Leiden University

Computer Science

A Data Structure Optimizing Compiler for tUPL

Name:	Dennis van der Zwaan
Date:	19th of June, 2019
1st supervisor:	Dr. K.F.D. Rietveld
2nd supervisor:	Prof. dr. H.A.G. Wijshoff

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

A Data Structure Optimizing Compiler for tUPL

Dennis van der Zwaan

Supervisors:
Dr. K.F.D. Rietveld
Prof. dr. H.A.G. Wijshoff

Abstract

Traditional programming languages have a range of artifacts that prevent compilers from being able to optimize the code on a grand-scale, such as explicit parallelization, explicit data structure definitions and data dependencies. tUPL is an alternative, high-level programming language that aims to avoid these problems. In this thesis we present libtupl, an optimizing compiler for tUPL, which has the ability to automatically generate efficient data structures for algorithms through a range of simple transformation passes. We show that advanced data structures derived through application of these transformation passes have the ability to outperform naive data structures and in certain scenarios outperform hand-optimized implementations of MKL. To enable tUPL to be used in any existing Python code base, we developed Tython, a front-end for libtupl that enables tUPL to be used in any existing Python code base. The use of Tython will be illustrated with a number of examples and its implementation will be discussed.

Contents

Chapter 1

Introduction

Optimizing compilers for traditional programming languages only have a limited ability to optimize the input code. For example, in traditional programming languages typically explicit programming constructs have to be used for multi-core parallelization in order to achieve near-optimal performance. Additionally, concrete data structures have to be defined by the programmer. Also, algorithm implementations tend to introduce a lot of explicit data dependencies which optimizing compilers must respect. Altogether this limits optimizing compilers of traditional languages mostly to small-scale optimizations and prevent such compilers from performing optimizations on the grand-scale.

tUPL is a widely applicable very high-level programming language for computational algorithms aiming to free itself from these artifacts [13]. Within tUPL the two core looping structures, the `forelem` and `whilelem` (pronounce: while—elem) loops, play a primary role in its power. These loops allow iterating through a set of (named) tuples: a *tuplespace* (or *reservoir*). A `forelem` loop iterates all tuples in a tuplespace once in an undefined order and executes the loop body atomically. A `whilelem` loop continues iterating tuples in an undefined order (possibly iterating some tuples more than once) until "nothing happens anymore". These looping structures naturally avoid locking down data dependencies across the entire program: only within a single loop body iterations data dependencies may exist *explicitly*. As no data dependencies are explicitly defined across different loop iterations these looping structures are inherently parallel. Within this thesis we will primarily look at the `forelem` loop.

Within loop bodies we can access *shared spaces* through an *address function*. A shared space is a storage location for data items indexed by any-dimensional integers. Address functions allow converting the tuple that is being iterated to the desired shared space index. For example, when we iterate a tuplespace containing two-dimensional tuples t with fields $\langle a, b \rangle$ we can access a one-dimensional shared space `X` using address function $\lambda\ t\ :\ (t.a,)$, indexing `X` using only the a component of the tuple. Usually we shorten this abstraction to `X[t.a]` for the sake of readability. Note that even though this syntax suggests `X` could be an array, the true data structure for this shared space is undefined: it can be seen as a key-value mapping without defined implementation. Similarly, we do not define the way tuples in a tuplespace are actually stored. Due to these high-level storage containers program specifications are free from

explicit data structures.

Let us consider an example. Listing 1.1 shows a sparse matrix-vector multiplication specification in tUPL. This specification iterates over the tuples in the `NZ` tuplespace (representing nonzeros) in any order and, for each nonzero, performs the multiply-add operation atomically. Here, the `NZ` tuplespace contains two-dimensional tuples with fields *row* and *col*, both integers.

```
1 forelem nz in NZ:
2   C[nz.row] += A[nz.row, nz.col] * B[nz.col]
```

Listing 1.1: Sparse matrix-vector multiplication tUPL specification.

Note how we store the nonzero value in a shared space `A` rather than with the tuples in the tuplespace. Although the alternative is allowed and equivalent, the convention is to store only *indices* in the tuples in reservoirs.

Listing 1.2 shows another example: sorting. This specification continues to swap adjacent elements until no element is out of order anymore. Generally, `whilelem` loops terminate when the program is in some state to which we can always return, no matter what sequence of (executable) tuples are executed. For this sorting specification it holds that when no element is out of order, no tuple is enabled at all, so detecting termination is trivial in this case.

```
1 whilelem adj in ADJS:
2     if A[adj.left] > A[adj.right]:
3         tmp = A[adj.left]
4         A[adj.left] = A[adj.right]
5         A[adj.right] = tmp
```

Listing 1.2: Sorting tUPL specification.

We have developed an optimizing compiler for tUPL, focusing on the automatic generation of efficient data structures. This project consists of two parts: libtupl and Tython. libtupl is a language-agnostic compiler that operates on a tUPL AST in which we perform optimization routines and generate output programs. libtupl only takes an initial AST and some other initialization structures as input. Tython is a front-end we have developed for libtupl which extends Python 3 with tUPL constructs. Tython interfaces with libtupl to then perform the actual optimization routines.

In Section 2 we look at an overview of the entire compiler. Within Section 3 we will look at the heart of the compiler: libtupl. In particular, two strategies for the generation of I/O routines will be discussed. Section 4 will describe at the Tython front-end we have developed in more detail. Section 5 enumerates a few additional extensions to tUPL. These extensions enable the compiler to derive more advanced implementations. In Section 6 we will demonstrate the effectiveness of various derived advanced implementations using a number of experiments on sparse matrix-dense matrix multiplication. Finally, Section 7 will conclude this thesis and propose some avenues for future work.

1.1 Related work

tUPL is an extension of the previously developed forelem framework [10, 11]. A range of the transformations described in this thesis resemble those from the

original forelem framework. In unreleased slides, Prof. dr. H.A.G. Wijshoff described various additions to the forelem framework which formed tUPL, such as the `whilelem` loop. In this thesis we extend the transformations to robustly transform input and output data streams and also introduce a range of new transformations.

Many algorithms have already been specified in tUPL. We have already seen sparse matrix-vector multiplication and sorting in Listings 1.1 and 1.2 respectively. Previously, specifications for maximum flow, finding strongly connected components, triangular solve [13], LU factorization [12], K-means clustering [8, 7], PageRank [15] and more have been constructed. For many of these specifications full implementations have been derived, such as for PageRank [15].

Within the experiments we will consider an extended version of tUPL which takes parallelization and runtime I/O into account. This is then applied on sparse matrix-dense matrix multiplication (SpMM) by deriving various sparse matrix data structures for the algorithm, similar to the experiments in the dissertation of Dr. K.F.D. Rietveld [10]. In this thesis we, however, perform experiments using tUPL, which has evolved significantly since those experiments. Additionally, we consider the automatic transformation of I/O routines so the input and output is transformed to the generated data structures automatically. Although a lot of research and library development has been performed to optimize the performance of SpMM in the case where all data is in-memory, such as in [12, 14, 1, 16], only a limited amount of research has been done in which data is loaded from persistent memory, such as in [17]. In our experiments we consider both cases: data can either be in-memory or has to be loaded from persistent memory.

1.2 Notation

Within this thesis we typically use a Python-like style to describe code, so certain Python-like notations are used. In fact, the syntax is mostly that of Tython, the front-end we have developed for tUPL. For example `a: Generator` indicates that the variable `a` is of type `Generator`. Similarly, `λ() -> Generator` indicates the function returns a `Generator`. We use the shorthand notation `{a: int, b: int}` to describe the type of a *named tuple* in code, here a two-tuple with fields `a` and `b`, both of type `int`. Outside of code we usually omit the type of values and write named tuples as $\langle a, b \rangle$.

Sometimes the function `assert` is used to indicate something is always truthy. This function also returns the object being asserted. Like in Python, empty structures are not truthy.

The unary postfix `++` and `--` operators from, for example, C++ are also used in code for brevity. Outside of C++ code, the unary prefix $*$ operator will unpack a structure. For example, with $a = \langle 1, 2 \rangle$, $\langle *a, 3 \rangle = \langle 1, 2, 3 \rangle$, but $\langle a, 3 \rangle = \langle \langle 1, 2 \rangle, 3 \rangle$. Unpacking a `Subscriptable` implies unpacking it into a `Stream` of key-value pairs (i.e. tuple of length two), as will be described in Section 3.

Note that in previous work involving the forelem framework and tUPL a different notation was used to specify algorithms. To ease into the new notation Listings 1.3 and 1.4 can be compared. The specification using the old

notation in Listing 1.3 is equivalent to the specification with the new notation in Listing 1.4.

```
1 forelem (r; r ∈ NZ.row)
2   forelem (nz; nz ∈ NZ.row[r])
3     C[nz.row] += A[nz.row, nz.col] * B[nz.col];
```

Listing 1.3: A tUPL specification using the old notation.

```
1 forelem r in NZ.row:
2   forelem nz in NZ where nz.row == r:
3     C[nz.row] += A[nz.row, nz.col] * B[nz.col]
```

Listing 1.4: A tUPL specification using the new notation (i.e. Tython syntax).

Chapter 2

Overview

Within this thesis we look at two components: libtupl and Tython. Tython, the front-end, can parse Tython code and generate an AST from it. Tython then converts this Tython AST to an agnostic tUPL AST part of libtupl. Initially, the code represented by the tUPL AST is expressed as operations on tuplereservoirs and shared spaces. In order to import data into these tuplereservoirs and shared spaces, a *load* I/O routine must be initialized. Similarly, for the reverse export operation an *unload* routine is defined. For every transformation that is performed by libtupl, both the algorithm AST as well as the load/unload routines are transformed. This ensures that for every step in the transformation chain input data can be transformed to the libtupl-generated data structures and vice versa.

Figure 2.1 visualizes this structure. Note that each `Function` contains the AST while each `IOGen` contains information on how to load/unload the data structures for that particular instantiation of the function. Each (parameterizable) optimization `Pass` can modify the `Function` and `IOGen` objects, forming a new node in the transformation tree. Tython (or another front-end) initializes the root `TransformationTreeNode`. In the end, *code generation* is performed on some final `TransformationTreeNode`. This will yield, for example, C++ code that can import (load) data, run the algorithm and export (unload) data.

The majority of the logic is part of the libtupl library, including a range of passes that can be applied on the transformation tree nodes, as we will see in Section 3.3. We primarily look at data structure transformations, not considering algorithmic transformations in great detail. Two different ways of handling I/O data transformations have been developed, each with certain advantages and disadvantages. Firstly, the I/O generation approach through combinations of simple *generators* transforming the data will be described in detail in Section 3.4. Secondly, the I/O generation approach by constructing *transformation graphs* of the data is described in Section 3.5. The definition of the `IOGen` objects and the way these objects are modified during each transformation `Pass` differs significantly between the two approaches as we will see in these two sections.

In Section 4 we will take a more detailed look at how Tython parses input code and interacts with libtupl to perform the compilation process.

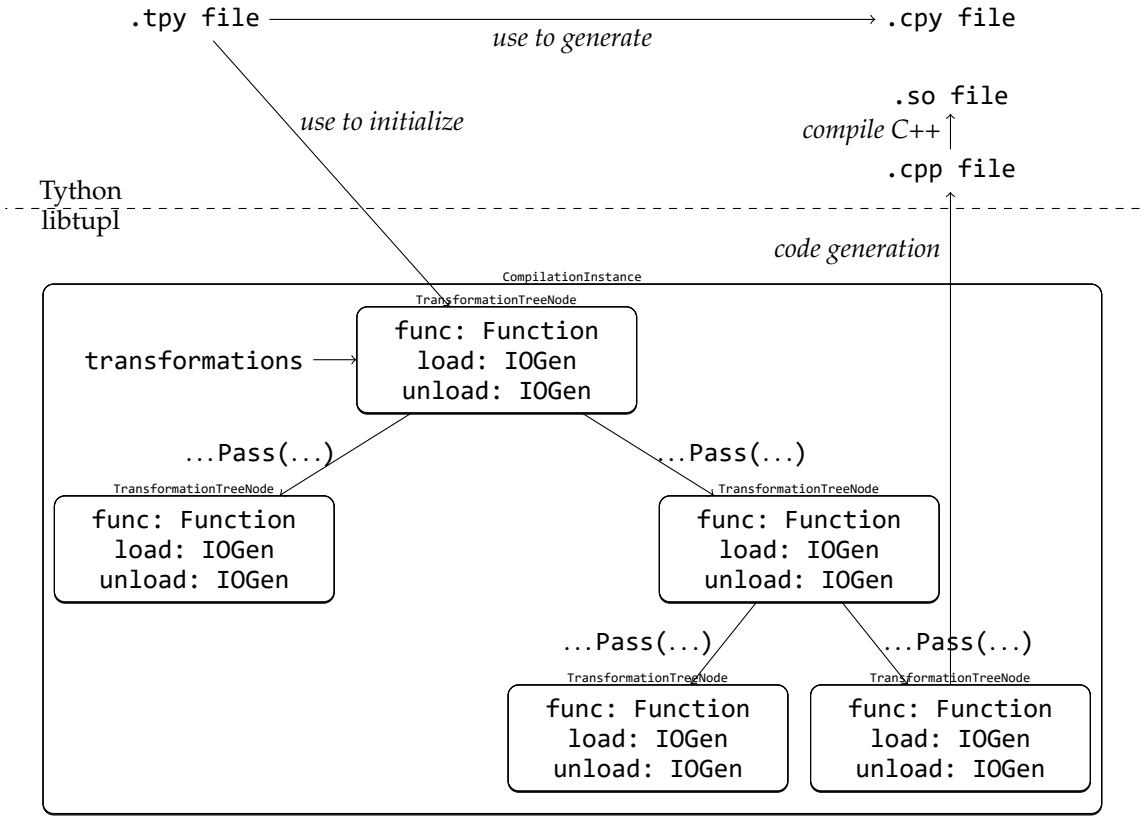


Figure 2.1: Overview of the compilation process.

Chapter 3

libtupl

In this chapter we will describe libtupl, a library to compile and optimize tUPL programs. libtupl does not include features to parse code, instead it operates on a more agnostic tUPL abstract syntax tree (AST), which allows any front-end to use this library. The frond-end we have additionally developed, Tython, is discussed in Section 4.

3.1 Compilation process

The compilation process can be divided into two phases: algorithmic optimization and data structure optimization. The algorithmic optimization process transforms the algorithm so that generally fewer operations are necessary to execute the algorithm. For example, the sparse matrix-vector multiplication specified in Listing 1.1 can be optimized to the specification in Listing 3.1. The latter specification iterates the tuples in tuplespace `NZ` in a row-wise fashion, iterating all tuples in a row at once. Compared to the first specification, which does not group the tuples in any way at all, a large amount of unnecessary reads and writes to shared space `C` can be saved. From this point on we will assume input specifications have already been algorithmically optimized unless mentioned otherwise and only focus the data structure optimization phase.

```
1 forelem row in NZ.row:
2   forelem nz in NZ where nz.row == row:
3     C[nz.row] += A[nz.row, nz.col] * B[nz.col]
```

Listing 3.1: A possible sparse matrix-vector multiplication tUPL specification after algorithmic optimization.

Data structure optimization consists of a number of simple transformations that are performed on the output of the algorithmic optimization phase. Note that because the output of the algorithmic optimization phase defines the starting point of the data structure optimization phase, it has a significant effect on the data structures that will be generated: the generated data structures will be tailored to the specification of the algorithm on which algorithm transformations have already been applied. Data structure optimization can significantly change the data structures further from this starting point. Algorithmic and

data structure optimization together can thus lead to a wide range of different data structures. Within the next few sections we will look at data structure transformations and two techniques to efficiently transform the input and output routines to match the target data structures. The vast majority of the data structure optimization process has been implemented as part of the *Tython* project as the C++ libtupl library.

3.2 Transformation tree

During the optimization process we construct a *transformation tree*. Each node in this tree is an algorithm specification, consisting of an AST, I/O information and some additional metadata. Each edge represents a transformation applied on a certain node, where the node at the arrowhead end is the result of the transformation. A transformation may be *parameterized*, indicating that we, for example, want a certain transformation to have an effect on a specific symbol only. Figure 2.1 visualizes this structure.

Technically we only have to store the initial AST, I/O information and metadata in the root node of the transformation tree. All transformations are deterministic, so we can derive child nodes directly from the root node and the sequence of transformations applied on this root node.

3.3 Transformations

A wide range of simple transformations have been implemented as *passes* to automatically transform the algorithm. Within this section we will describe these transformations and the effect they have on the algorithm. Within Sections 3.4 and 3.5 we will look at the effects these transformation passes have on the input and output data streams, as these have to be transformed to be able to store the actual data in the generated data structures.

Table 3.1 lists the implemented transformations. Some of these transformations are based on the forelem framework [11]. Appendix A can be referred to for a more detailed description of each of these passes.

The transformations can transform the initially *unmaterialized* specification (i.e. iteration order of the loops is undefined) to a *materialized* specification, where we fix the order we iterate through reservoir tuples. In materialized specifications all `SharedSpace` and `Reservoir` symbols have been transformed into `Subscriptable` symbols. `Subscriptable` symbols assign indices to each element, but do not enforce how these elements are stored. The ReservoirMaterializationPass, NStarMaterializationPass and SharedSpaceMaterializationPass passes can be used to materialize a specification. The *materialized* specification becomes a *concretized* specification after the final ConcretizationPass. Here all `Subscriptable` symbols have defined implementations, such as it becoming a multi-dimensional array or jagged linked list. In concretized specifications loops are also limited to just simple `for`- and `while`-loops. Note that various other transformations exist that operate on unmaterialized or materialized specifications that solely change the data structure in some way, leading to different outputs.

To illustrate the application of a transformation, consider the following ex-

Transformation name	Description	See also
EncapsulationPass	Transforms iterating a tuplespace its possible values (`forelem row in NZ.row`) to iterating a range (`forelem row in [0, max(NZ.row)]`) when possible.	A.1, [11]
AggregateReservoirPass	Transforms aggregations over a tuplespace its possible field values (`forelem row in [0, max(NZ.row)]`) to a scalar value (`forelem row in [0, max_NZ_row)]`), delegating the computation of that scalar to load time.	A.2
LocalizationPass	Merges (localizes) the values of a shared space into a reservoir so that indexing the original shared space is no longer necessary.	A.3, [11]
QueryForwardSubstitutionPass	When a loop has an equals-query, like in `forelem nz in NZ where nz.row == row`, substitutes `nz.row` for `row` in the loop body.	A.4
ReservoirMaterializationPass	Materializes a reservoir, constructing a subscriptable to store the reservoir its tuple data into. In the case of `forelem nz in NZ where nz.row == row` the materialization leads to a two-dimensional `Subscriptable PNZ` with indices `row` (the query) and `k` (an offset). The loop is substituted by `forelem k in N*`, iterating over all offsets.	A.5, [11]
NStarMaterializationPass	Materializes an `N* Reservoir`. If the original `forelem` loop has a query, this can lead to, for example, a `Subscriptable PNZ_len`, in which the number of tuples are stored matching those query values. `PNZ` then contains tuple data on indices $[query\_value, 0 \ldots PNZ\_len[query\_value] - 1]$.	A.6, [11]
SharedSpaceMaterializationPass	Materializes a shared space, converting it into a subscriptable.	A.7, [11]
HorizontalIterationSpaceReductionPass	Removes fields from a subscriptable containing tuples if that field is never used anywhere, reducing the width of each element in the subscriptable.	A.8, [11]
DelocalizationPass	Duplicates a subscriptable containing tuples into two subscriptables. The specification is modified to access the newly duplicated subscriptable to access certain fields. Usually followed up by a HorizontalIterationSpaceReductionPass to shrink both subscriptables so that they contain mutually exclusive fields.	A.9
StructureJaggedSplittingPass	Splits a Subscriptable containing tuples into a jagged Subscriptable. For example, can transform `Z[x, y].a += Z[x, y].b * Z[x, y].c` into `Z[x]._a[y].a += Z[x]._b_c[z].b * Z[x]._b_c[z].c`. Note how `Z` is now an array of structures containing two arrays each (i.e. a jagged structure).	A.10, [11][1]
ConcretizationPass	Concretizes each subscriptable to an actual data structure (usually an array of potentially multiple index dimensions). `forelem` loops are concretized to, for example, simple `for`-loops.	A.11, [11]

[1] Regular structure splitting only.

Table 3.1: All implemented transformations that have an effect on the algorithm specification.

ample application of the LocalizationPass. Listing 3.2 shows a scenario on which the LocalizationPass can be applied. We can decide to merge, for example, the shared space `A` into tuplespace `NZ`. This will insert a new field `merged_value` for each tuple `nz` inside of `NZ` with value `nz.merged_value = A[nz.row, nz.col]`. In order to access the value `A[nz.row, nz.col]` we can now simply read it from `nz.merged_value` instead, yielding the code in Listing 3.3.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem nz in NZ where nz.row == row:
3     C[nz.row] += A[nz.row, nz.col] * B[nz.col]
```

Listing 3.2: Example scenario on which the LocalizationPass can be applied.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem nz in NZ where nz.row == row:
3     C[nz.row] += nz.merged_value * B[nz.col]
```

Listing 3.3: Resulting code after applying the LocalizationPass.

Do note that it is not always safe to substitute indexing a shared space by such a merged value, as described in Appendix A.3. Other transformation passes also perform minor changes to the algorithm. Refer to Appendix A for full descriptions of each pass.

3.4 I/O generation through generators

One of the implemented techniques to generate data load and unload routines operating on *streams* of data (i.e. a lazily generated sequence of named tuples) is through creating a generator coroutine for each (potentially virtual) data structure each time a data structure is transformed during a transformation pass. Each generator takes one or more input generators (in the data format from the previous data structure), transforms these streams of elements and produces a new stream of elements (in the desired data format). Each generator is typically very simple, but connected together complex data structures can be loaded and unloaded.

Listing 3.4 shows an example generator that changes a subscriptable from containing tuples $\langle row, col \rangle$ to containing tuples $\langle col \rangle$. It does not modify the two-dimensional key of each value in the subscriptable. We do not yet store the elements, but just change the data that is to be stored later in an actual data structure and **`yield`** the resulting elements. Generators like this are typically produced by a HorizontalIterationSpaceReductionPass, changing a **`Subscriptable`** containing tuples $\langle row, col \rangle$ so that the *row* field is no longer stored if it is not accessed anywhere.

```
λ(input: Generator[Tuple[int, int], NTuple[row: int, col: int]]) -> Generator[Tuple[int, int], NTuple[col: int]]:
    while input:
        key, value = input.next()
        yield key, (value.col,)
```

Listing 3.4: A generator that reduces the width of the tuples in a subscriptable.

3.4.1 Transformations which modify data structures

Certain transformations change the data structure, such as the HorizontalIterationSpaceReductionPass. Data has to be transformed as well to make it fit the new structure. Some transformations thus produce generators that take the original data and generate transformed data that fits these new structures. Table 3.2 lists various transformations passes and the data transformations they perform. Note that these operations are an extension of the pass behavior described in Table 3.1. Detailed descriptions of the generators that are produced during these passes can be found in Appendix B.

Transformation name	Description	See also
HorizontalIterationSpaceReductionPass	Takes key-value pairs from a subscriptable, removes some fields from the tuple, outputs new key-value pairs with an unmodified key and width-reduced value. Listing 3.4 shows a generator that could be produced by this pass.	B.1
LocalizationPass	Reads tuples from a tuplespace, produces new tuples for the tuplespace with a new field (`merged_value`) for each tuple. The `merged_value` is found in a shared space depending on the input *query* (see Appendix A.3).	B.2
ReservoirMaterializationPass + NStarMaterializationPass	Two generators are typically produced for a reservoir `R`: one for the materialized data `PR`, assigning indices to each value, and one for the data in `PR_len`.	B.3
DelocalizationPass	Duplicates the generator for the original subscriptable to be used for the duplicated subscriptable.	B.4
ConcretizationPass	Takes the final generator producing key-value pairs and writes the output to an actual concrete data structure.	B.5

Table 3.2: Generators produced by various transformations. Descriptions here *extend* the behaviour of the original transformation passes described in Table 3.1.

Note that using coroutines to abstract the I/O will result in the chain of generators being controlled from the *bottom* (i.e. the end of the chain). Each generator then invokes the upper generator to produce a new element (through the `next()` call). The bottom generators here typically contain routines that construct the final data structure from the input and load concrete data in there, while the top generators typically read raw data elements from, for example,

the file system (or the other way around when unloading data from the generated data structures).

This can cause performance issues when more than one concretized data structure reads from a single source: that source is then read multiple times. For example, applying the ReservoirMaterializationPass and NStarMaterializationPass on some reservoir `A` can lead to the materialized reservoir `PA` (i.e. transforming the specification in Listing 3.5 into Listing 3.6), generated by the generator in Listing 3.7, and additionally a lookup subscriptable `PA_len`, generated by the generator in Listing 3.8, containing the number of entries matching each query. Both of these data structures are based on the contents in `A`, which results in the input being read twice in order to construct the data in both structures. See also Appendix B.3 for additional details about this case. Similar issues occur when performing the DelocalizationPass, as described in Appendix B.4.

```
1 forelem a in A.a:
2     forelem tuple in A where tuple.a == a: # A:
    Reservoir[a: int, b: int]
3         ... tuple ...
```

Listing 3.5: Some tUPL specification with tuplespace `A` before materialization.

```
1 forelem a in A.a:
2     forelem k in PA_len[a]:
3         ... PA[a, k] ...
```

Listing 3.6: Some tUPL specification with subscriptable `PA` after materialization.

```
1 λ(tr: Generator[NTuple[a: int, b: int]]) -> Generator[
    Tuple[int, int], NTuple[a: int, b: int]]:
2     counts = {} # some sort of lookup table, default value of
    0 if key does not yet exist
3     while tr:
4         tuple = tr.next()
5         query = (tuple.a,)
6         yield (*query, counts[query]), (*tuple, counts[
    query])
7         counts[query] += 1
```

Listing 3.7: A generator materializing a tuplereservoir.

```
1 λ(tr: Generator[NTuple[a: int, b: int]]) -> Generator[
    Tuple[int, int], NTuple[a: int, b: int]]:
2     counts = {} # some sort of lookup table, default value of
    0 if key does not yet exist
3     while tr:
4         tuple = tr.next()
5         query = (tuple.a,)
6         counts[query] += 1
7
8     for query, count in counts:
```

```
9           yield query, count
```

Listing 3.8: A generator materializing a `N*` reservoir into a `_len` subscriptable.

3.4.2 Initialization

Initial generators must be defined that stream the input data. For tuplespaces this is always a stream of tuples, for shared spaces this streams both the key and value of the contents of that shared space. Callback functions can be defined to export shared space data after running the algorithm, taking the key and value of each element in the shared space as function parameters. This callback function is invoked for each element in the shared space. A chain of generators can transform the data back to the desired output format before invoking the callback function for each data element.

For example, Listing 3.9 defines a coroutine generator in C++ which produces tuples $\langle row, col \rangle$ from a given input file in text format, which could be used to define where nonzero elements are stored in a sparse matrix. Generators like this are used to initialize the data structures from external sources.

```
1  struct tuple_row_col {
2      uint64_t row;
3      uint64_t col;
4  };
5
6  Stream<tuple_row_col> loader(std::ifstream text_stream) {
7      tuple_row_col nonzero_location;
8      while (text_stream >> nonzero_location.row &&
       text_stream >> nonzero_location.col) {
9          co_yield nonzero_location;
10     }
11 }
```

Listing 3.9: Coroutine to load nonzero position data for shared space `NZ`.

3.5 I/O generation through transformation graphs

Although generators are highly flexible to abstractly represent the data transformations applied by each pass, they have some limitations. Generators naturally have their control at the *end* of the generator chain. In other words, the generator produced at the ConcretizationPass will invoke parent generators to produce more data. Because of this certain things cannot easily be described efficiently using such a setup. For example, a DelocalizationPass wants to practically duplicate a data structure, which then requires reading the input twice (but typically processing this input slightly differently for the different concretized data structures).

To counter these problems we have developed a second implementation of I/O load/unload routine generation. This implementation stores the data transformations in *I/O transformation graphs*. Each (parameterizable) node in this graph represents a certain data transformation on the inputs. Directional

edges connect output sockets of one node to input sockets of others. A node may have zero or more input sockets and zero or more output sockets. It is permitted to connect multiple edges to a single output socket, in which case the data is duplicated. Sockets can be tagged by any amount of *expressions*, relating the data passing through a socket to the data structures used in the code. The exact way this I/O transformation graph is converted into imperative code is flexible, but here we primarily use a *top-down* compilation, unlike the generator approach, which is basically limited to a *bottom-up* compilation. Thus the control with this method lies at the topmost data generators, like a node where raw data is being read from a file. This is then *pushed* to other I/O nodes as data is being read.

Let us consider a simple example. Listing 3.10 shows the initial sparse matrix-vector multiplication specification without additional algorithmic optimizations applied on it. We can materialize shared space `A` to subscriptable `PA` using the SharedSpaceMaterializationPass and then concretize it to a flat 2D array `CPA` using the ConcretizationPass.

```
1 forelem nz in NZ:
2   C[nz.row] += A[nz.row, nz.col] * B[nz.col]
```

Listing 3.10: Initial sparse matrix-vector multiplication specification.

