Google Summer of Code 2023: NumFOCUS - CVXPY Boosting canonicalization performance by replacing N-dimensional sparse tensor representation.

William Zijie Zhang

September 1, 2023

Abstract

The performance of canonicalization procedures for optimization problems is an important metric for CVXPY users. Typically, this process is quite slow for large Disciplined Parametrized Programs (DPP). However, it is possible to improve upon the canonicalization by replacing the way 3D sparse tensors are represented. One of the major updates of CVXPY 1.3 was the addition of a SciPy-based backend, which allowed a performance speedup for certain types of optimization problems. This new backend offers an alternative to the original C++-based backend implementation (CVXCORE), and enables easier development. The SciPy backend has some limitations especially when it comes to manipulating sparse tensors with 3 dimensions. The current representation of a 3D sparse tensor is to use a list of 2D matrices. However, a sparse tensor of large dimension could be represented as a single strided matrix, which can make CVXPY backend operations orders of magnitude more efficient.

1 Introduction

This report documents the contributions I have made to CVXPY during the duration of Google Summer of Code 2023. The primary focus of the project was to improve the performance of canonicalization procedures for optimization problems, in particular, large Disciplined Parametrized Programs (DPPs). One key aspect of the project was to introduce a more efficient representation for 3D sparse tensors, going from a list of 2D matrices to a single strided matrix.

However, before being able to implement this novel idea, I had to understand the current backend and learn about the inner workings of CVXPY's canonicalization process. I also completed another backend in pure NumPy which served as a valuable reference implementation towards the project goal.

2 Community bonding period

During this period, I familiarized myself with the CVXPY codebase and tried to participate in online discussion forums. I quickly realized the project's significance as many members of the community brought up issues with their problems taking too long to compile. In particular, I collaborated with Andrash, a user who provided us with convex problems in plasticity modelling. This prompted my first pull request: benchmarks#14. The PR was a simplified reformulation of his research problem which could be benchmarked to evaluate future performance improvements. It was also quite encouraging to build upon the benchmarking foundation laid by Parth as part of Google Summer of Code 2022.

3 Background knowledge

This section serves as a reference to understand the scope of my contributions. I will first give a brief overview of CVXPY and its capabilities, then I will explain important concepts such as canonicalization and disciplined parametrized programming (DPP).

3.1 What does CVXPY actually do?

CVXPY is an open source Python-embedded modeling language for convex optimization problems [DB16]. It allows users to formulate their optimization problems in an intuitive way, following the standard mathematical notation. Provided this formulation followed a set of rules, commonly known as Disciplined Convex Programming (DCP), the user can typically obtain the optimal solution by calling the `solve` method.

However, it is important to note that CVXPY doesn't actually solve problems; instead, it acts as an interface for numerical solvers. For example, CVXPY is able to transform a problem formulated with high-level atoms into a standard form accepted by the solvers. It can also target a particular solver that would be best suited for the problem's optimization class. Convex problem classes such as Linear programming, quadratic programming and semidefinite programming form a hierarchy, where lower-class problems are subsets of higher-class problems. Naturally, there exists solvers for many different classes of problems, and it is often better to use as specific a solver as possible [AVDB18].

3.2 What is canonicalization?

The process of converting an optimization problem encoded in a domain specific language to a solver compatible form is called canonicalization [Gra04]. In many cases, canonicalizations can be separated into a sequence of reductions; which are functions that convert problems of one form to equivalent problems of another form. Two problems are equivalent if a solution for one can be converted to a solution for the other [AVDB18].

In the case of CVXPY, initial steps include turning maximization into minimization problems by negating the objective and expanding variable attributes to constraints. A later reduction is then to represent the constraints and objective as affine (a generalization of linearity which includes constants) expression trees. The leaves of the expression tree representing variables, constants and parameters, while the other nodes represent functions. Until the matrix stuffing step, the affine expression tree is kept in a symbolic form, preventing immediate matrix formation. The last step in the canonicalization is to transform the problem into the specific form required by each solver.

The following answer to a GitHub discussion#2142, written by my GSoC mentor, Philipp Schiele, contains an excellent summary of a step-by-step canonicalization for a quadratic problem.

3.3 What are parameters?

Contrary to variables which are unknown and optimized upon, parameters are treated as symbolic constants with unchanging properties whose value can be changed, but need to be specified before the problem is solved. Disciplined parametrized programming (DPP) is an extension to DCP which allows the user to solve convex optimization problems using parameters. DPP guarantees that the problem can be reduced to affine-solver-affine form which means that the canonicalization map can be represented as a sparse matrix $Q \in R^{n \times p+1}$, where n are the variables and $p+1$ are the parameters plus a scalar offset [AAB+19]. Solving a parametric problem multiple times with different parameter values will only require the canonicalization and matrix stuffing during the first solve. Subsequent solutions can be obtained by multiplying the sparse matrix Q with the corresponding parameter vector.

