The Thesis Committee for Muhammad Hannan Naeem
certifies that this is the approved version of the following thesis:

Towards Automatic Migration of Sequential Kernels: Numba to PyKokkos

SUPERVISING COMMITTEE:

Milos Gligoric, Supervisor

George Biros, Co-supervisor

Towards Automatic Migration of Sequential Kernels: Numba to PyKokkos

by

Muhammad Hannan Naeem

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2025

Dedication

To family, friends, and mentors. Without whom this journey would be stale.

Acknowledgments

This Thesis is a result of constant and unwavering support of many people in my journey.

Achieving this milestone without my supervisor, Dr. Milos Gligoric, would have been impossible. Throughout my two years as a student, he has instilled discipline, hard work, and every great quality of an excellent researcher in me. His support as a mentor, both inside and outside academia, has opened countless opportunities, and I am forever thankful to him.

I am also eternally grateful for my co-supervisor Dr. George Biros. His exceptional guidance has been the main steering force behind my work. The gracious financial support from him and the Oden family enabled my work, and I am truly thankful.

I am delighted to have worked with my dear college and soon-to-be Doctor, Nader Al Awar. His depth in knowledge, helpful nature, and kind demeanor were a source of constant hope in difficult stretches. I cannot thank him enough for his help. His contributions with PyKokkos are the backbone of my work.

In the fast-paced, two years, it was a treat to work with, and be around my research group. I am thankful to my colleagues, for helping me at times when it was most needed, and making the workspace fun and productive.

The Electrical and Computer Engineering staff has been absolutely heroic. Melanie Gulick, made navigating University bureaucracy super easy. Her resourcefulness and care for students was always exceptional. Thomas Atchity made hiccups in enrollment seem like a non-issue, and Cayetana Garcia's promptness was always a relief. I am grateful for countless other people who made the department worthwhile.

I would also like to thank Aleksandar Milicevic, my manager and mentor at Cubist, where I interned during the summer of 2024. I cannot thank him enough for

the professional growth I achieved under his shadow. Likewise, I am also thankful the rest of Cubist family, for the incredible opportunity.

The credit for this journey extends beyond University of Texas. No gratitude can repay the sheer trust and resources my parents put in me, the support my wife, sibling and friends offered. I am also thankful to my undergraduate advisor at Lahore University of Management Sciences, Dr. Fareed Zaffar, whose guidance and motivation landed me at UT Austin.

Work in this document was partially supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003969.

Abstract

Towards Automatic Migration of Sequential Kernels: Numba to PyKokkos

Muhammad Hannan Naeem, MSE
The University of Texas at Austin, 2025

SUPERVISORS: Milos Gligoric, George Biros

High performance computing (HPC) frameworks have notably been useful for scientific communities, however, for an average user, leveraging their power can prove to be challenging. Using these highly performant frameworks requires knowledge about computer architectures, familiarity with parallel programming, and getting across a steep learning curve. Recent work on PyKokkos has bridged this gap considerably by bringing HPC to Python. While PyKokkos provides remarkable performance, switching to it is non-trivial as PyKokkos follows a programming model that is different for an average Python user. It requires type annotations in kernels, and has a rigid resemblance to its parent, Kokkos's programming patterns. Numba, on the other hand, is a well known just-in-time (JIT) compiler for Python that is widely used. This work improves the usability and migration cost to PyKokkos by introducing a new tool – Caramba. Caramba enables automatic PyKokkos kernel generation from Numba - instantly opening doors for hardware optimized parallel performance. We also add type propagation for kernel arguments in PyKokkos to allow Caramba to function statically. Finally, we evaluate the overhead and performance trade-off with the aforementioned features and show that it is negligible.

Table of Contents

List of Tables

List of Figures

Code Snippets

Chapter 1: Introduction

This document presents a tool called Caramba for translating Numba source code to PyKokkos source code, and type propagation enhancements for PyKokkos. Together they make PyKokkos easy to pickup and use, hence enabling users to easily integrate a High Performance Computing framework.

We will first give a brief summary of the background work, motivation, and familiarize the reader with relevant terminology. This will follow an in-depth explanation of Caramba and the Type propagation enhancement. Each section will discuss inner workings and examples. We will then evaluate our work with respect to performance overheads and discuss the results, which will be followed by addressing limitations of our contributions and the way forward. Finally, we will conclude with a summary.

1.1 Background

Historically, High Performance Computing (HPC) frameworks have mostly been used by the scientific and academic communities. The technical nature of parallel programming, computer architecture, micro architecture, and performance optimization has kept the barrier to entry fairly high in this space. On the contrary, languages like Python have made programming more accessible. The simplicity of Python allows both industry, and indirect academic spheres to benefit from computational power. This is supplemented by various powerful libraries for Python that have simplified advance technologies. Similarly, advances in the hardware ecosystem has landed highly performant systems in the hands of the ordinary consumer and in turn libraries like NumPy, CuPy, and Numba [9] have made it significantly easy to harness the available power. However, while generally faster, these libraries do not completely leverage architecture based optimizations.

Nader et al. [2] [3] solve this problem by introducing PyKokkos, a parallel performance portable framework for Python, based on Kokkos [8] [14] that is tuned to several hardware backends but abstracts away the inner workings from the user. The result is easy to write, fast, and portable code that automatically leverages hardware based optimizations. Snippet 1.1 shows a simple example on how to use PyKokkos. PyKokkos is a sister project of Kokkos and hence inherits many programming patterns from it. While it is considerably easy to learn, Kokkos programming patterns, can be unconventional to a Python based novice user. Hence, there is room to make PyKokkos even more user-friendly and Pythonic. At the time of this work PyKokkos is still under development, while more improvements are expected, the solution we present should stand in supplement to PyKokkos on its own (apart from the direct contributions to PyKokkos).

```
1     import pykokkos as pk
2
3     @pk.workunit
4     def y_init(i, y_view):
5         y_view[i] = 1
6
7     def run(N):
8         y = pk.View([N], pk.double)
9         p = pk.RangePolicy(0, N)
10
11        pk.parallel_for(p, y_init, y_view=y)
```

Code Snippet 1.1: A simple example with PyKokkos that creates and initializes a PyKokkos view.

