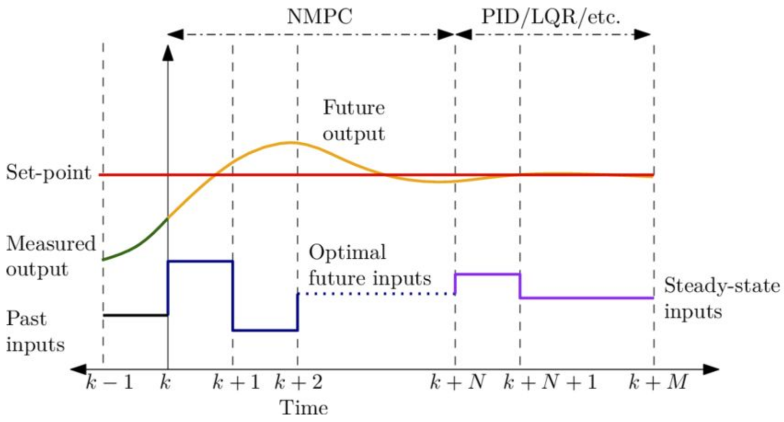
CPU Multi Threaded Dual-Gradient Projection for Embedded MPC

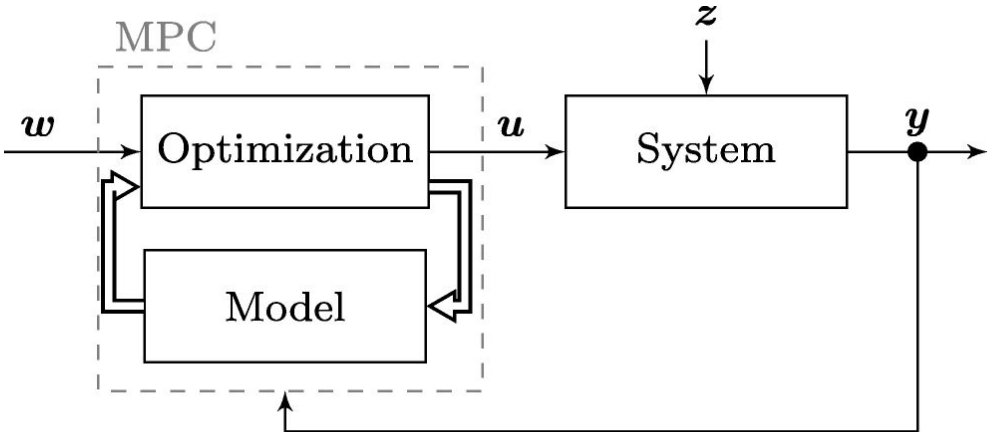
ECE 5772 - High Performance Embedded Programming

Joseph Volcic

Overview of Model Predictive Control (MPC)

- Model Predictive Control is a control strategy for dynamic systems. MPC solves for optimal inputs to a system by predicting its future behavior using a mathematical model and determining the best sequence of control actions that minimize a cost function while respecting system constraints.





Overview of The QP Problem

Consider the following finite-time optimal control problem formulation for MPC (bottom left). Using quadratic costs, we can repackage the optimal control problem on the left as a convex quadratic program (QP) (bottom right)

Cost Function

$$\min_{z} \ \ell_N(x_N) + \sum_{k=0}^{N-1} \ell_k(x_k, u_k)$$

Current State

$$\text{s.t. } x_0 = p$$

Dynamics and Constraints

$$x_{k+1} = A_k x_k + B_k u_k + f_k$$

$$F_k x_k + G_k u_k \leq c_k$$

$$k = 0, \ldots, N-1$$

Terminal State Constraints (good for stability)

$$F_N x_N \leq c_N$$

Quadratic Costs

$$\begin{aligned} \min_{z} \quad & \frac{1}{2} z' M z + (Cp + g)' z \\ \text{s.t.} \quad & Gz \leq Ep + b \end{aligned}$$

