GPU-GPAD Cookbook: Matrix Sizes, Evaluation of Sparsity, and Opportunities for Parallelization

Luke Nuculaj, Joseph Volcic, Shreyas Renukuntla, and Matthew Redoute
ECE 5770: GPU Accelerated Computing

November 2, 2024

Recall the optimal control problem

$$V^*(p) \triangleq \min_z \quad \frac{1}{2}z^\top M z + (Cp+g)^\top z + \frac{1}{2}p^\top Y p \tag{1a}$$
$$\text{s.t.} \quad Gz \leq Ep + b \tag{1b}$$

where m is the number of constraints, n_u is the size of the control vector u, and N is the length of the prediction horizon. The GPAD algorithm solves (1) by repeating the following steps:

$$w_\nu = y_\nu + \beta_\nu(y_\nu - y_{\nu-1}) \tag{2a}$$
$$\hat{z}_\nu = -M_G w_\nu - g_P \tag{2b}$$
$$z_\nu = (1-\theta_\nu)z_{\nu-1} + \theta_\nu \hat{z}_\nu \tag{2c}$$
$$y_{\nu+1} = [w_\nu + G_L\hat{z}_\nu + p_D]_+ \tag{2d}$$
$$y_0 = y_{-1} = 0, \;\; z_{-1} = 0,$$

where $y \in \mathbb{R}^m$ is the dual vector, $M_G \triangleq M^{-1}G^\top$, $g_P \triangleq M^{-1}(Cp+g)$, $G_L \triangleq \frac{1}{L}G$, $p_D \triangleq -\frac{1}{L}(Ep+b)$, $\nu \in \mathbb{N}$ is the iteration number, and β_ν is a scalar sequence developed by the following recursions

$$\theta_{\nu+1} = \frac{\sqrt{\theta_\nu^4 + 4\theta_\nu^2} - \theta_\nu^2}{2} \tag{3a}$$
$$\beta_\nu = \theta_\nu(\theta_{\nu-1}^{-1} - 1) \tag{3b}$$
$$\theta_0 = \theta_{-1} = 1.$$

This document first provides a table with the dimensions of all matrices used in (2), then breaks down the GPAD algorithm step by step, examining the sparsity of all associated matrices and evaluating the potential for GPU parallelization.

1 Matrix Sizes

We list the sizes of the GPAD matrices in Table 1: m is the number of constraints, n_u is the number of control inputs (that is, how many balancing currents in our system) and N is the prediction horizon.

Matrix	w	y	β	z	M_G	g_P	θ	G_L	p_D
Size	$m \times 1$	$m \times 1$	1×1	$n_u N \times 1$	$n_u N \times m$	$n_u N \times 1$	1×1	$m \times n_u N$	$m \times 1$

Table 1: GPAD Matrices Sizes

2 Algorithm Steps

In this section, we will look at each step in-depth, computing the number of flops required for each step, evaluating sparsity (i.e. if a matrix is 90% sparse, it means 90% of its elements are zero), and opportunities for parallelization. The variables in red indicate that they were computed offline and are constant variables.

2.1 Step 1: $w_\nu \leftarrow y_\nu + \beta_\nu(y_\nu - y_{\nu-1})$

This step takes the difference of two vectors, multiplies them by a scalar, and adds the result to another vector.

Evaluation of Sparsity: There are no redundant elements in y, so all matrices involved in this step are dense. **No sparsity**.

Flops Count: Because $y \in \mathbb{R}^m$, the subtraction requires m flops, the scalar multiply with β requires m flops and the addition to y_ν requires m flops, totaling **3m flops**.

Parallelization: This is SAXPY as we did in the first assignment, so each thread can compute a unique element of w_v. **Threaded SAXPY**.

2.2 Step 2: $\hat{z}_\nu \leftarrow -M_G w_\nu - g_P$

This step performs a matrix-vector multiplication and subtracts a vector from the result.

Evaluation of Sparsity: We first consider the general case of computing $M_G \leftarrow M^{-1}G^\top$. M is positive definite, so $M = LL^\top$ is a Cholesky decomposition (necessary and sufficient condition). $M^{-1} = (LL^\top)^{-1} = (L^\top)^{-1}L^{-1} = Q^\top Q$, where Q is a lower triangular matrix. $M_G = Q^\top Q G^\top$, and because the entries of G are circumstantial (that is, G may or may not have embedded triangular blocks), we cannot guarantee sparsity in the general case. However, we can consider the battery-balancing case and determine if there are any structural patterns we can exploit for computational gains. For the battery-balancing case, we have

