ETH zürich

ETH Library

Program Sparsification with DaCe

Master Thesis

Author(s):
Kleine, Jan

Publication date:
2024

Permanent link:
https://doi.org/10.3929/ethz-b-000670940

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Program Sparsification with DaCe

Master Thesis

Jan Kleine

April 19, 2024

Advisors: Prof. Dr. T. Hoefler, Philipp Schaad

Department of Computer Science, ETH Zürich

Abstract

The optimization of sparse computations is critical for a broad spectrum of applications, from material sciences to machine learning. These computations, characterized by their irregular data structures, present unique challenges in optimizing performance. This thesis investigates the limitations of existing frameworks like DaCe, which are primarily designed for dense data applications and struggle with the irregularity inherent in sparse data. Building on the framework's strengths, we propose enhancements that enable it to handle the optimization of sparse data applications better. By integrating a modular approach into DaCe, we facilitate the transition between dense and sparse implementations, allowing developers to interchange storage formats and compute kernels easily. This approach not only improves productivity by simplifying the optimization process but also has the potential to enhance performance and portability through the use of various underlying libraries suited to different hardware architectures. We evaluate our proposed workflow by applying it to synthetic and real-world applications, illustrating significant productivity improvements in optimizing sparse computations.

Contents

Contents ii

1 Introduction 1

2 Background 3
- 2.1 High-Performance Sparse Linear Algebra Libraries 4
- 2.2 Custom Kernel Generation 5
- 2.3 Domain-Specific Whole Program Optimization 6
- 2.4 Whole Program Optimization with DaCe 6

3 Storage Format Abstraction 9
- 3.1 Format Abstraction for Tensor Algebra Compiler 9
- 3.2 Tensor Data Format in DaCe 10

4 Library Node 14
- 4.1 Tensor Index Notation Library Node 15

5 Data-Centric Transformation 18
- 5.1 Container Transformation 18

6 Evaluation 20
- 6.1 3MM 20
- 6.2 BERT Encoder Forward Pass 25

7 Related Work 29

8 Conclusion 31
- 8.1 Future Work 31

Bibliography 33

Chapter 1

Introduction

The significance of sparse computations continues to grow, with diverse applications ranging from large-scale graph analytics and material simulations to molecular dynamics, finite element analysis, and machine learning. The challenge of optimizing these computations arises from the irregular nature of sparse data, making it a complex problem that has attracted considerable research attention.

The data-centric parallel programming framework, DaCe [1], has shown its effectiveness in optimizing scientific applications. However, its design is predominantly tailored to applications operating on dense data. The inherent irregularity of sparse data computations leads to an inefficient dataflow representation in DaCe as the dataflow becomes overwhelmed by control flow, rendering traditional dataflow optimizations ineffective.

While there have been notable strides in sparse program optimization, existing solutions have their limitations. For instance, Kjølstad et al. [2, 3] developed TACO, which excels at generating kernels for sparse tensor algebra computations. However, it falls short in performance when compared to specialized, hand-optimized libraries like Intel MKL for common operations. Similarly, Ivanov et al. [4] demonstrated with STen how to optimize machine learning models holistically, enabling the alteration of intermediate data's format and the corresponding computation update. Yet, this approach is confined to programs in the machine learning domain.

Despite the progress in sparse program optimization, a comprehensive approach to whole-program optimization for sparse data applications is still lacking. This thesis presents a novel solution to bridge this gap. We enhance the data-centric parallel programming framework, DaCe, to facilitate the optimization of general programs for sparse data. This empowers developers to interchange between various storage formats and choose different compute kernels to optimize their programs or adapt them to evolving requirements.

The remainder of this thesis is organized as follows: Chapter 2 provides an overview of the existing research landscape. The storage format abstraction and its integration into DaCe are detailed in Chapter 3. This is followed by the introduction of the tensor index notation library node in Chapter 4 and the discussion of data-centric transformations in Chapter 5. We evaluate the proposed workflow on two example applications in Chapter 6, a synthetic matrix multiplication application and the forward pass of the multi-head attention on the BERT encoder. Chapter 8 concludes the thesis.

Chapter 2

Background

In High-Performance Computing (HPC), the critical metrics – Performance, Portability, and Productivity, commonly referred to as the "three Ps" – have been extensively explored across numerous domains and use cases. Despite this widespread attention, the applicability of these principles to sparse linear algebra computations remains less than satisfactory. Ben-Nun et al. [5] offer the following definitions for these metrics:

- **Performance**: The efficiency with which a system utilizes its potential capabilities.
- **Portability**: The ease with which software can be executed across various platforms, optimally utilizing diverse hardware configurations.
- **Productivity**: The resources, both time and effort, required for the development, maintenance, and extension of the software.

Among the strategies for developing sparse linear algebra applications, one initial approach involves crafting the code from the ground up. This method, while straightforward, is fraught with challenges: it is labor-intensive, prone to errors, and seldom meets the criteria of the three Ps simultaneously. Successful implementation demands a robust understanding of the underlying storage formats and algorithms, ensuring the code's accuracy. However, such an approach severely lacks portability, necessitating considerable adjustments or complete redesigns for compatibility with various architectures. While achieving high performance is conceivable for those with an in-depth knowledge of the specific hardware in use, this level of expertise is typically beyond the purview of domain scientists, the likely authors of such applications. In essence, while attaining high performance or portability is possible, it substantially reduces productivity. Striving for both performance and portability concurrently leads to an untenably low level of productivity.

Fortunately, the necessity for such a burdensome approach is mitigated by

the advancements in today's HPC landscape, which offers more pragmatic solutions. These include utilizing specialized libraries with optimized kernels tailored for sparse linear algebra computations, generating sparse linear algebra kernels through specialized compilers, and domain-specific frameworks designed to enhance dense computational programs by leveraging the advantages of sparse intermediate data. In the subsequent sections, we will delve into these methodologies, examining their respective benefits and limitations.

2.1 High-Performance Sparse Linear Algebra Libraries

Adopting standardized interfaces for essential linear algebra operations has significantly streamlined development. These interfaces, often referred to as kernels, embody a set of predefined operations pivotal to linear algebra computations. A quintessential example of such an interface is BLAS, the collection of Basic Linear Algebra Subprograms, which has become a cornerstone in the development of linear algebra applications. In response to the widespread utilization of BLAS, hardware manufacturers have introduced highly optimized versions of these kernels, aiming to maximize the efficiency of hardware resources. While BLAS predominantly addresses dense vector and matrix operations, sparse counterparts have been developed to cater to the sparse data structures commonly encountered in various computational domains.

Intel's Math Kernel Library (MKL) is a prominent library for Intel CPUs, offering an extensive suite of dense and sparse optimized kernels. In the domain of NVIDIA GPUs, cuSPARSE assumes a similar role, providing specialized support for sparse vector and matrix operations. These libraries enhance performance and productivity by offering developers access to pre-optimized routines, thus alleviating the need for in-depth hardware-specific optimization knowledge.

Regarding the three Ps – Performance, Portability, and Productivity – these libraries yield substantial improvements in performance and productivity. They offer a level of portability within the ecosystem of a given hardware vendor, such as the cross-generational portability with MKL on Intel CPUs. However, this portability is often confined to the vendor's hardware spectrum, which presents limitations when transitioning between fundamentally different architectures, such as from Intel CPUs to ARM-based systems or GPUs.

Despite the advantages, these libraries are not without their constraints. Their general-purpose design inherently limits the support to widely adopted storage formats and operations. Consequently, in scenarios involving less conventional matrix sizes or storage formats, the libraries may revert to less

optimized routines, potentially necessitating costly data format conversions. In specific niche applications, these limitations may justify the development of custom code to realize optimal performance, echoing the initial approach of developing sparse linear algebra applications from the ground up. This trade-off between leveraging general-purpose libraries and crafting custom implementations underscores the ongoing challenge of balancing the three Ps within sparse linear algebra computations.

Another limitation in utilizing these libraries is committing to a specific storage format during development. Altering the storage format post-development necessitates revising the relevant sections of the codebase. This rigidity poses a significant challenge in the exploratory stages of algorithm design, where evaluating the performance of various storage formats is crucial. Moreover, it hampers adaptability in response to evolving requirements, further complicating the optimization process.

2.2 Custom Kernel Generation

Kjølstad et al. [2] introduced the Tensor Algebra COmpiler (TACO) to address the shortcomings of sparse BLAS libraries. Libraries have to balance the completeness of the tensor operations they support and the performance of these operations. This trade-off is primarily due to the combinatorial explosion in the variety of tensor operations and formats, making it impractical to hand-optimize code for every possible scenario.

The compiler uses a flexible storage format abstraction introduced by Chou et al. [6], representing all sparse and dense storage formats commonly used and many more. The compiler generates code from tensor index notation, which can express a wide variety of tensor operations. The code generator can target CPUs and NVIDIA GPUs, indicating progress with regard to portability.

However, the performance of the code generated by TACO can be suboptimal. Although specific optimizations have been integrated, such as workspaces [7], more theoretical advancements like auto-scheduling based on an asymptotic cost model [8] have yet to be implemented. Consequently, the compiler's ability to optimize is limited, resulting in performance that does not rival manually optimized code.