1.2 Motivation

Numba [9] is a JIT compiler for Python. It is well-known for its simplicity and familiarity to the Python flavor. Therefore, it is a good go-to choice for many users who care about performance. We created Caramba to automate the migration from Numba source code to PyKokkos source code. Not only does Caramba give a one-to-one comparison for a novice learner, but also makes it a great utility to

migrate larger code bases, without re-writing everything manually. It is important to note that while Numba provides performance boost, it is not an HPC framework, and mostly exploits compiler level analysis for optimizations.

Since PyKokkos works in tandem with Kokkos, which is written in C++, at the time of this work it explicitly requires static type definition. This is achieved by user provided type annotations. However, vanilla Python does not require type annotations from the user. By introducing dynamic type propagation for kernel arguments, we make PyKokkos more Pythonic, eliminating the need for the user to provide type annotations. Furthermore, this feature enables Caramba to work statically without ever running the kernels.

1.3 Terminology

PyKokkos defines two kinds of function bodies: a PyKokkos *workunit* or *kernel*, and a PyKokkos *function*. In this work we use the terms kernel and workunit interchangeably. A PyKokkos function, is like a PyKokkos workunit, but only differs in the decoration, that is, while a kernel is decorated with `PyKokkos.workunit`, a function is decorated with `PyKokkos.function`. This distinction plays an important role under the hood, but for our purposes this becomes relevant when calling a kernel from within a kernel.

Functions decorated with Numba properties are also referred to as kernels in this work, but we always make the distinction between the parent framework, PyKokkos or Numba. However, the term workunit is exclusive to PyKokkos.

Chapter 2: Caramba

Caramba serves as a stand-alone tool for Numba to PyKokkos migration while also exposing useful API to enable programmatic integration. Caramba offers an AST to AST solution that is entirely based on static analysis and heavily uses the AST module for Python and its visitor design patterns. It can also unparse the translated AST to produce PyKokkos source code. Figure 2.1 shows the general input, output, and flow for Caramba.

```
1    if __name__ == "__main__":
2        # Invoke this script to begin translating
3        path= sys.argv[1]
4        caramba_parser: Parser = Parser(path)
5        caramba_parser.translate()
6        caramba_parser.unparse("translated.py")
```

Code Snippet 2.1: Caramba migration steps.

The main entry point for the tool is its stand-alone script 2.1. This script also demonstrates simple calls to parse, translate, and unparse. The script accepts a source file and calls the three methods in the aforementioned order, generating a new source file that contains the translated PyKokkos code. Caramba currently assumes that kernels are within a single file.

2.1 Parser

Caramba's internal methods are also exposed as API, which can be used programmatically. The aforementioned methods can be used at a programmer's disposal along with various other helpful methods and variables. Illustration 2.2 shows how the internal components of Caramba work in tandem for translation. The `Parser`

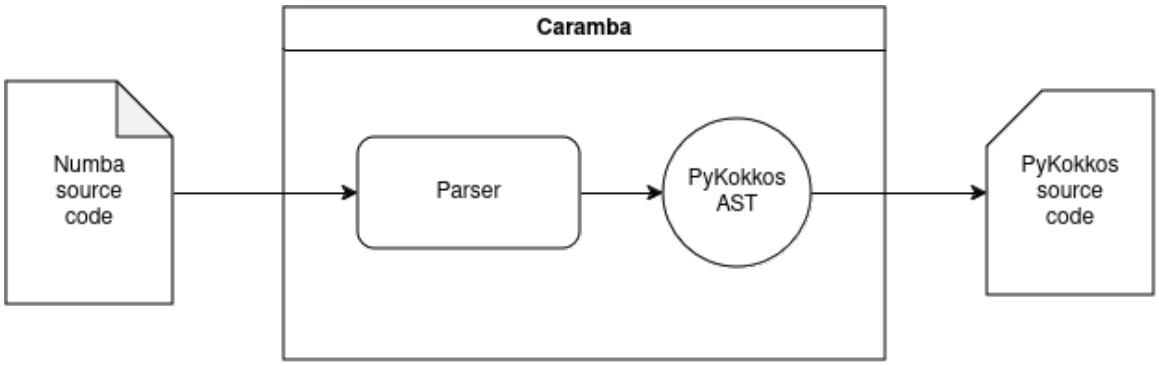

Figure 2.1: The input, operation, and output of Caramba.

constructs the AST of the source code and then marks all the valid kernel nodes that are to be translated. These kernels then become the primary input to the rest of the pipeline. At this stage, validation is simply checking if the kernel is decorated with Numba. Data wise the `Parser` exposes the AST for the user to fetch, and apply translation on. For translation the `Parser` houses a `Translator` object.

2.2 Translator

The `Translator` class manages the calls and data provided by the different visitors. Since the visitors work in tandem and share data, this class makes sure that dependent calls are wrapped in the right order. For example, the `ArrayTransformer` visitor can collect and provide information about different array variable identifiers that can then be used by the `SemanticTransformer` visitor to make necessary changes. Breaking the transformers into different visitors allows scalability, faster development, and it also allows the programmer to directly use a visitor if need be. In the intended flow of complete translation, the `Translator` is invoked by the `Parser` with the `translate_tree` method. Provided a list of kernels, from the `Parser`, `translate_tree` will first modify import nodes to insert PyKokkos import statement and remove the Numba import statement. Following this

