GPU-Centric Formulation and Techniques for SCAD-Penalized Multi-Sample Subclone Reconstruction

Yu Ding

April 14, 2025

1 Introduction

This document provides a comprehensive summary of the techniques used to implement a SCAD-penalized ADMM (Alternating Direction Method of Multipliers) algorithm on a GPU, focusing on multi-sample subclone reconstruction for cancer genomic data. The original objective addresses the problem of clustering Single Nucleotide Variants (SNVs) across multiple tumor samples, where each SNV is represented by an unknown parameter vector $\mathbf{p}_i \in \mathbb{R}^M$. A group SCAD penalty is imposed on the pairwise ℓ_2-norm differences $\|\mathbf{p}_i - \mathbf{p}_j\|_2$, enforcing that many SNVs share identical or near-identical parameter vectors.

Standard approaches to this problem can be computationally prohibitive when S (number of SNVs) or M (number of samples) grows large. Consequently, offloading major parts of the computation to a GPU becomes attractive. By harnessing `PyTorch`'s features for sparse and dense linear algebra, batch operations, and elementwise vectorized functions, we achieve a workflow that can handle thousands of SNVs efficiently.

Below is an explicit summary of the GPU-focused techniques:

1. **Data Representation on GPU**: Converting all major arrays—read counts, copy numbers, logistic-scale parameters, etc.—into `torch.tensor` objects on the GPU.
2. **Sparse Matrix Construction**: Building the large pairwise difference operator Δ as a sparse matrix on the GPU.
3. **Local Quadratic Approximation (IRLS)**: Formulating the logistic negative log-likelihood's second-order expansions as diagonal matrices plus constant vectors, stored as $\mathbf{B}$ and $\mathbf{A}$, on GPU.
4. **ADMM Splitting**: Introducing auxiliary variables $\boldsymbol{\eta}_{ij}$ (each living on GPU) for the pairwise differences and solving subproblems iteratively.
5. **Group SCAD Thresholding on GPU**: Executing the non-convex thresholding step in a single large, vectorized pass using boolean masks.
6. **Conjugate Gradient (CG) Solver**: Solving the linear system $(\mathbf{B}^\top\mathbf{B} + \alpha\,\Delta^\top\Delta)\mathbf{w} = \text{RHS}$ via a custom GPU-based CG loop, crucial for large-scale problem sizes.
7. **Clustering and Post-processing**: Final cluster assignment, labeling, and potential merges performed partially on GPU or CPU, depending on the scale of index-based merges.

The remainder of this document elaborates on these seven items and how they unify into a single GPU-based SCAD-penalized ADMM pipeline.

2 Problem Formulation with Focus on GPU Implementation

2.1 Multi-Sample SCAD Objective

We have S SNVs, each observed in M tumor samples. We define $\mathbf{p}_i = (p_{i1}, \ldots, p_{iM}) \in \mathbb{R}^M$ to be the logistic-scale parameters for SNV i. The read counts r_{ij} and coverages n_{ij} follow a (quasi)binomial model, so the negative log-likelihood for a single SNV–sample pair is

$$-\log P(r_{ij} \mid p_{ij}) \;=\; -\, r_{ij}\, \log(\theta_{ij}(p_{ij})) - (n_{ij} - r_{ij})\, \log\bigl(1 - \theta_{ij}(p_{ij})\bigr),$$

where $\theta_{ij} = \frac{e^{p_{ij}}}{1+e^{p_{ij}}}$ or adjusted forms incorporating copy numbers and tumor purity. Summing over all (i, j):

$$-\ell(\mathbf{p}) \;=\; \sum_{i=1}^{S}\sum_{j=1}^{M}\Bigl\{-\, r_{ij}\, \log\bigl(\theta_{ij}(p_{ij})\bigr) - \bigl(n_{ij} - r_{ij}\bigr)\, \log\bigl(1 - \theta_{ij}(p_{ij})\bigr)\Bigr\}.$$

To induce clustering of $\{\mathbf{p}_i\}$, we apply a group SCAD penalty on pairwise differences:

$$\sum_{i<j} \mathrm{SCAD}\bigl(\|\mathbf{p}_i - \mathbf{p}_j\|_2\bigr).$$

Hence, the final penalized objective (omitting constants) is

$$F(\mathbf{p}) = -\ell(\mathbf{p}) \;+ \sum_{1 \le i < j \le S} \mathrm{SCAD}\bigl(\|\mathbf{p}_i - \mathbf{p}_j\|_2\bigr).$$

We adopt an ADMM framework by introducing $\boldsymbol{\eta}_{ij} = \mathbf{p}_i - \mathbf{p}_j$ and a quadratic approximation to $-\ell(\mathbf{p})$ at each iteration via IRLS or a piecewise-linear logistic expansion.