The available Python frontend significantly enhances productivity by simplifying the use of TACO. However, one is still confined to the kernels generated by TACO, making it challenging to combine these with the highly optimized kernels available in specialized libraries for standard operations.

TACO's design considers the storage format during code generation and facilitates transitioning to alternative storage formats. Despite this flexibility,

the process remains predominantly manual, necessitating the integration of newly generated kernels into the existing codebase, which is not significantly more straightforward than with existing libraries.

2.3 Domain-Specific Whole Program Optimization

Both BLAS libraries and TACO have one common requirement: they must be considered while writing an application. Integrating generated or hand-tuned kernels into an existing application can be a significant task or even require a complete rewrite, primarily if written in a higher-level language like Python, which is incompatible with these libraries. These hurdles may be challenging for domain scientists who require an ergonomic language and high performance.

Ivanov et al. [4] explored this problem specifically for machine learning with PyTorch. Existing frameworks provide poor support for sparsity, while specialized frameworks focus predominantly on sparse inference. They introduced STen, a sparsity programming model and interface for PyTorch.

STen's model works by extracting the complete computation graph from a PyTorch model and replacing the PyTorch operations with STen wrappers. The STen wrappers automatically dispatch the correct implementation of an operation based on that operation's input and output storage formats. Adjusting the model to use different sparse formats becomes a trivial undertaking as one can easily exchange the storage format of intermediate data. The only restriction is that a compatible version of the operation must exist in STen's catalog of operations, i.e., a version that supports that exact combination of input and output formats. If such an implementation does not exist, STen will automatically convert the necessary input/output data to an appropriate format for which an implementation exists. In addition to this tunable sparsity, STen includes domain-specific capabilities, such as sparsifiers to prune output tensors.

STen's approach has several advantages. Regarding productivity, it allows for easy adaptability of existing models by directly hooking into an existing and widely used library. This integration allows for easy prototyping and development in Python and later optimization of the model for sparsity, potentially by a different performance engineer, without needing to manually re-implement the model using a different library or language.

2.4 Whole Program Optimization with DaCe

Ben-Nun et al. [1] introduced DaCe, a data-centric parallel programming framework that addresses the three P's in HPC. It allows domain scientists to express their programs in Python and translates them to an intermediate

representation called Stateful Dataflow Multigraphs (SDFGs). SDFGs are an intermediate representation (IR) that combine state machines for control flow with dataflow graphs for computation. Data-centric transformations can modify this IR to optimize the performance of the resulting code or adapt it to different hardware architectures. Transformations can be applied manually, by a performance engineer, or automatically, e.g., using the transfer tuning approach demonstrated by Trümper et al. [9]. DaCe currently supports CPUs (x86, ARM, and POWER9), GPUs (NVIDIA and AMD), and FPGAs (Xilinx and Intel).

DaCe has proven effective in all three metrics mentioned above. Ziogas et al. [10] used DaCe to speed up quantum transport simulations by up to two orders of magnitude on two different supercomputers, highlighting the performance and portability aspects. Ben-Nun et al. [11] also showed DaCe's productivity by porting a climate model, significantly reducing the code size while improving performance by nearly a factor of four.

However, DaCe's approach mainly targets applications working on dense data and quickly reaches its limits on sparse applications. While it works well for simple, sparse computations, such as sparse-dense matrix multiplication [9], the data-centric representation quickly reaches its limits with more complex examples, such as sparse-sparse matrix multiplication. The SDFG is soon dominated by control flow, as seen in Figure 2.1, making the application challenging to reason about and prohibiting data-centric optimizations due to a lack of dataflow.

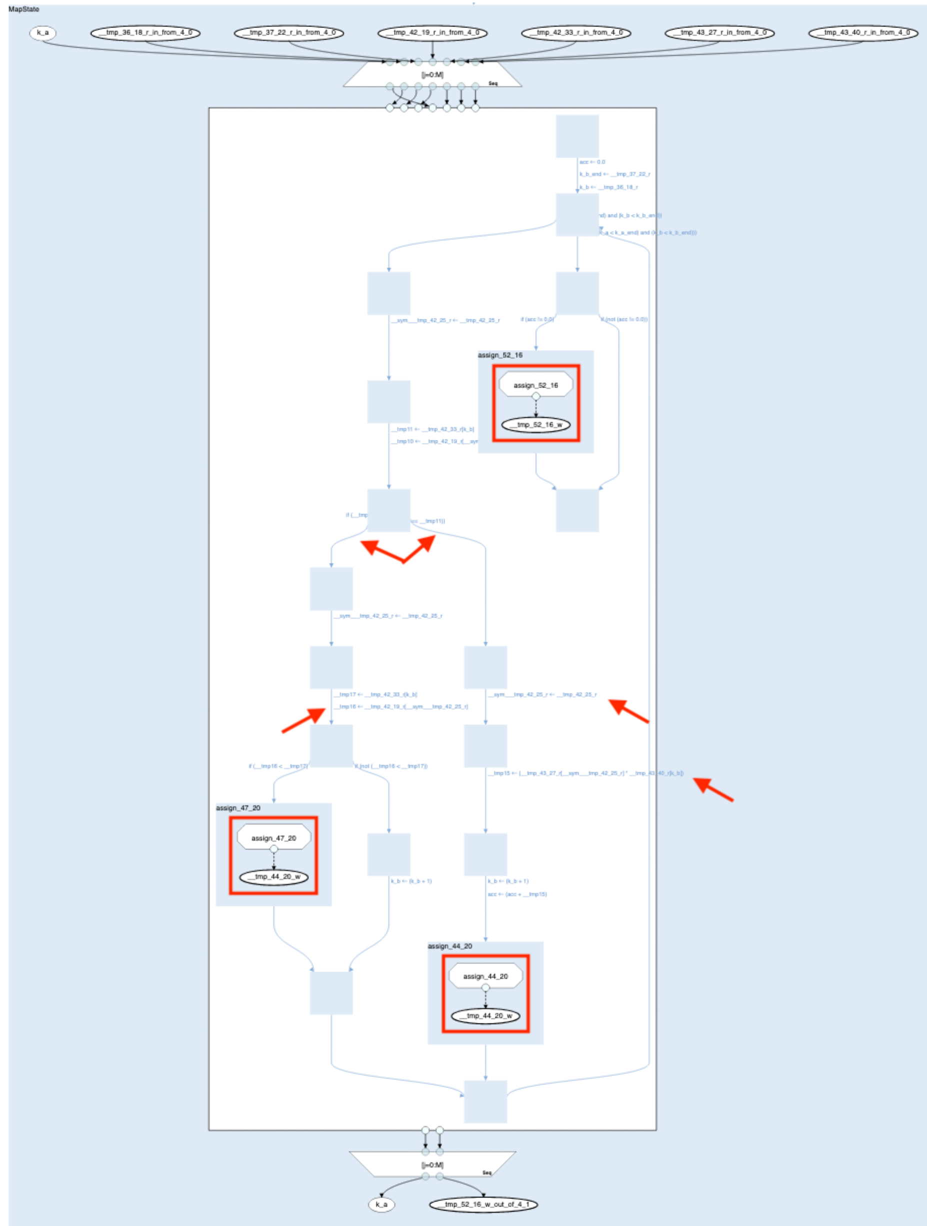


Figure 2.1: The inner map of sparse-sparse matrix-matrix multiplication in DaCe, containing the co-iteration logic. The red arrows highlight some of the logic that is pushed into state transition edges, including computations and memory accesses. The red boxes mark the only dataflow within the computation, which are mere single value assignments.

Chapter 3

Storage Format Abstraction

At the base of our workflow, we need a tensor storage format abstraction. An abstraction allows us to handle various formats uniformly without worrying about the implementation details of the specific format.

Tensor storage formats, both dense and sparse, consist of an index and values. The index defines how to interpret the values. For dense tensors, the index is simple, consisting of only a few scalars to encode the order and size of dimensions. This metadata is simple enough that most programs manually pass it along with a pointer to the array of values.

Sparse tensor storage formats, on the other hand, need more complex index structures involving additional auxiliary arrays. For example, the coordinate (COO) format needs an additional array per tensor dimension. The compressed sparse fiber (CSF) format, a generalization of the doubly compressed sparse row or DCSR format, needs two arrays per tensor dimension. Programmers need to keep track of and pass along the auxiliary arrays in addition to the necessary scalars and values. Handling this much auxiliary data soon becomes cumbersome, especially when mixing different storage formats.

To counter this, many libraries use abstractions to hold all the auxiliary information and values of a matrix or tensor. Some libraries, such as MKL, limit their abstractions to just a handful of matrix storage formats. To enable whole-program optimization, we need a more flexible and extensible abstraction. To this end, we seek inspiration from a more general-purpose abstraction.

3.1 Format Abstraction for Tensor Algebra Compiler

Kjølstad et al. [2, 6] propose a format abstraction for their tensor algebra compiler, TACO. The abstraction is a modular interface that can compose

most existing storage formats and variations thereof.