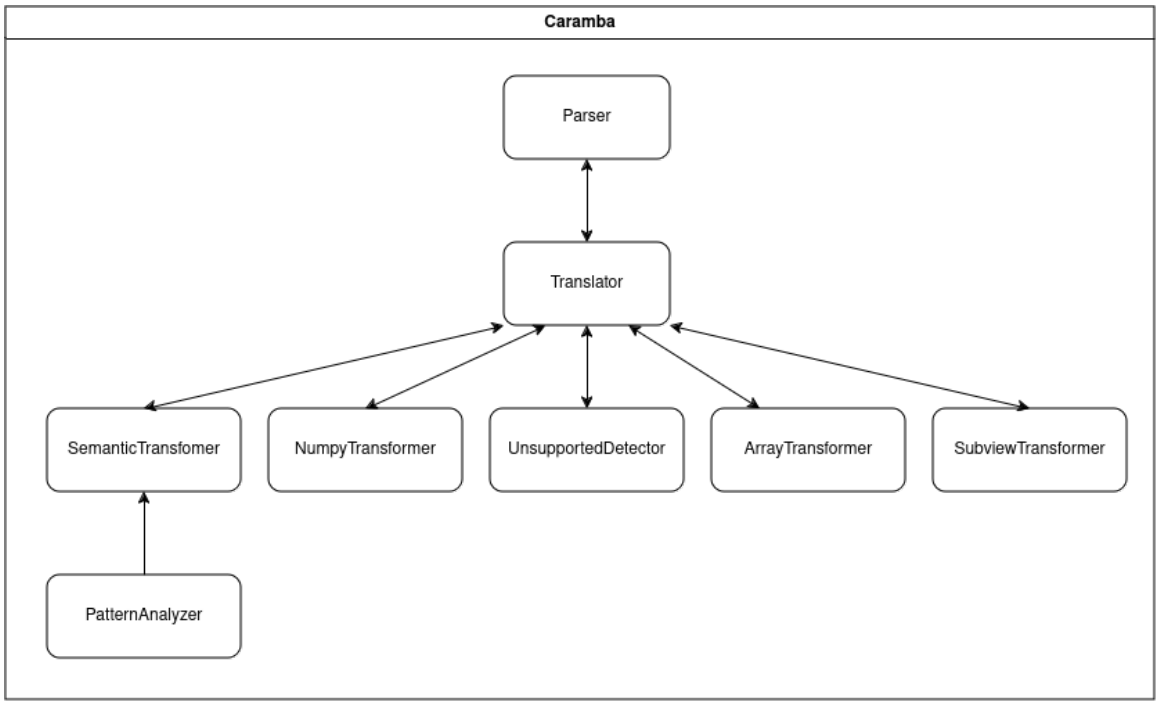

Figure 2.2: The Parser invokes its Translator which in turn manages all the visitors necessary to achieve translation.

`translate_kernel` is invoked for each of the kernels in the list provided by `Parser`. Here, an initial validation check is made for which we provide a separate visitor called `UnsupportedDetector`. `UnsupportedDetector` will rule out any kernels that use external libraries that PyKokkos cannot support (e.g., time, pandas, matplotlib, etc.), and will also detect if NumPy is being used in the kernel bodies. PyKokkos does not support NumPy within its kernel body, but provides a substitution for NumPy ufuncs. We follow the same theme with Caramba.

The `Translator` also provides some degree of inference - by collecting information about array variables, variables used for indexing, and external calls in kernels it can provide useful information to other visitors and conclude data types for views on function interfaces. For example, if an internal call to `kernelB` from `kernelA` accepts `cluster[i]` as an argument, which corresponds to `item` in `kernelB`, but `kernelB` performs an indexing on `item`, e.g, `x: int = item[0]`, then `cluster` must

be a 2D view. This information can then be passed to `SubviewTransformer` to rewrite `cluster[i]` as `cluster[i, :]` in `kernelA`. The snippet 2.2 shows this example written out. In snippet 2.2 `kernelB` indexes its parameter, `item`, which is passed as `cluster[i]`. In essence `kernelB` performs `cluster[i][0]` However by looking at `kernelA` body only it would have been impossible to know that `cluster` is a 2D view.

```
1   import pykokkos as pk
2
3   @pk.workunit
4   def kernelA(cluster):
5       ..
6       kernelB(cluster[i])
7
8   @pk.function
9   def kernelB(item)
10      ..
11      x: int = item[0]
```

Code Snippet 2.2: Infering view dimensions on function interfaces.

2.3 Visitors

Each visitor handles a different aspect of the translation. Broadly, these include PyKokkos specific semantics, views, subviews, and special cases like NumPy usage. Some visitors also provide utilities to existing visitors, e.g., validation. In this section we will discuss the main four visitors that work together.

2.3.1 Numpy Transformer

If the kernel is indeed using NumPy, then it is reverted to a normal Python function (by removing the decorators) and NumPy ufuncs are substituted by PyKokkos ufuncs. This is carried out by the `NumpyTransformer` which also validates that an equivalent ufunc exists in PyKokkos, or the translation cannot proceed. Otherwise, decorators are replaced with PyKokkos' own and `ArrayTransformer` and `SemanticTransformer` work in tandem to translate the kernel. Snippet 2.3 shows

the before and after in this case – the decorator is removed and NumPy ufuncs are replaced by PyKokkos ufuncs. Note that although Caramba can support NumPy in this manner, it comes at a great performance cost: most code will be executed in Python and only ufuncs will be optimized. Removing compilation to Kokkos and any possibility for optimizations is detrimental to kernel performance, but this is an area for future work. For now, we lay the stepping stones to build on.

```
1
2     # original numba kernel
3     @numba.njit(parallel=True)
4     def logistic_regression(Y, X, w, iterations):
5         for i in range(iterations):
6             w -= np.dot(((1.0 /
7                   (1.0 + np.exp(-Y * np.dot(X, w)))
8                   - 1.0) * Y), X)
9         return w
10    ...
11
12    # translated with Caramba
13    def logistic_regression(Y, X, w, iterations):
14        for i in range(iterations):
15            w = w - pk.dot((1.0 /
16            (1.0 + pk.exp(-1 * Y * pk.dot(X, w)))
17            - 1.0) * Y, X)
18        return w
```

Code Snippet 2.3: NumPy translation example. Note that in this case only ufuncs calls will use PyKokkos.