Figure 3.1 displays the simplified resulting input transformation graph, not displaying transformations on data structures other than `A`[1]. I/O graphs consist of a few concepts. Primarily it consists of I/O nodes, which typically define some sort of operation on data. I/O nodes have any number of input and output sockets (depending on the type of the I/O node). Output sockets can connect to input sockets of other nodes as long as no cycle forms. Connections between sockets are always annotated with the type of data sent. For example, between the "I/O reader" and "Tuple to KeyValue" node we send named tuples $\langle row, col, val \rangle$, and from "Tuple to KeyValue" to "Concretize to FlatArray" we send both a two-dimensional unnamed tuple (the index), denoted by $[2D]$, and named tuples $\langle row, col, val \rangle$. Note that when we send these two items they are always sent together: it can practically be seen as a two-tuple containing another two-tuple and the named tuple. We usually annotate each input socket with symbols (local to the I/O node) to which we assign incoming data. The output socket is typically annotated with a rough description of the transformation the I/O node performs. Finally we allow sockets to be *tagged* by expressions used in the algorithm, defining what data structures are represented by data going through that socket. In most cases these expressions are just a single symbol. For example, the data in `A`, `PA` and `CPA` is represented by the output of the "Tuple to KeyValue" node.

Note that in the example input transformation graph of Figure 3.1 the application of the SharedSpaceMaterializationPass to `A` did not transform the data, but did tag the "Tuple to KeyValue" node to indicate `PA` is also represented by the same output. The application of the ConcretizationPass did not transform the data as well. The I/O graph was initialized with just the "I/O reader" and "Tuple to KeyValue" nodes, the "Concretize to FlatArray" node is produced by

[1] Figure 3.1 is simplified. Concretization does not produce just a single I/O node, as we will see later in this section. The example illustrates the structure of I/O graphs well, though.

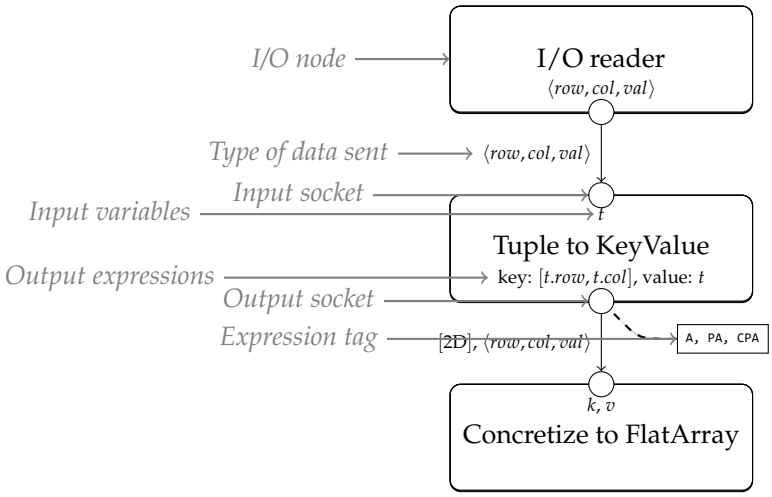


Figure 3.1: Simplified partial input transformation graph, for transforming a shared space `A` to a concretized subscriptable `CPA`, displaying nodes affecting shared space `A` only.

the ConcretizationPass. This node will actually write the data to the generated data structure.

These I/O transformation graphs specify the data transformations at a higher level than the coroutines in the previous section did, allowing us to perform transformations on this transformation graph more easily. Unlike the generator approach where we generate imperative code immediately, we only store high-level logical operations in the transformation tree describing what each node does. Only in the end we generate imperative code for each I/O node that actually performs the data transformations, unlike the approach in the previous sections where the produced coroutines are practically directly executable. It is, for example, possible to convert an I/O transformation graph back to a range of connected generators as described in the previous section, although in Section 3.5.1 we will see that a top-down compilation approach avoids certain problems a bottom-up compilation yields.

Table 3.3 enumerates all implemented I/O nodes. They are described in more detail in Appendix C.

3.5.1 Transformations which modify data structures

Table 3.4 summarizes the effect the transformation passes have on the I/O transformation graphs. Appendix C can be referred to for detailed descriptions and examples.

In the previous section we have seen that applying the ReservoirMaterializationPass and NStarMaterializationPass on `A` will yield two data structures `PA'` and `PA_len`. Both data structures have their own generator that can load its data, but both generators have to read the source input (i.e. `A` its data source), leading to the input being read twice.

This is no longer necessarily an issue when using transformation graphs to store transformations on the input data in. Figure 3.2 shows a typical graph

Node name	Input	Output	Description	Conv.
ConstantStream		`Tuple`, `NTuple`	For a range of keys outputs a constant tuple (see also Section 3.5.2).	D.4
DataStreamReader		`NTuple`	Imports data using an external data generator (see also Section 3.5.2).	D.3
DataStreamWriter	`NTuple`		Exports data using an external callback function (see also Section 3.5.2).	D.10
Aggregate	`NTuple`	`Tuple`, `NTuple`	Aggregates the value in a certain field of all input tuples depending on the aggregate function. Outputs single singleton tuple.	D.5
KeyValue to Tuple	`Tuple`, `any`	`NTuple`	Transforms input key-value pairs to a tuple depending on the configured expression.	Like D.1
Transform KeyValue	`Tuple`, `any`	`Tuple`, `any`	Transforms input key-value pairs depending on the configured expressions.	D.2
Tuple to KeyValue	`NTuple`	`Tuple`, `any`	Transforms input tuple to a key-value pair depending on the configured expressions.	Like D.2
Transform Tuple	`NTuple`	`Tuple`, `any`	Transforms input tuple depending on the configured expression.	D.1
Count Tuples	`NTuple`	`NTuple` & `Tuple`, `NTuple`	Assigns a number to each input tuple, extending the tuple with a new field containing that number. Tuples that match the same query have distinct numbers. Also outputs key-value pairs containing the total number of tuples in each query group.	D.6
Write Value	`Tuple`, `any`		Writes data to a concretized `Subscriptable`.	D.8
Deconcretize		`Tuple`, `any`	Streams data out of a concretized `Subscriptable`.	D.9
Jag	`Tuple`, `NTuple`	`Tuple`, `NTuple`	Reduces the width of the input key by a certain *offset*, creating a jagged structure (see also Section C.6).	D.7
Merge	`NTuple` & `Tuple`, `any`	`NTuple`	Extend each tuple in the first input with a new field whose value is looked up by key in the second input depending on the configured *query*.	—

Table 3.3: All implemented I/O nodes. The last column references to Appendix D where conversion routines for each I/O node are described, see also Section 3.5.3.

Transformation name	Description	See also
HorizontalIterationSpaceReductionPass	Creates a "Transform KeyValue" node modifying the tuple to the reduced tuple.	C.1
AggregateReservoirPass	Creates an "Aggregate" node aggregating a certain field of input tuples with the specified function.	C.2
LocalizationPass	Creates a "Merge" node with both the reservoir and shared space as input. The node is parameterized by the *query* describing where value to be merged is located in the shared space (see Appendix A.3).	C.3
ReservoirMaterializationPass + NStarMaterializationPass	Creates a "Count Tuples" node, assigning a unique index to each tuple for each query group, in the end forming `PR` from input reservoir `R`. Also outputs key-value pairs containing the number of tuples matching each query group for the `PR_len` subscriptable.	C.4
DelocalizationPass	When delocalizing `PR`, a duplicate `PR_deloc` is created. Whatever I/O node outputs the data of `PR` (i.e. is tagged with expression `PR`) is then also tagged with `PR_deloc`.	C.5
StructureJaggedSplittingPass	Creates a "Jag" node which converts the subscriptable to a jagged subscriptable. "Transform KeyValue" nodes are linked to that to perform the structure splitting.	C.6
MergeEliminationPass	This is a new pass that only has an effect on the I/O transformation graphs. It tries to eliminate "Merge" I/O nodes and replace them with a "Transform Tuple" I/O node, computing the desired output directly from some shared ascendant I/O node.	C.7
ConcretizationPass	Creates a "Write Value" node that actually writes resulting values to a concretized data structure.	C.8

Table 3.4: Generators produced by various transformations. Descriptions here *extend* the behaviour of the original transformation passes described in Table 3.1.

structure after the ReservoirMaterializationPass and NStarMaterializationPass have been applied on `A` (recall that $*Z$ will yield a stream of the key-value pairs in Z, i.e. the number of tuples that match each query). In this figure, "Any I/O node" indicates the node that generates the data of `A` can be any node: regardless of it the same new nodes are attached to its output socket. Note how a "Count Tuples" node has two outputs: one will represent `PA'` its data and one will represent `PA_len` its data (see also Section C.4). This node can be compiled top-down, avoiding the need to read the input twice and instead producing data for both connected I/O nodes at the same time.

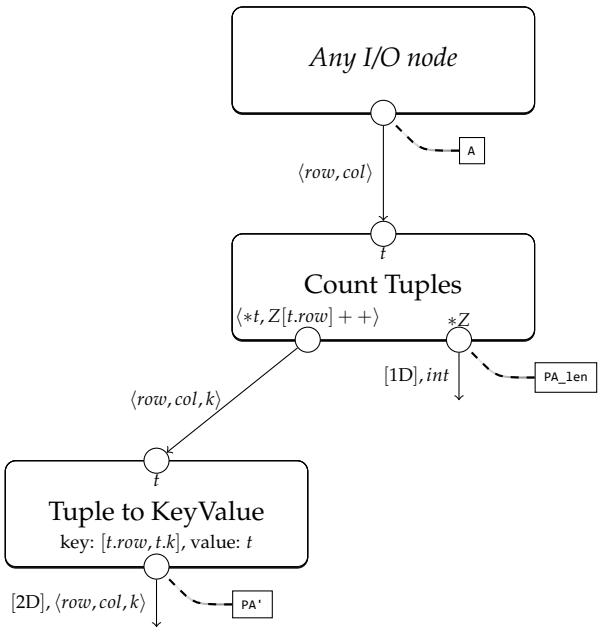


Figure 3.2: Example I/O transformation applied after performing a ReservoirMaterializationPass on `A` and the NStarMaterializationPass on `N*`.

The ConcretizationPass is another interesting case. This pass will introduce two nodes in the input transformation graph, as illustrated in Figure 3.3. The "Jag" node is used to ensure space exists (and potentially allocate it) in the `CPA` data structure so that `CPA[k[0], k[1]]` is writable. The "Write Value" node is then used to *write* the actual value to this allocated slot. Note that the "Jag" node is tagged with `CPA[_, _]`: this indicates that the output of the "Jag" node is local to `CPA[_, _]`, i.e. all keys of `CPA` are locked down. The "Write Value" node will thus only receive a value with a zero-dimensional key and writes this to the locked down index `CPA[k[0], k[1]]`. The "Jag" node can also be used to lock down a limited number of dimensions only using the StructureJaggedSplittingPass, creating a jagged data structure, as described in Section C.6.

3.5.2 Initialization

In order to initialize the I/O transformation graphs, initial graph nodes have to be created. These nodes have to be tagged with the tuplespace or shared

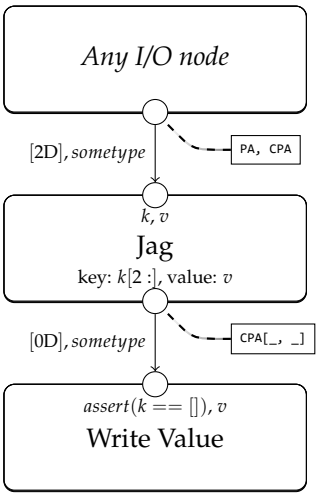


Figure 3.3: Example input transformation applied after performing a ConcretizationPass on PA.

space represented by a node. To provide input data, the I/O nodes "DataStreamReader" and "ConstantStream" can be used. The "DataStreamReader" takes an (external) generator coroutine as input, which can be used to, for example, read data from a file. The "ConstantStream" simply generates tuples with constant values $(v_0, v_1, \ldots)$ on each shared space storage location $(0 \ldots x_0, 0 \ldots x_1, \ldots, 0 \ldots x_{n-1})$, where x, n and v are parametrizable. It is mostly used to, for example, zero-initialize an output shared space.

The I/O node "DataStreamWriter" can be used to export any data after the algorithm has been executed. It takes an (external) callback function that is invoked for each tuple sent to this I/O node. This is mostly used to export the data of a shared space. Note that in many cases it is easy to inline this callback.

Let us consider an example I/O initialization for the sparse matrix-vector multiplication. We use the standard sparse matrix-vector multiplication specification as shown in Listing 3.11. The host language here is C++. Let us assume that A is an $N \times N$ sparse matrix and we start indexing from 0. So B and C both contain N elements. C is zero-initialized while B is initialized using doubles from 1 to n. The data for shared space A and tuplespace NZ is read from a simple COO-like file storing triplets. We thus have to split off the value into shared space A and only store the coordinates in tuplespace NZ.

```
1 forelem nz in NZ:
2     C[nz.row] += A[nz.row, nz.col] * B[nz.col]
```

Listing 3.11: Sparse matrix-vector multiplication specification in tUPL.

We can use a "DataStreamReader" to load data for A, NZ and B (where A and NZ share the same "DataStreamReader"). C can simply be initialized using a "ConstantStream". Listing 3.12 shows a possible coroutine that generates triplets of data from a text input file containing triplets of numbers. Listing 3.13 shows a possible generator for the integers 0 to $count - 1$, each put in a singleton tuple.

```
struct tuple_row_col_val {
    uint64_t row;
    uint64_t col;
    double val;
};

Stream<tuple_row_col_val> coo_loader(std::ifstream
   coo_file) {
    tuple_row_col_val triplet;
    while (coo_file >> triplet.row && coo_file >> triplet
    .col && coo_file >> triplet.val) {
        co_yield triplet;
    }
}
```

Listing 3.12: Coroutine to load data sparse matrix triplet data for `A` and `NZ`.

```
struct tuple_n {
    uint64_t n;
};

Stream<tuple_n> coo_loader(size_t count) {
    for (int n = 0; n < count; ++n) {
        co_yield {n};
    }
}
```

Listing 3.13: Coroutine to load vector index data for `B`.

We can use this to construct the input graph as shown in Figure 3.4. The upper "DataStreamReader" streams data produced by the coroutine from Listing 3.12 and the lower one the data produced by the coroutine in Listing 3.13. The unconnected sockets remain unconnected: transformations on the input specification will result in automatically generated I/O nodes connecting to these sockets.

Note that in this case we can also connect another "Tuple to KeyValue" to the lower "DataStreamReader" to zero-initialize `C`, instead of using the "ConstantStream" utility. Figure 3.5 shows this alternative zero-initialization approach for `C`.

For the output graph we want to export the data in `C` to, for example, the standard output. Before we can use the "DataStreamWriter" we have to specify an (external) callback routine that is invoked for each tuple of data sent to this writer. The input of this node will be a two-tuple containing the 1D location and the value at that location in `C`, so this callback function needs to take this as input. Listing 3.14 shows a possible callback function.

```
struct tuple_n_val {
    uint64_t n;
    double val;
};

```

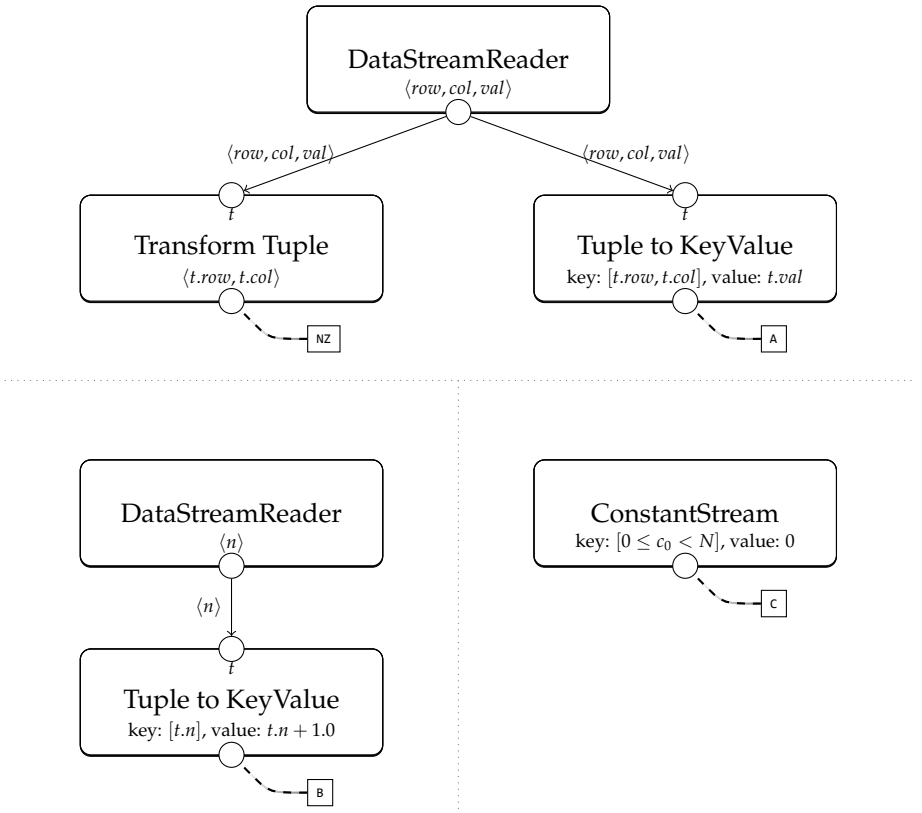


Figure 3.4: Initial input graph for tuplespace `NZ` and shared spaces `A`, `B` and `C` for the sparse matrix-vector multiplication specification.

```
6  void coo_writer(tuple_n_val c_value) {
7      printf("At␣%ju:␣%f.\n", c_value.n, c_value.val);
8  }
```

Listing 3.14: Callback function that dumps the contents of `C` to the standard output.

Figure 3.6 shows a possible initial output graph, using the callback function in Listing 3.14 for the "DataStreamWriter". Note how the "KeyValue to Tuple" node converts the input key and value into this node (representing each storage location and value in `C`) to a two-tuple, without explicit key, that can be used as input to the "DataStreamWriter".

3.5.3 Generating imperative code for transformation graphs

Before we can use the code generator to generate the output of the compilation process (Section 3.6), the input and output transformation graphs are converted to a tUPL AST without tUPL-specific constructs: each transformation graph then becomes a single function (`load` and `unload`). Each I/O node has its own conversion routine and will decide on its own how to invoke child

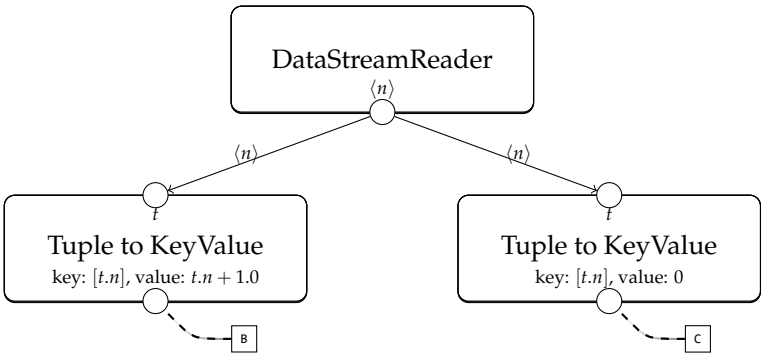


Figure 3.5: Alternative initial input graph (only displaying the initialization of `B` and `C`).

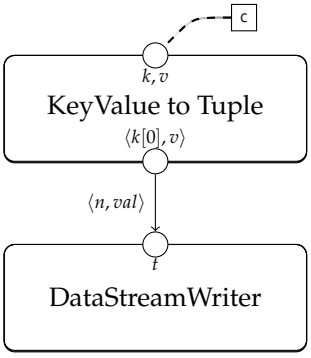


Figure 3.6: Initial output graph.

node conversion routines. After converting the two graphs the standard code generators can be used to convert the language-agnostic tUPL code to a target language.

Generally the implementation of these conversion routines is trivial. Each conversion routine takes a number of symbols as input in which the data is stored of the parent I/O nodes. The routine then creates *new* symbols containing the data that will be sent to the child I/O nodes. Typically these nodes are always compiled *top-down*. We start with the top (root) nodes that have no input sockets. These nodes will push data downwards, applying some transformation at each node. This is unlike the previous generator approach, where data would always need to be pulled from a parent generator. This tends to produce highly linear code. We rely on lower level compilers to optimize this code further by performing, for example, forward substitution and dead code elimination.

Details about how each I/O node is converted and examples can be found in Appendix D. Table 3.3 enumerates all implemented I/O nodes. The last column in this table refers to subsections in Appendix D containing details about its conversion routine.

3.6 Code generation

Once the ConcretizationPass has been performed, a *code generator* can be used to convert the output-agnostic transformation tree node containing the concretized algorithm specification and load/unload objects to some target language (see also Figure 2.1). Such code generators can be used to generate, for example, C++ output. This output can be compiled to an executable or library using, for example, *clang*.

After the various transformation passes have been applied the algorithm will only consist of basic constructs, such as `while` and `for` statements; `whilelem` and `forelem` statements have been eliminated. Additionally, all symbols referenced are now of simple types: concretized subscriptables or primitive scalars. This allows code generators to generate code in a straight-forward way as typical target languages support similar constructs too. Implementations of the low-level concretized subscriptables are part of the *libtupl runtime*, which allows target languages to manipulate such concretized subscriptables through a simple API.

When using the I/O approach with generators various coroutines can be produced. While it is possible to *flatten* these coroutines to low-level imperative code, many target languages support coroutines themselves too (and their compilers flatten simple coroutines as well). We thus generally do not eliminate coroutines, but let the lower level compilers optimize them away as they please instead.

The I/O approach using transformation graphs will output load and unload functions with basic constructs once the graphs have been transformed into imperative code as described in Section 3.5.3. It is thus also trivial to transpile this to a target language.

3.7 Sparse matrix-vector multiplication example

In the previous sections the examples were all primarily from sparse matrix-vector multiplication. In this section we will look at the complete output of a possible compilation of this algorithm using the transformation graph based I/O approach (which we generally prefer over the generator based approach). Initially the specification typically looks like the one shown in Listing 3.15. Let us assume the input transformation graph is initialized as in Figure 3.4 and the output graph as in Figure 3.6. After a simple algorithmic optimization process, transforming the specification such that the tuples being iterated row-by-row, the code in Listing 3.16 could be the output. The exact process behind algorithmic optimization is out of the scope of this thesis.

```
1 forelem nz in NZ:
2   C[nz.row] += A[nz.row, nz.col] * B[nz.col]
```

Listing 3.15: Example initial sparse matrix-vector multiplication example.

```
1 forelem row in NZ.row:
2   forelem nz in NZ where nz.row == row:
```

```
3       C[nz.row] += A[nz.row, nz.col] * B[nz.col]
```

Listing 3.16: Example initial sparse matrix-vector multiplication example after algorithmic optimization.

After algorithmic optimization, data structure optimization follows. In this example we will work towards a jagged data structure (i.e. a data structure whose elements are subscriptable data structures): an array indexed by row at the top level. Then, for each row two arrays are stored: one in which the column indices are stored and one in which the value is stored for each nonzero. Key advantage of such a structure over a non-jagged 2D array is that we do not have to pad each row to the maximum amount of nonzeros that occurs in a row anymore, potentially saving a significant amount of memory. Furthermore, splitting the column and nonzero values into two separate arrays could allow for better vectorization.

Let us first perform the EncapsulationPass on the code in Listing 3.16, followed by the LocalizationPass, localizing shared space `NZ` into tuplereservoir `A` on $[row, col]$, forming `NZ_merge_A`. Then, let us perform the AggregateReservoirPass on this `NZ_merge_A`, aggregating values for field *row* for function max. At this point the specification will look like the one shown in Listing 3.17, while the input transformation graph will be changed to Figure 3.7. The output transformation graph does not change during these first few transformations.

```
1 forelem row in [0, aggr_NZ_merge_A_max_row]:
2   forelem nz in NZ_merge_A where nz.row == row:
3     C[nz.row] += nz.merged_val * B[nz.col]
```

Listing 3.17: Algorithm specification after performing three transformations.

Let us now continue with the materialization of the data structures. First we perform the QueryForwardSubstitutionPass to avoid reading `t.row` from the tuples `t`: this value is always equal to `row` in the inner loop. After that we continue with the ReservoirMaterializationPass of `NZ_merge_A` into `PNZ_merge_A` and the NStarMaterializationPass to fully materialize the `NZ_merge_A` tuplespace into `PNZ_merge_A'` and `PNZ_merge_A_len`. Finally we perform the SharedSpaceMaterializationPass twice, once for `B` (into `PB`) and once for `C` (into `PC`). These passes will transform the algorithm to the code shown in Listing 3.18. The input transformation graph will become the one visualized in Figure 3.8. The only change to the output transformation graph of Figure 3.6 is that we also tag the "KeyValue to Tuple" node with `PC`. No additional I/O nodes are introduced in this output transformation graph.

```
1 forelem row in [0, aggr_NZ_merge_A_max_row]:
2   forelem k in [0, PNZ_merge_A_len[row]-1]:
3     PC[row] += PNZ_merge_A'[row, k].merged_val * PB[
    PNZ_merge_A'[row, k].col]
```

Listing 3.18: Algorithm specification after performing five additional transformations.

Before concretizing everything we first perform the HorizontalIterationSpaceReductionPass to remove the unused *row* and *k* tuple fields from the subscriptable `PNZ_merge_A'` storing the tuples, which results in `PNZ_merge_A''`. Additionally we perform a MergeEliminationPass to eliminate the "Merge" I/O

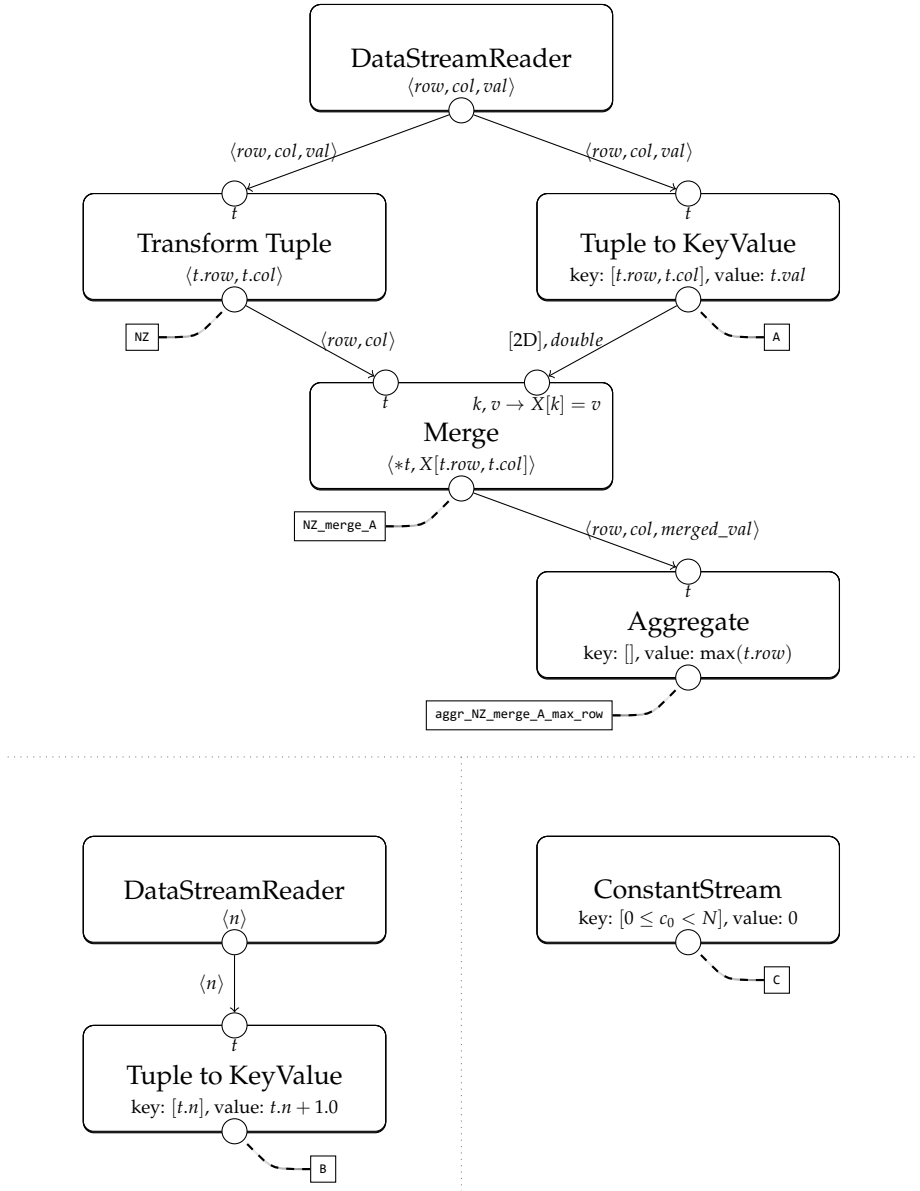


Figure 3.7: Input transformation graph corresponding with Listing 3.17 after performing three transformations.