The abstraction is based on coordinate hierarchies where each hierarchy level encodes coordinates along one tensor dimension. They propose six level formats that can be combined to create most storage formats used today and many more. We will give an overview of the three most essential formats here.

Dense As with fully dense tensors, storing a single dense dimension requires only storing the sine of the dimension. An example of this can be seen in Figure 3.1b to store a dense vector, or Figure 3.1f to store the dense row dimension of the CSR format.

Compressed This level format is well-known from CSR, CSC, and similar formats. It stores the coordinates as a segmented array. This requires two auxiliary arrays: the *pos* array, containing the segment bounds of the segments in the *crd* array. This level format has one segment for every coordinate encoded in the previous level. Figure 3.1f illustrates this. The first dense dimension encodes four coordinates. Thus, four segments are defined. The second segment is stored in $crd[2:4]$. The segments can contain duplicate coordinates as seen in Figures 3.1e and 3.1k in the row dimension of the COO format. Since no previous dimension exists, only one segment is defined, containing as many coordinates as explicitly stored values.

Singleton For every coordinate encoded in the previous level, the singleton format stores the next coordinate. The most prominent usage of this format is in the COO format, as seen in Figures 3.1e and 3.1k with all but the row dimensions.

The authors further define the *Range*, *Offset*, and *Hashed* storage formats, which enable diagonal and hash map storage.

We use this abstraction as the basis for our abstraction in DaCe.

3.2 Tensor Data Format in DaCe

Data in DaCe is modeled as data containers. A data container contains accessible data, both external (program inputs) and transient (internally managed data). There are several data container types built into DaCe, such as `Array`, `View`, and `Struct`. Other data types exist but are not relevant to this work.

An `Array` is a randomly accessible multidimensional array. A `View` is a reinterpretation of an array. It must be directly connected to the container

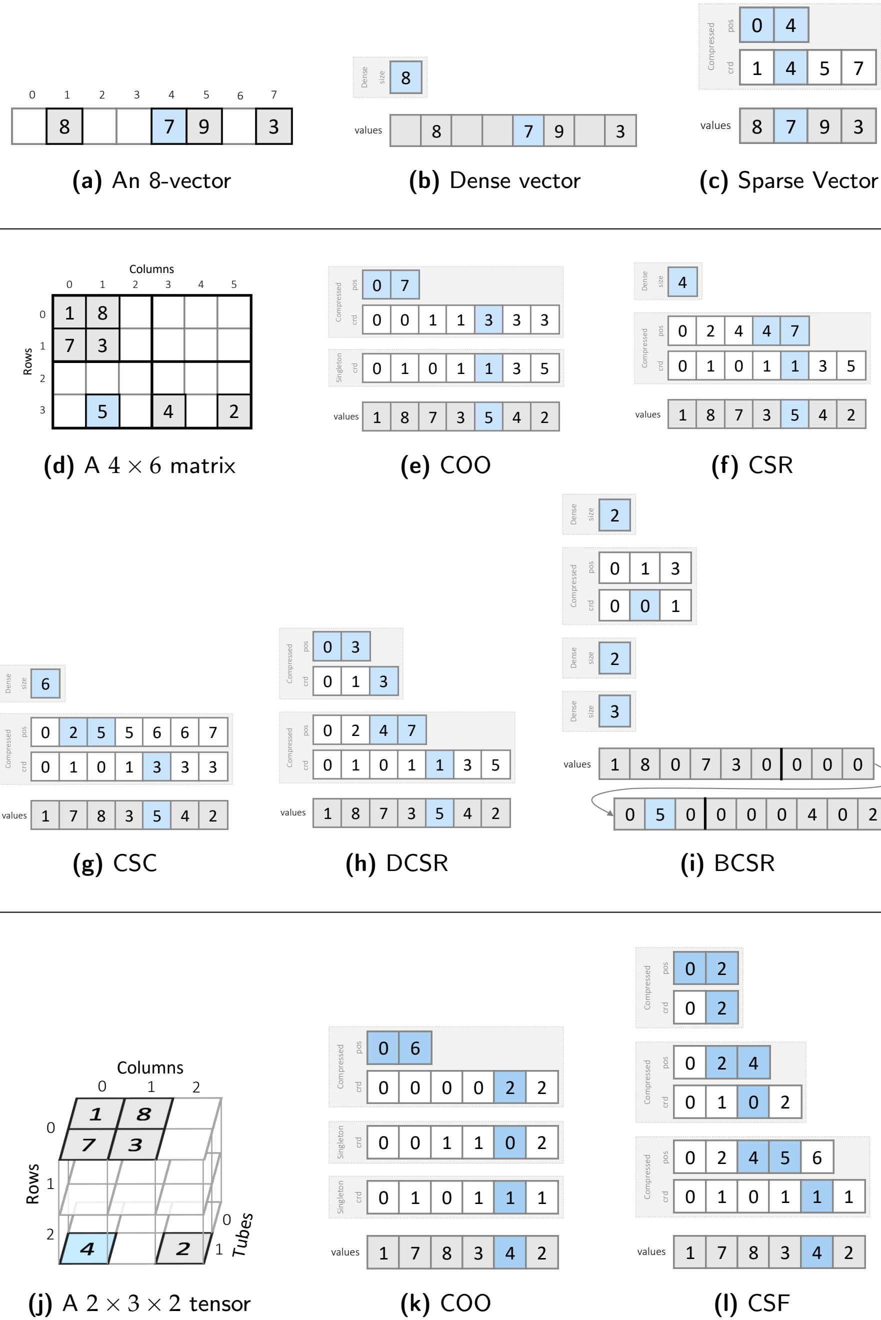


Figure 3.1: Examples of different vector, matrix, and tensor storage formats.

```python
CSR = Tensor(
    dace.float64,
    (M, N),
    [
        (Dense(), 0),
        (Compressed(), 1)
    ],
    nnz,
    "CSR_Matrix"
)

# or using shorthand

CSR = Tensor.CSR(
    (M, N), nnz,
    dace.float64
)
```

Listing 3.1: Python definition of a CSR Matrix

```python
CSR = Tensor(
    dace.float64,
    (M, N),
    [
        (Dense(), 1),
        (Compressed(), 0)
    ],
    nnz,
    "CSC_Matrix"
)

# or using shorthand

CSC = Tensor.CSC(
    (M, N), nnz,
    dace.float64
)
```

Listing 3.2: Python definition of a CSC Matrix

it is viewing and can be used, for example, to access subarrays or reshape an array (such as viewing a three-dimensional $M \times N \times K$ array as a two-dimensional $(M \cdot N) \times K$ array). Much like their C counterpart, `Structs` are nested data containers. They can have several fields, each being a data container of its own and can be passed along as one unit. `Views` can be used to look at individual fields within a struct.

Handling multidimensional sparse data typically involves dealing with many individual data containers or defining a custom struct to unite the necessary containers. To simplify this, we introduce a new data container type `Tensor` inspired by TACO's data abstraction. A `Tensor` can be defined by a few key parameters: The value data type, the number and size of dimensions, the index type and order, the number of non-zeros, and a name for the resulting struct.

Listings 3.1 and 3.2 show how to define $M \times N$ matrices in the compressed sparse row (CSR) and compressed sparse column (CSC) formats, respectively. Note that the only difference is in the order in which the dimensions are stored. Both formats have a dense level followed by a compressed level, with CSR compressing the columns in each row while CSC compresses the rows in each column.

We provide a further shorthand for the most common formats, such as CSR and CSC, to make usage even more ergonomic.

This abstraction achieves little on its own. It can generate the necessary struct for the code generation. However, its actual use is to allow other DaCe components to use the storage format of such a data container to decide on

implementation. We will look at an example of this in the next Chapter.

Chapter 4

Library Node

In DaCe, library nodes symbolize abstract units of work, such as matrix multiplication or transposition. Instead of having a native dataflow represent this operation, a library node can be used. The library node has two different purposes.

Expansion Firstly, a library node can be *expanded*, turning it from a generic node into a specific implementation. This implementation can be a native dataflow or, as the name suggests, it can use a library such as a fast Basic Linear Algebra Subprograms (BLAS) library like MKL. This allows DaCe to benefit from highly optimized libraries.

Transformation The second benefit of library nodes is that they enable *transformations* based on the semantics of a library node. This allows for transformations that may not be easily possible when using native dataflow or generic library nodes. One example of this is a matrix multiplication followed by a transposition. The matrix multiplication library node may be expanded to a specialization based on a specific library. Suppose that the library allows for the major order of the output to be changed (i.e., row- to column-major). In that case, a transformation can recognize this and fuse the multiplication and transposition into one multiplication that writes the output in the correct order.

Many library nodes exist, ranging from concrete operations like tensor transposition to more abstract operations such as evaluating an Einstein summation expression. We will use this more abstract approach to further enable our workflow and optimize programs for sparsity.

4.1 Tensor Index Notation Library Node

We introduce a new Tensor Index Notation (TIN) library node to DaCe. It evaluates a given tensor index expression, much like the `Einsum` library node evaluates an Einstein summation expression.