2.3.2 Array and Subview Transformers

While PyKokkos does provide a wrapper for NumPy arrays, one of the most contrasting difference between NumPy arrays and PyKokkos views is the indexing syntax and subviews. Slices of views are their own data type called subviews. At this point, while views are indexed in a nested fashion, much like Python lists, e.g., `A[i][j]`, subviews, like NumPy arrays are indexed by comma separated indices, e.g, `A[i:n-1, j]`. The only difference between views and subviews, syntactically, within PyKokkos is the presence of slice operator `':'`. Furthermore, in its cur-

rent state, PyKokkos requires all subviews to be declared in separate assignment statements and cannot otherwise exist in expressions. For example, given a view `A`, `call_function(A[i:n-1, j])` is invalid. Instead, `A_subview = A[i:n-1, j]` and then `call_function(A_subview)` is valid. These transformations are handled by the `ArrayTransformer` and `SubviewTransformer`. Both of these work in tandem with the `SemanticTransformer`. The `SubviewTransformer`, in particular, needs additional information from semantic and array transformers - view variable identifiers, and calls to other kernels. The `Translator` uses this information to infer if any passed arguments were subviews. The `SubviewTransformer` can then insert assignment statements for each subview, replace all references with the new subview declarations, and collect information about dimensions of each view. This information in turn can be used to generate `pk.function` annotations, which are explained in detail in towards the end of the next section. Finally, any array specific properties or functions, e.g., `shape`, are also substituted to their corresponding PyKokkos counterpart.

2.3.3 Semantic Transformer

Since PyKokkos has a lot of unique semantic properties, this particular visitor is the most complex. To begin with PyKokkos kernels are invoked with distinct parallel dispatch types. Most popular of these are `parallel_for`, `parallel_reduce`, and `parallel_scan`. Caramba currently supports `parallel_for` and `parallel_reduce` since both of these dispatches cover majority of supported use-cases. Snippet 2.4 shows how a PyKokkos kernel is invoked. While executing the chosen parallel dispatch type, kernel arguments and execution policy are passed along. Each parallel dispatch also expects some proprietary arguments for the kernel, namely thread ID and an accumulator variable (for `parallel_reduce` only). These are also taken care of by the `SemanticTransformer`. In PyKokkos the accumulator variable is automatically returned at the end of kernel execution, but this behavior must be explicitly present in Numba kernels. Therefore, `PatternAnalyzer`, a utility visitor to detect the parallel

dispatch type, can look for return statements and determine if the kernel is performing reduction, or in-place iterations, later of which correspond to `parallel_for`.

```
    import pykokkos as pk

    @pk.workunit # used with parallel_for
    def y_init(tid, y_view):
        y_view[tid] = 1

    @pk.workunit # used with parallel_reduce
    def yAx(tid, acc, cols, y_view, x_view, A_view):
        temp2: float = 0
        for i in range(cols):
            temp2 += A_view[tid * cols + i] * x_view[i]

        acc += y_view[tid] * temp2

    def run(N):
        y = pk.View([N], pk.double)
        p = pk.RangePolicy(0, N)
        pk.parallel_for(p, y_init, y_view=y)

        ..

        for i in range(nrepeat):
            result = pk.parallel_reduce(p, yAx,
                cols=M, y_view=y, x_view=x, A_view=A)
```

Code Snippet 2.4: An example of a PyKokkos kernel invocation.

Since these semantics are PyKokkos specific and may not be intuitive to pick up, Caramba generates a caller function for each kernel translated. Each of which automatically infers the policy and sets up the invocation for the programmer. As of now, Caramba chooses serial (single threaded) execution policy by default which behaves much like Numba's JIT. If parallelism is enabled in Numba decorated kernel, the `Range` execution policy is chosen and relevant for loops are eliminated. Snippet 2.5 highlights the execution policy selection by the `SemanticTransformer`. The generated caller function is identical to the protocol of the Numba source kernel, with the exception of the name, of course. Hence, it becomes a drop-in replacement instead for Numba kernel invocation. The caller also wraps any NumPy arrays into PyKokkos

views. Snippet 2.5 also shows how a caller function is intuitive to interface with as only a single parameter is required.

```

    # original kernel
    @numba.njit(parallel=True)
    def potential_numba_scalar_prange(cluster):
        energy = 0.0

        for i in numba.prange(len(cluster)-1):
            for j in range(i + 1, len(cluster)):
                    ..
                energy += e

        return energy

    # translated by Caramba
    @pk.workunit
    def potential_numba_scalar_prange(tid, acc, cluster):
        acc = 0.0
        for j in range(tid + 1, cluster.extent(0)):
                ..
            acc += e

    # caller function for PyKokkos kernel
    def call_potential_numba_scalar_prange(cluster):
        sample_exec_space = pk.ExecutionSpace.OpenMP
        space = pk.ExecutionSpace(sample_exec_space)
        cluster_view = pk.array(cluster)
        policy = pk.RangePolicy(space, 0, len(cluster) - 1)
        reduced = pk.parallel_reduce(policy, ..)
        return reduced


```

Code Snippet 2.5: Parallel kernels and caller functions.

Within the kernel body there are more semantics that need to be modified. For example, while type annotations are not required for arguments, they must be present for any declarations in the kernel body. The `SemanticTransformer` can infer the type from constant definitions, however, if the declaration stems from arguments, the static nature of Caramba may become a limitation. Furthermore, PyKokkos requires

different kinds of nested execution policies to support nested calls to other kernels, which are parallel in nature and complex to understand for a programmer unfamiliar with parallel programming. Fortunately, PyKokkos also has a function decorator, `Pyk.function`, which allows the decorated function to be invoked from within the kernel. The compiler automatically inlines the code decorated by `PyKokkos.function` within the caller kernel, appearing as it there was no external call. Therefore, to support nested calls, Caramba first needs to generate a PyKokkos function copy for the translated kernel and invoke it instead of the original kernel. The result is duplicated code, but PyKokkos, as of now, does not allow the same kernel definition to be decorated as both a kernel and function. PyKokkos function arguments must also be type annotated. The `SemanticTransformer` draws these annotations from the information collected by `ArrayTransformer` and the `SubviewTransformer`.