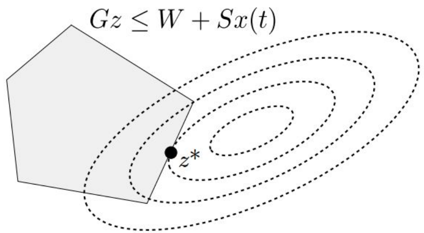
$$z \triangleq \begin{bmatrix} u'_0 & \ldots & u'_{N-1} \end{bmatrix}' \in \mathbb{R}^n$$

$$\ell_N(x) = \frac{1}{2} x' Q_N x + q'_N x + s_N$$

Overview of the GPAD Algorithm

- Normally, solving the QP for MPC is incredibly resource-intensive (active set, interior point methods, explicit MPC), and thus unfit for embedded applications.
- However, in “Simple and Certifiable Quadratic Programming Algorithms for Embedded Linear Model Predictive Control” (Bemporad, Patrinos), a dual fast gradient-projection approach (GPAD) is introduced for solving QP problems in a lightweight manner, fit for embedded systems, and can be easily executed on “p” parallel processors. The four steps are shown below:

$$\begin{aligned} \min_z \quad & \frac{1}{2} z' M z + (Cp + g)' z \\ \text{s.t.} \quad & Gz \leq Ep + b \end{aligned}$$



$$\begin{aligned} w_\nu &= \boxed{y_\nu + \beta_\nu (y_\nu - y_{\nu-1})} \\ \hat{z}_\nu &= \boxed{-M_G w_\nu} - g_P \\ z_\nu &= \boxed{(1 - \theta_\nu) z_{\nu-1} + \theta_\nu \hat{z}_\nu} \\ y_{\nu+1} &= [w_\nu + \boxed{G_L \hat{z}_\nu} + p_D]_+ \end{aligned}$$

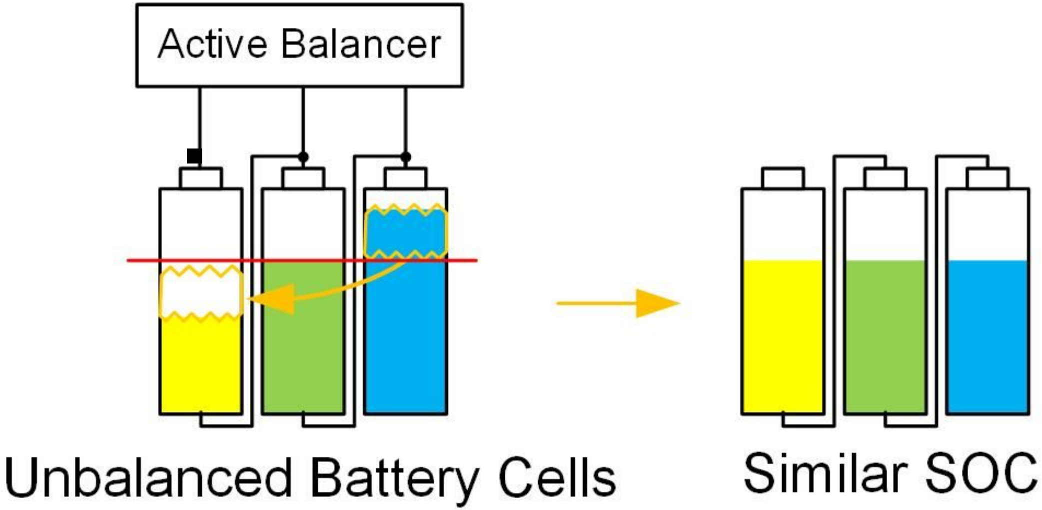
Step	**Flop Count**
(4a)	$3m$
(4b)	$2n_u N m$
(4c)	$3m$
(4d)	$2n_u N m + m$
Total	$4n_u N m + 7m$

Source: A. Bemporad “Model Predictive Control”

MPC Optimization Problem

- This work will focus specifically on the battery changing problem. However the approach will be generalized to any MPC.