Tensor index notation expresses how each element of the output tensor is comprised of the input tensors. It's a declarative language; it does not dictate in what order the computation is performed, just what the result should be. The tensor index expression for a matrix multiplication looks as follows:

$$C_{ij} = \sum_k A_{ik} * B_{kj}$$

Sometimes, a shorthand is used, where summations are implied over all variables not captured by the left-hand side. The matrix multiplication can thus be simplified to

$$C_{ij} = A_{ik} * B_{kj}$$

The TIN library node has one input for every tensor on the left-hand side and one output. We allow the inputs and output to be either of type `Array` or `Tensor`, where an array is just interpreted as a tensor with all dense dimensions.

There are two possible expansions: using TACO and mapping the expression to other existing library nodes.

TACO Expansion For this expansion, the library node first checks the input and output tensors to verify that they have a tensor storage format that TACO supports. At the time of writing, this is limited to a combination of dense and compressed level formats. If the formats are supported it calls TACO with the tensor expression to generate the kernel to compute the expression. The code is then augmented with the logic to convert from the DaCe structs to TACO's storage format. That code then replaces the library node in a tasklet. Figures 4.1 and 4.2 show the before and after of a tensor contraction example.

Expansion to Other Library Nodes Common operations, such as matrix multiplications, are usually supported by highly optimized libraries that outperform TACO-generated code. Further, TACO does not support every possible hardware configuration, such as AMD GPUs, further amplifying the case for falling back to these libraries when possible.

DaCe makes it easily possible to offer multiple expansions for a library node. We add another expansion that tries to map tensor contractions to existing calls in BLAS libraries. We will look at mapping a tensor index expression to a sparse-dense matrix-matrix multiplication in MKL (`mkl_sparse_?_mm`).

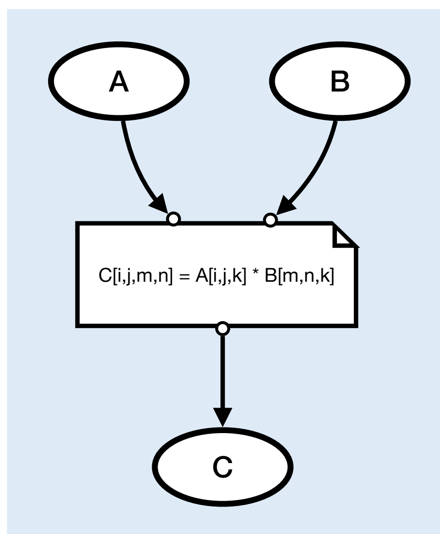


Figure 4.1: Tensor index notation library containing a tensor contraction.

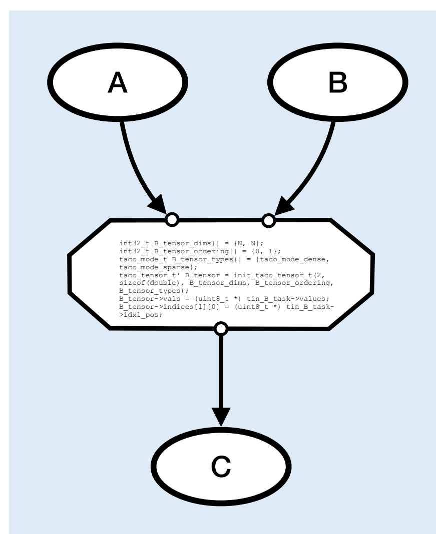


Figure 4.2: Tensor index notation library node expanded using TACO generated code.

The apparent case first. If the expression is of the form $C_{ij} = A_{ik} * B_{kj}$, A is in a supported sparse format (e.g., CSR), and both B and C are dense (both in row major or column major layout) we can directly map this to matrix multiplication because it is one.

However, there are other variations that we can map equally to matrix multiplications. If we have more leading or trailing dimensions, e.g., instead of i and j, we have mno and st respectively, so the expression becomes $C_{mnost} = A_{mnok} * B_{kst}$, we can still reduce this to a matrix multiplication. We can insert the necessary `Views` to reinterpret, for example, the B tensor as a matrix with dimensions $K \times (S \cdot T)$. Importantly, this also works with a sparse tensor where all but the last dimension is dense. We can thus reinterpret A as a $(M \cdot N \cdot O) \times K$ CSR matrix.

Finally, many libraries allow transposing the operands for matrix multiplication. This allows us to support cases where the contracted dimension is the trailing dimension in both tensors. When this is not the case (such as with MKLs `mkl_sparse_?_mm`, which only allows transposing the sparse operand A), we can insert transpositions where necessary.

Figure 4.3 shows such an expansion. The resulting `CSRMM` and `Transpose` library nodes can be expanded to specific implementations relying on tuned BLAS libraries like MKL or native data flow implementations.

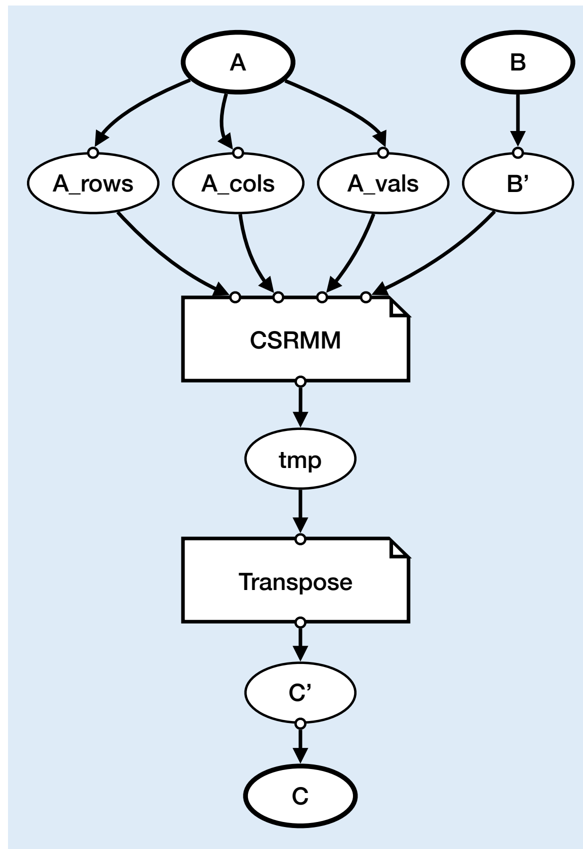


Figure 4.3: Expansion of the library node in Figure 4.1 to sparse matrix multiplication and transpose library nodes.

Chapter 5

Data-Centric Transformation

The power of DaCe lies in its data-centric transformations. Transformations recognize patterns in the form of sub-graphs in the SDFG and replace them with a sub-graph that achieves the same but may have different dataflow properties. The power here is that a performance engineer can do this without understanding the underlying math, simply by recognizing inefficient dataflow patterns. Further, this process can be automized with the help of machine learning, as showcased by Trümper et al. [9].

An example of such a transformation is map tiling. For example, matrix multiplication can be implemented naïvely with three nested loops. In DaCe this would be represented using a three-dimensional *map*. A map surrounds a sub-graph and can be viewed as (parallel) loop, executing the enclosed sub-graph once for every index combination. However, it is beneficial to break the computation into smaller blocks to improve cache locality. The `MapTiling` transformation replaces the original map with two nested maps. The outer iterates over the tiles, while the inner iterates within a tile. This turns the naïve matrix multiplication into a blocked matrix multiplication.

Transformations can also help identify more abstract operations. The `LiftEinsum` transformation can recognize sub-graphs that can be expressed via Einstein summation notation and replaces them with an `Einsum` library node. This enables the expansion of the library node to specialized versions such as `GEMM` for matrix multiplications and further to specialized implementations utilizing high-performance libraries.

5.1 Container Transformation

We introduce a simple transformation to enable our envisioned workflow. The transformation changes the data format of data containers connected to TIN library nodes. Since the implementation of the library node depends on

the connected data containers, we can use this to optimize the program for sparse intermediate data.

Specifically, two conditions have to be met to apply the transformation:

1. The data container can only be connected to TIN library nodes.
2. The data format of the container must be `Array` or `Tensor`, both before and after the transformation.

Based on condition 1, we can transform transient data between TIN library nodes, input data only read by TIN library nodes, and output data written by a TIN library node. This ensures that the change is handled appropriately by the rest of the graph.

Condition 2 allows us to switch the tensor from dense to sparse, sparse to dense, and sparse to sparse (with different formats).

Chapter 6

Evaluation

We evaluate our workflow using one synthetic and one real-world scenario.

We run all benchmarks on a system with 2× Intel Xeon Gold 6154 CPUs on the Ault testbed at the Swiss National Supercomputing Centre (CSCS). We execute all benchmarks on a single CPU core. While parallelization with OpenMP is possible for all examples discussed here, the potential (lack of) scalability would be attributed to the individual BLAS implementations and the TACO-generated code. We aim to evaluate the workflow, not the individual implementations' scalability.