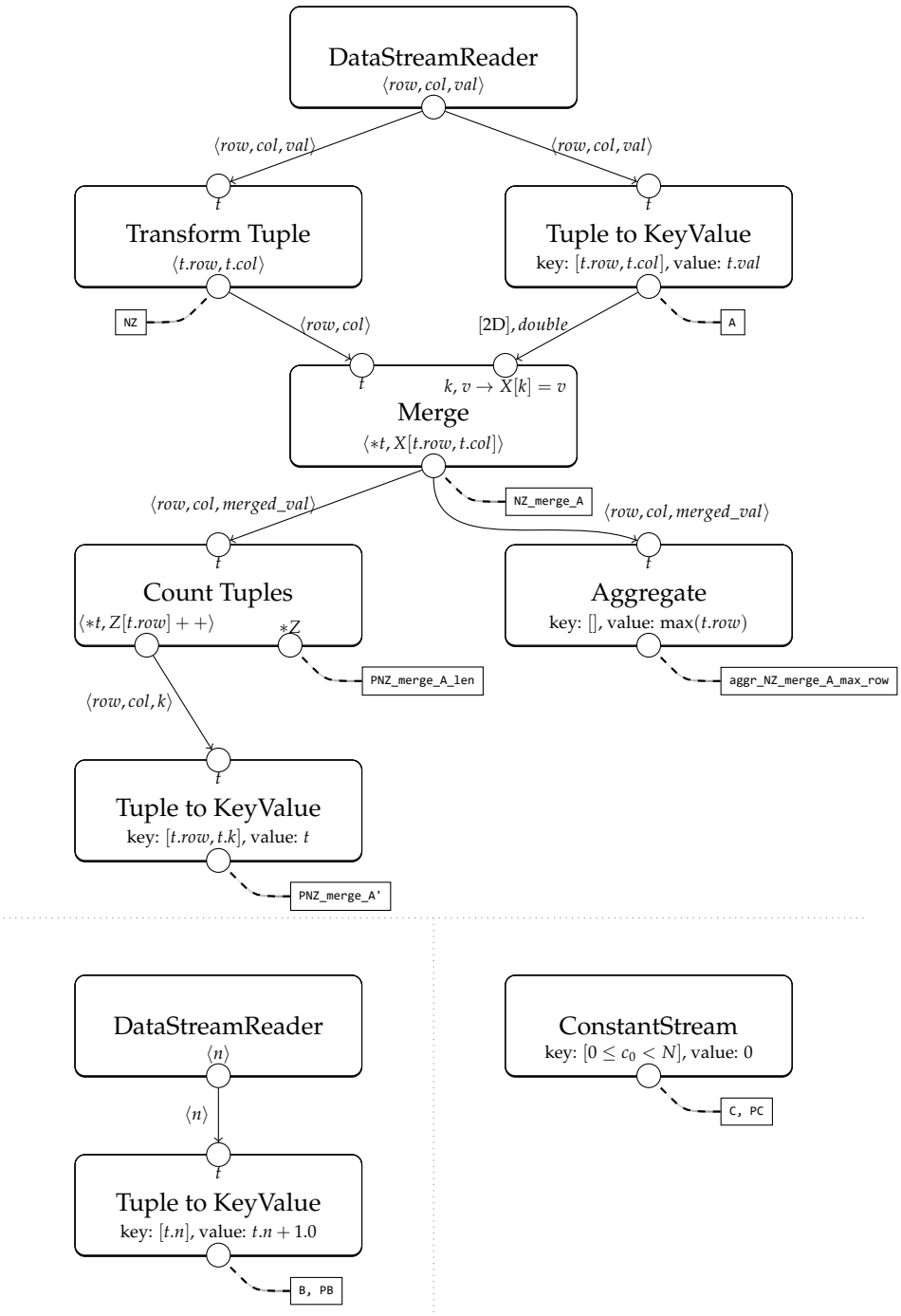


Figure 3.8: Input transformation graph corresponding with Listing 3.18 after performing six additional transformations.

node in the input transformation graph. Then we perform a StructureJaggedSplittingPass on `PNZ_merge_A''` to split the tuples $\langle col, merged\_val \rangle$ stored in this subscriptable into the groups $[\langle col \rangle, \langle merged\_val \rangle]$ at offset 1. To finish the data structure transformation process we perform the ConcretizationPass to concretize everything. This will yield the code in Listing 3.19 (note the conversion to C-style `for` loop specification) and the input transformation graph from Figure 3.9. The ConcretizationPass also changed the output transformation graph, as shown in Figure 3.10.

```
1 for row = 0, row <= aggr_NZ_merge_A_max_row, row += 1:
2   for k = 0, k <= PNZ_merge_A_len[row]-1, k += 1:
3     CPC[row] += CPNZ_merge_A'''[row].__merged_val[k].
    merged_val * CPB[CPNZ_merge_A'''[row].__col[k].col]
```

Listing 3.19: Algorithm specification after performing all transformations.

Finally the output code is generated using the C++ code generator. This code generator will also invoke routines to translate the transformation I/O graphs to tUPL code before converting those to C++ as described in Section 3.5.3. For reference, the full C++ output code of this example is provided in Appendix E.

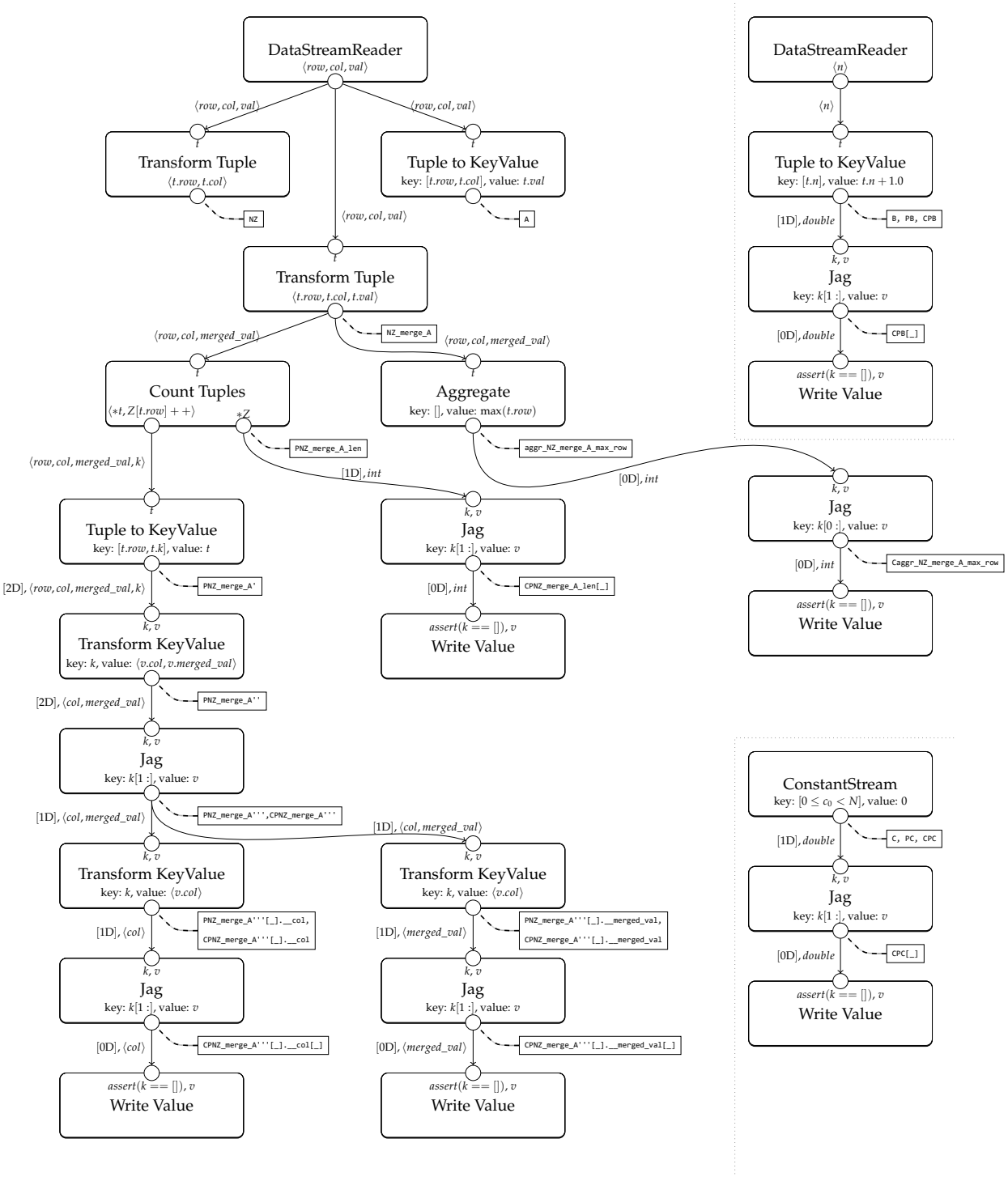


Figure 3.9: Input transformation graph corresponding with Listing 3.19 after performing all transformations.

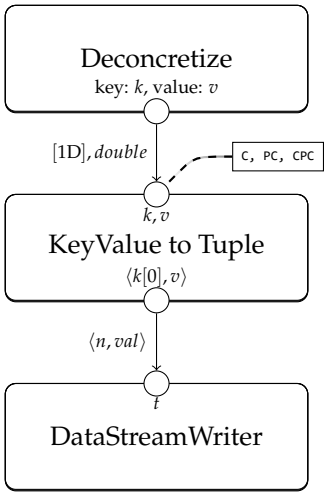


Figure 3.10: Final output transformation graph corresponding with Listing 3.19.

Chapter 4

Tython

In order to be able to easily use tUPL code in projects we have developed Tython, a front-end for libtupl. A key goal of Tython is to allow tUPL to be used within any existing Python 3 project wherever necessary, such as in functions demanding high performance, without burdening the user with a lot of interfacing hassle between Python and the libtupl output.

We introduce additional keywords to specify tUPL functions using a Python-like syntax. The Tython compiler can parse Tython code and compile the tUPL code blocks. All standard Python statements are compiled just like they would be if they were compiled with CPython directly. After compilation of Tython code the resulting Python compiled program (`.pyc`) can be directly executed using the standard CPython Python interpreter, or imported as a module in other Python or Tython programs just like normal Python modules. This makes Tython easily integrable in existing Python projects, as practically any Python program is compilable with the Tython compiler and Tython its output is always compatible with CPython.

We distinguish two compilation modes: *debug compilation* and *release compilation*. With debug compilation (Section 4.2) we compile all special Tython constructs to the output `.pyc` file directly, allowing end-users to test specifications quickly. When using release compilation (Section 4.3) all Tython constructs will be compiled to optimized C++ code using libtupl. This C++ code is then compiled to a Python C extension and invoked from the `.pyc` file automatically.

4.1 Syntax & parsing

In order to ensure high compatibility with Python, we have chosen to (automatically) extend the existing Python grammar and abstract syntax tree with the Tython constructs. Using CPython their own *pgen* tool, a parser can be generated from this grammar [6]. Additionally, the CPython project also developed a script to convert *Abstract-Type and Scheme-Definition Language* (ASDL) definitions to C header and source files that describe all AST nodes. Only some additional C code is then necessary to convert the parse tree into the AST for the newly introduced syntax. This yields us a highly robust Tython parser and semantic analyzer based upon the existing Python parser and semantic analyzer. Because we use the same tooling as the CPython project does

to generate parsers, AST definitions and semantic analyzers, existing compilation functions in the CPython project that parse text into Python ASTs can also be used to parse text into Tython ASTs.

In Tython we can specify sparse matrix-vector multiplication using, for example, the code in Listing 4.1. In this code we define a Tython module "MatVec" using the newly introduced `tdef` keyword. It consists of a tuplespace (`Reservoir`) and three shared spaces (`SharedSpace`), defined on lines 2—5 using the `ctxdef` keyword. This keyword can be used to define shared spaces and reservoirs for the entire specification. For example, `Reservoir[{row: int, col: int},]` indicates the reservoir contains tuples of type `{row: int, col: int}`, and `SharedSpace[2, int]` indicates the shared space has a two-dimensional index and contains integers. Within this module we define the "matvec" tUPL function on line 17, which operates on these spaces. Additionally, "load" and "unload" functions have been defined on lines 7 and 14 respectively. These functions are only symbolically analyzed and used to initialize the I/O state (in the case of release compilation this initializes the I/O transformation graphs, as described in Section 4.3, allowing libtupl to transform the I/O transformation graphs as it applies transformations). On line 9 we initialize shared space `A` from the input stream `Values`. For each tuple `v` in `Values` a value is inserted into `A` with index `(v.row, v.col)` and value `v.val`. Initialization of the other structures is similar. On line 15 we define how a data structure should be unloaded. Here, for each key-value pair `k, v` in `C` a tuple `(k[0], v)` is sent to the output stream `CVals`.

```
1  tdef MatVec:
2      ctxdef NZ: Reservoir[{row: int, col: int},]
3      ctxdef A: SharedSpace[2, int]
4      ctxdef B: SharedSpace[1, int]
5      ctxdef C: SharedSpace[1, int]
6
7      def load(Values: InStream[{row: int, col: int, val:
       int},],
8               BVals: InStream[{i: int, val: int},]):
9          BindSharedSpace(A, Values, lambda v: (v.row, v.
       col), lambda v: v.val)
10         BindSharedSpace(B, BVals, lambda v: (v.i,),
       lambda v: v.val)
11         BindReservoir(NZ, Values, lambda v: (v.row, v.col
       ))
12         BindSharedSpace(C, BVals, lambda v: (v.i,),
       lambda v: 0)
13
14     def unload(CVals: OutStream[{i: int, v: int},]):
15         BindSharedSpaceOut(CVals, C, lambda k, v: (k._0,
       v))
16
17     def matvec():
18         forelem t in NZ:
19             C[t.row] += C[t.row] + A[t.row, t.col] * B[t.
```

```
    col]
```

Listing 4.1: Sparse matrix-vector multiplication in Tython.

In order to actually invoke this tUPL code we can use standard Python code as shown in Listing 4.2. Whenever loading data, any Python iterable can be used as input, like a `list` (and is compatible with an **`InStream`** in Listing 4.1). Any container to which a tuple key can be assigned, like a `dict`, can be used to unload results into (such objects are compatible with an **`OutStream`** in Listing 4.1).

```
# like instantiating an instance of a class
matvec = MatVec()
# load data into the spaces
matvec.load(
    [(0, 0, 1), (5, 2, 4), (1, 3, 12.4)],
    [5, 6, 7, 8, 9, 10]
)
# run the algorithm
matvec.matvec()
# unload data to Python data structures
C = {}
matvec.unload(C)
print(C)
```

Listing 4.2: Invoking the sparse matrix-vector multiplication in Tython.

In Python, the *ast* module allows easy inspection and manipulation of the AST of Python code within Python scripts [18]. Similarly to this, we have created a new Python module *tast* that allows this for Tython code. Additionally, some utility routines to parse a string to such a Tython AST are included in this *tast* module. The Tython compiler will convert the Tython AST code to a standard Python AST by iterating through all Tython AST nodes using this *tast* module and creating equivalent Python AST nodes using the *ast* module (which is built into Python). For Tython-specific syntax (**`tdef`** blocks), a special compilation process takes over and replaces these Tython AST nodes with Python AST nodes. This process is described in more detail in the next two sections as this process differs between Tython its debug and release mode.

4.2 Debug compilation

The debug compilation mode transpiles all Tython specific syntax into equivalent Python code performing the same algorithm. We do not apply any special code transformations on this. Generally this leads to sub-optimal code, but it is useful to quickly test whether or not the specification works. This transpilation is fully implemented in Python, which generates a Python AST for the algorithm specification that is finally compiled to a `.pyc` file. It thus has no dependency on libtupl.

Each **`tdef`** is converted to a Python **`class`**, in which each **`Reservoir`** is represented as a Python `set` and each **`SharedSpace`** as a Python `dict` (indexed by tuples of integers). Python equivalents of the load and unload definitions can

be defined. In the case of debug compilation, `BindSharedSpace(A, Values, lambda v: (v.row, v.col), lambda v: v.val)` will insert, for each tuple `t` in `Values`, value `(lambda v: v.val)(t)` at key `(lambda v: (v.row, v.col))(t)` in `dict A`. `BindSharedSpaceOut(CVals, C, lambda k, v: (k._0, v))` will, for each key `k` and value `v` in `dict C`, insert the key and value `(lambda k, v: (k._0, v))(k, v)` into the `CVals` Python structure (typically a `dict`).

Now the algorithm can be transpiled. The algorithm in Listing 4.1 will be transpiled into a Python AST equivalent to the Python code shown in Listing 4.3. The `forelem` loop is simply converted into a `for`-loop and the subscripts into the `dicts` are now explicit Python tuples.

```
1  class MatVec:
2      ...
3      def matvec(self):
4          for t in self.NZ:
5              self.C[(t.row,)] = self.C[(t.row,)] + self.A
   [(t.row, t.col)] * self.B[(t.col,)]
```

Listing 4.3: Transpiled sparse matrix-vector multiplication.

`whilelem` loops will pseudo-randomly select tuples to execute from the tuplespace until no more tuples are enabled, then terminate. For example, the sorting specification in Listing 4.4 will compile to something roughly equivalent to the Python code in Listing 4.5.

```
1  tdef Sort:
2      ...
3      def sort():
4          whilelem adj in ADJS:
5              if A[adj.left] > A[adj.right]:
6                  tmp = A[adj.left]
7                  A[adj.left] = A[adj.right]
8                  A[adj.right] = tmp
```

Listing 4.4: Stable sorting in tUPL.

```
1  class Sort:
2      ...
3      def sort(self):
4          __all_tuples = set(itertools.product(self.ADJ,
   range(1)))
5          __enabled = set(__all_tuples)
6          while __enabled:
7              adj, __exec_seq_idx = random.sample(__enabled
   , 1)[0]
8              __enabled.remove((adj, __exec_seq_idx))
9              if __exec_seq_idx == 0 and self.A[adj.left] >
    self.A[adj.right]:
10                 tmp = self.A[adj.left]
11                 self.A[adj.left] = self.A[adj.right]
12                 self.A[adj.right] = tmp
```

```
13                    __enabled = set(_all_tuples)
```

Listing 4.5: Transpiled sort.

Note how the `whilelem` loop has been transpiled to a while-loop that keeps iterating a random tuple as long as the `__enabled` set of tuples and sequential code block pairs is nonempty. Whenever any tuple is selected it is removed from the enabled set. However, whenever any tuple successfully executes the `__enabled` set is reset back to `__all_tuples`, causing all previous disabled tuples to be attempted again.

The debug compiled `whilelem` loop does not check if the program ends up in a state to which it is always possible to return. The program should terminate in such a state, but when using debug compilation such specifications may never terminate.

4.3 Release compilation

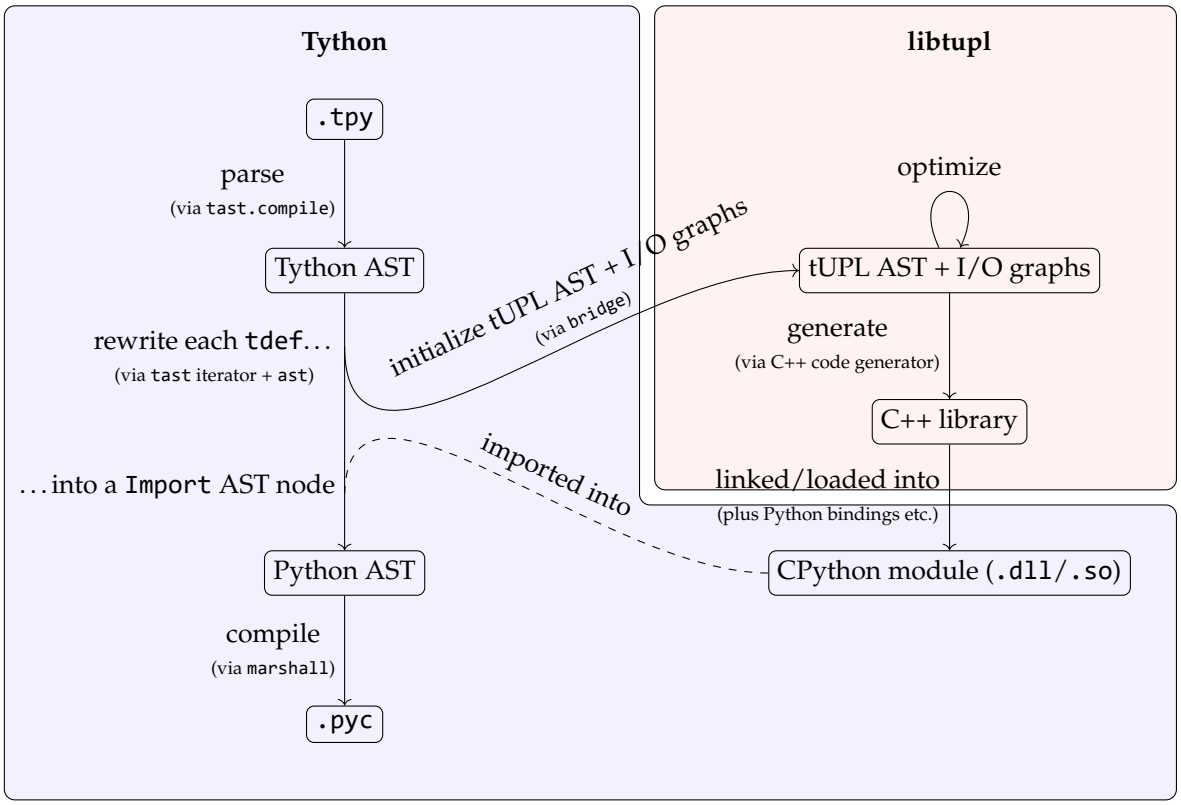


Figure 4.1: Data flow during the release compilation.

Whenever we compile using release mode we use libtupl to perform optimizations on the specified tUPL code. Figure 4.1 illustrates how data flows between each subsystem while the remainder of this section describes this process in more detail textually. As the Tython parser and utilities are imple-

mented in Python to allow rapid development, Python bindings are necessary for libtupl to allow Tython to interface with libtupl. This *bridge* exposes various libtupl classes to Python using *Boost.Python* [4]. It includes, for example, functions to define static types and symbols, functions to create AST nodes for tUPL code and functions to create I/O transformation graph nodes to allow constructing I/O transformation graphs from Python. The bridge also enables the various optimization transformations to be performed on the tUPL code and I/O graphs. Additionally, it exposes code generators, allowing it to dump, for example, C++ output. Some utilities are also exposed to, for example, determine the static result type of a tUPL AST expression.

For each `tdef` code block a separate tUPL compilation is performed (each `tdef` is a single `CompilationInstance` in Figure 2.1). First, the compiler will initialize the root node of the transformation tree, starting by copying the definitions of the shared spaces and tuplespaces that are defined through `ctxdef` to this node.

The load and unload function signatures are then analyzed statically. Each parameter of the load and unload functions, which must all be of type `InStream` and `OutStream` respectively in Tython, are converted to "DataStreamReader" and "DataStreamWriter" I/O node respectively. Tython constructs a generator for each "DataStreamReader" that iterates the values of any Python data structure and converts each value to a C++ struct which the generator will then yield[1]. For each "DataStreamWriter" Tython will generate a callback function that inserts each output tuple into a Python data structure.

Once these reader and writer I/O nodes have been initialized, tUPL will analyze the contents of the load/unload functions. Invocations to "Bind..." functions are then converted to I/O nodes. For example, `BindSharedSpace(A , Values, lambda v: (v.row, v.col), lambda v: v.val)` will create a "Tuple to KeyValue" node linking to the "DataStreamReader" loading the data of `Values`, with the two lambda-functions respectively specifying the key and value of each element in the shared space. `BindReservoir` behaves similarly to `BindSharedSpace`, just using a "Transform Tuple" I/O node. `ConstantInit (C, 0, (1000,))` will lead to the initialization of a "ConstantStream" (root) I/O node, which will stream key-value pairs for which the value is always 0. Tuples are produced for each of the one-dimensional keys $\langle k \rangle$ where $0 \leq k < 1000$. This "ConstantStream" node is then tagged with the shared space symbol expression `C`. `BindSharedSpaceOut(CVals, C, lambda k, v: (k._0, v))` is practically the reverse of `BindSharedSpace`. It takes the key-value data from `C`, converts it to a singular tuple using a "KeyValue to Tuple" I/O node (which is tagged with the symbol expression `C`), then links its output to the "DataStreamWriter" I/O node previously created for `CVals`. The load definition from Listing 4.1 is converted to the input transformation graph in Figure 3.5. The output definition is converted to the output transformation graph in Figure 3.6.

Finally, after initializing the I/O transformation graphs, the compiler will translate the Tython algorithm definition to a tUPL AST. The Tython and tUPL ASTs are quite similar, so this is a mostly straight-forward translation.

Now Tython will invoke the optimization routines on the fully initialized root transformation tree node. Once a concretized algorithm has been gener-

[1] The C++ code generator will generate structs for each tUPL (named) tuple.

ated Tython can invoke the C++ code generator to generate the optimized C++ implementation. Tython will add various additional C++ functions, such as the generator and callback routines that the "DataStreamReader" and "DataStreamWriter" need. Additionally, Tython generates Python C extension module specific code (such as a function bindings and other necessary runtime code) to allow compiling all generated code to a shared object that can be imported as a Python C extension module. This is then compiled using any C++ compiler, such as *clang*++ or *g*++.

Tython substitutes the `tdef` Tython AST node with a Python AST node that imports this generated module to allow invoking the optimized tUPL algorithm straight from Python. Once the compilation is completed the Tython AST is thus converted to a Python AST, which has been marshalled into a `.pyc` file. Additionally, for each `tdef` block Tython has generated and compiled a Python C extension that contains the optimized tUPL code, which is imported from the `.pyc` file.

Chapter 5

libtupl extensions

In order to support the experiments in Section 6, the set of transformations available in libtupl is extended by a number of additional experimental transformations. First we consider a simple algorithmic transformation that allows constructing *hybrid algorithms*, then a transformation to (safely) trivially parallelize the execution of loops, after that we take a look at a few adjustments to the I/O transformation trees to allow for runtime I/O and finally we take a look at dimensionality reduction.

The examples in this section operate on the example specification in Listing 5.1, in which *sparse matrix-dense matrix multiplication* (*SpMM*) is specified, with `VECCOUNT` being the number of columns in the dense matrix.

```
1 forelem i in [0, VECCOUNT-1]:
2   forelem nz in NZ:
3     C[i, nz.row] += A[nz.row, nz.col] * B[i, nz.col]
```

Listing 5.1: SpMM specification in tUPL.

We also use these transformations to derive CSR and Diagonal-CSR hybrid data structures from the SpMM specification in Sections 5.5 and 5.6 respectively.

5.1 Hybrid algorithms

It is possible to duplicate `forelem` loops and execute them separately in sequence while ensuring that each of the `forelem` loops iterate a different part of the tuplespace through `forelem` loop conditions. This can, for example, transform Listing 5.2 into Listing 5.3. Note that the tuples iterated in both `forelem` loops do not overlap, all tuples are still iterated only once.

```
1 forelem t in X:
2   ...
```

Listing 5.2: Scenario in which we can construct a hybrid algorithm.

```
1 forelem t in X where t.field0 == 0:
2   ...
```

```
3
4 forelem t in X where t.field0 != 0:
5   ...
```

Listing 5.3: Possible resulting code after constructing a hybrid algorithm.

The advantage of constructing hybrid algorithms is that certain scenarios can get more specialized implementations. For Listing 5.1 a possible hybrid variant could be as shown in Listing 5.4.

```
1 forelem i in [0, VECCOUNT-1]:
2   forelem nz in NZ where nz.row == nz.col:
3     C[i, nz.row] += A[nz.row, nz.col] * B[i, nz.col]
4
5   forelem nz in NZ where nz.row != nz.col:
6     C[i, nz.row] += A[nz.row, nz.col] * B[i, nz.col]
```

Listing 5.4: Hybrid SpMM specification in tUPL.

Note how in the first `forelem` loop we iterate all nonzeros on the *main diagonal*. If a matrix has many tuples on the main diagonal, this hybrid variant could be useful. With some additional transformations (such as the algorithmic transformation letting us iterate the upper loop row-by-row and localizing `A` into the tuplereservoir) the upper `forelem` loop can be transformed to the specification shown in Listing 5.5. Note how `NZ` is practically being split into two separate parts during this process (`NZ` and `PNZ_merged_A'`), indirectly performing *reservoir splitting* [7].

```
1 forelem i in [0, VECCOUNT-1]:
2   forelem row in [0, max(NZ.row)]:
3     C[i, nz.row] += PNZ_merged_A'[row].merged_val * B[i,
     nz.col]
4
5   forelem nz in NZ where nz.row != nz.col:
6     C[i, nz.row] += A[nz.row, nz.col] * B[i, nz.col]
```

Listing 5.5: Hybrid SpMM specification in tUPL after additional transformations.

As shown in Listing 5.5, we will in the end iterate the diagonal matrix values in a dense fashion, which often is much faster if the diagonal consists of primarily nonzeros. Note how we can transform the upper and lower `forelem` loop separately from one another: we may have localized `A` into `PNZ` in the upper `forelem` loop (and constructed an extended tuplespace), the bottom `forelem` loop still uses the original tuplespace without having the values of `A` merged into it.

