] = fEq[i,0,1:ly-1] + 18 * t[i] * cx[i] * cy
            [i] * \
           (fIn[7,0,1:ly-1] - fIn[6,0,1:ly-1] - fEq[7,0,1:ly-1] + fEq
               [6,0,1:ly-1])
        # Right boundary:
        fOut[i,lx-1,1:ly-1] = fEq[i,lx-1,1:ly-1] + 18 * t[i] * cx[i]
            * cy[i] * \
           (fIn[5,lx-1,1:ly-1] - fIn[8,lx-1,1:ly-1] - \
            fEq[5,lx-1,1:ly-1] + fEq[8,lx-1,1:ly-1])
        # Bounce back region:
        #fOut[i,bbRegion] = fIn[opp[i],bbRegion]
        #Using a explict mask
        if bbRegion is not None:
            masked = fIn[opp[i]].copy() * bbRegion
            fOut[i] = fOut[i] * ~bbRegion + masked
```

Figure A.1: Code snippet of Lattice Boltzmann D2Q9 using native Python loops

```
    def loop_body(cx, ux, cy, uy, fEq, fOut, fIn, omega, rho, t):
        i = get_iterator()
        cu = 3 * (cx[i:i+1] * ux + cy[i:i+1] * uy)
        fEq[i] = rho * t[i:i+1] * (1 + cu[i:i+1] + 0.5 * cu[i:i+1] **
             2 - 1.5 * (ux ** 2 + uy ** 2))
        fOut[i] = fIn[i:i+1] - omega * (fIn[i:i+1] - fEq[i:i+1])

    B.do_while(loop_body, 9, cx_arr, ux, cy_arr, uy, fEq, fOut, fIn,
        omega, rho, t_arr)

    # Microscopic boundary conditions
    def loop_body2(cx, ux, cy, uy, fEq, fOut, fIn, t, ly, lx):
        i = get_iterator()
        # Left boundary:
        fOut[i, 0, 1:ly-1] = fEq[i,0,1:ly-1] + 18 * t[i:i+1] * cx[i:i
            +1] * cy[i:i+1] * \
           (fIn[7,0,1:ly-1] - fIn[6,0,1:ly-1] - fEq[7,0,1:ly-1] + fEq
               [6,0,1:ly-1])
        # Right boundary:
        fOut[i,lx-1,1:ly-1] = fEq[i,lx-1,1:ly-1] + 18 * t[i:i+1] * cx
            [i:i+1] * cy[i:i+1] * \
           (fIn[5,lx-1,1:ly-1] - fIn[8,lx-1,1:ly-1] - \
            fEq[5,lx-1,1:ly-1] + fEq[8,lx-1,1:ly-1])

    def loop_body3(cx, ux, cy, uy, fEq, fOut, fIn, t, ly, lx):
        i = get_iterator()
        # Left boundary:
        fOut[i, 0, 1:ly-1] = fEq[i,0,1:ly-1] + 18 * t[i] * cx[i] * cy
            [i] * \
           (fIn[7,0,1:ly-1] - fIn[6,0,1:ly-1] - fEq[7,0,1:ly-1] + fEq
               [6,0,1:ly-1])
        # Right boundary:
        fOut[i,lx-1,1:ly-1] = fEq[i,lx-1,1:ly-1] + 18 * t[i:i+1] * cx
            [i:i+1] * cy[i:i+1] * \
           (fIn[5,lx-1,1:ly-1] - fIn[8,lx-1,1:ly-1] - \
            fEq[5,lx-1,1:ly-1] + fEq[8,lx-1,1:ly-1])
        masked = fIn[opp[i]].copy() * bbRegion
        fOut[i] = fOut[i] * ~bbRegion + masked


    if bbRegion is not None:
        B.do_while(loop_body3, 9, np.array(cx), ux, np.array(cy), uy,
             fEq, fOut, fIn, np.array(t), ly, lx)
    else:
        B.do_while(loop_body2, 9, cx_arr, ux, cy_arr, uy, fEq, fOut,
            fIn, t_arr, ly, lx)
```

Figure A.2: Code snippet of Lattice Boltzmann D2Q9 using `do_while`

Appendix B

C++ DSL Gaussian elimination