2.4 Discussion

It is important to note that both PyKokkos and Caramba are under active development. While we provide a solid stepping stone to build the translation infrastructure, it is easy to see that certain properties if updated within PyKokkos can notably simplify the translation pipeline. For example, making the subviews consistent with views in terms of syntax and getting rid of the assignment statement requirement for subviews, completely negates the need for a separate `SubviewTransformer`. Similarly, supporting multiple decorations for PyKokkos kernels will also help get rid of duplicate code generated as a result of nested calls. However, as of now, our objective is to support the current version of PyKokkos.

Python's AST module makes working with visitor design pattern straight forward, easy to understand, and scalable. We chose to perform the translation statically as it is not resource intensive, meaning that complex and intensive kernels do not have to run to achieve translation. Static translation for our purposes also keeps Caramba simple, especially by sticking with the AST module. We also avoid any possible per-

formance overheads in runtime. However, we note that static analysis has limitations, and acknowledge plans to include dynamic analysis in Caramba, particularly for type inference. At the same time, we show that Caramba is still functional with only static analysis.

Chapter 3: Type Propagation

Originally PyKokkos followed the statically typed nature of Kokkos in C++, which goes against Python's dynamic typing. As discussed earlier, this is one of the biggest reason for Python's appeal to a novice programmer. To bring PyKokkos on the same page, we partially remove the statically typed requirement by starting with kernel arguments and decorator arguments. Like the kernel body, PyKokkos requires all variable types to be annotated by the user, which is merely an option in Python and not a requirement. Moreover, workunit decorators require memory layout and space definition for all the corresponding views. Not only is this information readily available at compile time, but it creates superfluous annotations greatly hampering usability. In this part of our work we present an approach to automatically propagate the aforementioned type information without any input from the user.

3.1 Workunit Decorators

Snippet 3.1 shows an example of how a PyKokkos kernel is decorated. The decorator broadly serves two purposes: first, to allow the workunit to be identified as a PyKokkos kernel. Second, it provides memory space, layout, and trait configuration for every view that is an argument to the decorated workunit. This information is critical at compile time as different backends require different specifications of data. The view object being passed as an argument already has all this information, hence, the decorator can be abstracted away from the user and quietly injected into the AST. However, any user given configuration takes precedence over automatically inferred values.

```
    # Before type propagation
    @pk.workunit(
        y_view = pk.ViewTypeInfo(layout=pk.LayoutLeft,
        space=pk.CudaSpace)
    )
    def yAx(j :int, acc :pk.double,
        y_view: pk.View1D[pk.double]):
        ...

    # After Type propagation
    @pk.workunit
    def yAx(j, acc, y_view):
        ...


```

Code Snippet 3.1: Automatic type annotation and decorator generation.

3.2 Workunit Arguments

Similarly, the arguments to the kernel themselves are Python objects or variables with pre-inferred data types. Necessary information is already present within the variable object. For data types custom to PyKokkos, most prominently views, the information regarding dimension, size, and type of elements can also be extracted. This information can be collected when a parallel dispatch call is made at runtime. At this point, PyKokkos views, and the rest of arguments are already defined and are only being passed as arguments. Recall, that PyKokkos kernels also have some proprietary arguments, like thread ID, accumulator variable for reduction operations, and a boolean to indicate if `parallel_scan` has achieved completion. These arguments are always fixed in their types but the exact required combination depends directly on the execution policy and parallel dispatch type.

In case of primitive data types and views, a few additional steps are necessary. PyKokkos does not support strings or characters, so that mainly leaves us with integers and floats. Typically, these values are NumPy primitives, e.g., `NumPy.float64`, but either way the only transformation needed here is to infer these values as `PyKokkos.double` or verb—float— or `int`, depending on the actual value.

For views the element datatype is already stored in the object itself and only needs to be extracted. Once all annotations are collected, we abstract these away from the user, and inject them into PyKokkos AST before the compilation step eliminating the need for user to provide any explicitly. However, should a user choose to provide annotations, they will always take precedence over collected annotations.

3.3 Managing Compilation

While the aforementioned features are a great quality of life improvements, they do however present direct implications on compilation that must be addressed. Consider the case of invoking the same workunit with different type of arguments as shown in snippet 3.2. Under the hood PyKokkos generates C++ Kokkos code and compiles it into a shared object file. The compiled object file is indeed different for each differently annotated kernel. Therefore, we have this problem where PyKokkos will successfully compile a workunit based on the first set of arguments, but if the type of arguments change in the second call, the invocation will fail as type casting may not work for corresponding changes. This is because PyKokkos identifies a compilation with the kernel name only. Recall that we only abstract the annotations away, and behind the scenes they are present and part of the kernel definition, and by extension part of the C++ generated code. Hence, to fix this issue we need to make sure PyKokkos includes the collected annotations as a part of identification for the compilation. We do exactly that. By hashing the workunit name, view configurations, and annotation for each argument in order, PyKokkos can uniquely label each compilation. Next time the same kernel is invoked with different argument types, PyKokkos will check if the same hash exists, if not, a new compilation will be triggered and stored under the new hash. Cached compilations are reused in the same manner.

```
1
2       @pk.workunit
3       def yAx(j, acc, cols, y_view, x_view, A_view):
4
5           ...
6
7       def invoke1():
8           pk.set_default_space(pk.ExecutionSpace.OpenMP)
9           ...
10          # double datatype with Cuda as memory space
11          y: pk.View1D = pk.View([N], pk.double)
12          x: pk.View1D = pk.View([M], pk.double)
13          A: pk.View2D = pk.View([N, M], pk.double)
14          for i in range(nrepeat):
15              result = pk.parallel_reduce(p, yAx, cols=M, y_view=y,
16                      x_view=x, A_view=A)
17
18      def invoke2():
19          pk.set_default_space(pk.ExecutionSpace.Cuda)
20          ...
21          # float views with Cuda as memory space
22          y: pk.View1D = pk.View([N], pk.float)
23          x: pk.View1D = pk.View([M], pk.float)
24          A: pk.View2D = pk.View([N, M], pk.float)
25          for i in range(nrepeat):
26              result = pk.parallel_reduce(p, yAx, cols=M, y_view=y,
27                      x_view=x, A_view=A)
28
```

Code Snippet 3.2: Invoking PyKokkos kernel with different data types.