We repeat all benchmarks 1000 times (unless noted otherwise) and visualize the runtimes as box-and-whisker or line plots. In the case of box plots, the box denotes quartiles of the measurements, while the whiskers extend to the 2.5 and 97.5 percentiles, respectively. In the case of line plots, the bands show a 95% interval, ranging from the 2.5 to the 97.5 percentile. Where applicable, the annotations on the plot show the median time. The matrices are randomly generated, with uniformly distributed non-zero elements in the case of sparse data. The runtimes are collected using DaCe's build-in profiling tool and are the end-to-end runtime of the entire program unless noted otherwise.

6.1 3MM

For a synthetic example, we look at the multiplication of four matrices, $A = (B \cdot C) \cdot (D \cdot E)$, commonly referred to as 3MM. Figure 6.1 shows the corresponding SDFG using TIN library nodes for the matrix multiplications. We use this example to highlight the ease of prototyping. All implementations in this Section are working on matrices with double precision (64 bit) floating point values. For simplicity, we will look exclusively at square matrices.

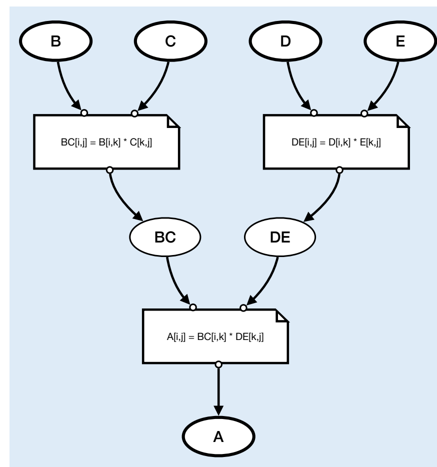


Figure 6.1: SDFG of 3MM.

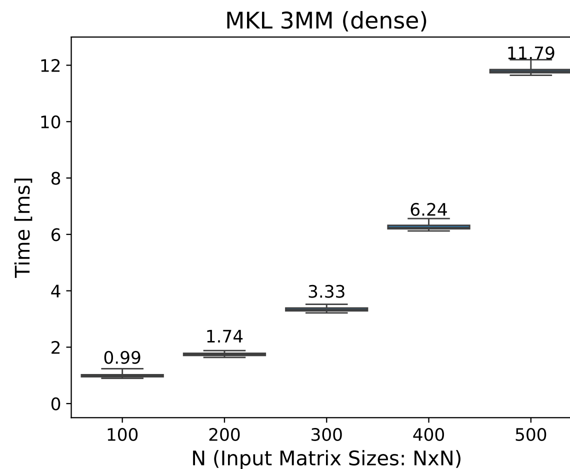
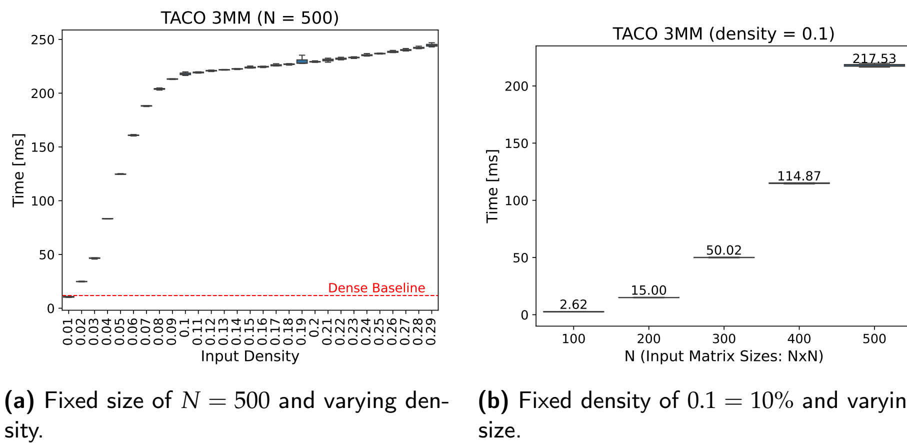


Figure 6.2: Execution time of 3MM using Intel MKL BLAS matrix multiplication kernels. Plot derived from 100 executions, annotated with median execution time.

We start with a dense baseline; for this, we expand the matrix multiplications to BLAS matrix multiplications with Intel MKL's BLAS implementation. Figure 6.2 shows the execution times of the dense baseline.

We can now very quickly adapt the program to work with sparse data. Assume we want all inputs sparse while being flexible on the output format.

For our first experiment, we apply the data container transformation to the original SDFG and change the data format for all data containers to CSR. We leave everything else to the default implementation so that DaCe will replace the library nodes with TACO-generated code. The execution time of this version is shown in Figure 6.3.



(a) Fixed size of $N = 500$ and varying density.

(b) Fixed density of $0.1 = 10\%$ and varying size.

Figure 6.3: Execution time of sparse 3MM using TACO generated matrix multiplication kernels. This version was benchmarked over 100 instead of 1000 runs.

Since we are working with sparse data, the density, i.e., the fraction of non-zero elements, becomes relevant to execution time. However, the dense version outperforms the sparse implementation even at low densities. For moderate degrees of sparsity, the execution time of the sparse code, as depicted in Figure 6.3b, is more than an order of magnitude slower than its dense counterpart. Looking at how the runtime varies depending on sparsity, we can see in Figure 6.3a that at a density of 1%, we are just barely outperforming the dense implementation at 10.76 ms vs. 11.33 ms.

Profiling the SDFG reveals the problem: The third matrix multiplication dominates the execution time. Figure 6.4 shows the visualization of such a profile, revealing a 19× difference in the execution time between the first multiplications and the last. This slowdown is due to decreased sparsity over subsequent multiplications, meaning the intermediate data is dense. Storing dense data in a sparse format and using sparse compute kernels is less than ideal.

In response, we can change the data type of the transient and output containers back to dense. Now that the final multiplication is consuming and producing dense data again, it can fall back to the dense MKL BLAS call from the baseline.

Figure 6.5a shows the execution time of the hybrid approach at varying densities. With this approach, the execution time increases gradually compared to the fully sparse version in Figure 6.3a. If we compare the two, we can see that with $N = 500$ at a density of 0.17, the execution time of the hybrid and dense versions are very close. Indeed, comparing the execution time of the hybrid version with a fixed density of 0.17, as seen in Figure 6.5b, to its dense counterpart (Figure 6.2), the execution times are virtually identical. We can

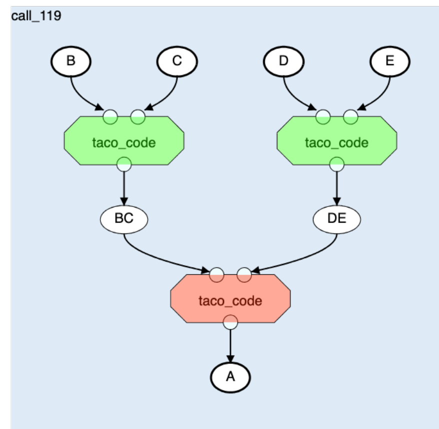
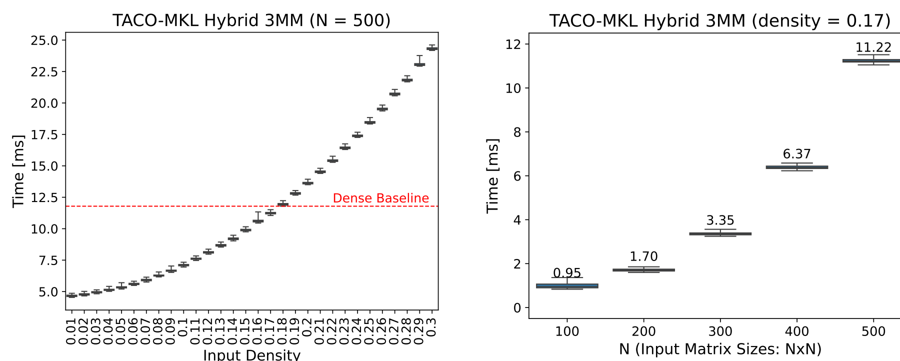


Figure 6.4: Screenshot of the DaCe SDFG editor with the profiling overlay visualizing the median execution time of the tasklets. $N = 500$, density $= 0.01$, execution time: $0.47\,\text{ms}$ for the green tasklets, $9.02\,\text{ms}$ for the red tasklet.



(a) Fixed size of $N = 500$ and varying density.