5.2 Trivially parallelizing execution of loops

Although `forelem` loops are parallel by definition, in many cases special care must be taken by the compiler to ensure that the loop body is always executed atomically, such as when data dependencies exist across loop iterations. The

compiler can, for example, apply synchronization techniques to ensure this. However, certain loops may be able to be *trivially parallelized*, like when the execution of any iteration of some loop only has read-read data dependencies with other loop iterations. In Listing 5.6 we are about to concretize the specification. It is easy to see that we can execute the outer loop in parallel without the need of additional synchronization techniques because writes to `PC` directly depend on the loop iterator `i`. In the end, loop blocking can be applied to divide the iterations among multiple processors [7].

```
1 @parallelize
2 forelem i in [0, VECCOUNT-1]:
3   forelem k in PNZ_len[]:
4     PC[i, PNZ'[k].row] += PA[PNZ'[k].row, PNZ'[k].col] *
    PB[i, PNZ'[k].col]
```

Listing 5.6: Trivially parallelizable SpMM specification in tUPL.

5.3 Runtime I/O

Runtime I/O allows loading and unloading data while the algorithm is being executed. One way to achieve this is by exposing a chunk of (externally allocated) memory to tUPL. tUPL can then directly access the data from this chunk of memory and read or write to it. In the case where this chunk of memory is, for example, a *memory mapped file*, this can lead to runtime I/O from persistent storage.

To enable accessing such external chunks of memory we can introduce a "View" I/O node. Figure 5.1 visualizes how such a node can be used. The transformation graph is transformed as usual by all transformation passes.

When compiling a "View" node we do not actually iterate through all data in the load/unload routine, but instead just attach some external chunk of memory (pointer, i.e. M in Figure 5.1) to each subscriptable whose data is sourced from this "View" node. Each access to the subscriptable is transformed depending on the nodes on the path between the "View" node and the "Write Value" node and will then directly access the memory behind it.

For example, a read `B[x]` will be compiled to `(double)(*(M + x * sizeof (double))) + 1.0` in C++, where `M` is some external (untyped) memory pointer. Note that the "Transform KeyValue" node will cause `+ 1.0` to be executed for each read access to shared space `B`[1]. No data structure will be generated for shared space `B` anymore, all accesses directly access the external memory.

In the case of SpMM this is mostly applicable to the large dense data containers `B` and `C`. The "Transform KeyValue" node can transform the key to, for example, $[k[0] \,\%\, N, k[0] \,//\, N]$, to transform the 1D (memory) index k to a 2D (shared space) index. Note that a $[k[0] \,\%\, N, k[0] \,//\, N]$ key transformation implies that the dense matrix data is stored *column-major*, and $[k[0] \,//\, N, k[0] \,\%\, N]$ implies that it is stored *row-major*. This key transformation is reversed for each access to the subscriptable: `B[x, y]` will be compiled to `(double)(*(M + ( x + y * N) * sizeof(double))) + 1.0` in C++ (for the column-major key

[1] This can cause complications if `B` is also writable. We assume `B` is read-only here.

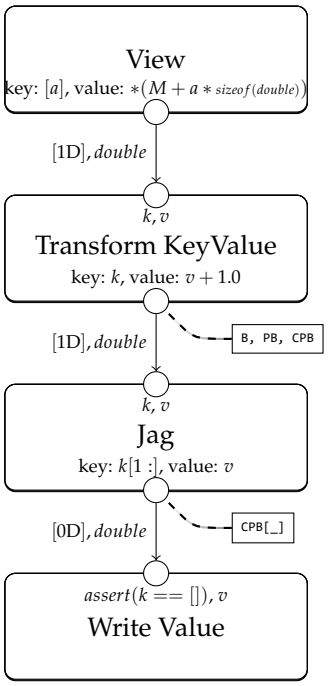


Figure 5.1: Simple input transformation graph utilizing a "View" node.

transformation). Runtime I/O prevents us from having to perform a lot of expensive copying to large generated data structures for `B` and `C`: we can just directly read and write to the external memory.

Note that not all key transformations can be reversed. For example, transforming the key k to $[k[0] \mathbin{/\!/} 2]$ cannot be reversed as $\lambda k : k \mathbin{/\!/} 2$ is not injective: we cannot recover the original k after the $[k[0] \mathbin{/\!/} 2]$ transformation.

5.4 Dimensionality reduction

Using the dimensionality reduction transformation we can convert two-dimensional structures accessed through a double loop into a one-dimensional structure [11]. This is best illustrated through an example. Figure 5.7 lists a specification on which dimensionality reduction can be effective. Note how the inner loop iterates tuples in `PNZ'` while the first subscriptable index remains locked to `row`. In other words, we just iterate a single *dimension* of the subscriptable using the inner loop.

```
1 forelem i in [0, VECCOUNT-1]:
2   forelem row in [0, max(PNZ.row)]:
3     forelem k in PNZ_len[row]:
4       PC[i, row] += PNZ'[row, k].val * PB[i, PNZ'[row, k
    ].col]
```

Listing 5.7: SpMM specification in tUPL after algorithmic optimization on which dimensionality reduction can be effective.

Rather than storing the tuples in a two-dimensional structure as in Listing 5.7, we can decide to place the values in each dimension one after another in a one-dimensional structure (without padding tuples). This is illustrated in Listing 5.8. A new one-dimensional subscriptable `PNZ''_ptr` then indicates where the data begins and ends in the one-dimensional `PNZ''` for each `row`.

```
1 forelem i in [0, VECCOUNT-1]:
2   forelem row in [0, max(PNZ.row)]:
3     forelem k in [PNZ''_ptr[row], PNZ''_ptr[row+1] - 1]:
4       PC[i, row] += PNZ''[k].val * PB[i, PNZ''[k].col]
```

Listing 5.8: SpMM specification in tUPL after performing dimensionality reduction.

5.5 Deriving a CSR implementation

The specification in Listing 5.1 can be transformed to an implementation using a CSR data structure [3] for the sparse matrix. We do not modify the `PC` and `PB` dense matrix data structures.

This data structure can be constructed by performing algorithmic optimization such that the tuples are iterated row-by-row, then the EncapsulationPass, AggregateReservoirPass, LocalizationPass, QueryForwardSubstitutionPass, ReservoirMaterializationPass, NStarMaterializationPass, SharedSpaceMaterializationPass, HorizontalIterationSpaceReduction, Dimensionality reduction, DelocalizationPass and the ConcretizationPass. Loops are trivially parallelized when possible. Runtime I/O is performed as well for all data structures when possible (i.e. for the `CPNZ_zipped_A''` array (containing the nonzero values) when the input format is the COO format, and the dense matrices). Listing 5.9 illustrates the resulting algorithm roughly.

```
1 for i in [0, VECCOUNT-1]:
2   for row in [0, max_NZ_row-1]:
3     for k in [CPNZ_zipped_A_ptr''[row], CPNZ_zipped_A_ptr
    ''[row+1] - 1]:
4       CPC[i, row] += CPNZ_zipped_A''[k] * PB[i,
    CPNZ_zipped_A''_deloc[k]]
```

Listing 5.9: SpMM implementation using a CSR data structure for the sparse matrix in tUPL.

5.6 Deriving a Diagonal-CSR hybrid implementation

As an example of the generation of a hybrid algorithm, we can derive an implementation which stores certain diagonal bands more efficiently. Nonzero elements that do not fit in such a diagonal band are demoted to an entry in the secondary CSR data structure.

This data structure can be constructed by first transforming the algorithm

to a hybrid algorithm, as in Listing 5.4 (where we split only the center diagonal band into a separate loop). Both loops can now be transformed independently of one another. The CSR implementation is derived like previously for the loop `where nz.row != nz.col`. It is here also decided that we only insert tuples for which `where nz.row != nz.col` holds into the CSR data structure, eliminating the need for the `where nz.row != nz.col` query.

The loop `where nz.row == nz.col` is transformed by first performing algorithmic optimization such that the tuples are iterated row-by-row. Forward substitution is performed inside of the loop, replacing all occurrences of `nz.col` with `nz.row`. Then, the EncapsulationPass, AggregateReservoirPass, QueryForwardSubstitutionPass, ReservoirMaterializationPass and NStarMaterializationPass are performed. Because only one tuple exists in each row, the `NZ` reservoir is materialized to a 1D `Subscriptable`. Finally, the SharedSpaceMaterializationPass, HorizontalIterationSpaceReduction and ConcretizationPass (to a dense array for the values) are performed.

Listing 5.10 illustrates a possible resulting algorithm. Note that various minor transformations can still be applied, such as loop interchange, splitting or merging. Additionally, multiple diagonals can be transformed to such a dense array.

```
1 for i in [0, VECCOUNT-1]:
2   for row in [0, max_NZ_row-1]:
3     CPC[i, row] += CPNZ''[row] * CPB[i, row]
4
5   for row in [0, max_NZ_row-1]:
6     for k in [CPNZ_zipped_A_ptr''[row], CPNZ_zipped_A_ptr
    ''[row+1] - 1]:
7       CPC[i, row] += CPNZ_zipped_A''[k] * CPB[i,
    CPNZ_zipped_A''_deloc[k]]
```

Listing 5.10: SpMM implementation using a Diagonal-CSR hybrid data structure for the sparse matrix in tUPL.

Chapter 6

Experiments

In order to show that solely using the transformations we have previously defined implementations can automatically be derived that perform competitively, we have conducted various experiments on four derived implementations. For these experiments we look at SpMM and compare a few of our derived implementations with highly hand-optimized SpMM implementations. Listing 5.1 shows a base SpMM implementation in tUPL.

6.1 Experimental configurations

We performed various experiments on the SpMM algorithm. Generally we do not transform the `B` and `C` data structures from their storage formats and utilize runtime I/O for these large dense matrices by memory mapping files containing their data. We do vary column-major and row-major storage for these two dense matrix data structures and different implementations are derived for both variants. We do derive various different data structures for `A`, which may lead to runtime I/O for this sparse matrix becoming unfeasible, forcing us to convert the sparse matrix to the desired format in memory before starting the algorithm itself. This format conversion is done at runtime. `A` is always stored in a file in three disjoint arrays (row, column and value) of equal length. Within these experiments we always order the elements by row, then column. All three arrays are aligned to 32 bytes. Sometimes (part of) these arrays can be used directly without any additional conversion, in which case runtime I/O still is performed. For example, the value and column arrays need no conversion when a CSR-like data structure is desired for `A`. Conversion routines are not parallelized.

We primarily consider four of our implementations as enumerated in Table 6.1: CSR (*Derived CSR*) and a Diagonal-CSR hybrid (*Derived Hybrid*) for both row major and column major inputs, as described in Sections 5.5 and 5.6 respectively. Preliminary experiments have shown that resorting the sparse matrix data for, for example, a CSC format is generally unfeasible and COO formats are additionally not competitive as conversion to CSR is very cheap. These derived implementations are compared with the Intel Math Kernel Library (MKL) their new Inspector-Executor sparse BLAS routines for CSR matrices (*MKL CSR*) [9]. We similarly do not consider COO here. Due to current

performance issues with MKL their CSC implementation we will not consider input that is sorted by column, then row in our experiments[1].

	See also
Derived CSR (For column major dense matrices)	Section 5.5
Derived CSR (For row major dense matrices)	Section 5.5
Derived Hybrid (For column major dense matrices)	Section 5.6
Derived Hybrid (For row major dense matrices)	Section 5.6

Table 6.1: The four derived implementations.

For the Derived Hybrid variant it is generally desired to minimize the amount of explicit zeros in the stored diagonal bands. If a diagonal band thus has few nonzeros, we elect to just store them in the secondary CSR data structure instead.

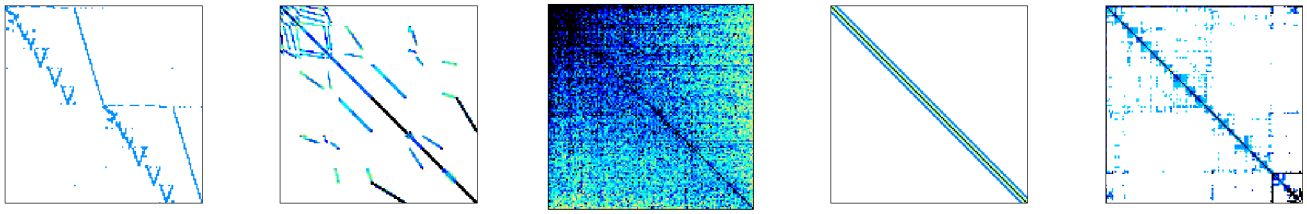
Clang 7 is used to compile the C++ code with `-O3 -march=native -ffast-math`. We rely on clang to perform low-level optimizations such as vectorization. The code generator could insert a limited number of compiler hints such as `restrict` and `assume_aligned`, which we can mechanically derive from the property that subscriptables do not share memory and that subscriptable data is always allocated at 32-byte boundaries. 64-bit doubles and 64-bit integers are used everywhere. We do not perform low-level (manual) optimizations that require extra domain knowledge.

All experiments are have been executed on the DAS-5 cluster at Leiden University [2]. Each node consists of two Intel Xeon E5-2630 v3 CPUs at 3.2 GHz. Each processor has 8 cores. While we will consider multi-processor computation (via NUMA), we do not consider simultaneous multithreading or multi-node computation. All matrix data is read from and written to a separate file server, which is connected to the compute nodes via a FDR InfiniBand interconnect. MKL version 2019.0.117 is used.

In the result figures each bar shows the mean runtime of the algorithm in a certain scenario. Each bar also is split into three parts with different shades. The bottom (lightest) part shows the *initialization* time. This consists of opening input files, creating and opening a new properly-sized output file[2] (or allocating memory in the heap and initializing the input array), memory mapping the files and converting the sparse matrix data to the desired format. The middle part of each bar shows the actual SpMM runtime. The top (darkest) part shows the cleanup time, which includes closing files and freeing allocated memory. Note that error bars are also drawn for each bar, but these are sometimes too small to be visible.

[1]The CSC sparse matrix-dense matrix multiplication routine in MKL is more than a magnitude slower than a naive CSC implementation. Additionally, as of this writing MKL only supports column major dense matrices in this configuration.

[2]This implicitly results in a zero-initialization of the output, but the implementations still do not assume any initialization for the output data.



(a) Mallya/lhr34c (b) Fluorem/HV15R (c) SNAP/sx-stackoverflow (d) Bourchtein/atmosmodl (e) Rajat/rajat30

Figure 6.1: The five sparse matrices used for the overview experiments.

	Width/height	Nonzeros	Kind
Mallya/lhr34c	35 152	764 014	Chemical Process Simulation Problem
Fluorem/HV15R	2 017 169	283 073 458	Computational Fluid Dynamics Problem
SNAP/sx-stackoverflow	2 601 977	36 233 450	Directed Temporal Multigraph
Bourchtein/atmosmodl	1 489 752	10 319 760	Computational Fluid Dynamics Problem
Rajat/rajat30	643 994	6 175 244	Circuit Simulation Problem
Kim/kim2	456 976	11 330 020	2D/3D Problem
Schenk_IBMSDS/matrix_9	103 430	1 205 518	Semiconductor Device Problem
Muite/Chebyshev4	68 121	5 377 761	Structural Problem
Simon/raefsky3	21 200	1 488 768	Computational Fluid Dynamics Problem
Bourchtein/atmosmodl-2	2 979 504	41 279 040	*(Derived)*
Bourchtein/atmosmodl-3	4 469 256	92 877 840	*(Derived)*

Table 6.2: Properties of all tested sparse matrices.

6.2 Overview experiments

We first consider five matrices with different sparse structures and compare the performance of a variety of implementations on these matrices. The test sparse matrices are all sourced from *The University of Florida sparse matrix collection* [5]. These first five matrices used are shown in Figure 6.1. Some additional properties of these matrices are shown in Table 6.2. Sparse matrix values are always converted to doubles. If a matrix its values are complex numbers, the imaginary part is discarded in a preprocessing step.

All SpMM implementations are executed in a few different scenarios. We first vary the number of columns in the dense matrices. Additionally, we also vary the CPU configuration the threads are run on, considering 2, 4 and 8 threads, but also $4+4$ and $8+8$ threads in a NUMA configuration (i.e. in the case of $8+8$ we run 8 threads on both NUMA nodes/processors). The input dense matrix consists of random doubles in the interval $[0, 500)$. We also test the scenario where the dense matrices are memory mapped separately from the scenario where dense matrices are fully in-memory (and initialized in-memory to a constant value too, in parallel). Sparse matrices are always loaded from memory mapped files, but they are generally much smaller in size than the dense matrices. We run each experiment at least 12 times and discard the first

two runs to reduce variety.

For the NUMA cases we use the *local* memory allocation policy. Due to the parallel initialization of the input dense matrix this will result in the first half of the data being on NUMA node 0 and the second half on NUMA node 1. The output data is not initialized, so the algorithms will determine the actual memory binding runtime instead (allocations are based on which thread triggers the allocation through a page fault). If the input and output dense matrices are memory mapped instead of heap-allocated the exact allocation behavior depends on the OS kernel instead. Preliminary experiments have shown that duplicating the shared sparse matrix data structures such that both NUMA nodes have a local copy does not improve performance significantly.

It is generally expected for the sparse matrices with obvious diagonals to have improved performance when using the Derived Hybrid algorithm compared to the Derived CSR implementation. We expect the Derived CSR implementation to perform slightly worse than the hand-optimized MKL CSR implementation.

Results Let us first look at the performance of *sx-stackoverflow*. Figure 6.2 shows the timings of the tested algorithms in various scenarios. Interestingly, the Derived CSR implementation generally outperforms MKL CSR in the column major scenarios. This could be due to the highly irregular structure of the sparse matrix: some rows are extremely dense and others are very spare, without obvious grouping. The MKL CSR may not be optimized to handle such matrices well. Interestingly, the MKL CSR performance worsens severely in the NUMA scenarios too, for this matrix. In the 256 columns, $8 + 8$ scenario Derived CSR even is nearly three times as fast as MKL CSR. When considering memory-mapped dense matrices, though, these significant gains start to fade, as shown in Figure 6.3. There likely is a major I/O bottleneck in this case. For the row major cases the differences are much smaller and MKL CSR tends to slightly outperform Derived CSR in most cases, being between 0.0 and 0.7 times faster.

The *atmosmodl* matrix contains more dense diagonal bands (unlike sx-stackoverflow), so we also test the Derived Hybrid implementation in this case. Figure 6.4 shows the results without memory-mapped dense matrices. Clearly, for the column major scenarios, the hybrid implementation significantly outperforms the Derived CSR implementation, for example being 31% faster in the 512 columns, 2 threads configuration. The Derived Hybrid implementation also comes close or even beats the MKL CSR implementation while the Derived CSR implementation does not. For example, in the 512 columns, $8 + 8$ threads scenario, it has a total running time of about 75% MKL its running time. Note that this is again in a NUMA scenario. However, in scenarios with few columns, the Derived Hybrid implementation may have a too high *initialization time* for the hybrid implementation to be worth it: converting the initial COO format to Derived CSC or the MKL CSC format is measurably faster than converting it to the Derived Hybrid format. This is, for example, visible in the 32 columns, 8 threads scenario, where the light shaded bottom section (initialization time) is where the majority of the total running time is spent.

For the row major scenarios the Derived Hybrid format is slightly slower. Derived CSR is on par with MKL CSR, though, in the worst case being up

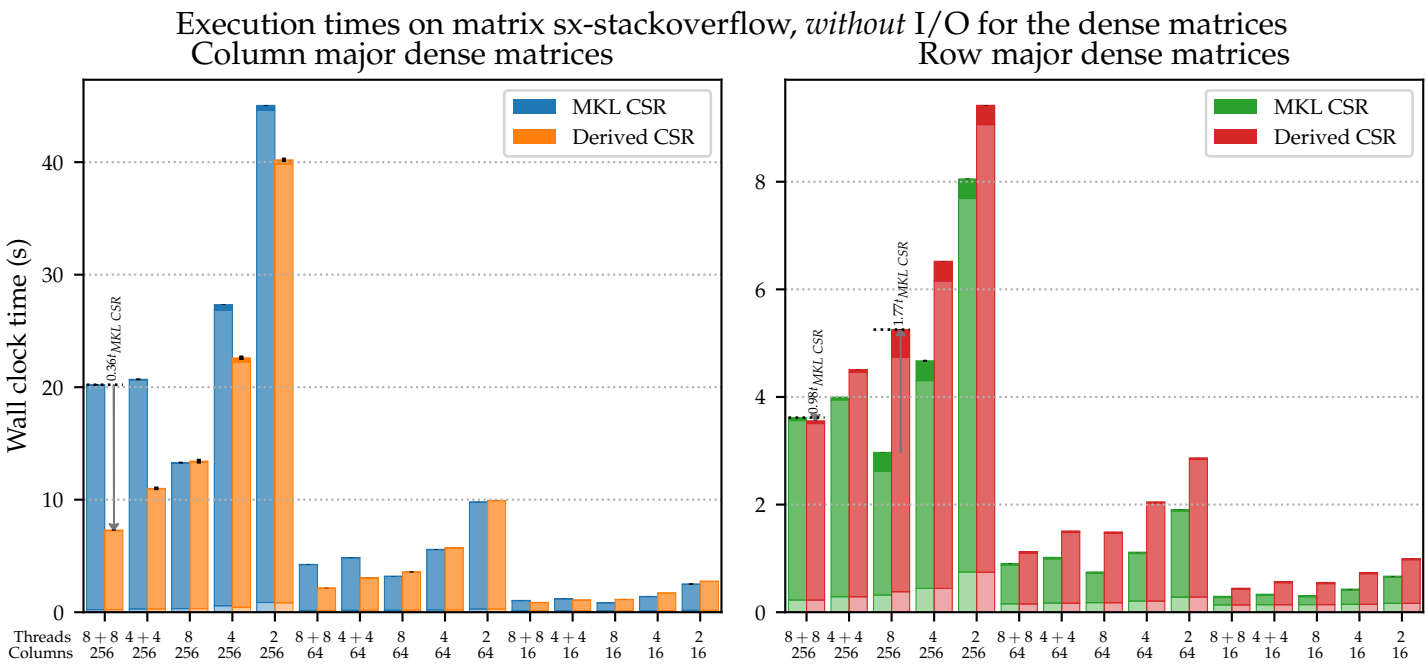


Figure 6.2: Performance on the sx-stackoverflow matrix where the dense matrices are not memory mapped.

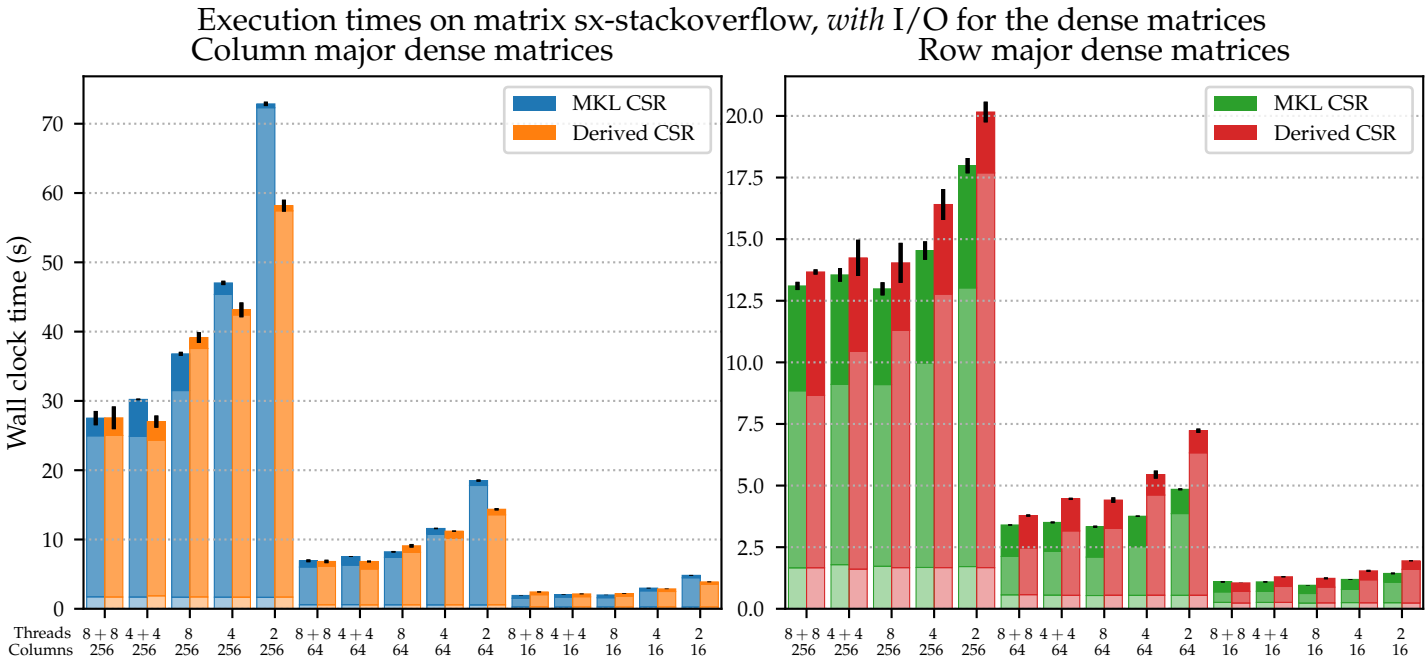


Figure 6.3: Performance on the sx-stackoverflow matrix where the dense matrices are also memory mapped.

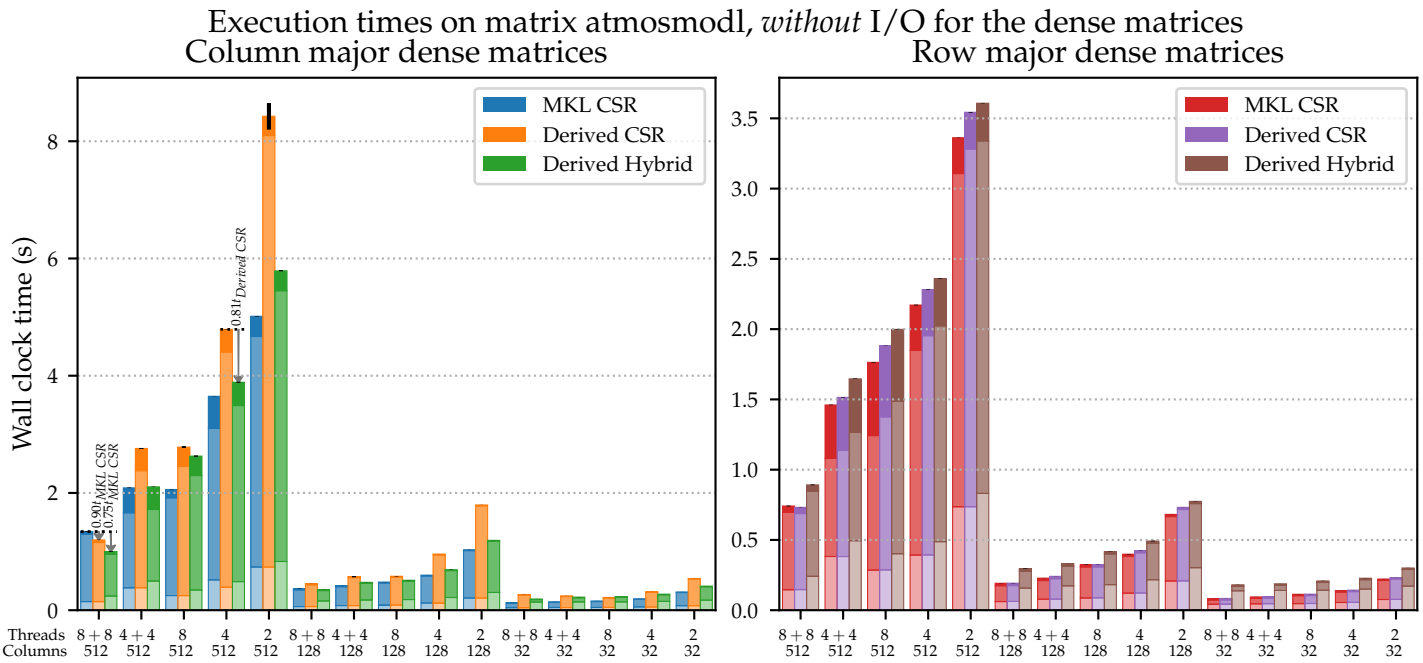


Figure 6.4: Performance on the atmosmodl matrix where the dense matrices are not memory mapped.

to 5% slower. Alternate hybrid data structures may be beneficial to try here instead due to the different iteration order, such as one where the 2D array containing the diagonal data is transposed. Just like with the sx-stackoverflow matrix, memory mapping the matrices will reduce the performance differences between the scenarios significantly, as shown in Figure 6.5. This is a common pattern among all tested matrices, although usually less extreme than for this matrix as most other matrices have more nonzeros per row, reducing the I/O time – CPU time ratio.

The *lhr34c* matrix has somewhat expected results: MKL CSR generally outperforms Derived CSR. The hybrid implementation does not apply here as there are no diagonal bands.