```
#include <iostream>
#include <bhxx/bhxx.hpp>
using namespace bhxx;

int main()
{
    // Shape of input matrix
    long int N = 500;
    Shape shape = {N,N};
    BhArray<float> a(shape, contiguous_stride(shape));

    // Fill the input matrix with random numbers
    BhArray<uint64_t> b(shape, {(long int)shape.at(0), 1});
    random(b, 1, 1);
    identity(a,b);
    free(b);
    divide(a,a,std::numeric_limits<uint64_t>::max());
    multiply(a,a,100);

    // Define the size the dimension which is iterated over
    size_t size = a.shape[0];
    Runtime::instance().flush();


    // Loop and loop body
    for (size_t c = 1; c < size ; c++) {
        // Python syntax:
        // a[c:, c - 1:] -= (a[c:, c - 1] / a[c - 1, c - 1:c])[:, None]
             * a[c - 1, c - 1:]
        // This is explicitly turned into:
        // a1 -= (a2 / a3) * a4
        // a5 = (a2 / a3)
        // a6 = a5 * a4
        // a1 = a1 - a6

        // Get the shape of the view into array a (shrinks every
            iteration)
        Shape shape = {size-c, size-c+1};

        // a1: a[c:, c - 1:]
```

```
        BhArray<float> a1 = BhArray<float>(a.base,
                     shape,
                     a.stride,
                     (c * a.stride.at(0)) + (c-1) * a.stride.at(1));
        // a2: a[c:, c - 1]
        BhArray<float> a2 = BhArray<float>(a.base,
                     shape,
                     {a.stride.at(0), 0},
                     (c * a.stride.at(0)) + (c-1) * a.stride.at(1));
        // a3: a[c - 1, c - 1:c]
        BhArray<float> a3 = BhArray<float>(a.base,
                     shape,
                     {0, 0},
                     ((c-1) * a.stride.at(0)) + (c-1) * a.stride.at(1))
                         ;
        // a4: a[c - 1, c - 1:]
        BhArray<float> a4 = BhArray<float>(a.base,
                     shape,
                     {0, a.stride.at(1)},
                     ((c-1) * a.stride.at(0)) + (c-1) * a.stride.at(1))
                         ;
        // Temporary arrays
        BhArray<float> a5(shape,  a.stride);
        BhArray<float> a6(shape, contiguous_stride(shape));

        // Perform the array operations
        divide(a5,a2,a3);
        multiply(a6,a5,a4);
        subtract(a1,a1,a6);

        // Free the views
        free(a1);
        free(a2);
        free(a3);
        free(a4);
        free(a5);
        free(a6);

        // Flush
        Runtime::instance().flush();
    }

    // Devide by the diagonal
    BhArray<float> diag = BhArray<float>(a.base,
        a.shape,
        {a.stride.at(0) + a.stride.at(1), 0}, 0);
    BhArray<float> a7(a.shape,  a.stride);
    divide(a7, a, diag);

    Runtime::instance().flush();