$G = \begin{bmatrix} M_{AB}^\top & -M_{AB}^\top & I_{(n_uN)} & -I_{(n_uN)} & K^\top & -K^\top \end{bmatrix}^\top$, where

$$M_{AB} \in \mathbb{R}^{(n_uN)\times(n_uN)} = \begin{bmatrix} B & 0 & 0 & \dots & 0 \\ AB & B & 0 & \dots & 0 \\ A^2B & AB & B & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A^{N-1}B & A^{N-2}B & A^{N-3}B & \dots & B \end{bmatrix}$$

$$K \in \mathbb{R}^{N\times(n_uN)} = \begin{bmatrix} \mathbb{1}_{n_u}^\top & 0 & 0 & \dots & 0 \\ 0 & \mathbb{1}_{n_u}^\top & 0 & \dots & 0 \\ 0 & 0 & \mathbb{1}_{n_u}^\top & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \mathbb{1}_{n_u}^\top \end{bmatrix}$$

Multiplying by $Q^\top Q$, we get the following:

$$Q^\top Q G^\top = \begin{bmatrix} Q^\top Q M_{AB}^\top & -Q^\top Q M_{AB}^\top & Q^\top Q & -Q^\top Q & Q^\top Q K^\top & -Q^\top Q K^\top \end{bmatrix} \quad (4)$$

$$Q^\top Q M_{AB}^\top = \begin{bmatrix} B & B & B & \dots & B & B \\ D_{2,1} & B & B & \dots & B & B \\ D_{3,1} & D_{3,2} & B & \dots & B & B \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ D_{N,1} & D_{N,2} & D_{N,3} & \dots & D_{N,N-1} & B \end{bmatrix} \quad (5)$$

where $D \in \mathbb{R}^{n_u \times n_u}$ are distinct diagonal matrices. So, (5) is $100 \times (1 - \frac{1}{n_u})\%$ sparse. Then

$$Q^\top Q = \begin{bmatrix} D_{1,1} & D_{1,2} & D_{1,3} & \dots & D_{1,N} \\ D_{2,1} & D_{2,2} & D_{2,3} & \dots & D_{2,N} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ D_{N,1} & D_{N,2} & D_{N,3} & \dots & D_{N,N} \end{bmatrix} \quad (6)$$

and is symmetric. (6) is also $100 \times (1 - \frac{1}{n_u})\%$ sparse. Finally, $Q^\top Q K^\top$ is not sparse. So, M_G is $100 \times (\frac{2n_u-2}{2n_u+1})\%$ **sparse** for $n_u \geq 1$. Notice how this is not dependent on N.

Flops Count: With a flattened block $M_G^{AB,*} \in \mathbb{R}^{(N\times n_uN)}$ (see "Parallelization" section for this step), taking the element-wise product of each row u of $M_G^{AB,*}$ with the associated section of w_ν requires n_uN flops. Then, we do $\approx n_uN$ adds for one element of the result. Repeating this process N times for one block of M_G gives $2n_uN^2$ flops per block. In the battery balancing case, we have four of these collapsible blocks in M_G, so $8n_uN^2$ flops. The other two $n_uN \times N$ blocks in M_G are not collapsible, albeit there are redundant structures contained therein. In the interest of time, we will perform normal matrix multiplication on this section, requiring $2n_uN^2$ flops. As for the subtraction with g_P, that takes another n_uN flops, which is negligible in comparison to the already-computed quadratic term. So, the total computation required for this step is $\approx 10n_uN^2$ **flops**.

Parallelization: To parallelize GPAD, the authors of the original paper suggest using "m parallel processors". For this step, there are a couple of ways to perform the parallel

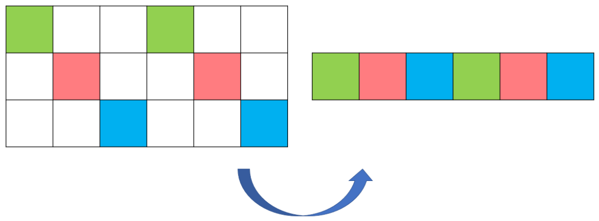

Figure 1: Condensing (5) for the case of $n_u = 3$

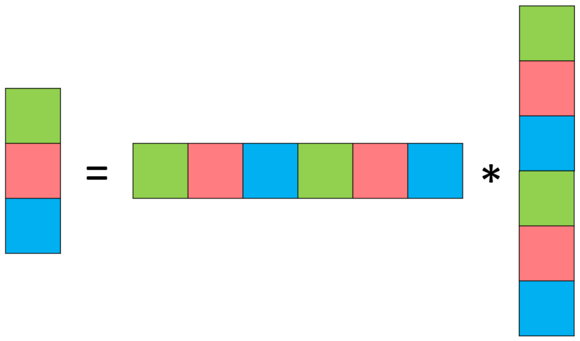

Figure 2: Multiplying $M_G^{AB,*}$ by $w_{\nu,i}$

computation with $\propto m$ threads. We can naively store the entire sparse matrix M_G in memory, and for each element in the result $-M_G w_\nu$, have all m threads fetch an entire row u from memory, and perform the dot product $u \cdot w_v$. Although this is the easiest to implement, recall from our earlier derivation that a significant majority of the elements in M_G are redundant zeros, so this approach sees a trade-off between ease of implementation and efficient use of memory.