4 Coding period

This section details the code contributions I have made to CVXPY.

4.1 Pre-midterm evaluation

I made one significant PR during the pre-midterm period: cvxpy#2186 which introduces a new NumPy backend along with improvements to the testing suite. In contrast to the existing Scipy backend, which represents tensors as a list of 2D sparse matrices, the NumPy backend operates efficiently with dense 3-dimensional arrays. The main benefits of this new representation is the simplified implementation process and the elimination of slow Python lists. For these reasons, the NumPy backend serves as an excellent reference to understand the backend functions and could also be used for future 3-dimensional sparse implementations. In addition, the NumPy backend has some use cases, in particular, it is quite fast for small quadratic problems.

4.1.1 Tensor representation

I will now proceed to explain the fundamental data structure that lies behind CVXPY's canonicalization backend. When referring to the tensor representation, it is often more convenient to focus on the innermost layer. However, the tensor comprises of two additional nested dictionaries, structured as follows: dict{dict{np.ndarray}}. This is necessary because constraints often involve a subset of variables/parameters and we can use these dicts to only access the corresponding parts of the tensor.
The first layer stores the variable ID and keeps track of the mapping between variables and specific columns in the tensor. For instance, when dealing with two variables, each having two dimensions, then you would get the following columns: $x1, x2, y1, y2$. This is stored in the following manner: 'variableID_to_col'={1:0, 2:2}, where the first variable 'x' is assigned an offset of 0 and the second variable 'y' is assigned an offset of 2. The second layer stores the parameter ID and is necessary to locate the offset of each parameter slice in the inner tensor.

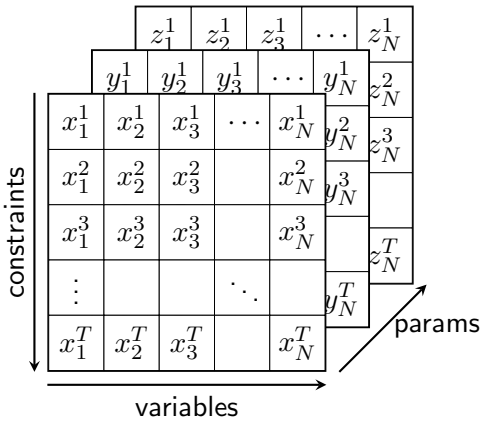

Figure 1: A 3-dimensional representation of the tensor.
(Note: the values inside the figure are not representative)

The central idea behind the canonicalization backend is to provide functions that manipulate the tensor representation, aligning with certain mathematical operations defined by the user. Typical operations include summation, multiplication, kronecker product, convolution, and horizontal/vertical stacking. In the following section, I will explain `sum_entries` which is the backend equivalent of using `cp.sum(x)` within the objective or constraints.

4.1.2 Example - sum_entries

The summation atom, often denoted as `sum_entries`, represents a fundamental mathematical operation that aggregates along the rows associated with a specific expression. It is important to note that in the context of the NumPy backend, which operates on 3d dense arrays, the row axis is located in the second dimension.
Consider the following simple example: define vector x of size n
x = Variable((n,))
$\begin{bmatrix} x_1 & x_2 & .. & x_{n-1} & x_n \end{bmatrix}$
x is represented as eye(n) in the A matrix, i.e.,
$\begin{bmatrix} x_1 & x_2 & .. & x_{n-1} & x_n \end{bmatrix}$
$\begin{bmatrix} 1 & 0 & ... & 0 & 0 \\ 0 & 1 & ... & 0 & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & ... & 1 & 0 \\ 0 & 0 & ... & 0 & 1 \end{bmatrix}$
`sum_entries(x)` means we sum along the row axis, i.e.,
$\begin{bmatrix} x_1 & x_2 & .. & x_{n-1} & x_n \end{bmatrix}$
$\begin{bmatrix} 1 & 1 & ... & ... & 1 & 1 \end{bmatrix}$

The following code snippet is a simplified implementation for `sum_entries` in the NumPy backend.
(Note: using the attribute `keepdims=True` ensures that the output is also a 3-dimensional tensor)

```
import numpy as np

def sum_entries:
        def func(x):
            return x.sum(axis=1, keepdims=True)

    view.apply_all(func)
    return view
```

4.2 Post-midterm evaluation

I made the following PR's during the final half of the project:

1. benchmarks#22 Adding all benchmarks from cvxpy main.
   This pull request transferred all the pre-existing benchmarks from 'test_benchmarks.py' to the official benchmark repository. From now on, every new PR to cvxpy will also benchmark over 20 unique optimization problems.