- In the battery charging case we have to abide by a couple rules. Current into a cell must equal current out of a cell, cell max voltage, cell min voltage. These are all defined in the M_G and G_L matrices.



Proposed Methodology for Optimization

- Partitioning the data into smaller section allows us compute parallel section of data.

- Running multiple steps of the algorithm at the same time.

- Matrix Compression reduces the total number of data elements operated on.

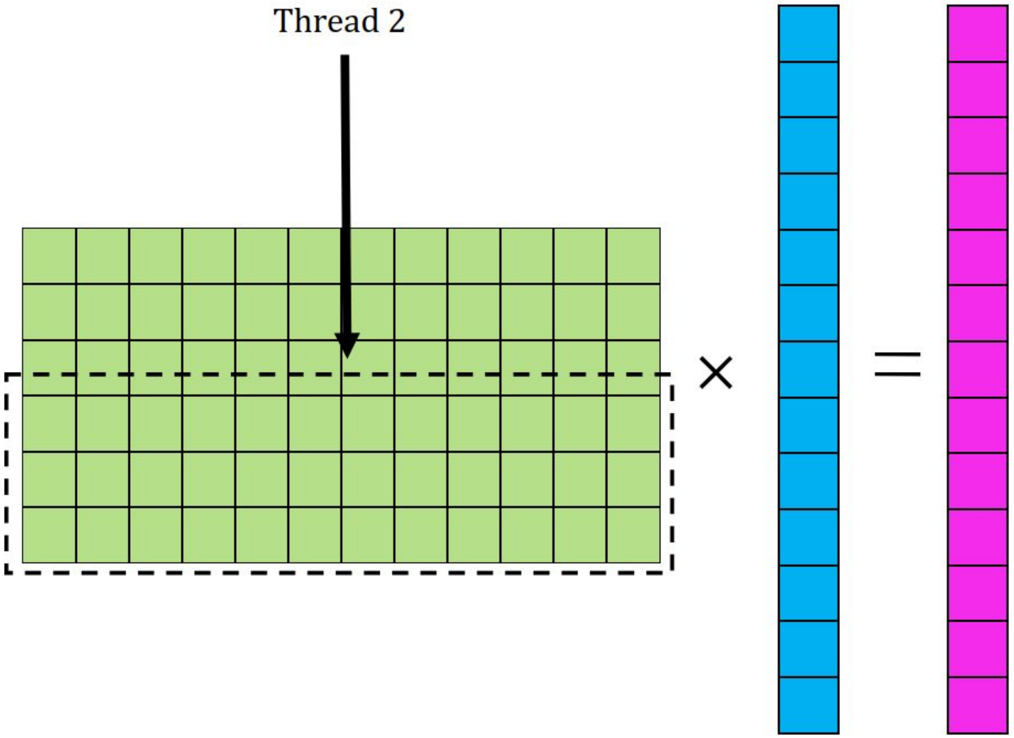
- Parallel SAXPY, however the data is too small to see benefit from this.

Optimization Methodology - Sparse Matrix-Vector Multiplication

- Matrix Vector multiplication is very similar to completing the dot product on multiple rows of data.