    // Use the result to force the calculation
    (void)a7;
    return 0;
}
```

Appendix C

Benchmark results

This appendix contains all data used for producing the plots in this thesis. The appendix is split a section for each benchmark. The concrete benchmark and related Figure is described in the title of each subsection. All the results are in seconds.

C.1 Heat Equation

Native Python Loops vs. do_while - Total Time Elapsed (Figure 3.5)

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
100	0.551365	0.551424	0.550672	0.552614	0.55024
500	2.065149	2.055291	2.065509	2.086963	2.060671
1000	3.941033	3.926163	3.934977	3.937326	3.905907
2500	9.618109	9.560994	9.503453	9.523078	9.539764
5000	18.968328	19.155854	18.979647	18.858298	18.962472

Table C.1: Total time elapsed using Bohrium with native Python loops in the Heat Equation benchmark

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
100	0.405761	0.404952	0.407058	0.405485	0.40457
500	1.314027	1.31838	1.320992	1.324067	1.320042
1000	2.450553	2.465876	2.465698	2.452402	2.477347
2500	5.894314	5.889495	5.960589	5.863008	5.855859
5000	11.60121	11.526704	11.576203	11.549222	11.524071

Table C.2: Total time elapsed using Bohrium with `do_while` with iterators in the Heat Equation benchmark

Native Python Loops vs. do_while - Time Elapsed Outside the Runtime Components (Figure 5.3)

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
100	0.0408219	0.0407622	0.0415508	0.0411443	0.0408223
500	0.0561009	0.0552037	0.055668	0.0558144	0.055066
1000	0.0727635	0.0735454	0.0739786	0.0728405	0.0746328
2500	0.124282	0.128667	0.126837	0.126612	0.122563
5000	0.208019	0.216121	0.211461	0.216888	0.216323

Table C.3: Time elapsed outside the runtime components using Bohrium with native Python loops in the Heat Equation benchmark

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
100	0.169864	0.168848	0.17049	0.169128	0.169286
500	0.719968	0.711464	0.71491	0.739862	0.720211
1000	1.38062	1.3818	1.35485	1.37052	1.36371
2500	3.44959	3.39281	3.35224	3.36349	3.36908
5000	6.69968	6.55374	6.77681	6.64432	6.79495

Table C.4: Time elapsed outside the runtime components using Bohrium with `do_while` with iterators in the Heat Equation benchmark

C.2 Gaussian elimination

Native Python Loops vs. do_while with Iterator - Total Time Elapsed (Figure 5.2)

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
100×100	0.129799	0.131297	0.130185	0.134141	0.131334
250×250	0.354861	0.360364	0.353304	0.357447	0.354173
500×500	0.841169	0.833103	0.84257	0.834011	0.835739
1000×1000	2.728231	2.691467	2.71399	2.713846	2.66621

Table C.5: Total time elapsed using Bohrium with native Python loops in the Gaussian elimination benchmark

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
100×100	0.1867	0.092105	0.09133	0.091643	0.091515
250×250	0.373242	0.235903	0.237144	0.23533	0.237175
500×500	0.776573	0.566323	0.560999	0.563564	0.564822
1000×1000	2.261963	1.85459	1.835167	1.842517	1.854306

Table C.6: Total time elapsed using Bohrium with `do_while` with iterators in the Gaussian elimination benchmark

Native Python Loops vs. do_while with Iterator - Time Elapsed Outside the Runtime Components (Figure 5.3)

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
100×100	0.0927506	0.0945023	0.0932487	0.0936039	0.0941453
250×250	0.187695	0.191167	0.186229	0.189007	0.186149
500×500	0.352158	0.349015	0.352721	0.346166	0.350361
1000×1000	0.739103	0.714742	0.721456	0.728655	0.712259

Table C.7: Time elapsed outside the runtime components using Bohrium with native Python loops in the Gaussian elimination benchmark

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
100×100	0.045075	0.0457731	0.0454868	0.0447828	0.0439029
250×250	0.0512157	0.0506299	0.0508269	0.0506915	0.0506955