With the *rajat30* matrix there are three diagonal bands, but only about 20% of the values fall in these bands. Additionally, these bands are not very dense. As shown in Figure 6.6, the Derived CSR implementation sometimes even outperforms the Derived Hybrid implementation, most likely because of this. In some NUMA scenarios, such as the 1024 columns, $8 + 8$ threads scenario with column major dense matrices, Derived CSR does outperform MKL CSR. These significant differences disappear when the dense matrices are memory mapped, as shown in Figure 6.7. Note how the cleanup time is significantly higher for the Derived Hybrid variant in Figure 6.7. This is most likely because the Derived Hybrid implementation does two passes over the dense matrix data for the diagonal and remaining CSR data, naturally worsening the performance of it, as output dense matrix pages written back during the diagonal data pass will have to be written back again once the CSR pass modifies it during the second pass. This results in more pages that have to be written back

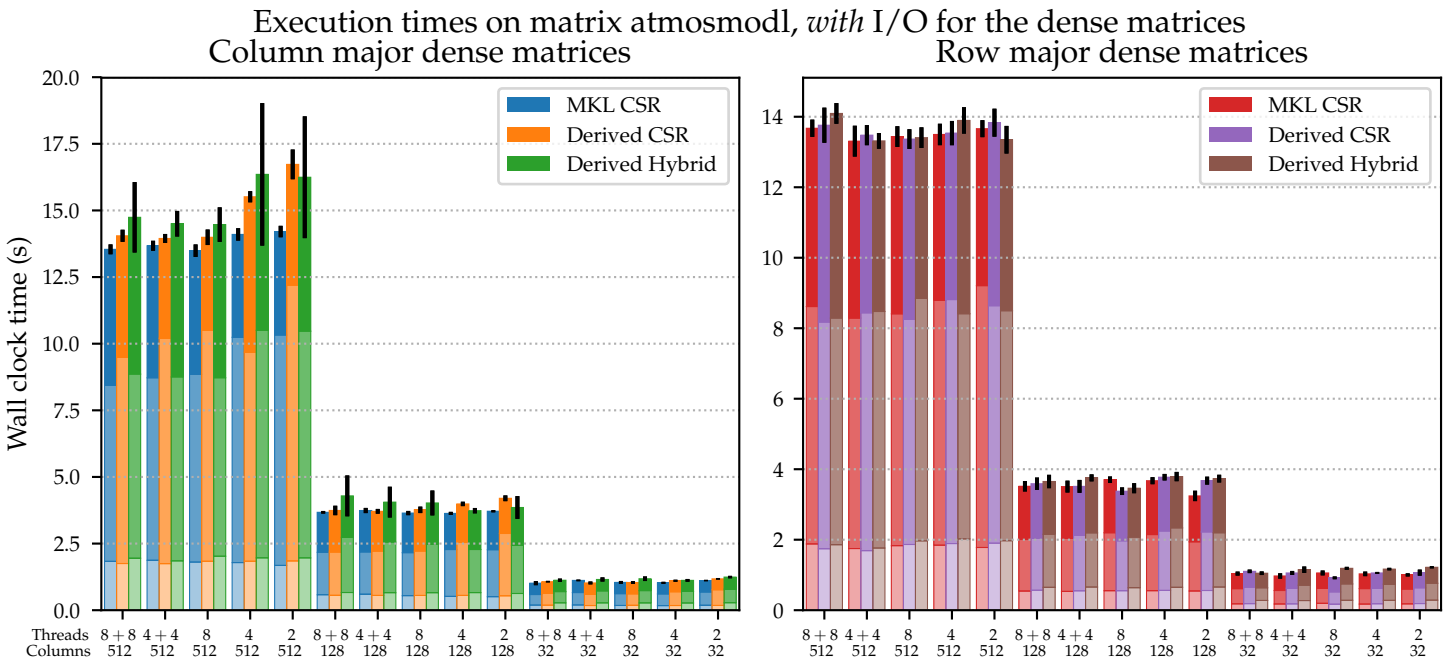


Figure 6.5: Performance on the atmosmodl matrix where the dense matrices are also memory mapped.

during the cleanup phase rather than during the algorithm its own runtime. This is, for example, clearly visible in the 1024 columns, 8 or 4 threads scenarios with column major dense matrices. Additionally, because not a lot of data is in the diagonal data array, a lot of time is spent waiting on the input matrix data to be loaded during the first pass: most operations on each dense matrix element occur in the second pass instead (the CSR pass), during which all data has already been loaded. MKL CSR and Derived CSR are mostly on par here, though.

Performance for the *HV15R* matrix is similarly to the rajat30 matrix, with only about 11% of the nonzeros being in a diagonal band, although when the dense matrices are memory mapped a significant performance improvement can be seen through the Derived Hybrid variant in the column major cases compared to Derived CSR, as shown in Figure 6.8, for example being twice as fast in the 512 columns, 4 threads scenario. MKL CSR still outperforms Derived Hybrid here, though, taking only 60% the amount of time Derived Hybrid does. The row major cases have low performance variance, but Derived CSR outperforms MKL CSR in the majority of the cases, for example being 14% faster in the 512 columns, 2 threads scenario. This matrix is a very large matrix with many nonzeros per row, but many nonzeros are not located in dense diagonal bands. Unlike rajat30, the HV15R matrix has 17 diagonal bands. As a result, the first Derived Hybrid pass is not as I/O bottlenecked as this first pass is for the rajat30 matrix.

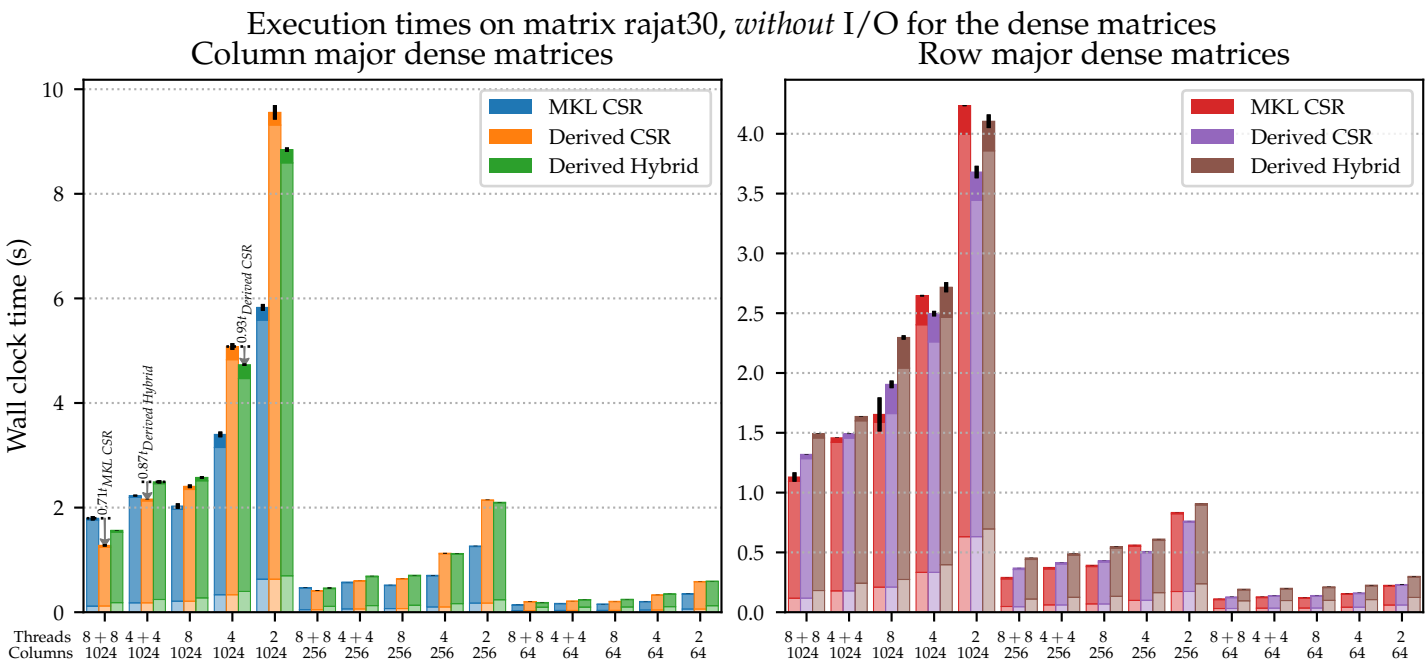


Figure 6.6: Performance on the rajat30 matrix where the dense matrices are not memory mapped.

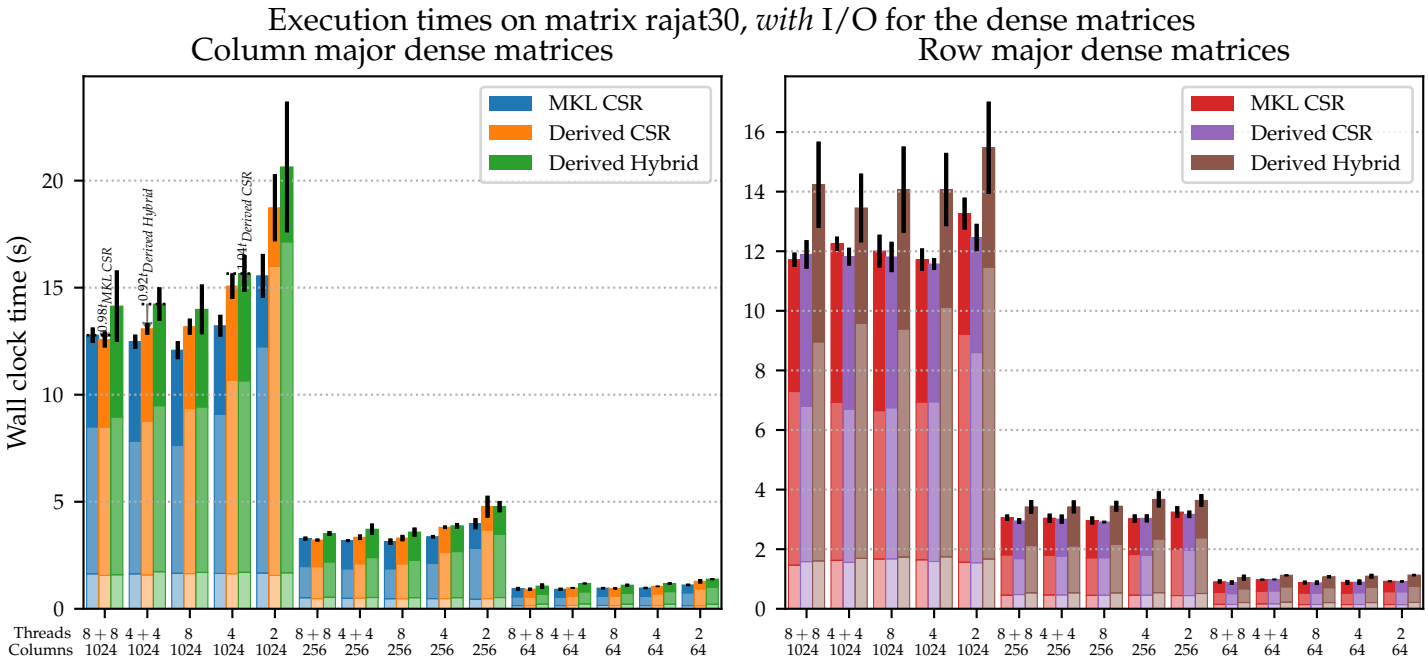


Figure 6.7: Performance on the rajat30 matrix where the dense matrices are also memory mapped.

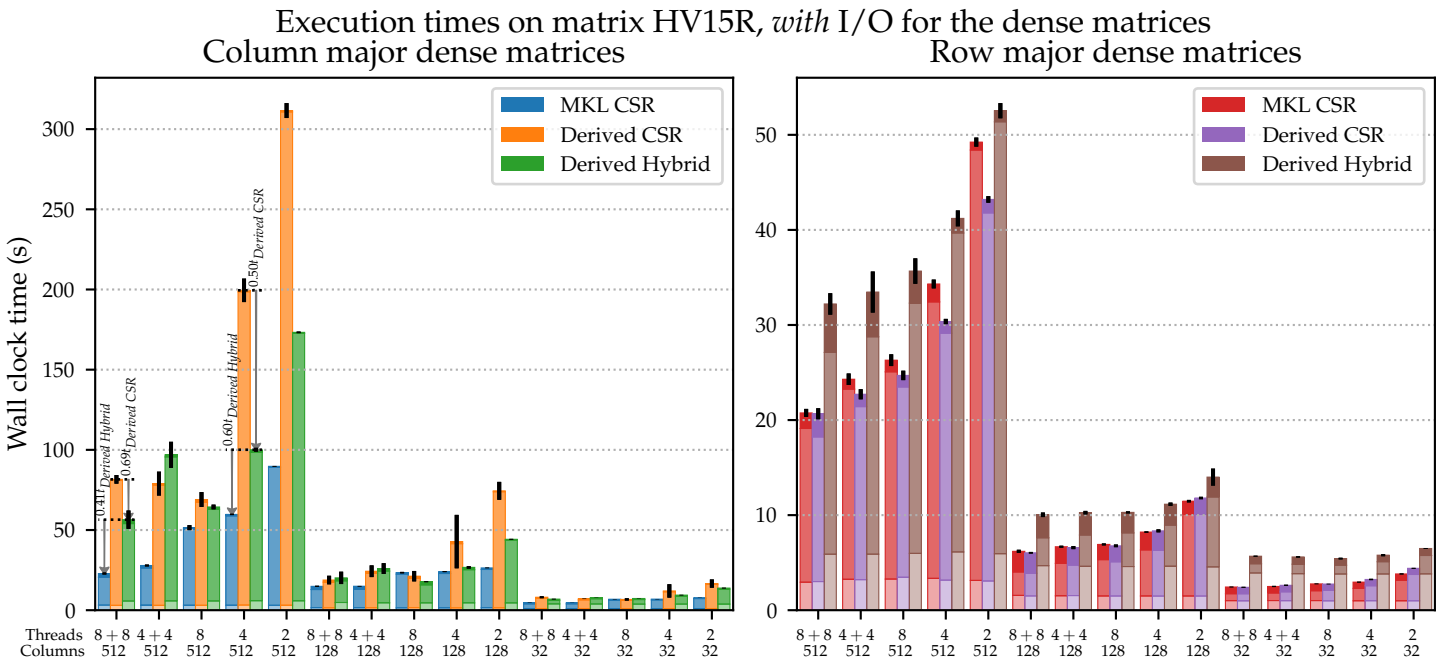
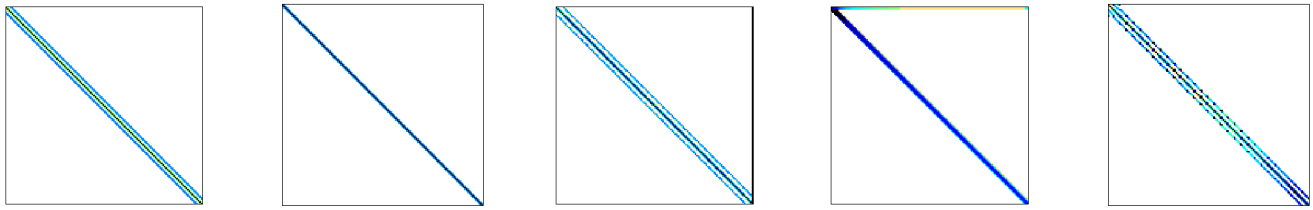


Figure 6.8: Performance on the HV15R matrix where the dense matrices are also memory mapped.

6.3 Diagonal experiments

The Derived Hybrid variant, as expected, has a positive influence on performance compared to Derived CSR if many nonzeros are in (dense) diagonals of the sparse matrix. In this followup experiment we study these cases in more detail. Figure 6.9 displays the matrices used in these experiments. Table 6.2 also lists some additional information about these matrices. We perform the same experiments on the diagonal matrices as we did on the overview matrices.

Results We have previously already seen and analyzed the performance of the atmosmodl matrix. The *kim2* matrix performs very similarly to this matrix. Like the atmosmodl matrix, the kim2 matrix has dense diagonal bands and no



(a) Bourchtein/atmosmodl (b) Kim/kim2 (c) Schenk_IBMSDS/matrix_9 (d) Muite/Chebyshev4 (e) Simon/raefsky3

Figure 6.9: The five sparse matrices used for the diagonal experiments.

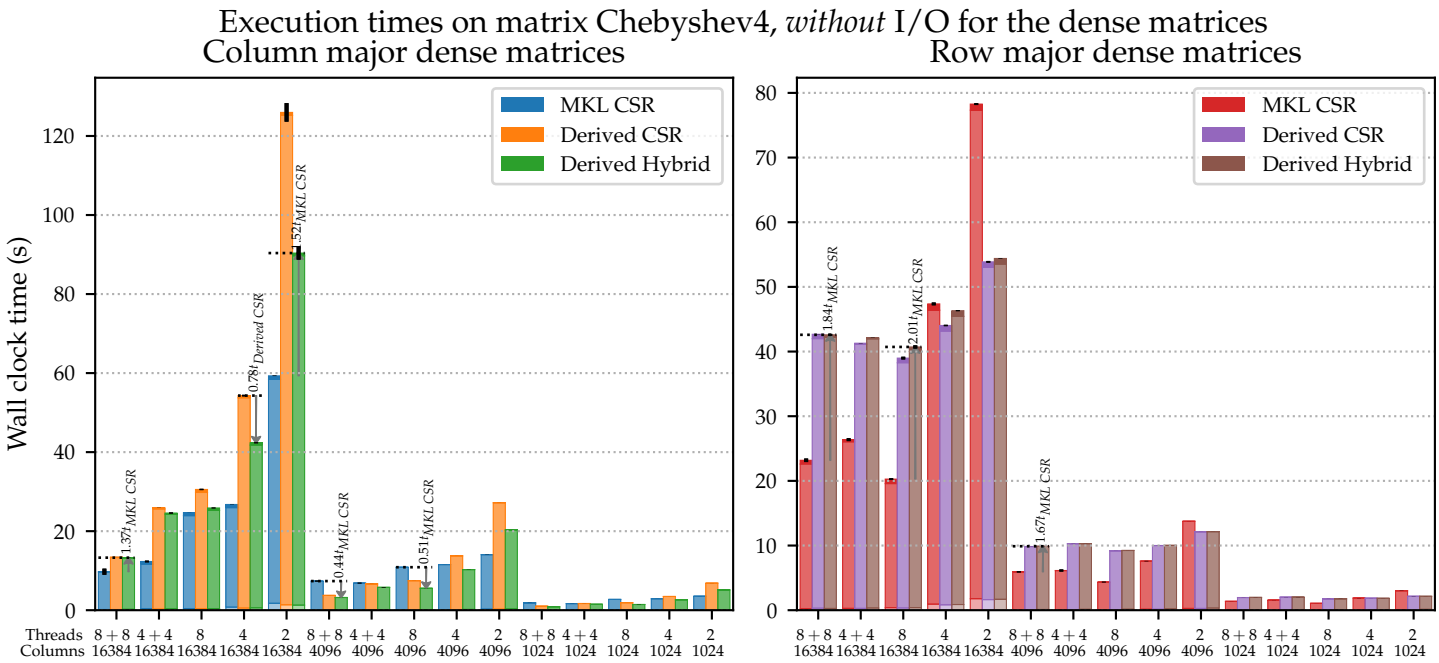


Figure 6.10: Performance on the Chebyshev4 matrix where the dense matrices are not memory mapped.

additional nonzeros outside of a clear diagonal band. The kim2 matrix does have 25 diagonal bands, though, whereas the atmosmodl matrix only has 7.

The *Chebyshev4* matrix has obvious diagonal bands, but only close to 31% of the nonzeros are located in these diagonal bands, all other elements are thus located in the secondary CSR format when using the Derived Hybrid format. As shown in Figure 6.10, the performance of the Derived Hybrid format does not come as close to the MKL CSR performance compared to the performance with the atmosmodl matrix in the column major scenarios with 16384 columns (roughly 40%–50% times slower than MKL CSR), but the Derived Hybrid implementation still significantly outperforms the Derived CSR implementation in many scenarios, even with such a small amount of nonzeros being located in diagonal bands, such as in the 16384 columns, 4 threads scenario, where it is 28% faster. Interestingly, with fewer columns, such as in the 4096 columns scenarios, Derived Hybrid often outperforms both Derived CSR and MKL CSR, being twice as fast in the $8 + 8$ and 8 threads scenarios. In the row major scenarios MKL CSR is in many scenarios much faster: up to two times as fast in the $8 + 8$ threads scenarios. This matrix also has various dense rows at the top of the matrix: a triple-hybrid format may improve performance further for this matrix.

The *raefsky3* matrix performs somewhat similar to *Chebyshev4*, as shown in Figure 6.11. For this matrix, more elements fit in the diagonal bands when using the Derived Hybrid format, close to 89%. However, most of these diagonal bands are not dense, like with rajat30. Although there still is a measurable performance improvement when using the Derived Hybrid implementation in the column major scenarios compared to Derived CSR, it may also be more bene-

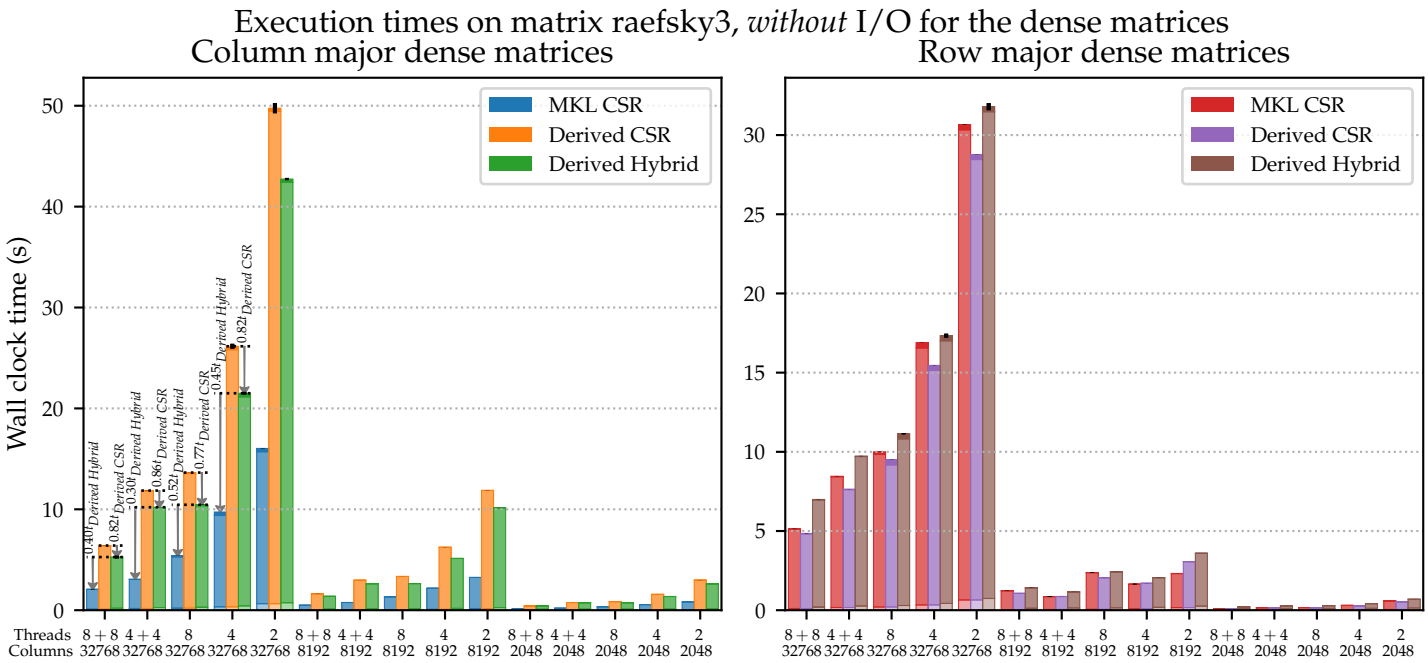


Figure 6.11: Performance on the raefsky3 matrix where the dense matrices are not memory mapped.

ficial to not store certain diagonal bands in the diagonal data array if many explicit zeros are required and instead store them in the secondary CSR format. MKL CSR tends to outperform the derived implementations significantly in most scenarios, although the performance differences are much smaller in the row major scenarios.

For the *matrix_9* matrix the Derived Hybrid format generally performs worse than the Derived CSR format as shown in Figure 6.12. Many diagonal bands are not dense in this matrix when using the Derived Hybrid format, some only consist of about $\frac{1}{3}$ nonzeros, leading to a lot of explicit zeros, causing this reduced performance. Additionally, this matrix has a range of nonzeros in the last few columns too, outside of any diagonal band.

6.4 Duplicated matrix experiments

Clearly the Derived Hybrid implementation can, in a range of scenarios, perform better than the Derived CSR implementation and in several cases even better than MKL CSR. The Derived Hybrid implementation generally sees a significant reduction in execution time over Derived CSR and MKL CSR when a fair amount of nonzeros are in this diagonal data structure and when the secondary CSR structure contains a low amount of explicit zeros.

Within this section we will look at effects of this balance on performance. We look at the *atmosmodl* matrix, for which we previously used the 7 diagonal bands -39204, -198, -1, 0, 1, 198 and 39204. These bands were fairly densely filled with nonzeros. We *duplicate* the atmosmodl matrix and derive atmosmodl-2

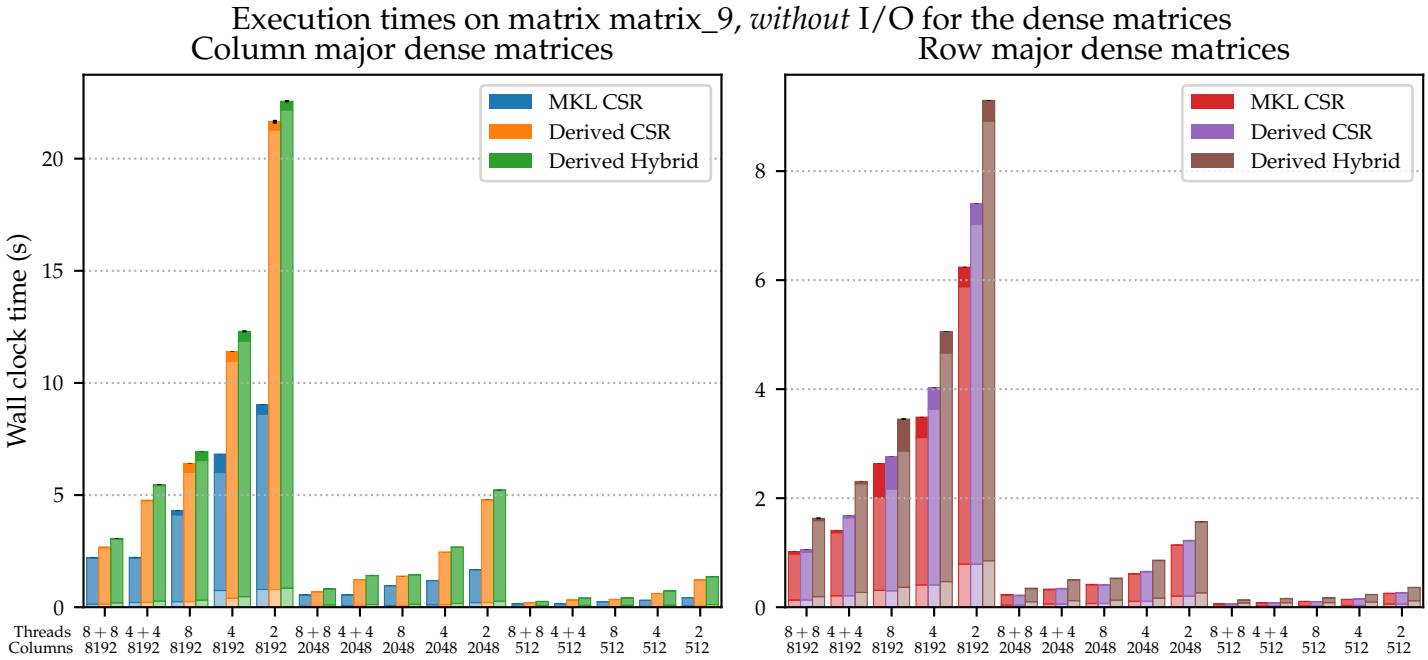


Figure 6.12: Performance on the matrix_9 matrix where the dense matrices are not memory mapped.

and atmosmodl-3. For atmosmodl-2 we perform a two-duplication, in which every nonzero becomes four nonzeros in a square shape (with the same value), as visualized in Figure 6.13. For atmosmodl-3 each nonzero becomes nine nonzeros. Whenever performing a X-duplication, the matrix its width and height thus get multiplied by X, while the total number of nonzeros gets multiplied by X^2.

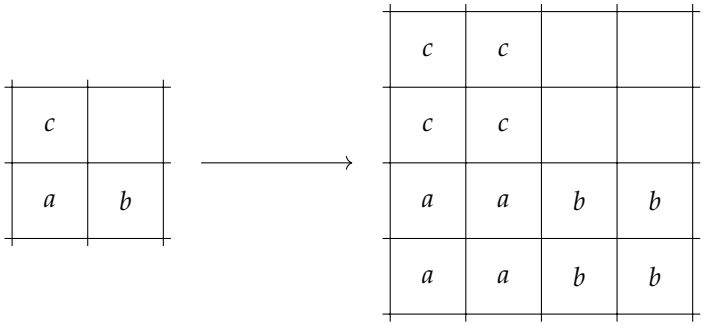


Figure 6.13: A two-duplication of the matrix elements visualized.