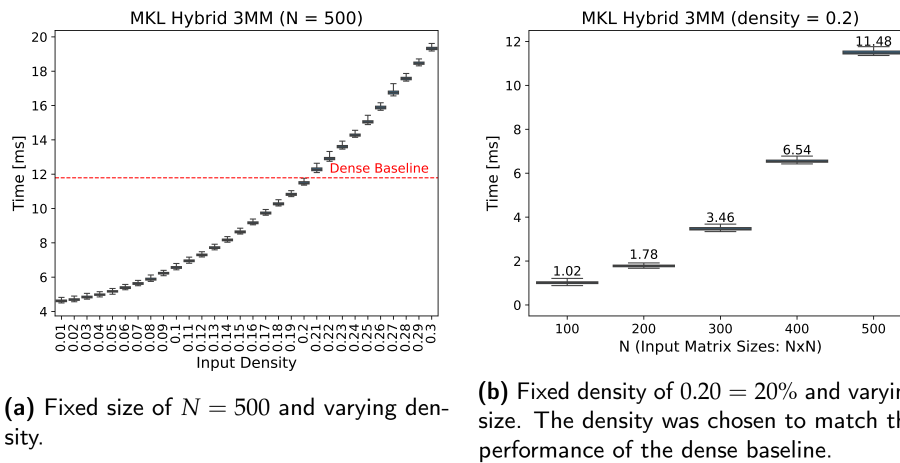
(b) Fixed density of $0.17 = 17\%$ and varying size. The density was chosen to match the performance of the dense baseline.

Figure 6.5: Execution time of hybrid (sparse & dense) 3MM using TACO generated kernels for sparse multiplications and Intel MKL BLAS for dense multiplications.

thus conclude that the sparse TACO-MKL hybrid implementation matches the performance of the dense implementation when the density is 17% and surpasses it at lower densities.

Finally, since matrix multiplications are very common, MKL offers a sparse-sparse matrix multiplication with dense output (`mkl_sparse_?_sp2md`). We can use this function if both inputs have the same storage format (CSR, CSC, or BSR). Since this is the case in our example, we can use a specialized expansion as discussed in Chapter 4 to expand the TIN library node to a version using the MKL sparse library.

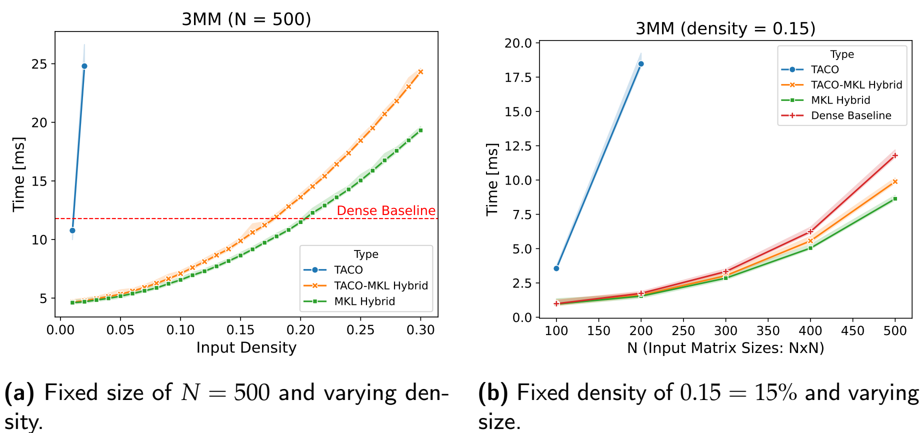
This final specialization further improves performance, as seen in Figure 6.6,



(a) Fixed size of $N = 500$ and varying density.

(b) Fixed density of $0.20 = 20\%$ and varying size. The density was chosen to match the performance of the dense baseline.

Figure 6.6: Execution time of hybrid (sparse & dense) 3MM using MKL sparse and MKL BLAS libraries for matrix multiplication kernels.



(a) Fixed size of $N = 500$ and varying density.

(b) Fixed density of $0.15 = 15\%$ and varying size.

Figure 6.7: Comparison of all explored approaches. The pure TACO approach is only partially shown to keep the y-axis scale within useful bounds.

pushing the border at which the sparse implementation matches the speed of the dense implementation to 20%.

Figure 6.7 shows a complete comparison of all discussed approaches. Figure 6.7a highlights the "break-even" point, at which the sparse implementations match the performance of the dense implementation. The plot further shows how, at lower densities, the difference between the two hybrid approaches becomes negligible. This is to be expected as, with decreasing density, the computation becomes increasingly irregular, negating optimizations MKL might have over the TACO-generated kernel. Figure 6.7b shows how the performance scales with increased matrix size.

We can explore these optimizations with a few clicks of a button without man-

ually changing the code. This allows for very convenient manual exploration and optimization, as we have done here. However, more importantly, it lays the groundwork for automatic optimizations. Auto-tuners and brute-force optimizers could enumerate the best choice for storage formats and kernel implementations, enabling non-performance engineers to benefit from these optimizations.

6.2 BERT Encoder Forward Pass

To examine our workflow on a real-world program, we look at the forward pass of the multi-head attention of the BERT encoder. Listing 6.1 contains the corresponding Python code, and Figure 6.8 shows the resulting SDFG. We want to investigate whether we can benefit from using sparse weights (namely `wq`, `wk`, and `wv`). Note that we are not concerned with sparsifying the weights; we want to investigate the performance of a potentially sparse version.

The weights are used in a tensor contraction at the beginning of the program, specifically, lines 14–16 of Listing 6.1, which correspond to the highlighted sections of the SDFG in Figure 6.8. The Einstein summation $phi, bki \to phbk$ corresponds to the tensor index notation $qq'_{phbk} = wq_{phi} \cdot x_{bki}$ (where qq' is

```
1  @dace.program
2  def mha_forward(
3      x: dace.float32[BB, SM, N],
4      wq: dace.float32[P, H, N],
5      wk: dace.float32[P, H, N],
6      wv: dace.float32[P, H, N],
7      wo: dace.float32[P, H, N],
8      bq: dace.float32[P, H, 1, 1],
9      bk: dace.float32[P, H, 1, 1],
10     bv: dace.float32[P, H, 1, 1],
11     bo: dace.float32[N],
12     scaler: dace.float32,
13 ):
14     qq = np.einsum("phi,bki->phbk", wq, x) + bq
15     kk = np.einsum("phi,bki->phbk", wk, x) + bk
16     vv = np.einsum("phi,bki->phbk", wv, x) + bv
17
18     beta = scaler * np.einsum("phbk,phbj->hbjk", kk, qq)
19     alpha = softmax(beta)
20     gamma = np.einsum("phbk,hbjk->phbj", vv, alpha)
21     out = np.einsum("phi,phbj->bji", wo, gamma) + bo
22
23     return out
```

Listing 6.1: Python code of mutli head attention forward pass.

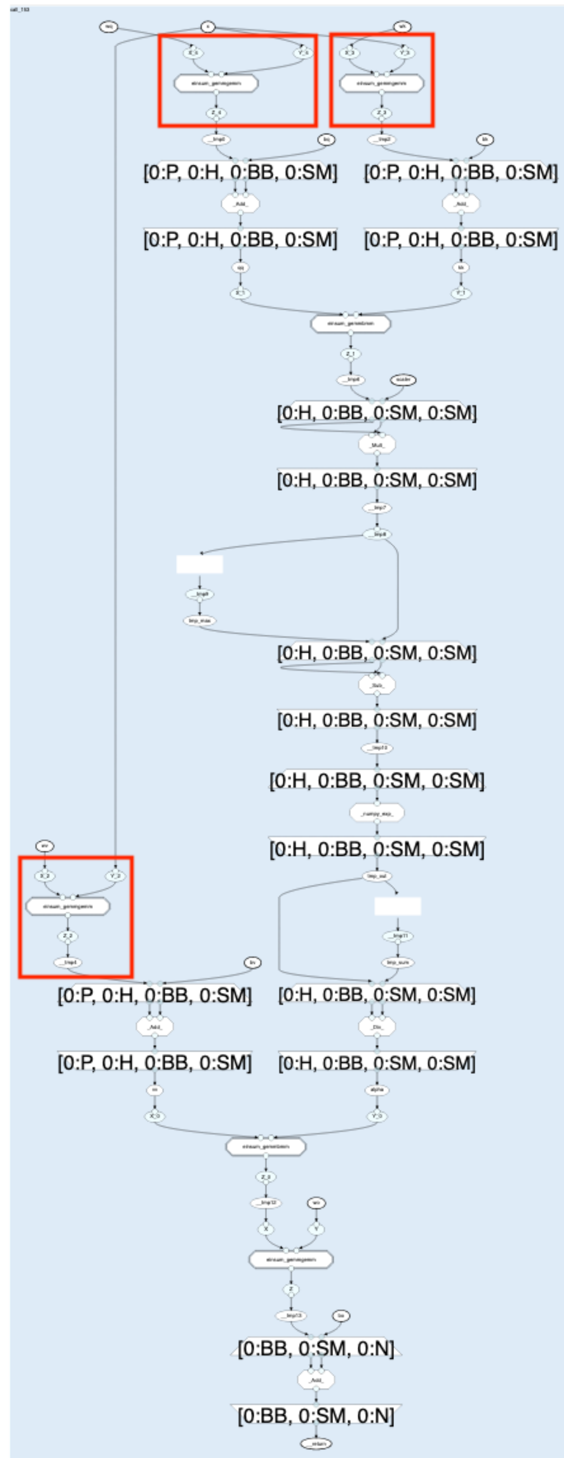
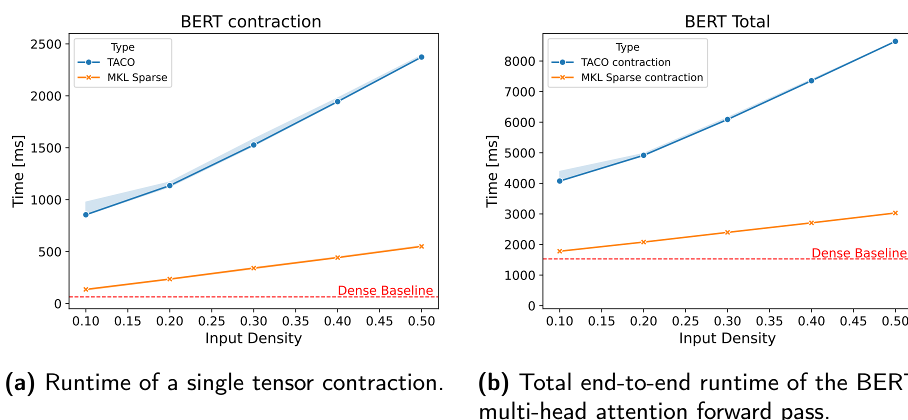


Figure 6.8: SDFG of the BERT encoder multi-head attention forward pass. The targeted tensor contractions are highlighted in red.