Alternatively, consider the method of condensing the first two blocks of $-M_G$ shown in **Fig. 1**, where we consider every block in (5) and flatten them. For convenience, we will use $M_G^{AB} := (5) \in \mathbb{R}^{(n_u N \times n_u N)}$, and its flattened form will be $M_G^{AB,*} \in \mathbb{R}^{(N \times n_u N)}$. From here we can perform threaded multiplication as shown in **Fig. 2**, where the memory accesses are naturally coalesced, making full use of the GPU's memory bandwidth. For a flattened row u of $M_G^{AB,*}$, we can use $n_u N$ threads to perform the element-wise multiplication with w_ν, with the result $x \in \mathbb{R}^{n_u N} = u \odot w_\nu$. Then, we can assign N threads to do the accumulation on x, where each thread will have a distinct starting index, adding up every n_u^{th} element in x. Repeat this process for every row until the matrix multiplication is complete.

As a third option, we can expand on the previous approach by partitioning $M_G^{AB,*}$ into halves or quarters along the rows and launching separate kernels to compute the corresponding section of the result. **Matrix flattening, element-wise operations, interleaved sums, and *possibly* shared memory tiling**.

2.3 Step 3: $z_\nu \leftarrow (1 - \theta_\nu) z_{\nu-1} + \theta_\nu \hat{z}_\nu$

This step multiplies two vectors by two distinct scalars and adds the results

Evaluation of Sparsity: There are no redundant elements, so all the matrices involved in this step are dense. **No sparsity**.

Flops Count: Because $z \in \mathbb{R}^{(n_u N)}$, the scalar multiplications each require $n_u N$ flops, and the vector addition requires $n_u N$ flops, totaling **$3n_u N$ flops**.

Parallelization: This is SAXPY as we did in the first assignment, so each thread can compute a unique element of z_ν. **Threaded SAXPY**.

2.4 Step 4: $y_{\nu+1} \leftarrow [w_\nu + G_L \hat{z}_\nu + p_D]_+$

This step ...

Evaluation of Sparsity: ... **...**.

Flops Count: ... **... flops**.

Parallelization: ... **...**.

As an applied example case, we consider the series-connected battery balancing problem, which has a number of constraints $m = 4n_u N + 2N$.

The averaged execution times are visualized in **Fig. 3**, which are averaged over various sizes of the optimization variable z and executed in MATLAB R2023b on an Intel i7 CPU running at 2.9 GHz with 32 GB of RAM.

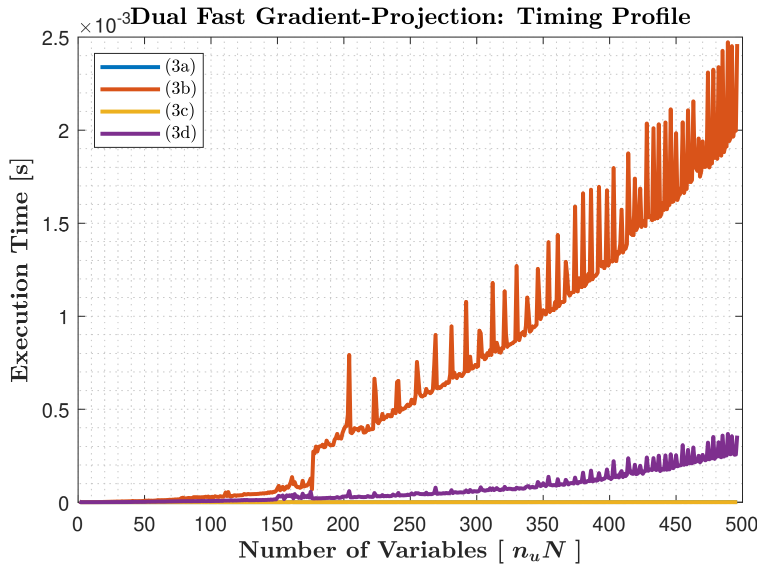

Figure 3: Timing profile for the GPAD algorithm averaged over 100 trails for each number of variables.

It is clear to see that (3b) and (3d) are the bottlenecks in the GPAD algorithm – this is reinforced by the fact that their computation grows quadratically with the number of variables in the MPC problem while (3a) and (3c) are only linear. Upon inspection, the bottlenecks are matrix multiplications, additions, and element-wise comparisons, which are easily parallelizable ripe for a GPU implementation. To complicate things further, the larger matrices involved in these computations are sparse (almost all of their elements are zero), so special considerations will have to be made to matrix multiplication kernel functions so as to cut back on unnecessary computation and memory overhead.

References