Chapter 4: Performance Evaluation

4.1 Caramba

There are mainly two aspects of Caramba that need to be evaluated. Correctness and performance. Correctness because we want to ensure integrity of the original Numba source code, and performance because we want to make sure that the cost of translation does not out-weigh the utility Caramba offers. We use differential testing and check correctness by comparing the outputs of original and translated code. This testing also drove the development of Caramba.

Performance wise there are two more branches that need to be considered. First, we measure the time cost for the actual translation – which is the primary cost at this stage of our project that we are concerned about. Second, we want to analyze that the performance of kernels themselves against their original Numba counterparts so that we can directly compare any differences in reported times. To evaluate these aspects we collected different Numba based kernels from across several repositories with a focus the official Numba documents and the examples within. We then invoked Caramba on these source files and recorded the translation times, followed the by the runtimes of the translated code. The kernels are arranged as follows in the source files:

- **simple1.py:** `sum2d`
- **simple2.py:** `sum3d`
- **lennard_jones_prange.py:** `lennard_jones_prange`
- **stencil_numba.py:** `star` and `star2`

4.1.1 Translation

As discussed earlier, Caramba primarily has two phases, parsing and translation. The parsing phase is primarily dominated by AST construction and organizing information for the `Translator` to start the second phase. In the translation phase, the actual transformations of the AST take place – nodes are replaced, modified, added or removed by the various visitors described earlier in this document. Table 4.1 shows the reported time in milliseconds it takes Caramba to fully translate a source file. We distinguish between the two phases, and table 4.1 clearly shows that the translation phase dominates the total reported time. This result is as expected – the parsing phase is mostly just the AST module constructing the AST and depends on the number of lexical tokens in the Python source code. Translation visitors on the other hand make multiple passes and perform multiple transformations depending on the size of the tree and the type of its nodes. For example, `stencil_numba.py` has many subview nodes, which as previously noted, require a lot of work in conjunction with views, hence, as expected the translation phase of `stencil_numba.py` is expensive.

However, we note that compared to the compile time and run time, time taken by Caramba is a one time occurrence. The user will only ever use Caramba once on their source code. Moreover, as we will see in the succeeding sections, the time taken to translate a kernel is comparable but significantly less than the run time for sizable inputs. Hence, we deem time taken by Caramba to be insignificant versus the run times of the kernels.

Table 4.1: Time taken by Caramba to translate a source file.

Source	**#Kernels**	**Parse time (ms)**	**Translation time (ms)**	**Total time (ms)**
simple1.py	1	0.69	2.12	2.81
simple2.py	1	0.41	2.95	3.36
lennard_jones_prange.py	3	0.83	6.31	7.15
stencil_numba.py	3	3.44	14.07	17.51

4.1.2 Kernel Performance

In this section the goal is to see how Caramba translated PyKokkos kernels perform against their Numba based counterparts. We expect the performance to be on-par if not better. Table 4.2 shows the reported times for each kernel from the source files mentioned in Table 4.1. For each kernel, the table reports an average time for different problem sizes. The average is taken from 13 total runs. PyKokkos wall time denotes the total time including the overhead that PyKokkos introduces over Kokkos, while the kernel time represents the actual time spent executing the kernel. We follow the official Numba documents to time the performance of Numba kernels. The suggested method is exactly the same as measuring the PyKokkos wall time, i.e., `start_time` subtracted from `end_time`. Authors of PyKokkos, on the other hand, report both the times noted here [2] but relate performance to the actual kernel time. Both frameworks cache compilations, hence, we provide a warm-up run to both, and only then start recording time. Currently, Caramba will set CPU to be the default execution space when translating. While future Caramba versions will support automatic selection between the CPU and GPU, the user can still easily switch the execution space in the generated code by making a simple edit. We provide more explanation and an example in the following section and snippet 4.1. However, not all kernels can benefit from running on GPU.

We find the reported kernel times in Table 4.2 to be similar to their Numba counterparts in all but one case. `lennard_jones_prange` uses a partially parallel

iterative solution, and it appears that Numba is able to optimize the iterations much better. To achieve better times in PyKokkos we use the portability of PyKokkos and run the kernel on GPU. By default, Caramba generated code uses OpenMP for parallel `pranges`. Since GPUs are much better at handling such parallel workloads, switching to GPU as shown in snippet 4.1 yields Table 4.3. We immediately see a significant improvement as the problem size increases. It is important to note that while PyKokkos does offer the same compilation utility to increase speedup, it is inherently a parallel programming framework, and hence designed around this paradigm. As it is, running serial workloads with parallel dispatches is unconventional. We discuss, in section 5.1 that in future versions of Caramba we plan to automatically introduce parallelism. Specifically, iterations appear to be much better optimized by Numba. Hence, it is clear that PyKokkos is superior in parallel workloads, but Numba, for now, can outperform it in serial workloads. `sum2d` and `sum3d` also rely on nested iterations to solve for an accumulated value in a serial fashion. We can also clearly see Numba getting a slight advantage in both of these cases confirming our suspicions. However, as expected, even without necessary parallel optimizations, PyKokkos is on-par with Numba. PyKokkos beats Numba in `star2` which updates slices of an array iteratively, in PyKokkos this is achieved by updating smaller subviews within a larger view. It appears that this scenario regarding data updates favors PyKokkos.

```
1     import pykokkos as pk
2
3     def call_potential_numba_scalar_prange_gpu(cluster):
4         # Manually changed from pk.ExecutionSpace.OpenMP
5         sample_exec_space = pk.ExecutionSpace.Cuda
6         ...
7         reduced = pk.parallel_reduce(policy,
    potential_numba_scalar_prange_gpu, cluster=cluster_view)
8         return reduced
9
10
```

Code Snippet 4.1: Running translated parallel code on GPU.