These duplications of matrices will result in new, not as dense diagonal bands appearing, next to the originally dense diagonal bands. For example, in atmosmodl, the 39204 band is nearly completely dense. In atmosmodl-2 this diagonal band is shifted over to 78408 and is still as dense. However, the two diagonal bands next to it, at 78407 and 78409, will be half as dense. Table 6.2

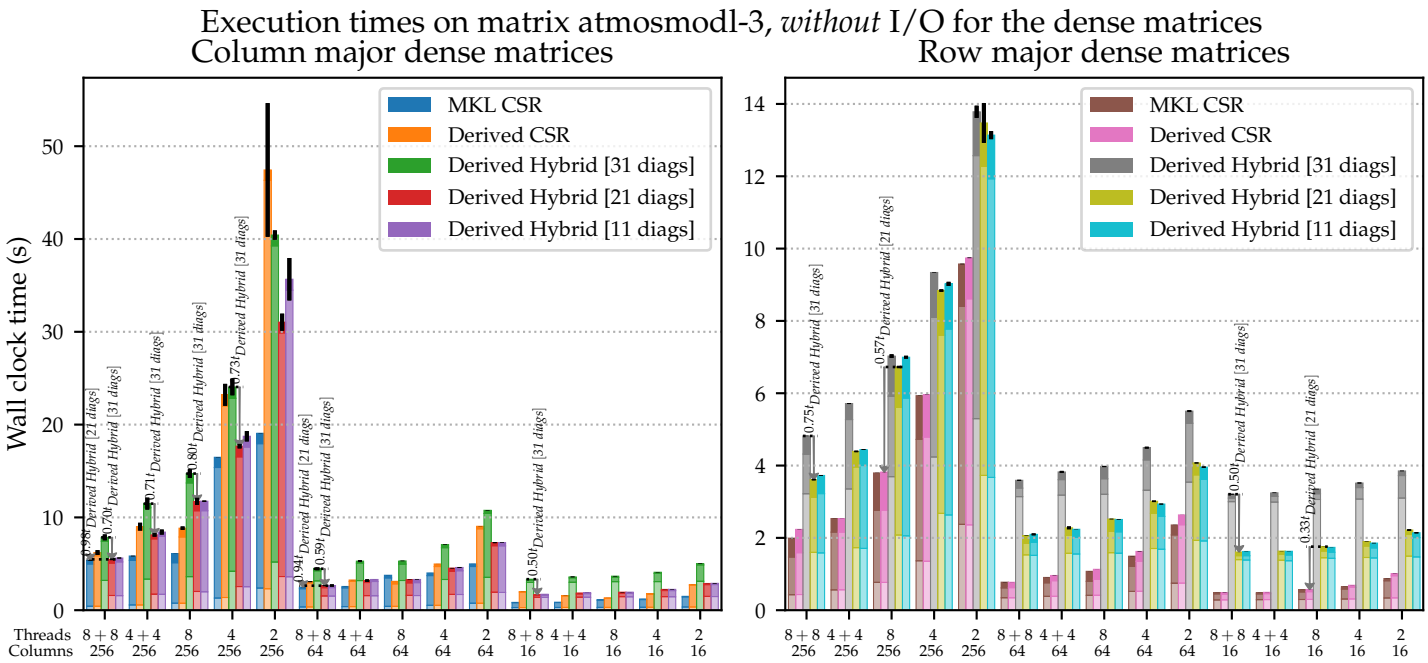


Figure 6.14: Performance on the atmosmodl-3 matrix where the dense matrices are not memory mapped.

shows some properties of these derived matrices.

Results First, we will look at the results of *atmosmodl-3*. Figure 6.14 displays the performance of the derived implementations on this matrix. We consider three variants of the Derived Hybrid implementation: one which considers all 31 diagonals, one only considering the 21 densest diagonals and finally one considering the 11 very dense diagonals only. Variants with fewer diagonal bands have nonzeros stored in the secondary CSR format. This allows us to analyze the performance effect of including sparse diagonal bands in the dense diagonal storage format (introducing explicit zeros as a result). Note how, when we use column major dense matrices, the Derived Hybrid format performs best when we remove the boundary diagonal bands from the diagonal matrix, i.e. reducing the amount of stored diagonal bands in the diagonal format from 31 to 21: these outer bands are only $\frac{1}{3}$ full, only having between 0.6–0.8 times the original runtime. For example, in the 256 columns, $8 + 8$ threads scenario this nearly matches MKL CSR performance. For such sparse bands it is thus better to store the data in the secondary CSR format. Removing the second-outer band as well, which here is about $\frac{2}{3}$ full, usually leads to a slight reduction in performance. For the row major dense matrix cases it is similar: considering 21 diagonals leads to the best performance in most cases, up to two times as fast in for example the 16 columns, $8 + 8$ threads scenario. Derived CSR and MKL CSR outperform the Derived Hybrid variants here, though. Derived CSR is 3 times as fast as Derived Hybrid in the 16 columns, 8 threads scenario, for example.

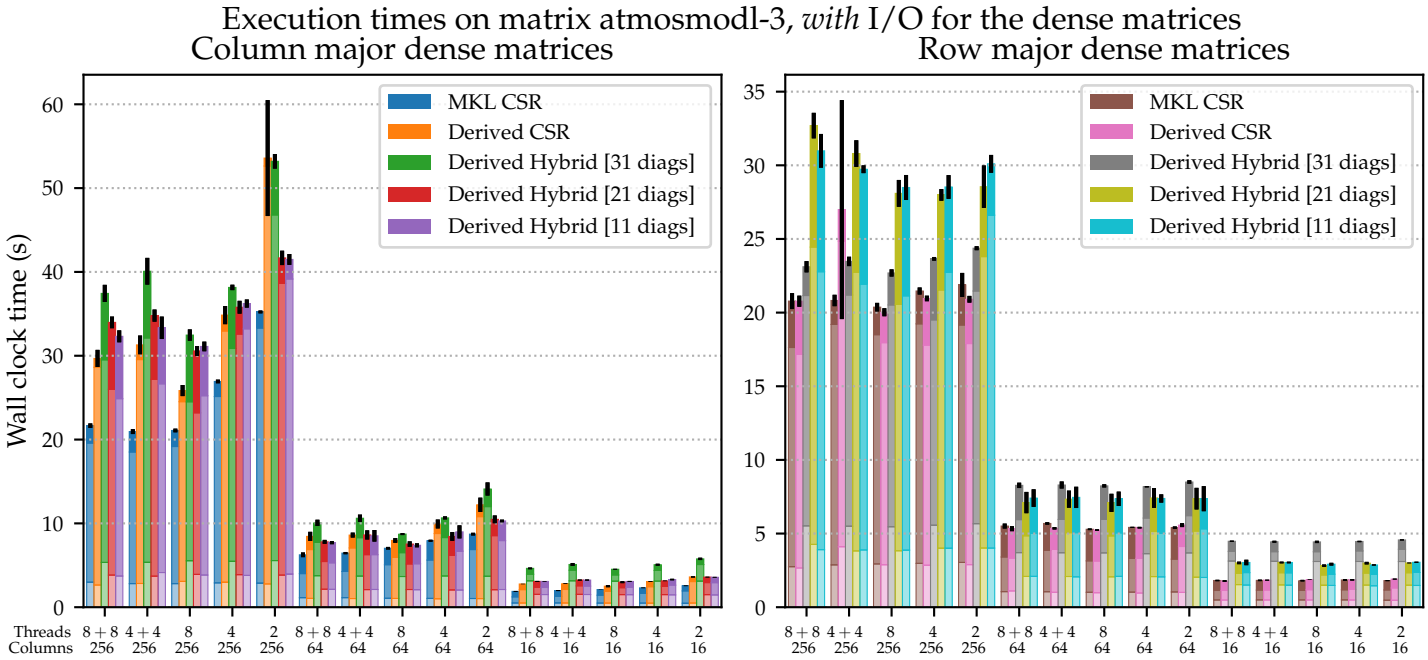


Figure 6.15: Performance on the atmosmodl-3 matrix where the dense matrices are also memory mapped.

Figure 6.15 also considers memory mapped dense matrices. Interestingly, for the row major variants with large dense matrices, the variant in which we consider all 31 diagonal bands for the Diagonal Hybrid performs significantly better compared to the other Diagonal Hybrid formats, possibly due to the I/O overhead outweighing the relatively large number of wasted operations on explicit zeros. The other implementations also have to perform a second pass over the dense matrix data to process the secondary CSR data, which is no longer bounded by input load times, as the first pass already caused all data to be loaded into memory: for the implementations where secondary CSR data is stored the termination time (upper bar) can be seen to be much higher. Additionally, when the dense matrices are small, the conversion to the Derived Hybrid format can cause a much larger overhead when all data is converted to the diagonal storage format.

Figure 6.16 shows the results on the *atmosmodl-2* matrix. From the column major cases it becomes clear that including diagonals which are only 50% dense reduces performance marginally in a few scenarios, such as in the 256 columns, 8 threads scenario. In many other scenarios they perform about even, like in the 256 columns, 2 threads scenario. Interestingly, in the row major cases considering just 9 diagonals is significantly slower than considering all 19 diagonals. It is, for example, 32% slower in the 256 columns, 8 threads scenario. This is in contrast with the performance on the atmosmodl-3 matrix, where considering just a subset of the diagonals performs better than considering all of the diagonals. This may be because the outer diagonals are still 50% dense, unlike on the atmosmodl-3 matrix where these bands are only 33% dense. Improved caching may also have an effect, as only 19 elements are accessed each itera-

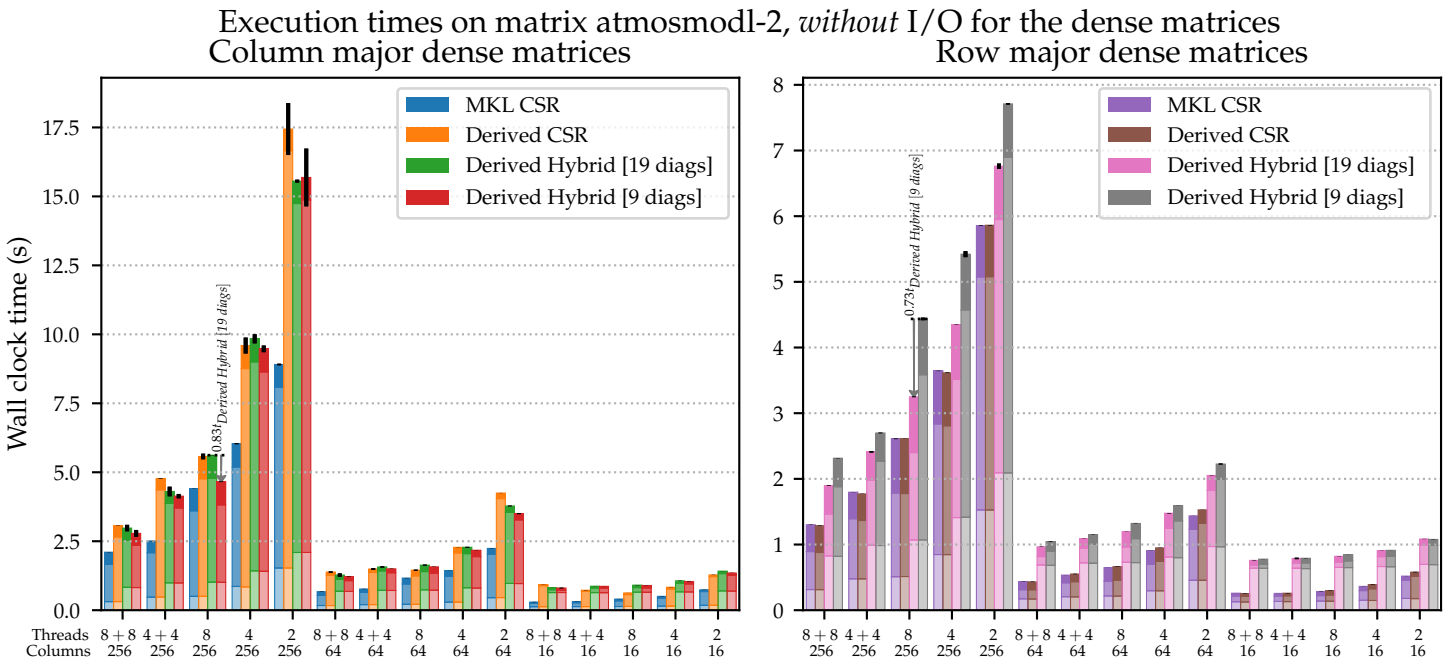


Figure 6.16: Performance on the atmosmodl-2 matrix where the dense matrices are not memory mapped.

tion instead of not 31. Considering 19 bands on the atmosmodl-2 matrix will also often have three diagonal bands next to one another, whereas considering 31 bands on the atmosmodl-3 matrix will often have five diagonal bands next to one another. This may lead to improved vectorization, as at most 4 packed doubles can be multiplied at once on Intel Xeon E5-2630 v3 processors.

In general there are various scenarios in which the Derived Hybrid implementation outperforms Derived CSR, even though the Derived Hybrid implementation has a longer initialization time and needs to perform two passes over the dense matrix data. In some cases that overhead can cause Derived CSR to outperform Derived Hybrid, though. This can also happen when the diagonals stored in Derived Hybrid are not very dense: about half of the elements in a diagonal band should be nonzero for it to be worth considering the Derived Hybrid implementation when the dense matrix data is in-memory. When dense matrices are loaded from files additional nonzeros do not negatively affect performance as much. In fact, avoiding the second CSR pass entirely in such cases can improve performance. Generally, the Derived Hybrid format is much less effective when the dense matrices are row major compared to column major dense matrices.

The derived implementations in some cases outperform MKL CSR. There is no clear pattern in which cases this occurs, but it is interesting to note that MKL does not always perform well in especially NUMA scenarios. In many cases the Derived Hybrid implementation comes closer to the MKL CSR performance.

Chapter 7

Conclusions

Within this section we conclude the project. Additionally some future work will be suggested.

7.1 Summary

In this thesis we have described libtupl: a library to optimize tUPL programs. It can perform data structure optimization by performing various simple transformations, constructing a transformation tree in the process.

libtupl also keeps track of transformations that have to be done on the algorithm its input and output to make it compatible with the automatically generated data structures, generating load and unload routines automatically. Two ways have been explored to keep track of such data transformations: one in which each transformation is described through a coroutine generator and one in which we construct a higher-level I/O transformation tree.

We have seen that the coroutine generator approach has the disadvantage that, for example, the input may be read multiple times unnecessarily. This is because each generated data structure will try to fill the data of this data structure completely on its own: if multiple data structures are based on the same input, this leads to the input being read multiple times during the load phase, once for each data structure. The coroutine generator approach does have a lot of control over *merging* multiple input streams into one, though.

I/O transformation trees do not explicitly define how the tree is actually converted to imperative code, but we generally convert them in a way such that the input *sources* push data towards the generated data structures (the coroutine generator approach always does this the other way around, *pulling* data from input sources). This avoids problems where the input may be read multiple times and has the additional advantage that I/O transformation trees are much easier to transform further, yielding more optimized load and unload routines.

Additionally, we have developed Tython, an extension of Python 3, to serve as a front-end for libtupl. Tython code can contain tUPL code blocks in addition to standard Python code. These tUPL code blocks can be optimized by the Tython compiler using libtupl. Tython outputs compiled Python files which automatically invoke the optimized implementations, which can then be exe-

cuted using the standard CPython interpreter or imported from any Python or Tython script. Any Python project can thus easily integrate tUPL code using Tython.

Finally, to illustrate the effectiveness of the data structures that are automatically derived using the presented framework, we have performed various experiments on derived SpMM implementations using an extended version of tUPL: also considering parallelization and runtime I/O. We have seen that Derived Hybrid implementations (hybrids of CSR and compressed diagonal storage) of sparse matrix-dense matrix multiplication from a simple input specification can result in performance that, for various sparse matrix structures, significantly improves performance over the simpler Derived CSR implementations, sometimes being twice as fast. Such Derived Hybrid implementations can also be very competitive with state-of-the-art hand-optimized implementations, in certain scenarios being about 50% faster than MKL CSR. These differences are mostly noticeable with column major dense input matrices; with row major dense input matrices Derived Hybrid implementations are generally slower than Derived CSR and MKL CSR. Performance differences tends to be more apparent in NUMA scenarios. Additionally, the Derived Hybrid format performs best when the stored diagonal bands primarily consist of nonzeros: if more than about half of the values in a band consists of zeros it is suggested to store that band in the secondary CSR format instead.

7.2 Future work for the derivation of data structures

We have measured the performance of various SpMM implementations we have derived in the experiments in Section 6. Some suggestions for future research are enumerated in this section to potentially improve the performance even further.

MKL CSR tends to perform worse in NUMA configurations. It could be interesting to investigate if it is possible to automatically optimize the generated implementations for NUMA configurations, as currently we re-use the same implementation for each thread configuration. Optimized implementations for other parameters, such as dense matrix column count, may be interesting to explore too. It might be beneficial to use a different iteration order when there are very few dense matrix columns, for example.

Storing diagonals with few explicit zeros in a dense data structure can yield improved performance. Some matrices have other dense structures, such as dense rows, columns or blocks. It could be interesting to construct hybrid data structures with such dense rows, columns or blocks too. For the Chebychev4 matrix a triple-hybrid may be interesting, for example: there are dense diagonals, dense rows and parts without a dense structure.

There are a wide range of small variations that can be applied on the currently derived implementations. Investigating the value of transformations such as loop interchange, loop blocking or permuting dimensions of multidimensional arrays in greater detail could help optimize performance further.

It would also be interesting to use the tUPL framework to derive implementations for other sparse matrix algorithms, such as triangular solve. Algorithms like LU decomposition present additional challenges: the sparse structure becomes writable. Finding techniques to automatically minimize problems such

as fill-in would be valuable to research.

Another interesting case could be to derive efficient indexing data structures to perform certain data queries quickly. For example, we could specify a query that finds all distinct persons that are at least 30 years old that own a boat that has a value of at least 500 using the tUPL code in Listing 7.1. This is similar to the work done by Dr. K.F.D. Rietveld in [10], where he transformed SQL queries into specifications for the forelem framework.

```
1 forelem boat in Boats where boat.value >= 500:
2   forelem person in Persons where person.age >= 30 and
      boat.owner_id == person.id:
3     ExpensiveBoatOwners.add(person) # ExpensiveBoatOwners
      is a tuplespace (set) too
```

Listing 7.1: Data query example in tUPL.

7.3 Future work for the transformation framework & implementation

Tython and libtupl can be extended with a large amount of additional features to improve the compiler. In this section a number of suggestions are enumerated.

Tython and libtupl do not currently support fully automatic exploration: the user has to input a number of transformations to be applied. The compiler can be extended to support automatically iterating the possible implementations (in some sort of a guided way using heuristics), trying to automatically find implementations that perform well on some test inputs. This can be taken a step further by allowing generation of new optimized algorithms during the runtime of an algorithm, similar to how JIT compilers work.

It could be considered to merge the ReservoirMaterializationPass and the NStarMaterializationPass. Currently, in order to materialize a reservoir one must always execute both of these passes and additional passes in-between the two to later influence the NStarMaterializationPass are currently not possible. The NStarMaterializationPass practically finishes the partly-completed materialization of the reservoir.

Currently the StructureJaggedSplittingPass does two things: split the structure (tuple) into multiple parts and convert the subscriptable in which these structures are located to a subscriptable of subscriptables (of subscriptables...) in which the split structures are located (i.e. a jagged subscriptable). Although this is a highly useful transformation, it could be split into two independent passes performing simpler operations instead, where one splits the structure and the other cuts a subscriptable at a certain offset, transforming it into a jagged subscriptable.

The I/O transformation graph is currently always compiled to a tUPL AST top-down: I/O nodes do not have control of when they receive input data like the coroutines have with the generator I/O approach. A hybrid of the two I/O approaches could be valuable in some cases too. For example, a "Merge" node could then try to partially consume data from the shared space input socket until it finds the input that is to-be-merged. I/O nodes could then support to be

compiled like a generator (child node needs to be request this generator to produce data) or traditionally (outputs data/invokes children as data is pushed to the node).

Whenever the boundaries of data structures are not known in advance a lot of reallocations may be required as new data is loaded, like when concretizing into a dense array. These reallocations can be very expensive. Additional input passes could be generated to try to determine data structure index boundaries before allocating these structures if they are not known beforehand, which can be much cheaper than reallocating the data every time to accommodate for larger sizes. If the input data is ordered in some way, data structure boundaries may be able to be determined much faster too. Note that more than one additional pass may be required to avoid reallocations. For example, `CPNZ` may be a $(max(t.row) + 1) \times max(CPNZ\_len)$ dense array. Here the data structure `CPNZ_len` is a one-dimensional array with $max(t.row) + 1$ values. Using an initial pass only $max(t.row)$ can be determined without reallocations: only then we know the dimensions of `CPNZ_len`. The second pass allows filling `CPNZ_len` and computing $max(CPNZ\_len)$ as well without reallocations. A third pass can then finally fill `CPNZ` without reallocations, now that the dimensions of that two-dimensional dense array are known.

In Section 5.3 we looked at runtime I/O and coined the "View" node. This concept can be extended to support *random access iterators* instead of just a pointer to raw memory. Such a random access iterator can be indexed directly to obtain a random value, just like accessing a random value in raw memory using the "View" node, but it can also be iterated through sequentially, like an **`InStream`** can. This allows a compilation flow where runtime I/O is considered, i.e. where each data structure access is directly mapped to this random access iterator, but also still allows generating optimized data structures.

An alternative approach for runtime I/O is embedding the streaming operations (i.e. iterating through **`InStream`** or emitting data to **`OutStream`**) right into the algorithm itself, also eliminating the need for a generated concretized data structure. This would also allow, for example, reading data from a file without requiring a full random access iterator (or a pointer to raw memory) during the algorithm itself.

Bibliography

[1] H. M. Aktulga et al. "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations." In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 1213–1222.

[2] H. Bal et al. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term." In: *Computer* 49.5 (2016), pp. 54–63. ISSN: 0018-9162. DOI: 10.1109/MC.2016.127. URL: doi.ieeecomputersociety.org/10.1109/MC.2016.127.

[3] N. Bell and M. Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors." In: *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM. 2009, p. 18.

[4] *Boost.Python — 1.66.0*. https://www.boost.org/doc/libs/1_66_0/libs/python/doc/html/index.html. Accessed: 2019-01-04.

[5] T. A. Davis and Y. Hu. "The University of Florida sparse matrix collection." In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), p. 1.

[6] *GitHub — The Python programming language*. https://github.com/python/cpython. Accessed: 2019-01-04.

[7] A. Hommelberg. *Using the Forelem Framework to Express and Optimize K-means Clustering*. https://theses.liacs.nl/pdf/AnneHommelberg3.pdf. Accessed: 2019-03-28. 2017.

[8] A. Hommelberg, K. F. D. Rietveld, and H. A. G. Wijshoff. "Abstracting Parallel Program Specification: A Case Study on K-means Clustering." In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. CF '19. Alghero, Italy: ACM, 2019, pp. 279–282. ISBN: 978-1-4503-6685-4. DOI: 10.1145/3310273.3322828. URL: http://doi.acm.org/10.1145/3310273.3322828.

[9] *Intel Math Kernel Library Developer Reference*. https://software.intel.com/en-us/articles/mkl-reference-manual. Accessed: 2019-01-04.

[10] K. F. D. Rietveld. *A versatile tuple-based optimization framework*. Leiden Institute for Advanced Computer Science (LIACS), Faculty of Science, Leiden University, 2014.

[11] K. F. D. Rietveld and H. A. G. Wijshoff. "Forelem: A versatile optimization framework for tuple-based computations." In: *CPC 2013: 17th Workshop on Compilers for Parallel Computing*. Citeseer. 2013.

[12] K. F. D. Rietveld and H. A. G. Wijshoff. "Optimizing sparse matrix computations through compiler-assisted programming." In: *Proceedings of the ACM International Conference on Computing Frontiers*. ACM. 2016, pp. 100–109.

[13] K. F. D. Rietveld and H. A. G. Wijshoff. "Towards a new tuple-based programming paradigm for expressing and optimizing irregular parallel computations." In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM. 2014, p. 16.

[14] E. Saule, K. Kaya, and Ü. V. Çatalyürek. "Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi." In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2013, pp. 559–570.

[15] B. van Strien, K. F. D. Rietveld, and H. A. G. Wijshoff. "Deriving highly efficient implementations of parallel pagerank." In: *Parallel Processing Workshops (ICPPW), 2017 46th International Conference on*. IEEE. 2017, pp. 95–102.

[16] R. Vuduc, J. W. Demmel, and K. A. Yelick. "OSKI: A library of automatically tuned sparse matrix kernels." In: *Journal of Physics: Conference Series*. Vol. 16. 1. IOP Publishing. 2005, p. 521.

[17] D. Zheng et al. "Semi-external memory sparse matrix multiplication for billion-node graphs." In: *IEEE Transactions on Parallel and Distributed Systems* 28.5 (2017), pp. 1470–1483.

[18] *ast — Abstract Syntax Trees — Python 3.7.2 documentation*. `https://docs.python.org/3/library/ast.html`. Accessed: 2019-01-04.

Appendix A

Transformation passes

Various passes have been implemented that affect the algorithm its code specification. This appendix contains a more detailed description of these implemented passes.

A.1 EncapsulationPass

If we iterate through a tuplespace its possible field values, like "`forelem row in NZ.row`", the EncapsulationPass can try to substitute this with a range iterator, like "`forelem row in [0, max(NZ.row)]`". The advantage of such a range iterator is that it can in the end be concretized to a plain `for`-loop and we do not have to materialize the sequence of possible field values (i.e. `NZ.row`) at all if we also apply the AggregateReservoirPass, which we will see later. This is only allowed if the iterator value (i.e. `row`) is solely used in equals conditions of loops over the same tuplespace (i.e. `NZ`) on the same field (i.e. `nz.row == row`) or inside of loop bodies of such loops. This can, for example, transform the code in Listing A.1 into the code in Listing A.2.

```
1 forelem row in NZ.row:
2   forelem nz in NZ where nz.row == row:
3     C[row] += A[row, nz.col] * B[nz.col]
```

Listing A.1: Example scenario on which the EncapsulationPass can be applied.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem nz in NZ where nz.row == row:
3     C[row] += A[row, nz.col] * B[nz.col]
```

Listing A.2: Resulting code after applying the EncapsulationPass.

In Listing A.3 the EncapsulationPass cannot be applied: transforming `NZ.row` into `[0, max(NZ.row)]` could reset additional `C[row]` values to zero for values of `row` that are not also in `NZ.row`. Loop splitting could be applied here, splitting the reset off of the actual sparse matrix-vector multiplication, after which the EncapsulationPass can be applied on the second loop.

```
1 forelem row in NZ.row:
2   C[row] = 0
3   forelem nz in NZ where nz.row == row:
4     C[row] += A[row, nz.col] * B[nz.col]
```

Listing A.3: A situation in which the EncapsulationPass cannot be applied on the outer loop.

A.2 AggregateReservoirPass

The AggregateReservoirPass can be used to transform aggregations of tuplespace fields into a single scalar value. This can allow us to, for example, transform the code in Listing A.4 into Listing A.5.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem nz in NZ where nz.row == row:
3     C[row] += A[row, nz.col] * B[nz.col]
```

Listing A.4: Example scenario on which the AggregateReservoirPass can be applied.

```
1 forelem row in [0, max_NZ_row]:
2   forelem nz in NZ where nz.row == row:
3     C[row] += A[row, nz.col] * B[nz.col]
```

Listing A.5: Resulting code after applying the AggregateReservoirPass.

The computation of the value of `max_NZ_row` is, as a result of applying this pass, delegated to a value that is loaded in load routine, which we will look at in Section C.2. Key advantage of this is that `max(NZ.row)` no longer has to be computed before running the loop: this value can be computed while reading the input. In other words, we do not have to iterate `NZ.row` another time just to determine the maximum value if we just do it during the load routine.

A.3 LocalizationPass

The LocalizationPass can copy the values of a shared space into a new field of all tuples in a tuplespace (i.e. *localize* the values in the shared space). This typically eliminates the shared space completely and the values of the shared space are then retrieved from tuple fields directly: the shared space is practically merged into the tuplespace. The only requirement is that this shared space is solely indexed using tuple fields from tuples in the tuplespace into which we are going to merge the shared space. For example, if `A` is accessed through `A[nz.row, nz.col]` only, with `nz` being tuples from `NZ`, then we can merge safely. If `A[row, nz.col]` were used to access the shared space, substituting `A[row, nz.col]` with a merged value would not necessarily be safe. Additionally, we typically do not merge shared spaces into the tuplespace if the shared space is writable to avoid complications.

The LocalizationPass can be used to, for example, merge shared space `A` into tuplespace `NZ` in Listing A.6. This will produce the code in Listing A.7. Tuplespace `NZ` now thus contains tuples $\langle row, col, merged\_value \rangle$, where `merged_value = A[row, col]` for each tuple. We call this `[row, col]` the *query* of the LocalizationPass. In other words, for each tuple `t` in the tuplespace `NZ` we look up the value in the shared space that is to-be-merged using the query. That value thus is `A[t.row, t.col]` here.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem nz in NZ where nz.row == row:
3     C[nz.row] += A[nz.row, nz.col] * B[nz.col]
```

Listing A.6: Example scenario on which the LocalizationPass can be applied.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem nz in NZ where nz.row == row:
3     C[nz.row] += nz.merged_value * B[nz.col]
```

Listing A.7: Resulting code after applying the LocalizationPass.