(a) Runtime of a single tensor contraction.

(b) Total end-to-end runtime of the BERT multi-head attention forward pass.

Figure 6.9: Runtime comparison at varying weight densities of the BERT encoder multi-head attention forward pass. The program was executed ten times for every tested density, resulting in 30 executions of the relevant tensor contraction.

the temporary value used to compute $qq = qq' + bq$).

DaCe can expand this Einstein summation to a BLAS matrix multiplication. We use this with MKL's BLAS implementation as our performance baseline. For our first sparse version, we transform the weight tensor to a compressed sparse fiber (CSF) tensor and use the TACO expansion of the tensor index expression.

As we saw the tensor index notation previously in Figure 4.1 in Chapter 4, we know that, as long as we can reduce the sparse tensor to a CSR matrix, we can expand the library node to a sparse-dense matrix multiplication followed by a transpose, as seen in Figure 4.3. To test this, we transform the sparse tensor to only compress the third dimension, which allows us to use the expansion. We further expand the resulting `CSRMM` and `Transpose` library nodes to optimized MKL implementations and refer to it as "MKL Sparse".

Figure 6.9 shows the runtime of the three versions. It is apparent that, even at very optimistic densities, the sparse version of the encoder is significantly slower than the dense version. The TACO contraction is $13.3\times$–$37.0\times$ slower than the dense baseline, resulting in a $2.66\times$–$5.56\times$ slowdown in end-to-end time. While the MKL sparse version is significantly faster than TACO, the contraction is still $2.23\times$–$8.57\times$ slower than the baseline, with a $1.16\times$–$1.96\times$ slowdown in end-to-end runtime.

Theoretically, it is possible to improve the execution time of the MKL Sparse version by fusing the transpose with the addition following it. However, the transposition accounts for approximately 15 ms of the contraction time (constant over density). Thus, even if we remove it entirely, the runtime will still be slower than the dense baseline. Because of this, we do not attempt any further optimizations.

Our primary focus here is not to optimize the application but to highlight the ease with which we can explore the different approaches. We show that we can apply the same workflow as before to a real-world application like BERT, switching between storage formats and algorithm implementations with the click of a button. Doing the same on an application written with just TACO or MKL would be more complicated, requiring manual code changes. Changing a conventional application to use MKL instead of TACO would require more extensive manual intervention. We show how to do it without even touching the code.

Chapter 7

Related Work

Many previous works have focused on optimizing sparse applications. In the following, we highlight a few of the most closely related approaches and point out some ways our holistic workflow enriches sparse program optimization capabilities.

Tensor Algebra Compiler Bik et al. [12] discussed integrating TACO's techniques into the MLIR open-source compiler architecture. Tian et al. [13] have done a similar thing, developing a sparse tensor algebra dialect for the MLIR infrastructure based on their domain-specific tensor contraction compiler, COMET [14].

Henry et al. [15] generalized the concepts from TACO to sparse array programming. They extended tensor index notation to array index notation, which can handle arbitrary user-defined functions instead of just additions and multiplications. Further, the framework allows compressing other values than zeros (e.g., ones).

Extending and integrating these tensor algebra compilers into a compiler enables more optimizations and a more integrated workflow. Our approach offers a similar convenience while also allowing the exploration of external, hand-optimized libraries.

Other Code Generation Approaches Du et al. [16] showed how to generate storage formats and GPU kernels for sparse matrix-vector products (SpMV) based on input sparsity pattern and hardware architecture.

Yadav et al. [17, 18] built a distributed tensor algebra compiler, DISTAL, targeting CPU and GPUs. They later extended it to support sparse tensor computations with SpDISTAL.

DaCe can adapt applications to run in distributed settings. While a single sparse computation can not yet be easily adapted to run in a distributed

environment, it should already be possible to distribute separate sparse computations across different nodes.

Hardware Approaches Hsu et al. [19] build the Stardust compiler as a separate compilation path in the TACO that compiles sparse tensor algebra to reconfigurable dataflow architectures. They later developed the sparse abstract machine [20], an abstract machine model for targeting sparse tensor algebra to reconfigurable fixed-function dataflow accelerators.

Srivastava et al. [21] similarly propose a hardware accelerator to accelerate dense and sparse tensor factorization.

While we have not explored utilizing reconfigurable fixed-function hardware in this work, DaCe does support FPGAs. In its current form, this enables offloading different parts of an application to different accelerators while optimizing CPU codes with the approach we have explored here.

Inspector-Executor Compiler Optimizations Strout et al. [22], Nandy et al. [23], Zhao et al. [24], and Venkat et al. [25, 26, 27] adapted polyhedral optimizations to the sparse world with the sparse polyhedral framework. It is based on the inspector-executor model, where an inspector analyzes the data at runtime and selects an optimized executor based on that knowledge. They also explore a series of other compiler transformations and automatic parallelization for sparse matrix computations.

DaCe currently does not implement these inspector-executor workflows. However, it should generally be possible to integrate them using data-centric transformations, further expanding DaCe's optimization capabilities.

Probibalistic Modeling for Cache and Locality Improvements Heras et al. [28, 29], Pichel et al. [30, 31], Hong et al. [32] modeled and improved cache locality of sparse matrix-vector products through reordering.

Fraguela et al. [33, 34, 35] and Doallo et al. [36, 37] explored probibalistic cache miss modeling.

DaCe's analytical capabilities regarding locality and cache optimizations are currently limited to dense data. These approaches could guide auto-tuners in their search for the best optimizations.

Chapter 8

Conclusion

This work showed how a modular approach to sparsity can be implemented in DaCe. The presented workflow allows developers to easily switch from a dense to a sparse implementation of the same program and quickly iterate between different sparse versions. It makes it possible to explore different storage formats and kernel implementations with the figurative click of a button instead of wasting valuable time on lengthy manual modifications of programs.

Looking back at the three Ps in HPC – Performance, Portability, and Productivity – our workflow drastically improves productivity. While performance and probability depend on the underlying libraries used for the sparse computations, it is clear that the workflow can accelerate the choice of algorithm and library for optimal performance. Further, our modular approach enables the integration of further libraries supporting different hardware architectures, which should aid the portability of sparse applications.

8.1 Future Work

This work lays the groundwork for further developments of sparse optimizations with DaCe.

Portability To facilitate portability to different architectures, such as GPU, adding support for different libraries is necessary. A notable candidate is cuSPARSE, targeting NVIDIA GPUs. The modular approach of DaCe makes it easy to integrate new libraries and associated storage formats (e.g., $m : n$ or, more specifically, $2 : 4$ sparsity).

Native Dataflow Support The advantages of the storage format abstraction are currently only utilized by library nodes that expand to library calls or fixed C++ code. Implementing sparse algorithms using DaCe's data flow

abstraction still needs to be more convenient. DaCe could be extended with map schedules that allow iterating and co-iterating the different level formats of the storage abstraction. This could replace the TACO code generation with a native dataflow implementation. A native dataflow implementation would also enable data-centric transformations on the sparse algorithms, such as tiling or fusing maps iterating sparse data structures.

Stochastic Modeling The abovementioned native dataflow representation could also enable stochastic modeling of the algorithms in order to predict, for example, cache misses. Such information could guide the optimization of sparse algorithms.

Data Optimizations we have only explored the algorithm side of sparse program optimization. Reordering sparse matrices can significantly increase locality and, thus, performance. Transformations could be added that insert reordering logic and modify the dataflow to account for the new order.

Inspector Executor Workflows Similarly, a transformation could insert an inspector to gather statistics about the sparse data that can be used in later stages of the program.

Bibliography

[1] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for high-performance parallel programs," *CoRR*, vol. abs/1902.10345, 2019. arXiv: 1902.10345. [Online]. Available: http://arxiv.org/abs/1902.10345.

[2] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, Oct. 2017. DOI: 10.1145/3133901.