Looking at PyKokkos wall time, as noted by the authors [2] [4] originally, the

compilation overhead introduced by PyKokkos becomes more and more negligible as the input size increases. Between the kernel and the wall time, following the authors, we primarily looked at the kernel times when comparing against Numba, however we do note that difference between wall time and kernel time decreases with increasing problem sizes. With sizable inputs we do not observe a stark difference, but for smaller cases, as expected, the overhead appears to be significant.

Table 4.2: Kernels from aforementioned source files in table 4.1 profiled with different problem sizes.

Kernel	Size	PyKokkos wall time (ms)	PyKokkos kernel time (ms)	Numba time (ms)
lennard_jones_prange	10^2	1.62	0.22	0.08
lennard_jones_prange	10^3	5.03	3.46	2.23
lennard_jones_prange	10^4	80.34	79.00	60.83
lennard_jones_prange	10^5	7068.38	7067.12	4668.79
sum2d	10^2	1.42	0.01	0.02
sum2d	10^3	1.41	0.01	0.02
sum2d	10^4	1.40	0.06	0.07
sum2d	10^5	1.91	0.53	0.50
sum2d	10^6	6.39	5.00	4.96
sum2d	10^7	37.38	36.04	36.23
sum2d	10^8	139.30	137.98	136.88
sum3d	10^2	1.45	0.01	0.02
sum3d	10^3	1.48	0.01	0.02
sum3d	10^4	1.41	0.06	0.07
sum3d	10^5	1.88	0.48	0.52
sum3d	10^6	6.21	4.78	4.92
sum3d	10^7	33.48	32.16	36.30
sum3d	10^8	141.13	139.76	136.44
star	10^2	2.77	0.01	0.03
star	10^3	2.71	0.01	0.03
star	10^4	1.89	0.01	0.03
star	10^5	2.66	0.02	0.03
star	10^6	2.02	0.05	0.03
star	10^7	2.77	0.14	0.04
star	10^8	3.00	0.40	0.06
star2	10^2	1.97	0.01	0.03
star2	10^3	2.70	0.02	0.06
star2	10^4	1.96	0.06	0.50
star2	10^5	3.34	0.53	4.47
star2	10^6	7.71	5.03	35.16
star2	10^7	66.25	63.49	160.44
star2	10^8	346.54	343.95	1322.59

4.1.3 Possible Improvements

Table 4.2 reveals that kernels are on par with Numba is most cases, however, we note that to fully take advantage of the performance PyKokkos offers, most kernels will need to be restructured. Which implies that the next version of Caramba could include transformations based on performance optimizations. For example, PyKokkos would greatly benefit if `lennard` and `sum` kernels had all their iterations in parallel. However, a better semantic analysis is required to make this a general solution, as serial operations may sometime depend on their order. Furthermore, we need to investigate the issue with nested kernel calls with PyKokkos functions.

Table 4.3: PyKokkos kernels with parallelism benefits greatly by executing on GPU for larger problem sizes.

Kernel	**Size**	**PyKokkos GPU kernel time (ms)**	**Numba time (ms)**
lennard_jones_prange	10^2	0.76	0.08
	10^3	10.98	2.71
	10^4	104.90	86.12
	10^5	2178.50	5468.85

4.2 Type Propagation

For type propagation the overhead introduced is purely during compile time. This is when PyKokkos will inject the collected annotations into the AST, which is then translated into C++. It is important to note that compilation only occurs in the very first invocation of a kernel for a give set of argument types. The compilation is then cached, and at every subsequent run we only check the argument data types at run time and then proceed to call the correct compilation. Hence, a small overhead is also introduced at every subsequent run by checking the argument data types.

Table 4.4 shows the reported time of the first run versus subsequent run for each of the PyKokkos source file. The noted times are an average of 5 runs. To get these times take two measurements within PyKokkos: the start time when a parallel dispatch is invoked, and the end time just before the actual execution of translated call happens. These sources were collected from the PyKokkos repository examples.

We can clearly see, that the compilation overhead times do not change by much. We attribute this to that fact that type propagation is insignificant compared to PyKokkos' own compilation machinery. The most overhead by introducing Type Propagation is observed in `bytes_and_flops` where team policy is used, and the main kernel has a total of six arguments three of which are views. `gather` on the other hand uses the range policy and also has three views. Hence, we note the slight overhead comes from resolving team policy arguments. All in all, the times are very comparable and even within margin of error.

Table 4.4: We measure the total PyKokkos overhead before and after introducing type propagation. W/ Types heading indicates that user provided all the type annotations., while W/O Types indicates that the types were inferred and propagated.

Source	**Compilation run**		**Pre-compiled run**	
	W/ Types (s)	**W/O Types (s)**	**W/ Types (ms)**	**W/O Types (ms)**
01	10.19	10.15	0.33	0.85
bytes_and_flops	11.23	12.09	1.61	2.29
gather	11.68	11.57	1.56	2.61

Chapter 5: Limitations and Future Work

In this chapter we will discuss the limitations of Caramba's current version, address the shortcomings that we have noted previously in this document, and finally discuss related future updates for Caramba.

5.1 Caramba

In its current form Caramba shows great promise and utility, but as its parent project, PyKokkos, develops there are several potential improvements on the table. PyKokkos is an HPC framework, designed to be used to write highly performant parallel code. Numba on the other hand is not necessarily for parallel computing, but to provide faster compilation and runtimes. This difference ultimately raises conflicts in semantics that prove difficult to resolve. For example, Numba has operator overloading for element-wise operations on NumPy arrays, e.g., Array1 * Array2. While PyKokkos might support this in the future, when writing parallel kernels in PyKokkos, the programmer would typically be expected to take control of each thread working on each element in parallel, e.g, `View1[thread_id] * View2[thread_id]`, rather than letting the framework decide how to complete this operation on the entire views. PyKokkos does provide ufuncs, utility kernels for common operations, to invoke in such cases. However, as we noted while discussing `SemanticTransformer`, to avoid complex execution policies we can only invoke a PyKokkos function from a PyKokkos workunit, and not another workunit. This implies that to invoke multiply ufuncs, we must create a PyKokkos function copy for it, as internally it only exists as a workunit. While a plausible strategy, this would greatly increase the length of the translated source code causing confusion for the novice programmer. Which brings us to another limitation.