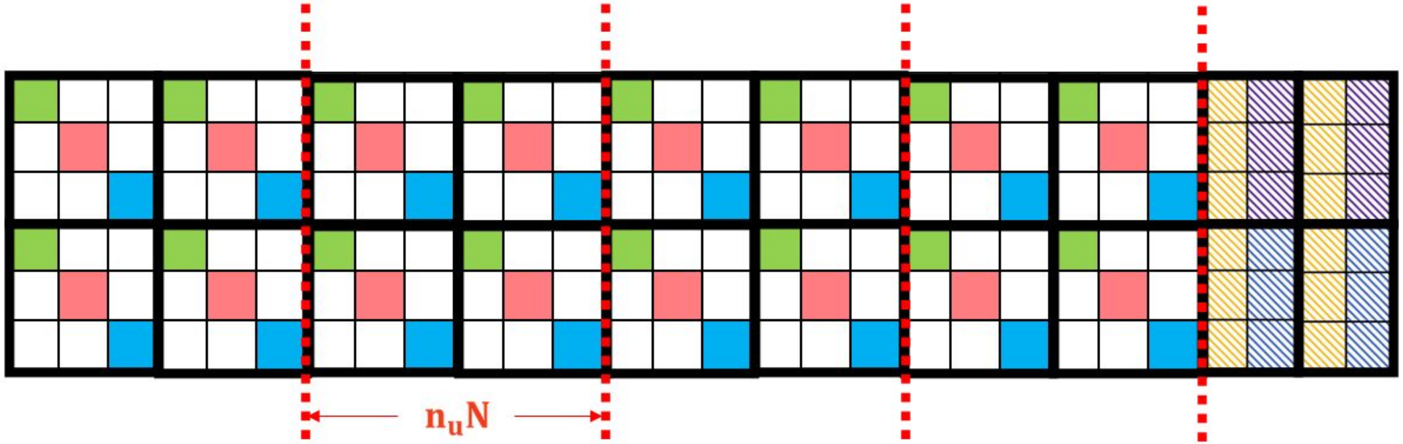
- To parallelize matrix vector multiplication the matrix can be partitioned into multiple chunks for threads to compute simultaneously.

- Built using a Blas framework and pthreads. Pthreads allows me to calculate how to split the data before computation.

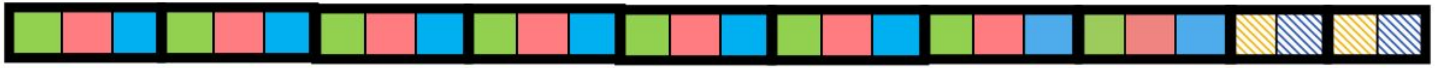


Optimization Methodology - Matrix Compression

- Large matrices in step2 and step 4 can be compressed to remove zero elements.
- The compressed array is represented as two array to maintain data and position.
- Padding is applied to rows with less elements to ensure each row has the same number of elements. (Important for GPU operations)



Non-Compressed Matrix



Compressed Matrix

Proposed Methodology - Parallel Steps

- Recall the GPAD algorithm: Upon direct observation, we notice that some steps further ahead do not depend on the results calculated in previous steps.
- We can draw up a directed acyclic graph to represent tasks can be done in parallel, and which need to wait for others to be completed.
- This problem cannot be pipelined due to each steps dependence of completion of the previous step.

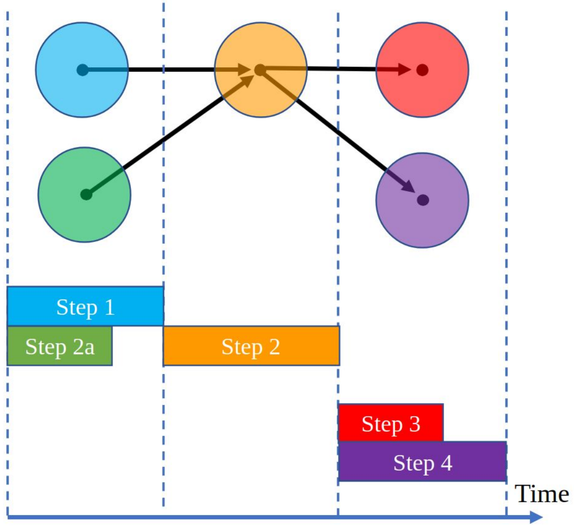
$$w_\nu = y_\nu + \beta_\nu (y_\nu - y_{\nu-1})$$