500×500	0.0601827	0.0602726	0.0604314	0.0604773	0.0595189
1000×1000	0.0818881	0.0823171	0.0819901	0.0820531	0.082154

Table C.8: Time elapsed outside the runtime components using Bohrium with `do_while` with iterators in the Gaussian elimination benchmark

do_while with Iterator vs. C++ DSL - Total Time Elapsed (Figure 7.1)

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
100×100	0.0529207	0.05121	0.0510289	0.0522914	0.0503707
250×250	0.141809	0.141106	0.144294	0.144398	0.142096
500×500	0.354403	0.361248	0.357103	0.351951	0.353549
1000×1000	1.20138	1.21517	1.19638	1.19015	1.2023

Table C.9: Total time elapsed using the C++ bridge in Bohrium in the Gaussian elimination benchmark

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
100×100	0.00249912	0.00251592	0.00249131	0.00247485	0.00249956
250×250	0.00578952	0.00582764	0.00583745	0.00581401	0.00576394
500×500	0.0113224	0.011443	0.0114209	0.0114151	0.0113455
1000×1000	0.025108	0.0251653	0.0251166	0.0240978	0.0251604

Table C.10: Time elapsed outside the runtime components using the C++ bridge in Bohrium in the Gaussian elimination benchmark

C.3 Lattice Boltzmann D2Q9

Native Python Loops vs. do_while with Iterator vs. Pure Numpy - Total Time Elapsed (Figure 5.5)

Cylinder size	Run 1	Run 2	Run 3	Run 4	Run 5
100×100	3.249304	3.265206	3.256104	3.230815	3.233663
1000×1000	8.775606	8.886555	8.659007	8.752588	8.67693
1500×1500	14.493795	14.595372	14.680029	14.796234	15.198797
2000×2000	22.93049	23.097057	23.428149	23.204801	23.037299
2500×2500	33.824476	34.504985	34.426627	33.436218	33.664412

Table C.11: Total time elapsed using Bohrium with native Python loops in the Lattice Boltzmann D2Q9 benchmark

Cylinder size	Run 1	Run 2	Run 3	Run 4	Run 5
100×100	2.324748	2.250807	2.259893	2.249042	3.863414
1000×1000	9.614365	7.651047	7.977149	7.894877	7.60438
1500×1500	13.535015	13.943496	13.938182	13.794283	13.803483
2000×2000	22.461099	22.248939	22.550734	22.14467	21.865968
2500×2500	32.380847	33.559291	32.957853	32.613421	32.512398

Table C.12: Total time elapsed using Bohrium with `do_while` with iterators in the Lattice Boltzmann D2Q9 benchmark

Cylinder size	Run 1	Run 2	Run 3	Run 4	Run 5
100×100	0.082871	0.083295	0.083175	0.083985	0.083243
1000×1000	9.959279	10.159842	10.231873	10.119377	10.067706
1500×1500	21.974908	22.068648	22.093072	22.092101	22.12603
2000×2000	38.333089	38.388296	38.462096	38.29349	38.261624
2500×2500	62.274634	61.816113	62.643529	62.219366	62.579144

Table C.13: Total time elapsed using pure Numpy in the Lattice Boltzmann D2Q9 benchmark

Native Python Loops vs. do_while with Iterator - Time Elapsed Outside the Runtime Components (Figure 5.6)

Cylinder size	Run 1	Run 2	Run 3	Run 4	Run 5
100×100	1.49079	1.49831	1.4981	1.4748	1.47312
1000×1000	1.63469	1.6085	1.5869	1.60458	1.57881
1500×1500	1.67606	1.68202	1.68147	1.66413	1.66158
2000×2000	1.79732	1.79011	1.75148	1.78564	1.78409
2500×2500	1.93117	1.90184	1.92216	1.91073	1.99965

Table C.14: Time elapsed outside the runtime components using Bohrium with native Python loops in the Lattice Boltzmann D2Q9 benchmark

Cylinder size	Run 1	Run 2	Run 3	Run 4	Run 5
100×100	0.538803	0.476644	0.480248	0.48458	0.497984
1000×1000	0.574835	0.568751	0.553141	0.560328	0.567333
1500×1500	0.641202	0.628749	0.633278	0.633303	0.634975
2000×2000	0.729576	0.736783	0.734492	0.736652	0.735045
2500×2500	0.868004	0.868804	0.869129	0.866674	0.87108

Table C.15: Time elapsed outside the runtime components using Bohrium with `do_while` with iterators in the Lattice Boltzmann D2Q9 benchmark

Native Python Loops vs. do_while with Iterator - Total Time Elapsed (Figure 5.7)

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
25	12.932254	12.845485	12.807704	13.386647	13.402197
50	26.491114	25.969812	26.951535	26.977483	25.739319
75	38.726643	40.175776	40.32093	39.757458	38.46214