2. cvxpy#2213: Adding docstrings to the canonicalization backend.
   This pull request addresses documentation by adding detailed docstrings to every function present in canon_backend.py. This helps improve the readability and clarity of the codebase which is crucial for the on-boarding of future contributors.

3. cvxpy#2216: Stacked-Slices backend.
   This pull request is the main contribution of the project, it introduces the stacked-slices backend along with minor unit-tests updates. The following subsection details a parametrized example for `sum_entries` within the new backend. For a comprehensive understanding of other backend functions, please refer to the docstrings located under test_python_backend.py.

4.2.1 Comparison - sum_entries

Consider a similar example to 4.1.2, but this time we have a parametrized expression instead of a variable. To keep things simple we will also use a 3-dimensional expression instead of arbitrary n.
define parameter vector x with shape 9 by 3
x = Parameter((3,))

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`sum_entries(x)` means we sum along the row axis, i.e.,

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now how can we get `sum_entries(x)` from the original x?
For starters, we must multiply x with a 3 by 9 matrix to output a 3 by 3
We also need a matrix that will sum the rows of a particular 3 by 3 slice while ignoring the other parameter slices, i.e.,

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

This happens to be the Kronecker product between the identity with length three, sp.eye(3): $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ and a row vector filled with ones, np.ones(3): $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$

The following code snippet is a simplified implementation for `sum_entries` in the stacked-slices backend. Notice that there is an additional argument 'p', which is simply the size of the parameters in the tensor. The variable 'm' represents the number of rows per parameter slice.

```
import numpy as np
import scipy.sparse as sp

def sum_entries:
    def func(x, p):
        if p == 1:
            return sp.csc_matrix(x.sum(axis=0))
        else:
            m = x.shape[0] // p
            return (sp.kron(sp.eye(p, format="csc"), np.ones(m)) @ x).tocsc()

    view.apply_all(func)
    return view
```

4.3 Performance analysis

The new stacked-slices backend performs quite well in the aforementioned benchmarks suite. Most optimization problems have seen drastic improvements, in some cases the compilation time is even 100 times faster. It is safe to say that the stacked-slices backend should outperform the original SciPy backend in most use cases. Despite the improvements, when compared to the CPP backend a few benchmarks are still slower. In the figure below you can see a performance comparison for both DPP and `ignore_dpp`.

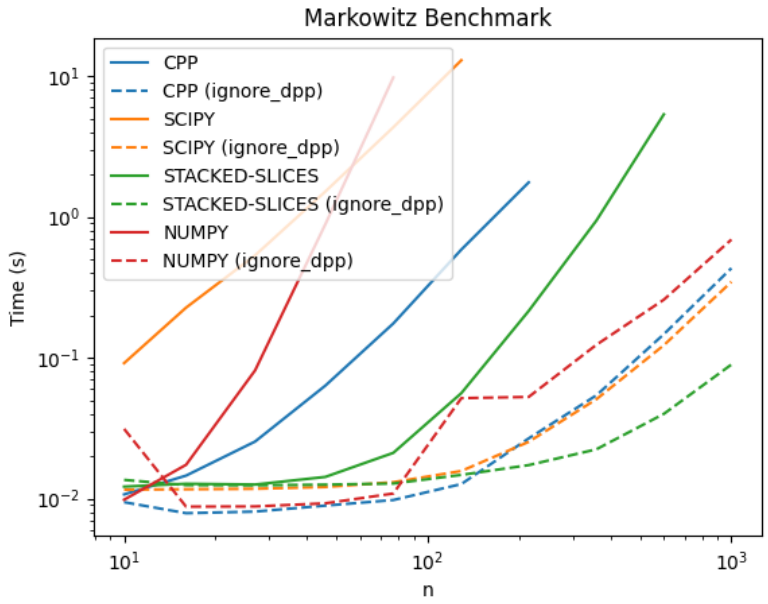

Figure 2: Comparison between all CVXPY backends for a Markowitz covariance matrix

4.4 Complexity analysis

The new stacked-slices backend brought along interesting discoveries in complexity and in the selection of sparse formats. To begin, we noticed that when dealing with many empty rows, compressed sparse column (CSC) outperforms compressed sparse row (CSR). This is because CSR stores indices pointers for every row, which becomes redundant when most rows are empty. Unfortunately, row slicing in CSC is much slower than in CSR because it traverses through every column to locate the corresponding row indices. This presents us with a classic trade-off between less memory usage and slightly higher computation time.
In terms of space complexity, it is important to note that changes have only been made to structural representations, leaving the underlying data unchanged. Therefore I believe the memory usage will be relatively similar to the SciPy backend. In terms of computational complexity, the stacked-slices backend capitalizes on the strength of sparse linear algebra by constructing more extensive sparse matrices. Furthermore, it is able to exploit mathematical simplifications where the original SciPy backend is constrained to iterating through every parameter slice.