$$\hat{z}_\nu = -g_P$$

$$\hat{z}_\nu = -M_G w_\nu$$

$$z_\nu = (1-\theta_\nu) z_{\nu-1} + \theta_\nu \hat{z}_\nu$$

$$y_{\nu+1} = [w_\nu + G_L \hat{z}_\nu + p_D]_+$$



Results - Testing Methodology

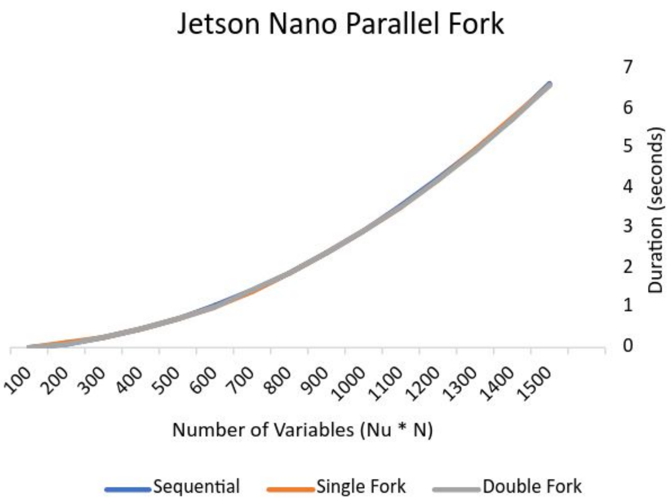
Testing Machines

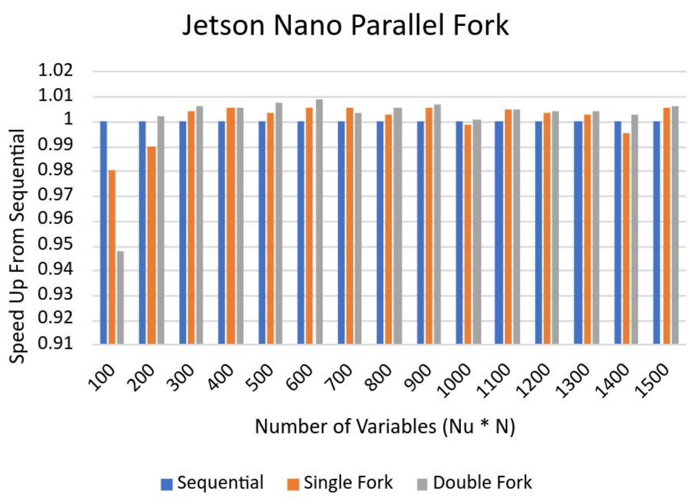
Jeston Nano	Desktop
Cores: 4 Threads: 4 Clock Speed: 1.9 GHz	Cores: 8 Threads: 16 Clock Speed: 3.2 GHz

Results - Parallel Steps Jetson

- Running multiple steps in parallel provided a slight speed up, however nearly negligible.

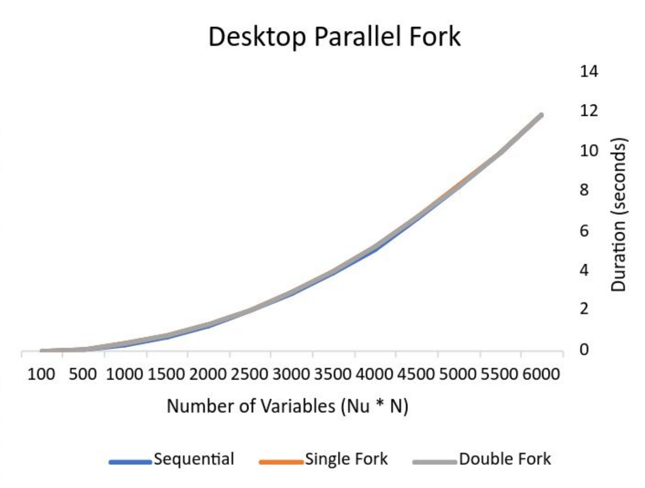
- Two implementations were tested. Running steps 1 and 2a in parallel with steps 3 and 4 in parallel. As well as running only steps 3 and 4 in parallel.