100	51.482113	51.895326	52.638627	53.452204	51.676184

Table C.16: Total time elapsed using Bohrium with native Python loops in the Lattice Boltzmann D2Q9 benchmark

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
25	12.015178	12.051683	12.393348	12.098721	12.320924
50	24.476696	24.706707	23.779818	24.519277	24.774788
75	35.674723	35.80345	36.629309	37.307324	37.315177
100	48.942532	47.549734	47.703445	47.514934	49.852277

Table C.17: Total time elapsed using Bohrium with `do_while` with iterators in the Lattice Boltzmann D2Q9 benchmark

Native Python Loops vs. do_while with Iterator - Time Elapsed Outside the Runtime Components (Figure 5.8)

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
25	1.64534	1.62705	1.64491	1.67007	1.66079
50	3.1313	3.11601	3.07705	3.18869	3.15514
75	4.62297	4.67616	4.57591	4.67388	4.63237
100	6.16528	6.08662	6.04979	6.08533	6.06204

Table C.18: Time elapsed outside the runtime components using Bohrium with native Python loops in the Lattice Boltzmann D2Q9 benchmark

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
25	0.634419	0.633796	0.622464	0.623784	0.616501
50	1.0556	1.07968	1.07712	1.07329	1.05112
75	1.52663	1.50137	1.50905	1.49545	1.49822
100	1.93977	1.9543	1.95423	1.9222	1.96732

Table C.19: Time elapsed outside the runtime components using Bohrium with `do_while` with iterators in the Lattice Boltzmann D2Q9 benchmark

C.4 Tridiagonal Matrix Solver

Native Python Loops vs. do_while with Iterator vs. Pure Numpy - Total Time Elapsed (Figure 5.10)

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
10000×100	0.605732	0.608763	0.604871	0.609622	0.60588
100000×100	1.145819	1.145353	1.136464	1.143399	1.143842
1000000×100	6.655974	6.684266	6.664184	6.69185	6.663966

Table C.20: Total time elapsed using Bohrium with native Python loops in the Tridiagonal Matrix Solver benchmark

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
10000×100	0.478152	0.476207	0.478979	0.476796	0.477811
100000×100	1.001286	0.985472	0.996482	0.994958	1.004217
1000000×100	6.553664	6.552341	6.537191	6.537886	6.534618

Table C.21: Total time elapsed using Bohrium with `do_while` with iterators in the Tridiagonal Matrix Solver benchmark

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
10000×100	0.043034	0.042669	0.043055	0.043606	0.042918
100000×100	0.846918	0.845367	0.845282	0.843576	0.842647
1000000×100	9.71465	9.738713	9.921835	9.73837	9.735608

Table C.22: Total time elapsed using pure Numpy in the Tridiagonal Matrix Solver benchmark

Native Python Loops vs. do_while with Iterator - Time Elapsed Outside the Runtime Components (Figure 5.11)

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
10000×100	0.171298	0.174296	0.173302	0.174177	0.171772
100000×100	0.202653	0.202609	0.197653	0.204273	0.199138
1000000×100	0.389999	0.393599	0.392416	0.396901	0.389587

Table C.23: Time elapsed outside the runtime components using Bohrium with native Python loops in the Tridiagonal Matrix Solver benchmark

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
10000×100	0.049193	0.0497132	0.0497154	0.0496756	0.0497674
100000×100	0.0676653	0.0679306	0.0681772	0.0678729	0.0680173
1000000×100	0.242852	0.245263	0.238223	0.241983	0.238063

Table C.24: Time elapsed outside the runtime components using Bohrium with `do_while` with iterators in the Tridiagonal Matrix Solver benchmark

Native Python Loops vs. do_while with Iterator vs. Pure Numpy - Total Time Elapsed (Figure 5.12)

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
1000×1000	2.283535	2.267172	2.259402	2.265105	2.276266
10000×1000	2.502261	2.484794	2.483215	2.473133	2.478464
100000×1000	7.790881	7.809673	7.792907	7.781869	7.8292

Table C.25: Total time elapsed using Bohrium with native Python loops in the Tridiagonal Matrix Solver benchmark