[3] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "Taco: A tool to generate tensor algebra kernels," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, Oct. 2017. DOI: 10.1109/ase.2017.8115709.

[4] A. Ivanov, N. Dryden, T. Ben-Nun, S. Ashkboos, and T. Hoefler, "Sten: Productive and efficient sparsity in pytorch," Apr. 15, 2023. DOI: 10.48550/ARXIV.2304.07613. arXiv: 2304.07613 [cs.LG].

[5] T. Ben-Nun, T. Gamblin, D. S. Hollman, H. Krishnan, and C. J. Newburn, "Workflows are the new applications: Challenges in performance, portability, and productivity," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, IEEE, Nov. 2020. DOI: 10.1109/p3hpc51967.2020.00011.

[6] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, Oct. 2018. DOI: 10.1145/3276493.

[7] F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe, "Tensor algebra compilation with workspaces," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, Feb. 2019. DOI: 10.1109/cgo.2019.8661185.

[8] P. Ahrens, F. Kjolstad, and S. Amarasinghe, "Autoscheduling for sparse tensor algebra with an asymptotic cost model," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ACM, Jun. 2022. DOI: 10.1145/3519939.3523442.

[9] L. Trümper, T. Ben-Nun, P. Schaad, A. Calotoiu, and T. Hoefler, "Performance embeddings: A similarity-based transfer tuning approach to performance optimization," in *Proceedings of the 37th International Conference on Supercomputing*, ser. ICS '23, Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 50–62, ISBN: 9798400700569. DOI: 10.1145/3577193.3593714. [Online]. Available: https://doi.org/10.1145/3577193.3593714.

[10] A. N. Ziogas, T. Ben-Nun, G. I. Fernández, T. Schneider, M. Luisier, and T. Hoefler, "Optimizing the data movement in quantum transport simulations via data-centric parallel programming," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, Nov. 2019. DOI: 10.1145/3295500.3356200.

[11] T. Ben-Nun, L. Groner, F. Deconinck, T. Wicky, E. Davis, J. Dahm, O. D. Elbert, R. George, J. McGibbon, L. Trümper, E. Wu, O. Fuhrer, T. Schulthess, and T. Hoefler, *Productive performance engineering for weather and climate modeling with python*, 2022. DOI: 10.48550/ARXIV.2205.04148.

[12] A. Bik, P. Koanantakool, T. Shpeisman, N. Vasilache, B. Zheng, and F. Kjolstad, "Compiler support for sparse tensor computations in mlir," *ACM Transactions on Architecture and Code Optimization*, vol. 19, no. 4, pp. 1–25, Sep. 2022, ISSN: 1544-3973. DOI: 10.1145/3544559.

[13] R. Tian, L. Guo, J. Li, B. Ren, and G. Kestor, "A high performance sparse tensor algebra compiler in mlir," in *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, IEEE, Nov. 2021. DOI: 10.1109/llvmhpc54804.2021.00009.

[14] E. Mutlu, R. Tian, B. Ren, S. Krishnamoorthy, R. Gioiosa, J. Pienaar, and G. Kestor, "Comet: A domain-specific compilation of high-performance computational chemistry," Feb. 13, 2021. DOI: 10.48550/ARXIV.2102.06827. arXiv: 2102.06827 [cs.MS].

[15] R. Henry, O. Hsu, R. Yadav, S. Chou, K. Olukotun, S. Amarasinghe, and F. Kjolstad, "Compilation of sparse array programming models," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. DOI: 10.1145/3485505. [Online]. Available: https://doi.org/10.1145/3485505.

[16] Z. Du, J. Li, Y. Wang, X. Li, G. Tan, and N. Sun, "Alphasparse: Generating high performance spmv codes directly from sparse matrices," Nov. 7, 2022. DOI: 10.48550/ARXIV.2212.10432. arXiv: 2212.10432 [cs.DC].

[17] R. Yadav, A. Aiken, and F. Kjolstad, “Distal: The distributed tensor algebra compiler,” PLDI ’22, Mar. 15, 2022. DOI: 10.1145/3519939.3523437. arXiv: 2203.08069 [cs.PL].

[18] R. Yadav, A. Aiken, and F. Kjolstad, “Spdistal: Compiling distributed sparse tensor computations,” Jul. 28, 2022. DOI: 10.48550/ARXIV.2207.13901. arXiv: 2207.13901 [cs.DC].

[19] O. Hsu, A. Rucker, T. Zhao, K. Olukotun, and F. Kjolstad, “Stardust: Compiling sparse tensor algebra to a reconfigurable dataflow architecture,” Nov. 7, 2022. DOI: 10.48550/ARXIV.2211.03251. arXiv: 2211.03251 [cs.PL].

[20] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, “The sparse abstract machine,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ACM, Mar. 2023. DOI: 10.1145/3582016.3582051.

[21] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, “Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, Feb. 2020. DOI: 10.1109/hpca47549.2020.00062.

[22] M. M. Strout, M. Hall, and C. Olschanowsky, “The sparse polyhedral framework: Composing compiler-generated inspector-executor code,” *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1921–1934, Nov. 2018. DOI: 10.1109/jproc.2018.2857721.

[23] P. Nandy, M. Hall, E. Davis, C. Olschanowsky, M. Mohammadi, W. He, and M. Strout, “Abstractions for specifying sparse matrix data transformations,” *Proceedings of the Eighth International Workshop on Polyhedral Compilation Techniques*, Jan. 23, 2018. [Online]. Available: https://par.nsf.gov/biblio/10309097.

[24] T. Zhao, T. Popoola, M. Hall, C. Olschanowsky, and M. Strout, “Polyhedral specification and code generation of sparse tensor contraction with co-iteration,” *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 1, pp. 1–26, Dec. 2022. DOI: 10.1145/3566054.

[25] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, “Non-affine extensions to polyhedral code generation,” in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ACM, Feb. 2014. DOI: 10.1145/2581122.2544141.

[26] A. Venkat, M. Hall, and M. Strout, “Loop and data transformations for sparse matrix code,” *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 521–532, Jun. 2015. DOI: 10.1145/2813885.2738003.

[27] A. Venkat, M. S. Mohammadi, J. Park, H. Rong, R. Barik, M. M. Strout, and M. Hall, "Automating wavefront parallelization for sparse matrix computations," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, Nov. 2016. DOI: 10.1109/sc.2016.40.

[28] D. Heras, J. Cabaleiro, and F. Rivera, "Modeling data locality for the sparse matrix–vector product using distance measures," *Parallel Computing*, vol. 27, no. 7, pp. 897–912, Jun. 2001. DOI: 10.1016/s0167-8191(01)00089-8.

[29] D. Heras, V. Blanco, J. Cabaleiro, and F. Rivera, "Modeling and improving locality for the sparse-matrix–vector product on cache memories," *Future Generation Computer Systems*, vol. 18, no. 1, pp. 55–67, Sep. 2001. DOI: 10.1016/s0167-739x(00)00075-3.

[30] J. Pichel, D. Heras, J. Cabaleiro, and F. Rivera, "Improving the locality of the sparse matrix-vector product on shared memory multiprocessors," in *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004. Proceedings.*, IEEE, 2004. DOI: 10.1109/empdp.2004.1271429.

[31] J. C. Pichel, J. A. Lorenzo, F. F. Rivera, D. B. Heras, and T. F. Pena, "Using sampled information: Is it enough for the sparse matrix-vector product locality optimization?" *Concurrency and Computation: Practice and Experience*, vol. 26, no. 1, pp. 98–117, Oct. 2012. DOI: 10.1002/cpe.2949.

[32] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ACM, Feb. 2019. DOI: 10.1145/3293883.3295712.

[33] B. B. Fraguela, R. Doallo, and E. L. Zapata, "Cache misses prediction for high performance sparse algorithms," in *Euro-Par'98 Parallel Processing*, Springer Berlin Heidelberg, 1998, pp. 224–233. DOI: 10.1007/bfb0057857.

[34] B. B. Fraguela, R. Doallo, and E. L. Zapata, "Modeling set associative caches behavior for irregular computations," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 1, pp. 192–201, Jun. 1998. DOI: 10.1145/277858.277910.

[35] B. B. Fraguela, R. Doallo, and E. L. Zapata, "Memmory hierarchy performance prediction for blocked sparse algorithms," *Parallel Processing Letters*, vol. 09, no. 03, pp. 347–360, Sep. 1999. DOI: 10.1142/s0129626499000323.

[36] R. Doallo, B. Fraguela, and E. Zapata, "Cache probabilistic modeling for basic sparse algebra kernels involving matrices with a non-uniform distribution," in *Proceedings. 24th EUROMICRO Conference (Cat. No.98EX204)*, IEEE Comput. Soc, Aug. 1998. DOI: 10.1109/eurmic.1998.711825.

[37] R. Doallo, B. Fraguela, and E. Zapata, "Direct mapped cache performance modeling for sparse matrix operations," in *Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing. PDP'99*, IEEE, 1999. DOI: 10.1109/empdp.1999.746696.