Note that applying the LocalizationPass does not necessarily eliminate the shared space entirely. For example, the specification in Listing A.8 accesses shared space `A` in two locations, but with different address functions. We can, in this example, only replace shared space accesses with the merged value in the tuple if the address function output matches the merge query. After a single application of the LocalizationPass with query `[row, col]`, as shown in Listing A.9 the tuple has been changed to $\langle row, col, merged\_value \rangle$, where *merged_value* contains `merged_value = A[row, col]`.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem nz in NZ where nz.row == row:
3     C[nz.row] += A[nz.row, nz.col] + A[nz.col, nz.row]
```

Listing A.8: A scenario in which a single application of the LocalizationPass does not eliminate the shared space.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem nz in NZ where nz.row == row:
3     C[nz.row] += nz.merged_value + A[nz.col, nz.row]
```

Listing A.9: A single application of the LocalizationPass did not eliminate the shared space.

Additionally applying the LocalizationPass a second time with query `[col, row]` will merge the shared space access with the other address function into the tuplespace, as shown in Listing A.10. The tuples will then get a second merged value: `merged_value_2 = A[col, row]`. The shared space `A` now is fully eliminated.

```
forelem row in [0, max(NZ.row)]:
  forelem nz in NZ where nz.row == row:
    C[nz.row] += nz.merged_value + nz.merged_value_2
```

Listing A.10: Two applications of the LocalizationPass do eliminate the shared space entirely.

A.4 QueryForwardSubstitutionPass

The QueryForwardSubstitutionPass tries to perform forward substitution based on the query (i.e. condition) of a loop. For example, in Listing A.11, we have the query `where nz.row == row` on tuplespace `NZ`. Clearly, `nz.row` can be substituted with just `row` in the inner loop body, which leads to Listing A.12.

```
forelem row in [0, max(NZ.row)]:
  forelem nz in NZ where nz.row == row:
    C[nz.row] += A[nz.row, nz.col] + A[nz.col, nz.row]
```

Listing A.11: A scenario where applying the QueryForwardSubstitutionPass is beneficial.

```
forelem row in [0, max(NZ.row)]:
  forelem nz in NZ where nz.row == row:
    C[row] += A[row, nz.col] + A[nz.col, row]
```

Listing A.12: The resulting code after applying the QueryForwardSubstitutionPass.

Note that after performing the QueryForwardSubstitutionPass in this example scenario we no longer read field *row* from any tuple in tuplespace `NZ`. Later on in the data structure optimization process the HorizontalIterationSpaceReductionPass can be applied to reduce the amount of data we have to store in the tuplespace. In Section A.8 we will look at this pass.

A.5 ReservoirMaterializationPass

The ReservoirMaterializationPass can transform a tuplespace into a *materialized* tuplespace. In other words, we now assign storage indices to each tuple in the tuplespace. We do not define an actual data structure in which the tuplespace data is truly stored yet, though.

This entails creating a new *subscriptable* symbol in which the tuple data can be looked up (virtually through the assigned storage indices). We will denote such derived symbols by prefixing the name with a `P`. Their type is always a `Subscriptable`. Looping structures and tuple accesses are then transformed to use this subscriptable instead of the non-materialized tuplespace.

We will illustrate what this pass truly does using an example. Listing A.13 can be transformed into Listing A.14 using the ReservoirMaterializationPass, materializing the `NZ` tuplespace. The subscriptable `PNZ` is created in which

data can be looked up. The *equals query* from the original `forelem` loop (i.e. `NZ where nz.row == row`) is here moved into first dimension of the `PNZ` subscriptable. Additionally, the subscriptable gets an additional dimension in which we pass the new iterator value `k`: an offset. Generally speaking the amount of equal queries the loop has, plus one dimension for the offset value `k`, will be the total number of index dimensions of the resulting subscriptable. We also extend each tuple in the tuplespace with this new offset value `k`.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem nz in NZ where nz.row == row:
3     C[row] += A[row, nz.col] * B[nz.col]
```

Listing A.13: Example scenario on which the ReservoirMaterializationPass can be applied.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem k in N*:
3     C[row] += A[row, PNZ[row, k].col] * B[PNZ[row, k].col
    ]
```

Listing A.14: Resulting code after applying the ReservoirMaterializationPass.

If n tuples match the query `NZ where nz.row == row`, then each of those n tuples has some (typically distinct) offset value k so that the tuple can be retrieved through `PNZ[row, k]`. The exact offset value k assigned to each tuple depends on `N*`: a tuplespace that will iterate the offset values (potentially different offset values depending on the outer loop). How `N*` (and indirectly `PNZ`) will look will be determined through the NStarMaterializationPass.

A.6 NStarMaterializationPass

The NStarMaterializationPass can be seen as a continuation of the ReservoirMaterializationPass. The ReservoirMaterializationPass produced a `N*` reservoir containing the offsets at which each tuple is stored. The NStarMaterializationPass materializes this reservoir and determines what this reservoir actually looks like.

Materializing the `N*` reservoir from Listing A.14 could yield the code in Listing A.15. Here we assign each tuple an offset value between `0` and `PNZ_len[row]-1` and the inner loop iterates a varying number of times depending on `row`. `PNZ_len[row]` then contains an integer indicating the number of tuples that originally matched the query `where t.row == row` for that `row`. This technique of materializing the `N*` reservoir can always be applied. Note how assigning the offset values of each tuple will define what `PNZ` will actually look like, so `PNZ` also becomes `PNZ'` (they are practically the same in the code, but we know more information about the `PNZ'` subscriptable).

```
forelem row in [0, max(NZ.row)]:
  forelem k in [0, PNZ_len[row]-1]:
    C[row] += A[row, PNZ'[row, k].col] * B[PNZ'[row, k].
   col]
```

Listing A.15: Possible result code after applying the NStarMaterializationPass.

Alternatively, one could materialize the `N*` reservoir to the specification in Listing A.16. This is only possible if either the same number of tuples match the query `where t.row == row` for each `row` iterated (in the above example for all values of `row` in the range `[0, max(NZ.row)]`, because previously the EncapsulationPass was applied), or if *padding tuples* could be inserted into the materialized reservoir `PNZ` such that executing these padding tuples does not change the outcome of the algorithm. This may require introducing new values into shared spaces as well to prevent changing the behavior of the algorithm. For example, the tuple values $\langle row, max(NZ.col) + 1 \rangle$ could be used as padding. Iterating this tuple still matches the original query `where t.row == row`, but no other tuple uses this *col* value. Do note that we now have to introduce values at `A[row, max(NZ.col)+1]` and `B[max(NZ.col)+1]`. Storing 0 at these locations will ensure executing these padding tuples are a no-op, so in this case it is possible to use this alternative `N*` materialization. Because we also change the data in `A` and `B`, we create derived shared spaces `A'` and `B'`.

```
forelem row in [0, max(NZ.row)]:
  forelem k in [min(PNZ_len), max(PNZ_len)]:
    C[row] += A'[row, PNZ'[row, k].col] * B'[PNZ'[row, k
   ].col]
```

Listing A.16: Possible alternative result code after applying the NStarMaterializationPass.

Generally a key advantage of the latter `N*` materialization is that the inner loop iterator bounds no longer depend on the outer loop iterator `row`. This allows us to, further down the line, perform a loop interchange. However, detecting when this kind of `N*` materialization can be applied safely can be very complicated.

A.7 SharedSpaceMaterializationPass

The SharedSpaceMaterializationPass can materialize a shared space. Similarly to the ReservoirMaterializationPass we now generate a subscriptable symbol for the shared space. The indices for this subscriptable symbol can directly be derived from the address function used to access the shared space.

In the case of Listing A.17 we can apply the SharedSpaceMaterializationPass three times to materialized all three shared spaces. This transforms the specification into Listing A.18. Although this visually does not change much because we use shorthand notations for shared space indexing, omitting the explicit address function, the address function is now converted into code to index the subscriptables directly: no address function exists anymore (i.e. the type of `A` is a `SharedSpace` while that of `PA` is a `Subscriptable`).

```
forelem row in [0, max(NZ.row)]:
  forelem k in [0, PNZ_len[row]-1]:
    C[row] += A[row, PNZ[row, k].col] * B[PNZ[row, k].col
    ]
```

Listing A.17: Example scenario on which the SharedSpaceMaterializationPass can be applied.

```
forelem row in [0, max(NZ.row)]:
  forelem k in [0, PNZ_len[row]-1]:
    PC[row] += PA[row, PNZ[row, k].col] * PB[PNZ[row, k].
    col]
```

Listing A.18: Resulting code after applying the SharedSpaceMaterializationPass.

A.8 HorizontalIterationSpaceReductionPass

If subscriptables contain tuples with tuple fields which are not accessed anywhere, the HorizontalIterationSpaceReductionPass can eliminate that field from the tuples, reducing the width of each tuple. For example, in Listing A.19, `PNZ` is the direct result of reservoir materialization and contains tuples $\langle row, col, k \rangle$. We only read the field `col` from tuples in this subscriptable, so the HorizontalIterationSpaceReductionPass will create a new subscriptable `PNZ'` in which we only store tuples with a single field: $\langle col \rangle$.

```
forelem row in [0, max(NZ.row)]:
  forelem k in [0, PNZ_len[row]-1]:
    PC[row] += PA[row, PNZ[row, k].col] * PB[PNZ[row, k].
    col]
```

Listing A.19: Example scenario on which the HorizontalIterationSpaceReductionPass can be applied.

A.9 DelocalizationPass

The DelocalizationPass splits accesses to a subscriptable symbol containing tuples off into a duplicate of that subscriptable symbol for a certain set of tuple fields. For example, Listing A.20 can be transformed into Listing A.21 using a DelocalizationPass, splitting accesses on tuple fields $\{col\}$ off into `PNZ_deloc`. Typically the HorizontalIterationSpaceReductionPass is executed afterwards to try to reduce both subscriptables their containing tuples into smaller tuples to not duplicate the tuple data.

```
forelem row in [0, max(NZ.row)]:
  forelem k in [0, PNZ_len[row]-1]:
    C[row] += PNZ[row, k].merged_value * B[PNZ[row, k].
    col]
```

Listing A.20: Example scenario on which the DelocalizationPass can be applied.

```
forelem row in [0, max(NZ.row)]:
  forelem k in [0, PNZ_len[row]-1]:
    C[row] += PNZ[row, k].merged_value * B[PNZ_deloc[row,
     k].col]
```

Listing A.21: Resulting code after applying the DelocalizationPass.

A.10 StructureJaggedSplittingPass

The StructureJaggedSplittingPass is similar to the DelocalizationPass, but has more capabilities. It can transform expressions in the form of `Subscriptable[_, _, _].field` into, for example, `Subscriptable[_, _].fieldgroup[_].field`, where fieldgroup is an array of tuple types containing at least `field`, but potentially additional tuple fields (i.e. a jagged structure). Aside from the expression the pass is applied on, the *offset* at which we split the subscriptable (in the previous example 2) and the previously described *field groups* are parameters of this pass. The offset must be between 0 and the number of dimensions of the subscriptable (exclusive upper bound).

We will illustrate the exact behavior through an example. Listing A.22 shows a possible scenario. We apply the StructureJaggedSplittingPass on expression `PNZ` with offset 1 and groups $[\langle row \rangle, \langle col, val \rangle]$. This leads to the code shown in Listing A.23. Note how we now have a 1D subscriptable `PNZ'` (indexed with `row`), with at each location a tuple $\langle \_\_row, \_\_col\_val \rangle$. At each tuple location $\_\_row$ we have another 1D subscriptable storing tuples $\langle row \rangle$, where row is a scalar value. Similarly, at each $\_\_col\_val$ a 1D subscriptable is stored containing tuples $\langle col, val \rangle$.

```
forelem row in [0, max(NZ.row)]:
  forelem k in [0, PNZ_len[row]-1]:
    PC[PNZ[row, k].row] += PNZ[row, k].val * PB[PNZ[row,
     k].col]
```

Listing A.22: Example scenario on which the StructureJaggedSplittingPass can be applied.

```
forelem row in [0, max(NZ.row)]:
  forelem k in [0, PNZ_len[row]-1]:
    PC[PNZ'[row].__row[k].row] += PNZ'[row].__col_val[k].
     val * PB[PNZ'[row].__col_val[k].col]
```

Listing A.23: Possible result after applying the StructureJaggedSplittingPass on PNZ with offset 1.

Note how applying the StructureJaggedSplittingPass on Listing A.22 with offset 0 and groups $[\langle col, val \rangle, \langle row \rangle]$ is roughly equivalent to a DelocalizationPass plus HorizontalSpaceReductionPasses. The only difference is that the root

is now a symbol PNZ' containing a struct of two elements rather than two entirely disjoint symbols, as shown in Listing A.24.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem k in [0, PNZ_len[row]-1]:
3     PC[PNZ'.__row[row, k].row] += PNZ'.__col_val[row, k].
    val * PB[PNZ'.__col_val[row, k].col]
```

Listing A.24: Possible result after applying the StructureJaggedSplittingPass on PNZ with offset 0.

Note that it can be beneficial to apply this pass multiple times with different parameters to construct more advanced data structures. For example, we can apply the pass on `PNZ'.__col_val[_, _]` with offset 1 with groups $[\langle row \rangle, \langle col \rangle]$ on the code in Listing A.24. The result is shown in Listing A.25.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem k in [0, PNZ_len[row]-1]:
3     PC[PNZ''.__row[row, k].row] += PNZ''.__col_val[row].
    __val[k].val * PB[PNZ''.__col_val[row].__col[k].col]
```

Listing A.25: Possible result after applying the StructureJaggedSplittingPass on `PNZ'.__col_val[_, _]` with offset 1.

It is also reasonable to just use a single group containing all tuple fields whenever applying the pass at a nonzero offset. For example, applying the pass on Listing A.22 again with offset 1, but groups $[\langle row, col, val \rangle]$ will yield the code in Listing A.26. Here we change the 2D subscriptable to a 1D subscriptable containing single (separate) 1D subscriptables at each location, without performing any additional grouping. In the end this could be concretized as an array of arrays (regular jagged array) in which the tuples are stored.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem k in [0, PNZ_len[row]-1]:
3     PC[PNZ'[row].__row_col_val[k].row] += PNZ'[row].
    __row_col_val[k].val * PB[PNZ'[row].__row_col_val[k].
    col]
```

Listing A.26: Possible result after applying the StructureJaggedSplittingPass on PNZ with offset 1, but using only a single group.

A.11 ConcretizationPass

The ConcretizationPass will concretize all subscriptables. A concretized subscriptable, whose name we typically prefix with a `C`, will have a defined concretization, such as an array or linked list. Nested data structures, such as an X-dimensional array containing linked lists, can be achieved through applying the StructureJaggedSplittingPass before the ConcretizationPass. For example, for the code in Listing A.26, `PNZ'` could be concretized as a linked list while `PNZ'[_].__row_col_val` could be concretized as an array. libtupl currently always concretizes everything as an array.

Additionally, the ConcretizationPass will convert all tUPL-based looping structures to more traditional looping structures. For example, range-based `forelem` loops can be converted to a `for`-loop.

Let us concretize the code from Listing A.24. This will yield the code in Listing A.27. The exact data structures of `CPC`, `CPNZ'` and `CPB` are not visible in this code, but tUPL concretizes these data structures to dense arrays. `CPC` and `CPB` thus become 1D arrays of doubles. `CPNZ'` is then concretized to a 1D array containing a tuple of two 1D arrays at each location (i.e. `CPNZ'[_].__row` and `CPNZ'[_].__col_val` are concretized to 1D arrays containing 1D and 2D tuples respectively).

```
1 for row = 0, row <= max(NZ.row), row += 1:
2   forelem k = 0, k <= PNZ_len[row]-1, k += 1:
3     CPC[CPNZ'[row].__row[k].row] += CPNZ'[row].__col_val[
    k].val * CPB[CPNZ'[row].__col_val[k].col]
```

Listing A.27: Possible result after applying the ConcretizationPass.

Appendix B

I/O generation through generators

Certain transformation passes will also have to transform the input/output data. When using the I/O generation approach through generators, transformations may create new generators defining how the data generated by other generators should be transformed. In this appendix the produced generators are described for various passes. We mostly focus on loading generators, unloading generators are typically similar.

B.1 HorizontalIterationSpaceReductionPass

Horizontal iteration space reduction simply reduces the width of tuples inside of a subscriptable structure. The produced generators look like the example in Listing B.1. As this transformation is performed on subscriptables, a key and value is passed in. This transformation does not modify the key at all, but constructs a new, smaller `NTuple` for each input value.

```
1 λ(input: Generator[Tuple[int, int], NTuple[a: int, b: int
    , c: int]]) -> Generator[Tuple[int, int], NTuple[a:
    int, b: int]]:
2     while input:
3         key, value = input.next()
4         yield key, (value.a, value.b)
```

Listing B.1: A generator adjusting the data in a subscriptable after horizontal iteration space reduction, removing the `c` field from the value tuple.

B.2 LocalizationPass

The LocalizationPass relates data from a shared space to tuples in a tuplespace. Multiple variations for the LocalizationPass are possible, each relating the data in different ways.

A naive approach is to simply write each shared space value to an interme-

diate data structure that allows looking up values by keys, practically converting this stream of data to a temporarily concretized data structure. This always works but can be highly inefficient, especially if this utility data structure is chosen is unsuitable for the keys of the shared space. Listing B.2 shows such a possible implementation.

```
λ(tr: Generator[NTuple[a: int, b: int]], ss: Generator[
    Tuple[int, int], double]) -> Generator[NTuple[a: int,
    b: int, merged_val: double]]:
    # convert shared space to some immediate lookup table
    lut = {}
    while ss:
        key, value = ss.next()
        lut[key] = value

    # finally stream through the reservoir: appending the
    merged tuple value
    while tr:
        tuple = tr.next()
        yield (*tuple, lut[tuple.a, tuple.b])
```

Listing B.2: A generator merging the data in a shared space into a tuplespace.

Often it is possible to merge both generators together more efficiently. As this generator can control on its own when to advance each generator, it could decide to advance both at the same time. This only works well when the value to be merged from the shared space is at the same stream offset as the stream offset of the tuple in the reservoir stream into which the value should be merged. Listing B.3 shows such a generator.

```
λ(tr: Generator[NTuple[a: int, b: int]], ss: Generator[
    Tuple[int, int], double]) -> Generator[NTuple[a: int,
    b: int, merged_val: double]]:
    # finally stream through the reservoir: appending the
    merged tuple value
    while tr:
        tuple = tr.next()
        key, value = ss.next()
        assert(tuple.a == key[0] and tuple.b = key[1]) #
    either prevent this from happening using an additional
    analysis pass or fall back to another approach once this
    occurs
        yield (*tuple, value)
```

Listing B.3: A generator merging the data in a shared space into a tuplespace without an intermediate lookup table.

B.3 ReservoirMaterializationPass + NStarMaterializationPass

Whenever applying the ReservoirMaterializationPass, followed by a NStarMaterializationPass on the produced `N*` reservoir, we have to assign indices to

each tuple in the tuplereservoir to convert it to a subscriptable, usually starting from 0. Usually we also produce the associated `_len` subscriptable, indicating how many tuples match a certain query. Let us assume that is the case in the following example too.

For the materialized reservoir the generator shown in Listing B.4 generates the key-value data for the subscriptable. The associated `_len` subscriptable is generated in a highly similar fashion, as shown in Listing B.5. Note how the data in the `_len` reservoir cannot be yielded until all tuples have been counted, as the input order of tuples is generally undefined. Additionally note that we need two separate generators for the two subscriptables. Both of them are very similar: they both have to count the tuples and read the exact same input stream. As a result, production of the `_len` subscriptable requires reading the input stream twice as the input generator is practically duplicated. Using the generator approach it is not possible to merge these two generators and generate data for both data structures at once, as the control lies at the bottom.

```
1 λ(tr: Generator[NTuple[a: int, b: int]]) -> Generator[
    Tuple[int, int], NTuple[a: int, b: int]]:
2     counts = {} # some sort of lookup table, default value of
    0 if key does not yet exist
3     while tr:
4         tuple = tr.next()
5         query = (tuple.a,)
6         yield (*query, counts[query]), (*tuple, counts[
    query])
7         counts[query] += 1
```

Listing B.4: A generator materializing a tuplereservoir.

```
1 λ(tr: Generator[NTuple[a: int, b: int]]) -> Generator[
    Tuple[int, int], NTuple[a: int, b: int]]:
2     counts = {} # some sort of lookup table, default value of
    0 if key does not yet exist
3     while tr:
4         tuple = tr.next()
5         query = (tuple.a,)
6         counts[query] += 1
7
8     for query, count in counts:
9         yield query, count
```

Listing B.5: A generator materializing a `N*` reservoir into a `_len` subscriptable.

B.4 DelocalizationPass

Although the DelocalizationPass does not truly change data, it does basically duplicate a subscriptable, aside from changing the algorithm in a sensible way. The subscriptable being delocalized had a `Generator` generating key-value pairs for this subscriptable. For the delocalized subscriptable this generator is cloned and will in the end be executed twice to generate data for both subscriptables. A delocalization thus also leads to reading the input multiple times.

B.5 ConcretizationPass

The ConcretizationPass actually writes data to the desired data structures. For example, the function in Listing B.6 may be produced to concretize subscriptable PA into CPA.

```
λ(tr: Generator[Tuple[int, int], NTuple[a: int, b: int]]):
    while tr:
        key, value = tr.next()
        CPA.ensure_writable(*key)
        CPA[*key] = value
```

Listing B.6: A function concretizing a subscriptable.

Note how this function will consume a generator completely and write the data to a concrete data structure. The implementation of the concretized data structure is part of the *libtupl runtime*, which implements things like subscripting a data structure and the `ensure_writable` method.

For the output a generator is produced instead, enumerating all key-value pairs in the concrete data structure, as shown in Listing B.7.

```
λ() -> Generator[Tuple[int, int], NTuple[a: int, b: int]]:
    for dim0 in range(0, CPA.bounds<0>()):
        for dim1 in range(1, CPA.bounds<1>()):
            yield (dim0, dim1), CPA[dim0, dim1]
```

Listing B.7: A generator deconcretizing a concretized subscriptable. Here we assume the subscriptable is concretized as some dense data structure.

Appendix C

I/O generation through transformation graphs

C.1 HorizontalIterationSpaceReductionPass

Whenever we perform horizontal iteration space reduction on a subscriptable, we reduce the width of the value tuple and leave the key unchanged. Figure C.1 shows a "Transform KeyValue" node reducing the tuple its width. Such an I/O node can arbitrarily transform a key and value based on the expressions it has been parameterized with. Here, the output key is equal to the input key and the output value becomes $\langle t.col \rangle$ where t is the input value. Because we performed the HorizontalIterationSpaceReductionPass on subscriptable `A` (whose associated I/O node is not shown here, but could be located above the "Transform KeyValue" I/O node in Figure C.1) we tag this node with `A'` as a result.

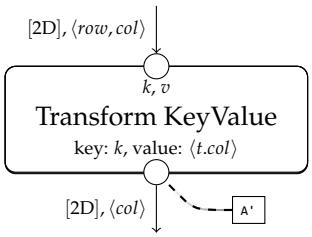


Figure C.1: Example I/O transformation applied after performing a HorizontalIterationSpaceReductionPass on subscriptable `A`.

C.2 AggregateReservoirPass

We can aggregate values from a reservoir of tuples using the AggregateReservoirPass. An "Aggregate" I/O node is then used to actually perform this aggregation. Figure C.2 illustrates this. For "Aggregate" nodes only the function that should be used for the aggregation (such as min, max, $+$ and $\times$) and the tuple field to be aggregated are parameterizable. Typically this node is compiled

to something that completely processes the input passed to it and whenever all input has been processed a scalar value is output. This pass also outputs a zero-dimensional key (the output of this I/O node is always stored for which some sort of key is required later down the line — alternatively, a "Tuple to KeyValue" node could also be used to create this zero-dimensional key).

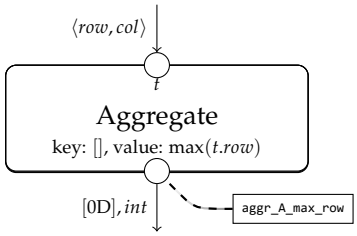


Figure C.2: Example I/O transformation applied after performing an AggregateReservoirPass on tuplespace `A`.

C.3 LocalizationPass

The LocalizationPass will use a "Merge" node to merge the value of a shared space into a new tuple field for all tuples sent to this node. Figure C.3 illustrates this. "Merge" nodes are parameterized by an expression that indicates what shared space value should be merged into each tuple, in this example `[t.row, t.col]` for each tuple `t`. Although the "Merge" node in Figure C.3 is visualized using an intermediate lookup table `X`, the implementation of a "Merge" node may not have to use such a lookup table. Sometimes "Merge" nodes can be eliminated entirely too using the MergeEliminationPass, as we will see later.

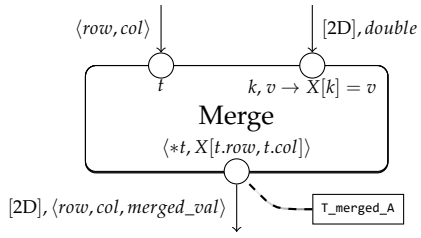


Figure C.3: Example I/O transformation applied after performing a LocalizationPass on tuplespace `T` and shared space `A`.

C.4 ReservoirMaterializationPass + NStarMaterializationPass

Figure C.4 shows the resulting partial I/O graph after executing the ReservoirMaterializationPass and the NStarMaterializationPass. Here we introduce a "Count Tuples" I/O node, which is parameterized by the query. In this exam-

ple, $[row]$ is the query. It outputs two things: first a stream of tuples which mirrors the input steam, but with the generated offset value appended to each tuple. The "Tuple to KeyValue" node then transforms this to the desired key-value stream for the subscriptable `PNZ'`.

The second "Count Tuples" output is a key-value stream which indicates the number of tuples matching each unique query value. This output represents subscriptable `PNZ_len`.

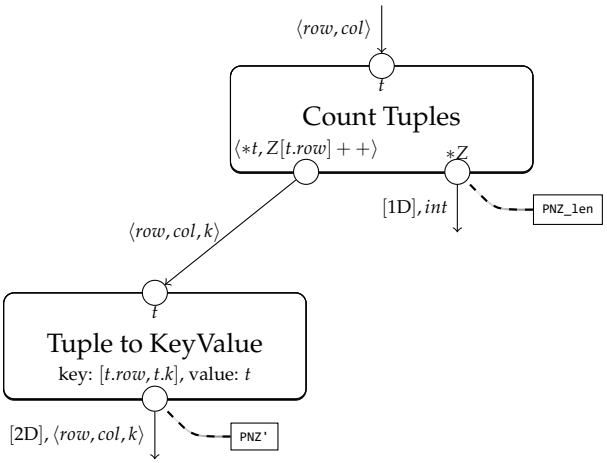


Figure C.4: Example I/O transformation applied after performing a ReservoirMaterializationPass on `NZ` and the NStarMaterializationPass on on `N*`.

Note how this "Count Tuples" I/O node can compute the two outputs simultaneously (depending on the actual implementation, which we discuss in Section D.6). This is unlike the generator I/O approach, for which two disjoint generators are required, each computing one output at a time.

C.5 DelocalizationPass

The DelocalizationPass on its own only duplicates a subscriptable symbol and tries to let different pieces of code use different data structures. Thus, no actual data transformations need to take place. HorizontalIterationSpaceReduction is usually performed after a DelocalizationPass to create two simpler data structures, which is visualized in Figure C.5.

C.6 StructureJaggedSplittingPass

Whenever we perform a StructureJaggedSplittingPass on a subscriptable we split the index of the subscriptable into two pieces. For example, jagging `A[_, _, _]` at offset 2 will yield `A[_, _][_]`. Additionally, tuple fields are regrouped. Together, this can transform `A[_, _, _].a + A[_, _, _].b` where `A` contains tuples $\langle a, b \rangle$ into, for example, `A[_, _].__a[_].a + A[_, _].__b[_].b` (a and b split apart) or `A[_, _].__a_b[_].a + A[_, _].__a_b[_].b` (a and b grouped together). For the index splitting we introduce a "Jag" node

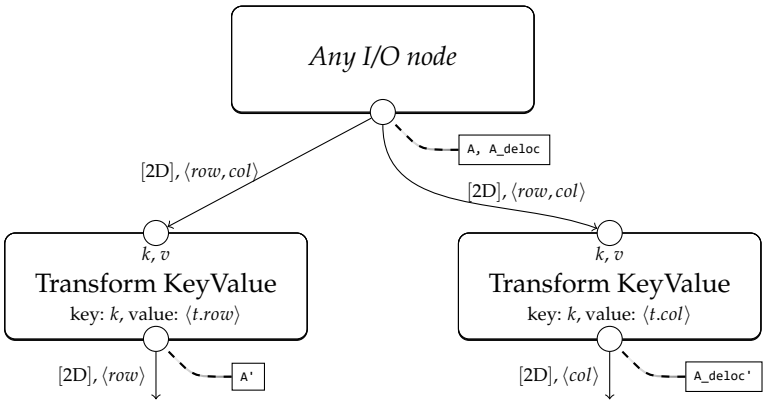


Figure C.5: Example I/O transformation applied after performing a DelocalizationPass on `A` followed by two HorizontalIterationSpaceReductionPasses.

and for the field grouping we use previously introduced "Transform KeyValue" nodes.

Let us look at the simple example in Listing C.1. Applying the StructureJaggedSplittingPass with offset 1 and groups $[\langle col, val\rangle, \langle row\rangle]$ on this code will produce the code in Listing C.2. This pass also produces a single "Jag" I/O node and one or more "Transform KeyValue" I/O nodes, as shown in Figure C.6.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem k in [0, PNZ_len[row]-1]:
3     PC[PNZ[row, k].row] += PNZ[row, k].val * PB[PNZ[row,
    k].col]
```

Listing C.1: Example scenario on which the StructureJaggedSplittingPass can be applied.

```
1 forelem row in [0, max(NZ.row)]:
2   forelem k in [0, PNZ_len[row]-1]:
3     PC[PNZ'[row].__row[k].row] += PNZ'[row].__col_val[k].
    val * PB[PNZ'[row].__col_val[k].col]
```

Listing C.2: Possible result after applying the StructureJaggedSplittingPass on `PNZ` with offset 1.