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
1000×1000	1.069338	1.057709	1.05805	1.061003	1.057852
10000×1000	1.253604	1.237343	1.245131	1.245278	1.243612
100000×1000	6.342957	6.342455	6.361713	6.344982	6.335569

Table C.26: Total time elapsed using Bohrium with `do_while` with iterators in the Tridiagonal Matrix Solver benchmark

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
1000×1000	0.078835	0.07927	0.083725	0.075786	0.077584
10000×1000	0.473027	0.471747	0.471406	0.471019	0.46214
100000×1000	10.088353	10.126048	10.12359	10.135305	10.144265

Table C.27: Total time elapsed using pure Numpy in the Tridiagonal Matrix Solver benchmark

Native Python Loops vs. do_while with Iterator - Time Elapsed Outside the Runtime Components (Figure 5.13)

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
1000×1000	1.33165	1.33287	1.32219	1.32443	1.32779
10000×1000	1.37491	1.35531	1.36766	1.3579	1.36007
100000×1000	1.66266	1.67509	1.66655	1.63947	1.68261

Table C.28: Time elapsed outside the runtime components using Bohrium with native Python loops in the Tridiagonal Matrix Solver benchmark

Input size	Run 1	Run 2	Run 3	Run 4	Run 5
1000×1000	0.0935777	0.0950387	0.0940152	0.0933085	0.0929237
10000×1000	0.112965	0.113139	0.113189	0.112484	0.113019
100000×1000	0.288704	0.28963	0.288711	0.28784	0.288834

Table C.29: Time elapsed outside the runtime components using Bohrium with `do_while` with iterators in the Tridiagonal Matrix Solver benchmark

C.5 Quasicrystal

Native Python Loops vs. do_while with Iterator vs. Pure Numpy - Total Time Elapsed (Figure 5.16)

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
100	3.228212	3.957611	3.536279	2.900109	2.97331
250	8.410754	7.979065	7.862272	6.845308	8.125205
500	17.734834	14.699996	18.868326	15.212735	18.245579
750	26.458329	29.03801	24.036743	27.513468	27.565146
1000	26.662883	27.802139	39.271105	36.247685	32.3884

Table C.30: Total time elapsed using Bohrium with native Python loops in the Quasicrystal benchmark

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
100	3.867842	4.403844	3.963481	3.853888	4.121947
250	8.47053	9.458399	8.906092	9.640613	9.326346
500	17.361965	18.212112	17.8157	18.943099	18.277322
750	29.231706	27.15125	26.85696	28.6248	29.216997
1000	33.259588	36.486951	39.155676	36.999727	34.622333

Table C.31: Total time elapsed using Bohrium with `do_while` with iterators in the Quasicrystal benchmark

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
100	2.838947	2.795257	2.853274	2.8331	2.83561
250	7.117084	7.004081	7.128945	7.157839	7.0882
500	14.230319	14.236146	14.324889	14.341689	14.273399
750	21.443025	21.306093	21.527961	21.345427	21.150914
1000	28.502478	28.461436	28.470303	28.49579	28.692074

Table C.32: Total time elapsed using pure Numpy in the Quasicrystal benchmark

Native Python Loops vs. do_while with Iterator - Time Elapsed Outside the Runtime Components (Figure 5.17)

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
100	0.131602	0.130737	0.133235	0.13664	0.136542
250	0.289988	0.281111	0.291749	0.288868	0.290223
500	0.509648	0.534741	0.508497	0.533247	0.514987
750	0.74059	0.740976	0.748654	0.741889	0.746788
1000	1.01546	1.03136	0.966942	1.01426	0.985012

Table C.33: Time elapsed outside the runtime components using Bohrium with native Python loops in the Quasicrystal benchmark

Iterations	Run 1	Run 2	Run 3	Run 4	Run 5
100	0.0452866	0.0449474	0.0439239	0.0451709	0.044921
250	0.0514993	0.0509138	0.0513696	0.0508669	0.0508603
500	0.0705181	0.0609793	0.0613474	0.0602982	0.0608908
750	0.0702075	0.0710871	0.0712522	0.0702665	0.0711574
1000	0.0794059	0.0854311	0.0808981	0.0827319	0.0805686

Table C.34: Time elapsed outside the runtime components using Bohrium with `do_while` with iterators in the Quasicrystal benchmark