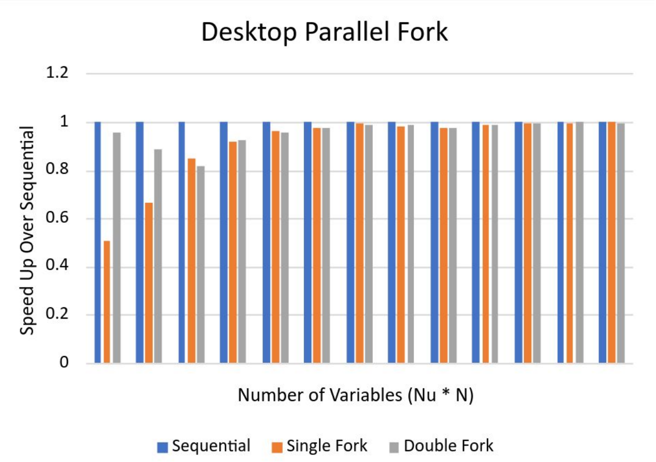




Results - Parallel Steps Desktop

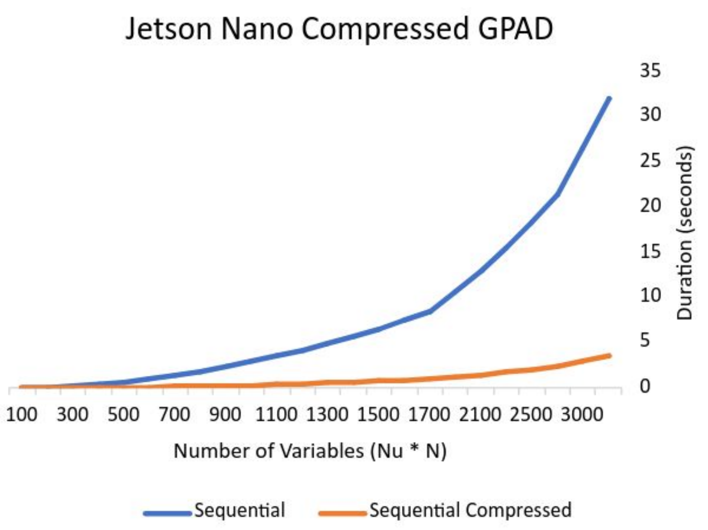
- The same tests were performed on the desktop, and the sequential version out performed both parallel versions.
- The steps the can be run in parallel are very small causing more overhead then computation gain.
- Due to the lack of performance this was cut in future tests.

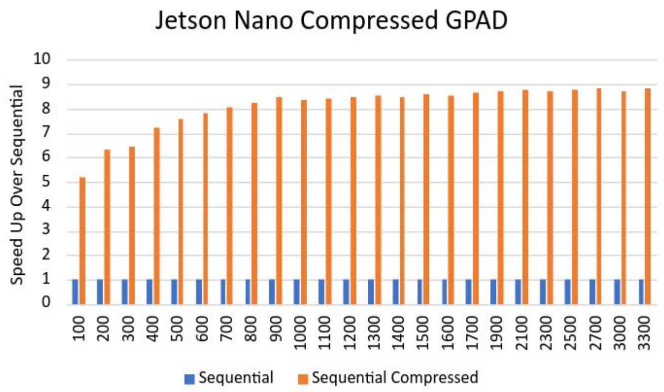




Results - Sequential Matrix Compression Jetson

- Compressing the matrices and then performing multiplication offered around a 9 times speed up on the Jetson.