5 Future work

This section details potential future improvements to the canonicalization backend of CVXPY.

5.1 python-graphBLAS Backend

The integration of this backend into CVXPY is already in the planning stages, and its development will commence once the stacked-slices backend has been successfully merged. The GraphBLAS standard provides a high-level abstraction of using graph algorithms and semi-rings to perform linear algebra operations. Notably, SuiteSparse comprises a collection of C libraries designed to address sparse linear algebra problems, and Python-graphBLAS serves as the interface for SuiteSparse's implementation of graphBLAS.
Although python-graphBLAS has mainly been developed to accelerate graph algorithms, preliminary benchmarks indicate that its sparse matrix operations are often already much faster than the SciPy equivalent. Concerning implementation, the Python-graphBLAS backend will have the same tensor representation as the stacked-slices backend. It will only require a few modifications to the sparse library and related operations. Beyond performance enhancements, another potential benefit of adopting a graphBLAS backend is the ability to use their just-in-time compiler with simple configuration changes. Finally, the GraphBLAS user guide presents various parameter settings such as hyper-sparsity, custom semi-rings and masking which can be fine-tuned for CVXPY.

5.2 Rust backend

Exploring other backend options, a particularly interesting idea would be using Rust, a system language that has increased significantly in popularity recently. Rust is an excellent language due to its type safety, performance and concurrency support. Its ownership and borrowing system help avoid common mistakes when dealing with memory management. However, being a relatively new language, Rust lacks Python's vast range of matrix libraries and will generally be a bit more challenging to maintain. There is also an open-source Rust interior-point solver developed by the Oxford Control Group named Clarabel which demonstrates the growing significance of the language amongst the optimization community.

5.3 Adding parameter support to 'exotic' backend functions

There are currently four functions which don't support parameters in the backend: division, convolution and right and left Kronecker product. These functions are labeled "exotic" due to their relative increase in complexity and less frequent usage. Mathematically, these functions should follow a similar pattern to the others, i.e., keep track of slices of a tensor, apply an operation on each slice and stack the slices back vertically. Although these operations don't typically involve parameters, there are some use cases and it would render the backend more complete.
However, we must keep in mind that new features must be added to all backends to avoid discrepancies in the API functionality. The default CPP backend (CVXCORE) faces many challenges when attempting to make changes due to lack of maintenance and legacy reasons. Furthermore, the CPP backend can still outperform other backends in certain scenarios, thereby necessitating its continued availability. It would probably be prudent to wait until the CPP backend has been phased out before going forward with this new feature.

6 Summary

The main focus of the GSoC project was to improve the compilation time of convex optimization problems. Before tackling this project, I was told that improving canonicalization performance was one of the most high impact improvements for CVXPY. At the time I didn't know what any of these things meant, but after spending a few months as a member of the community and developing new backends, I finally understand the importance of this project. Practically all CVXPY users will benefit from canonicalization performance, in fact, it is by far the most requested feature from the community.

In the beginning of the report, I outlined some important concepts that I found extremely important in this ongoing journey of learning. I then cover some technical examples of the backend along with the equivalent code contribution. The first major milestone was the completion of a canonicalization backend written in NumPy using 3 dimensional dense tensor representations. Although the performance of this backend cannot compare to its sparse counterparts, the NumPy backend lays a foundation for future N-dimensional sparse libraries such as SciPy sparse arrays or Pydata sparse.

The next important contribution was the stacked-slices backend, where the tensor representation is now a strided 2d sparse matrix. It is simpler to think of it as a dimension reduction from a 3d tensor, where every slice is stacked vertically, hence the name. This backend introduces many interesting mathematical shortcuts that perform equivalent operations to the other backends. Many of these operations make clever use of the Kronecker and index changes. I really recommend the curious reader to dive into the source code to learn more about it.

Finally, I showcase some performance results from preliminary benchmarks and give my perspective on the complexity of this new tensor representation. I also give detailed explanations on the next steps for further canonicalization improvements. Although we have accomplished the main objectives of the GSoC project, there is a lot more work to be done and even more ideas to be explored.

I want to conclude this report by thanking my mentors, Philipp Schiele and Steven Diamond, for which the completion of this project wouldn't have been remotely close to possible.

References

[AAB+19] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and Zico Kolter. Differentiable convex optimization layers, 2019.

[AVDB18] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[DB16] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[Gra04] Michael Grant. *Disciplined Convex Programming*. PhD thesis, Stanford University, 2004.