The different nature of both frameworks also brings about some performance

implications for the current version of Caramba. Numba, focuses on compiler based optimizations, while PyKokkos does constitute the same principle, the broader theme is parallelism. The current version of Caramba does not introduce parallelism automatically, but only converts the code to run serially. This is the most interesting area of work for the next Caramba versions, where we introduce semantic analyses to exploit possible parallelism. This change will eliminate the shortcomings of always running serial workloads.

Currently, Caramba produces substantial amount of auxiliary and critical code. This includes caller functions for each kernel, assignment statements for each subview used, annotations for new declarations, and PyKokkos function body for each kernel invoked in a nested manner. Together these supplements can compound to produce larger than expected source code. Currently, Caramba needs usability improvements in PyKokkos to address these shortcomings. Allowing the same kernel body to be decorated as a function, directly gets rid of any need to create a function duplicate. Similarly, allowing subview instantiations in line within expressions would completely disregard the `SubviewTransformer`, making PyKokkos and Caramba simpler. Lastly, with regard to type inference there are two points: first, type inference within the kernel bodies is expected in future PyKokkos versions, this would take away the need to handle this problem in Caramba altogether. Second, during the while type annotations are required by PyKokkos functions, a dynamic approach to read the arguments values and generate their type annotations is also a noteworthy improvement.

Finally, NumPy, CuPy, and external libraries have limited support in PyKokkos. While PyKokkos supports NumPy and CuPy arrays, their methods and properties may not be supported at all. Complete NumPy support is an integral part of Numba, and one of its biggest strengths. Unfortunately, PyKokkos is yet to mature in this category. Similarly, PyKokkos currently does not support calls to external libraries, except math functions.

5.2 Type Propagation

While the aforementioned improvements make a sizable improvement towards PyKokkos' usability, it is important to note that it is partial in nature. Currently, annotations are required in the kernel bodies themselves. The PyKokkos team plans to change this in the future updates. The primary purpose of this feature was to allow Caramba to work statically.

Chapter 6: Related Work

There are existing frameworks that follow the theme of bringing performance to Python because of its de facto status. Cython [5] adds static typing and translates the source to C or C++ natively. However, Cython like Numba [9], is not inherently designed as an HPC and does not offer hardware tuned parallel performance. Portability wise there are two notable works, PyTorch [13] and TensorFlow [1]. Both these frameworks are geared towards building machine learning applications and pipelines in Python. They support both CPU and GPU backends but are not a good choice for writing general purpose kernels. PyKokkos by contrast is much more general, promising performance for any kind of workload, including machine learning. IrGL [12] introduces an intermediate representation enabling parallel graph algorithms to be compiled to CUDA.

For migrating between code, Opdyke [10] [11] defines general operations to help refactor and incorporate changes automatically. Stratego/XT is a toolset [6] that takes in user defined transformation rule and can then transform programming languages. On the other hand, TXL [7] exists as a programming language itself, that is specifically designed to help in source transformation with rule definitions.

Chapter 7: Conclusion

In this work we introduced two improvements to usability for PyKokkos. Our main contribution is a framework called Caramba that offers migration from Numba to PyKokkos. Caramba is an AST to AST solution and therefore heavily uses the visitor design pattern to achieve translation. This pattern makes it modular and scalable. However, since PyKokkos and Caramba are both heavily in development, there are still some improvements to be made, both in terms of performance and user-friendliness. In most of the cases, however, we do show that the performance of Caramba translated code is on par with Numba source code. To enable Caramba to work statically, we also introduced Type Propagation, a feature that abstracts away user typed annotations from PyKokkos workunit arguments. This feature greatly reduces superfluous annotations allowing the code to look cleaner and more understandable. Our evaluation shows that the addition of Type Propagation does not add any significant overhead to PyKokkos compile time.

Works Cited

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016. URL `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`.

[2] Nader Al Awar, Steven Zhu, George Biros, and Milos Gligoric. A performance portability framework for Python. In *International Conference on Supercomputing*, pages 467–478, 2021.

[3] Nader Al Awar, Steven Zhu, Neil Mehta, George Biros, and Milos Gligoric. PyKokkos: Performance portable kernels in Python. In *International Conference on Software Engineering, Tool Demonstrations Track*, pages 164–167, 2022. doi: 10.1145/3510454.3516827.

[4] James Almgren-Bell, Nader Al Awar, Dilip S Geethakrishnan, Milos Gligoric, and George Biros. A multi-GPU Python solver for low-temperature non-equilibrium plasmas. In *International Symposium on Computer Architecture and High Performance Computing*, pages 140–149, 2022.

[5] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science and Engineering*, 13(2):31–39, 2011. doi: 10.1109/MCSE.2010.118.

[6] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science*

of computer programming, 72(1-2):52–70, 2008.

[7] James R Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[8] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2014.07.003. URL `http://www.sciencedirect.com/science/article/pii/S0743731514001257`. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[9] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[10] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[11] William F. Opdyke and Ralph E. Johnson. Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In *Symposium on Object-Oriented Programming Emphasizing Practical Applications*, pages 145–161, 1990.

[12] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on GPUs. pages 1–19, 2016. doi: 10.1145/2983990.2984015.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: an imperative style, high-performance deep

learning library. In *International Conference on Neural Information Processing Systems*, pages 8026–8037, 2019.

[14] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. The Kokkos ecosystem: Comprehensive performance portability for high performance computing. *Computing in Science Engineering*, 23(5):10–18, 2021. doi: 10.1109/MCSE.2021.3098509.

Vita

Muhammad Hannan Naeem is currently a Software Engineer at Cubist. He works on developing and designing blockchain solutions that meet the security and performance needs of real world systems. Muhammad Hannan Naeem received his undergraduate degree in Computer Science from Lahore University of Management Sciences.

Address: hannan@utexas.edu

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.