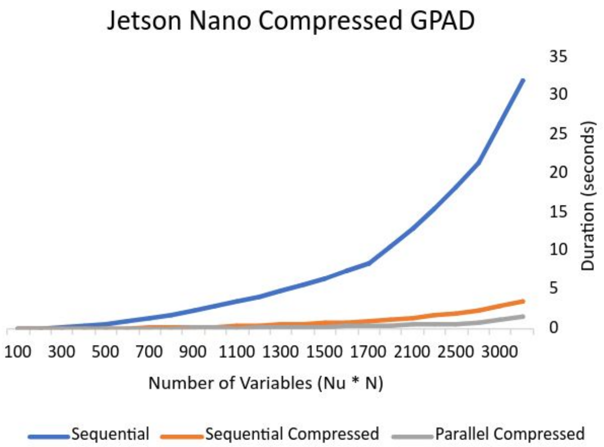


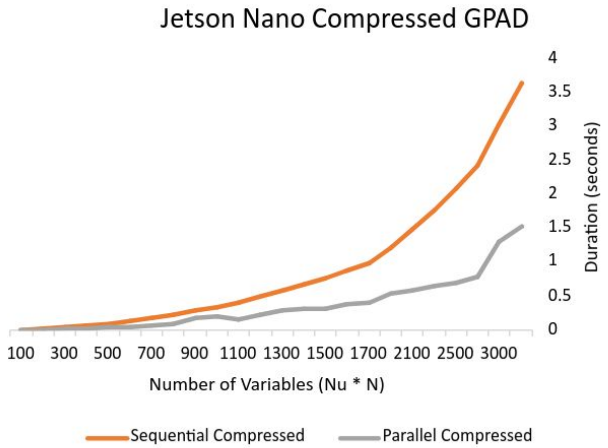


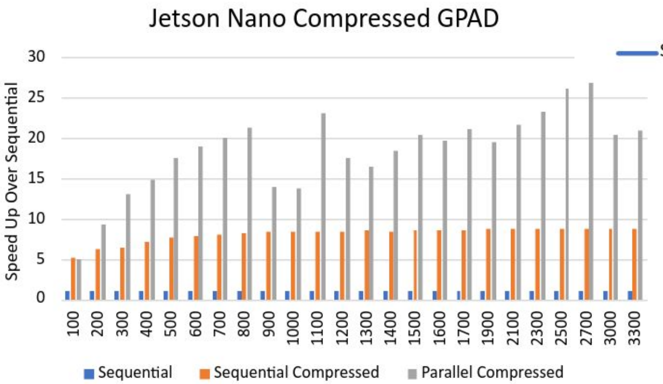
Results - Parallel Matrix Multiplication Jetson

- Running the matrix vector multiplication in parallel lead to a significant additional speed up, around 2-3 times faster