The "Jag" node will cut *offset* dimensions off of the key, in this example just one. Here, this I/O node then represents `PNZ'[_]`: the output only has one key dimension left as first key dimension is locked down by this I/O node (hence the "_" placeholder).

"Jag" I/O nodes will guarantee the *existence* of a certain integer key in the data structures represented by the node (regardless of how the data structure will be concretized eventually). The "Jag" node in Figure C.6 will thus guarantee the existence of `PNZ'[k[0]]` for each `k` that is pushed to the node. The node does *not* fill the contents of the node with explicit values yet. It may, however, initialize inner data structures in a data-independent way, i.e. invoke the *default constructor* of these inner structures.

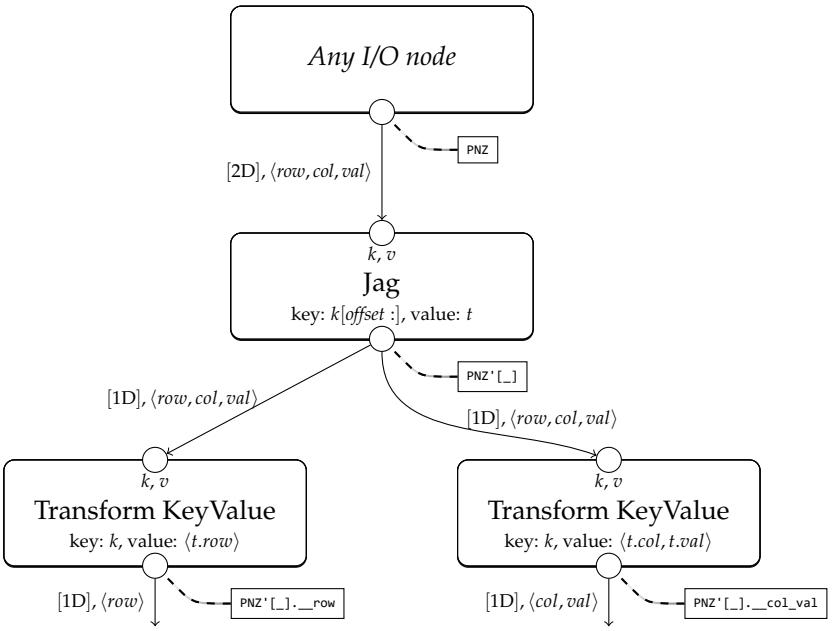


Figure C.6: Example I/O transformation applied after performing a StructureJaggedSplittingPass on `PNZ` with offset 1.

C.7 MergeEliminationPass

The MergeEliminationPass is a pass that is specific for the transformation graph I/O generation approach. This pass does not transform the algorithm its specification in any way, but instead changes the I/O graph only. This pass tries to eliminate "Merge" nodes, typically produced by the LocalizationPass.

Let us illustrate the behavior of this pass through an example. Figure C.7 shows an I/O graph in which a "Merge" node exists. Note that the inputs of this "Merge" node are a "Transform Tuple" and "Tuple to KeyValue" node, which have a shared I/O node (which outputs tuples of data). The MergeEliminationPass can realize that in this scenario we can avoid complicated merging logic by bypassing the "Transform Tuple" and "Tuple to KeyValue" nodes entirely and immediately generating the desired merged output through a single "Tuple to KeyValue" node directly linking to the shared source node. This will yield the I/O graph in Figure C.8. The existing I/O nodes that represent `T` and `A` have their output no longer connected to the previous "Merge" node, but may still have their outputs connected to *other* I/O nodes. If this is not the case, future *dead code elimination* will just eliminate the effect of these I/O nodes entirely.

Caution must be taken when implementing this MergeEliminationPass, as it cannot always eliminate a "Merge" node, even when a shared I/O node exists. Every time a tuple is produced as input for the "Merge" node, the value at `X[t.row, t.col]` must be produced at the shared space input in Figure C.7. Static symbolic analysis can guarantee this.

Here, whenever the shared I/O node produces value $t_0 = \langle a, b, c \rangle$, the "Merge" node will receive $t = \langle a, b \rangle$ from the "Transform Tuple" node (which

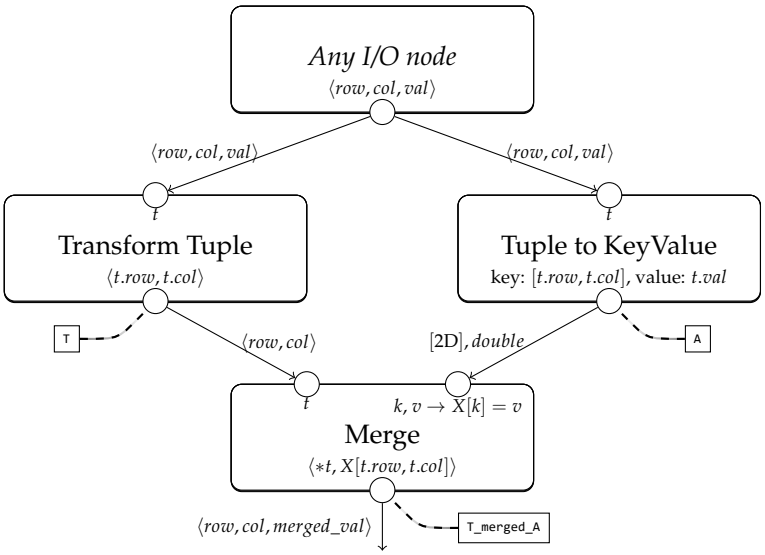


Figure C.7: Example I/O graph on which the MergeEliminationPass is effective.

just eliminates the latter value c). The "Tuple to KeyValue", in this case, produces value $v = c$ for storage on location $k = [a, b]$. The "Merge" I/O node wants to merge, for the incoming $t = \langle a, b \rangle$, the value $X[t.row, t.col] = X[a, b] = v$ into that tuple. Because this value in $X[a, b] = v$ is produced at the same time $\langle a, b \rangle$ is streamed to the node (i.e. also produced when t_0 is output by the shared I/O node), it is safe to eliminate the "Merge" node here.

For example, when the "Transform Tuple" node in Figure C.7 would transform the input tuple t to $\langle t.col, t.row \rangle$ (now also flipping *row* and *col*), the "Merge" node cannot be eliminated safely. When the shared I/O node would produce the value $t_0 = \langle a, b, c \rangle$, the "Merge" node will now receive $t = \langle b, a \rangle$, $k = [a, b]$ and $v = c$. The "Merge" node wants to try to merge $X[t.row, t.col] = X[b, a]$ into the tuple, but the "Tuple to KeyValue" node will produce only the value v at the key $k = [a, b]$, which is not $[b, a]$. Thus the MergeEliminationPass cannot eliminate the "Merge" node safely in this case.

C.8 ConcretizationPass

The ConcretizationPass will also have to actually write data to the data structures selected. For this, we introduce a "Write Value" I/O node. This I/O node can only write *scalar* values to some previously fixed storage location, though, so we first use a "Jag" node to remove (or actually, *fix*) the key entirely. Note that due to the parent/child relations between the I/O nodes the concrete storage location this "Write Value" node should write to will be passed down to this node via the "Jag" node implicitly (all children of this "Jag" node will have a limited *scope* in which subscriptable keys are locked down).

Let us consider an example in which we concretize `PA`, a two-dimensional subscriptable containing doubles. Figure C.9 shows the resulting I/O graph.

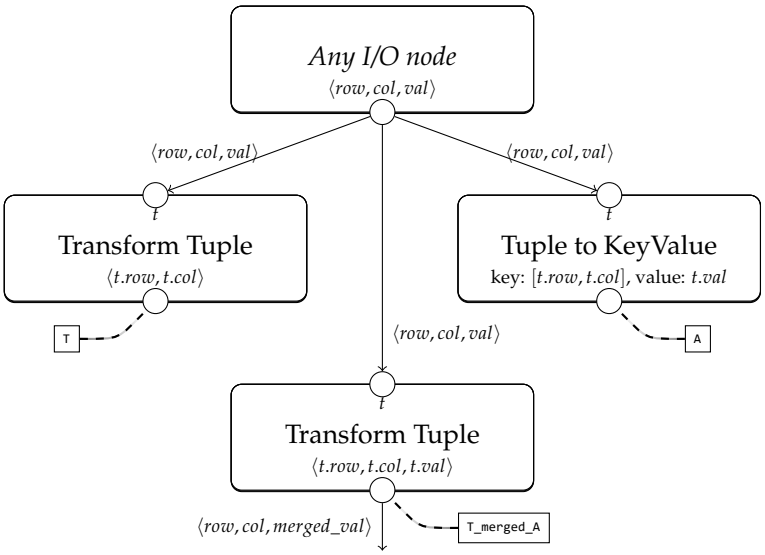


Figure C.8: I/O graph after applying the MergeEliminationPass.

As the output socket of the "Jag" node has been tagged with `CPA[_, _]`, the "Write Value" will write the incoming values to this `CPA` concretized subscriptable.

Note that, for unloading data structures, we need to "deconcretize" the data structures too in the unload I/O transformation graph. For this we introduce a "Deconcretize" node, which reads all key-value pairs in a subscriptable and streams them out. Such a "Deconcretize" node is a root I/O node in the unload graph, as shown in Figure C.10.

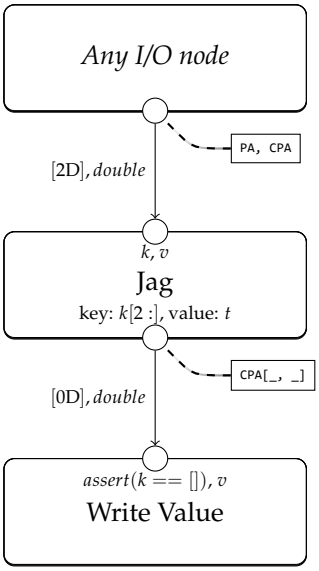


Figure C.9: I/O graph after the concretization of PA.

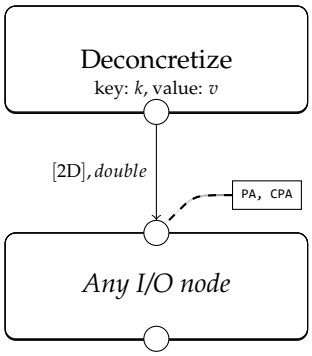


Figure C.10: Unload transformation graph after the concretization of PA.

Appendix D

Imperative code generation for I/O nodes

I/O nodes need to be converted to imperative code during the code generation step. This appendix describes how various I/O nodes are converted to the output-agnostic internal AST before the output-specific code generator transpiles it to the desired target language.

D.1 Transform Tuple

The "Transform Tuple" I/O node takes a single symbol as input, transforms the data in this depending on the anonymous function (expression) this node has been parameterized with and then stores this in a new symbol that the child I/O nodes will then use.

Let us assume the input tuple is of the format $\langle row, col \rangle$ and we want to reduce the width of the tuple, i.e. the output will be $\langle col \rangle$. Then, the "Transform Tuple" has been parameterized by the function `lambda t: (t.col,)`. For this example, Listing D.1 shows the code generated by this I/O node.

```
1 ...
2 input_symbol = ... # assigned to by parent I/O node
3 output_symbol = (input_symbol.col,) # this line of code is
      generated by the conversion routine
4 ...
```

Listing D.1: Code generated by the "Transform Tuple" conversion routine.

D.2 Transform KeyValue

The "Transform KeyValue" I/O node is just like the "Transform Tuple" node, except for that this node is parameterized by two anonymous functions, each taking a key and value from the parent I/O node. This node generates two new symbols: one for the output key and value. Based on the anonymous functions and the key and value symbols of the parent I/O node, these two

output symbols are assigned to and the symbols are forwarded to all child nodes.

D.3 DataStreamReader

The "DataStreamReader" node is always a root node of the I/O graph (i.e. this node has no input sockets) and is parameterized by an external generator coroutine only. This node will create a loop to iterate the entire external generator until it is exhausted and push the generated elements to the child I/O nodes. Listing D.2 shows the code generated by this I/O node during this conversion process.

```
... # other disjoint root node code
while input_stream:
    output_symbol = input_stream.next()
    ... # child nodes
... # other disjoint root node code
```

Listing D.2: Code generated by the "DataStreamReader" conversion routine.

D.4 ConstantStream

The "ConstantStream" node is, just like the "DataStreamReader", always a root node of the I/O graph. It will generate a constant value for a range of keys. Listing D.3 shows the code generated by this I/O node. `N` here is one less than the number of desired key dimensions. `limit` describes the limits of each key for each key dimension, such as $(1000,)$ for a one-dimensional "ConstantStream" of 1000 constants.

```
... # other disjoint root node code
for dim0 in range(0, limit[0]):
    for dim1 in range(0, limit[1]):
        ...
            for dimN in range(0, limit[N]):
                output_symbol_key = [dim0, dim1, ..., dimN]
                output_symbol_value = constant_value
                ... # child nodes

... # other disjoint root node code
```

Listing D.3: Code generated by the "ConstantStream" conversion routine.

D.5 Aggregate

The "Aggregate" node will process all the tuples streamed to it (in fact, only processes a single field of the tuples streamed to it), then finally produce a single scalar output. Some scalar aggregator is initialized to `initial_value`, depending on the aggregate function (often 0, but when multiplying this would

be 1, for example). Then for each tuple the value in the to-aggregate field is taken and the aggregate function is applied on the current aggregator and the incoming value, updating the aggregator value. The final aggregator value is sent to all child I/O nodes once all tuples have been processed. Listing D.4 shows the code generated by this I/O node.

```
aggregator = initial_value
...
# data iterators...:
    ...
    input_symbol = ... # assigned to by parent I/O node
    aggregator = aggregate_func(aggregator, input_symbol.target_field)
    ...
...
# after all data has been iterated
output_symbol_key = []
output_symbol_value = aggregator
... # output socket 0 code
...
```

Listing D.4: Code generated by the "Aggregate" conversion routine.

D.6 Count Tuples

The "Count Tuples" node has two output sockets: one on which the numbered tuples per offset are streamed out and one on which the total number of tuples per group is streamed out. In all cases, the "Count Tuples" node can do both of these tasks in a single pass over the input data. Listing D.5 shows the code generated by the "Count Tuples" node.

```
counts = {} # some sort of lookup table, initialized to 0 if element not found
...
# data iterators...:
    input_symbol = ... # assigned to by parent I/O node
    query = (input_symbol.queryfield1, input_symbol.queryfield2, ...)
    counts[query] += 1
    output_symbol = (*input_symbol, counts[query])
    ... # output socket 0 code
...
# after all data has been iterated
for query, count in counts:
    output_symbol2_key = query
    output_symbol2_value = count
    ... # output socket 1 code
...
```

Listing D.5: Code generated by the "Count Tuples" conversion routine.

In some cases it is possible to generate more efficient output code, like when one would know in advance that the input query would be ordered. This would eliminate the need for a full fledged lookup table, such as `counts` in Listing D.5. The code at *output socket 1* could also be executed immediately once we move on to another query value, rather than forcing us to wait until all tuples have been processed.

D.7 Jag

The "Jag" I/O node will generate code to *ensure space is allocated* to write at the index of the key that has been split off. The exact implementation may vary depending on the underlying data structure. For example, let us assume a flat array is used and the "Jag" splits a 4D index $\langle a, b, c, d \rangle$ at offset 2 and is tagged with symbol expression `CPNZ'''`. The node will then ensure that `CPNZ'''[a , b]` is writable. It will also pass on a reference to child I/O node generation routines that $\langle a, b \rangle$ is the index that data should be written on (i.e. further limit the scope of child I/O nodes).

The "Jag" node does not actually write data, but only (re)allocates it. It may initialize inner data structures, though, through a data-independent *default constructor*. The "Jag" node does not modify or use the value passed to the node, but does pass the value symbol reference on to child I/O nodes. Listing D.6 illustrates the code generated by this I/O node.

```
1 ...
2 input_symbol_key = ... # assigned to by parent I/O node
3 ensure_writable(CPNZ'''', input_symbol_key.a,
      input_symbol_key.b)
4 output_symbol_key = (input_symbol_key.c, input.symbol_key
      .d)
5 ...
```

Listing D.6: Code generated by the "Jag" conversion routine.

In the above example, another "Jag" I/O node can be generated as some descendant of the previous "Jag" I/O node as part of the concretization. The remaining two index values $\langle c, d \rangle$ will then also be split off and locked down. This descendant "Jag" node will be tagged to generate data for `CPNZ'''[_, _].__a_b`. The two placeholder slots here will be occupied by the referenced indices from the ancestor "Jag" node: a and b. The code generated by this second "Jag" node will thus roughly look like the code shown in Listing D.7.

```
1 ...
2 input_symbol_key = ... # assigned to by some ancestor I/O
      node
3 ...
4 input_symbol2_key = ... # assigned to by parent I/O node
5 ensure_writable(CPNZ'''[input_symbol_key.a,
      input_symbol_key.b].__a_b, input_symbol2_key.c,
      input_symbol2_key.d)
6 output_symbol2_key = ()
```

```
...
```

Listing D.7: Code generated by the second "Jag" conversion routine.

D.8 Write Value

The "Write Value" I/O node simply writes a scalar value (which may be a tuple containing multiple scalars) to a storage location. This storage location must be allocated in advance by a "Jag" node. Let us assume a "Write Value" node is writing to `CPNZ'''[_, _].__a_b[_, _]` (i.e. the parent "Jag" node has been tagged with that expression). Then Listing D.8 is the code that is generated by this node.

```
...
input_symbol_key = ... # assigned to by some ancestor I/O node
input_symbol2_key = ... # assigned to by some ancestor I/O node
...
input_symbol3_key = ... # assigned to by parent I/O node
input_symbol3_value = ... # assigned to by parent I/O node
assert(input_symbol3_key == ())
CPNZ'''[input_symbol_key.a, input_symbol_key.b].__a_b[input_symbol2_key.c, input_symbol2_key.d] = input_symbol3_value
...
```

Listing D.8: Code generated by the "Write Value" conversion routine.

D.9 Deconcretize

The "Deconcretize" I/O node will read all the data out of a concretized data structure and pushes each key and value to child I/O nodes. Let us assume that `CPA` is being deconcretized and that `CPA` is concretized as some N-dimensional dense data structure. Then Listing D.9 is the code generated by this I/O node to deconcretize the data.

```
... # other disjoint root node code
for dim0 in range(0, CPA.bounds<0>()):
    for dim1 in range(1, CPA.bounds<1>()):
        ...
            for dimN in range(N, CPA.bounds<N>()):
                output_symbol_key = (dim0, dim1, ..., N)
                output_symbol_value = CPA[*output_symbol_key]
                ... # child nodes
... # other disjoint root node code
```

Listing D.9: Code generated by the "Deconcretize" conversion routine for `CPA`. Here we assume the subscriptable is concretized as some dense data structure.

D.10 DataStreamWriter

The "DataStreamWriter" I/O node simply consumes each tuple streamed to it and invokes a callback function with (a reference to) this tuple as parameter. Listing D.10 is the code that can be generated by this node.

```
1 ...
2 input_symbol = ... # assigned to by parent I/O node
3 callback_func(input_symbol)
4 ...
```

Listing D.10: Code generated by the second "DataStreamWriter" conversion routine.

Appendix E

Example C++ output

Example C++ output for the sparse matrix-vector multiplication example compilation process in Section 3.7.

```
1  struct _tuple__col_int___ {
2      int _col;
3  };
4  struct _tuple__zipped_val_int___ {
5      int _zipped_val;
6  };
7  struct
       _tuple____col_FlatArraySubscriptablem1mm_tuple__col_
       int___m______zipped_val_FlatArraySubscriptablem1mm_
       tuple__zipped_val_int___m___ {
8      FlatArraySubscriptable<1, _tuple__col_int___> ___col;
9      FlatArraySubscriptable<1, _tuple__zipped_val_int___>
       ___zipped_val;
10 };
11 struct _tuple__row_int___ {
12     int _row;
13 };
14 struct _tuple__0_int___ {
15     int _0;
16 };
17 struct _tuple__row_int____col_int____val_int___ {
18     int _row;
19     int _col;
20     int _val;
21 };
22 struct _tuple__i_int____val_int___ {
23     int _i;
24     int _val;
25 };
26 struct _tuple_ {
27 };
28 struct _tuple__row_int____col_int____zipped_val_int____k_
```

```
    int___ {
    int _row;
    int _col;
    int _zipped_val;
    int _k;
};
struct _tuple__row_int____k_int___ {
    int _row;
    int _k;
};
struct _tuple__row_int____col_int____zipped_val_int___ {
    int _row;
    int _col;
    int _zipped_val;
};
struct _tuple__0_int____1_int___ {
    int _0;
    int _1;
};
struct _tuple__col_int____zipped_val_int___ {
    int _col;
    int _zipped_val;
};
struct _tuple__i_int____v_int___ {
    int _i;
    int _v;
};
class MatVec {

private:
    FlatArraySubscriptable<1, int> _PB;

private:
    FlatArraySubscriptable<1,
    _tuple____col_FlatArraySubscriptablem1mm_tuple__
    col_int___m_____zipped_val_FlatArraySubscriptablem1mm
    _tuple__zipped_val_int___m___> _PNZ_zip_Ammm;

private:
    FlatArraySubscriptable<1, int> _PC;

private:
    FlatArraySubscriptable<0, int>
    _accum_NZ_zip_A_row_max;

private:
    FlatArraySubscriptable<1, int> _PNZ_zip_A_len;

public:
    void _matvec() {
```

```
74          _tuple__row_int___ _rowval;
75          _tuple__0_int___ _kv;
76
77          for (_rowval._row = 0; _rowval._row <=
    _accum_NZ_zip_A_row_max.get(); ++_rowval._row) {
78              if (true) {
79                  for (_kv._0 = 0; _kv._0 <= (
    _PNZ_zip_A_len.get(_rowval._row) - 1); ++_kv._0) {
80                      if (true) {
81                          _PC.get(_rowval._row) = (_PC.get(
    _rowval._row) + (_PNZ_zip_Ammm.get(_rowval._row).
    ___zipped_val.get(_kv._0)._zipped_val * _PB.get(
    _PNZ_zip_Ammm.get(_rowval._row).___col.get(_kv._0).
    _col)));
82                      }
83                  }
84              }
85          }
86      }
87
88  public:
89      void _load(InStream<
    _tuple__row_int____col_int____val_int___>& _Values,
    InStream<_tuple__i_int____val_int___>& _BVals) {
90          _tuple__0_int___ _tmp_26;
91          int _tmp_25;
92          int _tmp_23;
93          _tuple_ _tmp_6;
94          int _tmp_35;
95          int _tmp_27;
96          int _itr0;
97          _tuple_ _tmp_34;
98          _tuple__0_int___ _csgen_key;
99          _tuple__0_int___ _tmp_31;
100         int _tmp_33;
101
    _tuple__row_int____col_int____zipped_val_int____k_int___
     _tmp_10;
102         _tuple__row_int____k_int___ _tmp_9;
103         int _tmp_5;
104         _tuple__0_int___ _tmp_24;
105         int _tmp_17;
106         _tuple__row_int____col_int____zipped_val_int___
    _tmp_3;
107         _tuple_ _tmp_29;
108         int _tmp_2;
109         int _tmp_32;
110         _tuple__row_int____col_int____val_int___ _tmp_0;
111         _tuple__0_int___ _tmp_14;
112         FlatArraySubscriptable<1, int> _tmp_7;
```

```
113         int _csgen_value;
114         _tuple__i_int____val_int___ _tmp_30;
115         _tuple_ _tmp_22;
116         _tuple__0_int____1_int___ _tmp_1;
117         _tuple__row_int____k_int___ _tmp_11;
118         int _tmp_21;
119         _tuple__col_int____zipped_val_int___ _tmp_12;
120         _tuple_ _tmp_36;
121         int _tmp_13;
122         _tuple__0_int___ _tmp_15;
123         _tuple_ _tmp_18;
124         int _tmp_28;
125         _tuple__0_int___ _tmp_19;
126 
    _tuple__row_int____col_int____zipped_val_int____k_int___
     _tmp_8;
127         _tuple__col_int___ _tmp_16;
128         _tuple__zipped_val_int___ _tmp_20;
129 
130         _tmp_5 = 0;
131         while (_neof(_Values)) {
132             _tmp_0 = _read(_Values);
133             _tmp_1 = (_tuple__0_int____1_int___){_tmp_0.
    _row, _tmp_0._col};
134             _tmp_2 = _tmp_0._val;
135             _tmp_3 = (
    _tuple__row_int____col_int____zipped_val_int___){
    _tmp_0._row, _tmp_0._col, _tmp_0._val};
136             _tmp_5 = _max(_tmp_5, _tmp_3._row);
137             _ensure_writable(_tmp_7, _tmp_3._row);
138             _tmp_8 = (
    _tuple__row_int____col_int____zipped_val_int____k_int___
    ){_tmp_3._row, _tmp_3._col, _tmp_3._zipped_val, _tmp_7
    .get(_tmp_3._row)};
139             _tmp_7.get(_tmp_3._row) = (_tmp_7.get(_tmp_3.
    _row) + 1);
140             _tmp_9 = (_tuple__row_int____k_int___){_tmp_8
    ._row, _tmp_8._k};
141             _tmp_10 = _tmp_8;
142             _tmp_11 = _tmp_9;
143             _tmp_12 = (
    _tuple__col_int____zipped_val_int___){_tmp_10._col,
    _tmp_10._zipped_val};
144             _tmp_13 = _tmp_11._row;
145             _ensure_writable(_PNZ_zip_Ammm, _tmp_11._row)
    ;
146             _tmp_14 = (_tuple__0_int___){_tmp_11._k};
147             _tmp_15 = _tmp_14;
148             _tmp_16 = (_tuple__col_int___){_tmp_12._col};
149             _tmp_17 = _tmp_15._0;
```

```
150             _ensure_writable(_PNZ_zip_Ammm.get(_tmp_13).
    ___col, _tmp_15._0);
151             _tmp_18 = (_tuple_){};
152             _PNZ_zip_Ammm.get(_tmp_13).___col.get(_tmp_17
    ) = _tmp_16;
153             _tmp_19 = _tmp_14;
154             _tmp_20 = (_tuple__zipped_val_int___){_tmp_12
    ._zipped_val};
155             _tmp_21 = _tmp_19._0;
156             _ensure_writable(_PNZ_zip_Ammm.get(_tmp_13).
    ___zipped_val, _tmp_19._0);
157             _tmp_22 = (_tuple_){};
158             _PNZ_zip_Ammm.get(_tmp_13).___zipped_val.get(
    _tmp_21) = _tmp_20;
159         }
160         while (_neof(_BVals)) {
161             _tmp_30 = _read(_BVals);
162             _tmp_31 = (_tuple__0_int___){_tmp_30._i};
163             _tmp_32 = _tmp_30._val;
164             _tmp_33 = _tmp_31._0;
165             _ensure_writable(_PB, _tmp_31._0);
166             _tmp_34 = (_tuple_){};
167             _PB.get(_tmp_33) = _tmp_32;
168         }
169         for (_itr0 = 0; _itr0 <= 1137; ++_itr0) {
170             _csgen_key = (_tuple__0_int___){_itr0};
171             _csgen_value = 0;
172             _tmp_35 = _csgen_key._0;
173             _ensure_writable(_PC, _csgen_key._0);
174             _tmp_36 = (_tuple_){};
175             _PC.get(_tmp_35) = _csgen_value;
176         }
177         _ensure_writable(_accum_NZ_zip_A_row_max);
178         _tmp_6 = (_tuple_){};
179         _accum_NZ_zip_A_row_max.get() = _tmp_5;
180         for (_tmp_23 = 0; _tmp_23 <= (_tmp_7.bound<0>() -
     1); ++_tmp_23) {
181             _tmp_24 = (_tuple__0_int___){_tmp_23};
182             _tmp_25 = _tmp_7.get(_tmp_23);
183             _tmp_26 = _tmp_24;
184             _tmp_27 = _tmp_25;
185             _tmp_28 = _tmp_26._0;
186             _ensure_writable(_PNZ_zip_A_len, _tmp_26._0);
187             _tmp_29 = (_tuple_){};
188             _PNZ_zip_A_len.get(_tmp_28) = _tmp_27;
189         }
190     }
191
192 public:
193     void _unload(OutStream<_tuple__i_int____v_int___>&
```

```
    _CVals) {
194         int _tmp_37;
195         _tuple__0_int___ _tmp_38;
196         _tuple__i_int____v_int___ _tmp_40;
197         int _tmp_39;
198 
199         for (_tmp_37 = 0; _tmp_37 <= (_PC.bound<0>() - 1)
    ; ++_tmp_37) {
200             _tmp_38 = (_tuple__0_int___){_tmp_37};
201             _tmp_39 = _PC.get(_tmp_37);
202             _tmp_40 = (_tuple__i_int____v_int___){_tmp_38
    ._0, _tmp_39};
203             _write(_CVals, _tmp_40);
204         }
205     }
206 };
```