- Testing was done with 4 threads

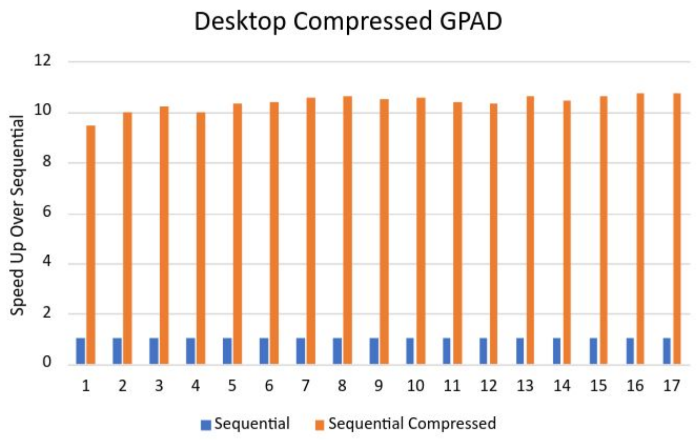


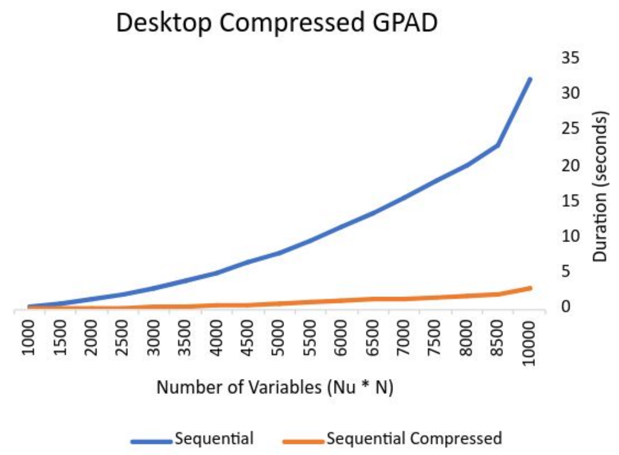




Results - Sequential Matrix Multiplication Desktop

- Similar to the Jetson there was around a 10 times speed up after matrix compression.

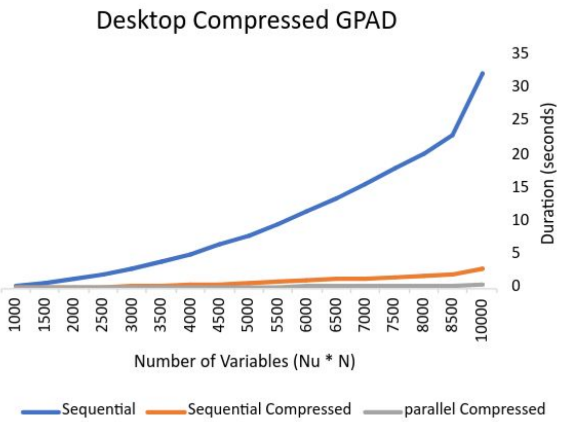


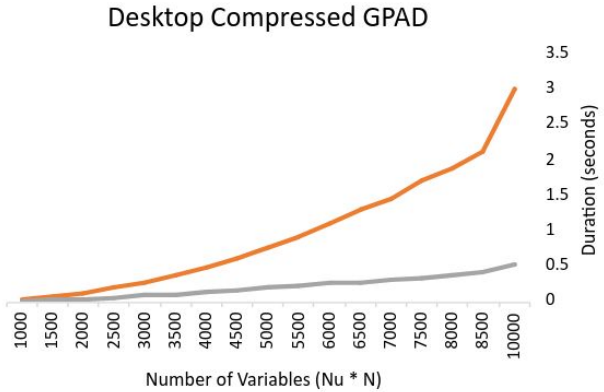


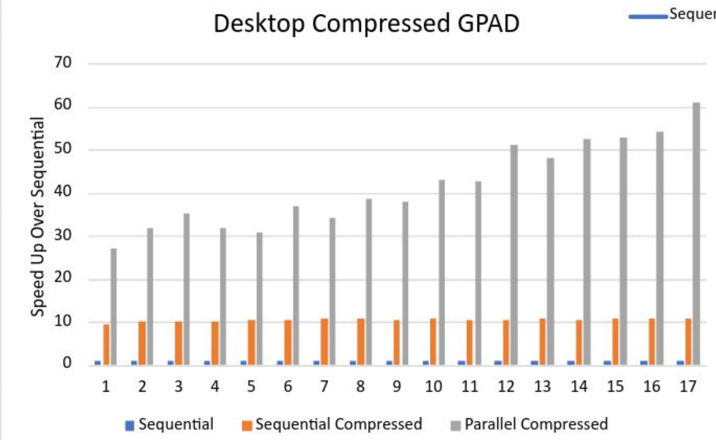
Results - Parallel Matrix Multiplication Desktop

- Running the matrix vector multiplication in parallel lead to a significant additional speed up, around 3 - 6 times faster

- Testing was done with 8 threads







Conclusions

- CPU multi-threaded acceleration of GPAD demonstrates significant performance improvements over traditional a single thread CPU implementation.

- Optimizations like matrix compression, data partitioning, are key to achieving real-time performance.

- Future work includes improving the parallel steps approach, as well as comparing the GPU and CPU multi-threaded results.