2.2 ADMM Splitting and GPU Perspective

Standard ADMM. The usual ADMM scheme sets up:

$$\boldsymbol{\eta}_{ij} \;=\; \mathbf{p}_i - \mathbf{p}_j, \quad \boldsymbol{\tau}_{ij} \text{ dual variables.}$$

The augmented Lagrangian is

$$L(\mathbf{p}, \boldsymbol{\eta}, \boldsymbol{\tau}) = \; -\ell(\mathbf{p}) \;+\; \sum_{i<j} \mathrm{SCAD}\bigl(\|\boldsymbol{\eta}_{ij}\|_2\bigr) \;+\; \sum_{i<j} \Bigl[\frac{\alpha}{2}\, \|\mathbf{p}_i - \mathbf{p}_j - \boldsymbol{\eta}_{ij}\|_2^2 - \langle \boldsymbol{\tau}_{ij}, \mathbf{p}_i - \mathbf{p}_j - \boldsymbol{\eta}_{ij}\rangle\Bigr].$$

At iteration k, we do:

1. $\mathbf{p}$-update (using the local quadratic approximation): solve a linear system in $\mathbf{p}$.
2. $\boldsymbol{\eta}$-update: apply group SCAD thresholding to each pair's difference.
3. $\boldsymbol{\tau}$-update: $\boldsymbol{\tau}_{ij}^{(k+1)} \leftarrow \boldsymbol{\tau}_{ij}^{(k)} - \alpha\bigl[\mathbf{p}_i^{(k+1)} - \mathbf{p}_j^{(k+1)} - \boldsymbol{\eta}_{ij}^{(k+1)}\bigr]$.

From a GPU perspective, the key is to *implement each sub-step in a vectorized, parallel manner* to avoid expensive CPU loops over $\binom{S}{2}$ pairs.

3 Data Representation and Initialization on the GPU

3.1 Converting Data to GPU Tensors

When reading arrays (e.g. $\mathbf{r}, \mathbf{n}, \mathbf{minor}, \mathbf{total}$, and the precomputed logistic expansions or $\mathbf{A}_{ij}, \mathbf{B}_{ij}$) from disk as `NumPy` arrays, we immediately wrap them:

$$\texttt{torch.as\_tensor}(\text{NumPy array}, \text{dtype} = \texttt{float32}, \text{device} = \texttt{'cuda'}).$$

This ensures that all subsequent operations on these arrays happen on the GPU. If S or M is large, the memory usage can be significant, so we must confirm the GPU has enough VRAM.

3.2 Logistic Parameter Initialization

Each SNV i has an initial guess $\mathbf{p}_i^{(0)}$. Typically, we do:

$$\theta_{ij}^{(0)} = \frac{r_{ij} + \epsilon}{n_{ij} + 2\,\epsilon}, \quad \text{clamp } \theta_{ij}^{(0)} \text{ in [low, up]}, \quad p_{ij}^{(0)} = \log\bigl(\tfrac{\theta_{ij}^{(0)}}{1-\theta_{ij}^{(0)}}\bigr).$$

Everything is done in a GPU tensor. We also might clamp the logistic scale in $[-\text{control\_large}, +\text{control\_large}]$ (e.g. ± 4) to avoid numerical extremes.

4 Sparse Matrix Construction of Δ

4.1 Vectorizing Pairwise Differences

To handle pairwise differences $\mathbf{p}_i - \mathbf{p}_j$ for $1 \le i < j \le S$, we define an operator Δ that, given a flattened $\mathbf{p} \in \mathbb{R}^{(S \times M)}$, yields all pairwise differences, also flattened. In practice:

$$(\Delta \mathbf{p})_{((i,j),\,m)} = p_{i,m} - p_{j,m} \quad \text{with } i < j.$$

Hence, Δ has dimension $\bigl(\binom{S}{2} \times M\bigr) \times (S \times M)$. Each row is mostly zero, containing a $+1$ in the column for (i, m) and -1 in the column for (j, m).

4.2 Implementation in PyTorch

We do a fully GPU-based method to build Δ as a `torch.sparse_coo_tensor`:

1. `torch.triu_indices(S, S, offset=1)` obtains all $(i < j)$. This yields $\text{No\_pairs} = \frac{S\,(S-1)}{2}$ pairs, stored in two 1D tensors `i_idx` and `j_idx`.
2. For each pair and each sample index m, define the row index in Δ:

   $$\text{row} = \text{pair\_index} \times M + m,$$

   and the column indices for (i, m) and (j, m). We assign $+1$ in the $(\text{row}, \text{col\_i})$ location and -1 in the $(\text{row}, \text{col\_j})$ location.
3. Concatenate all row, column, and value arrays in vectorized form on the GPU, then call `torch.sparse_coo_tensor` to finalize Δ.
4. The final shape is $\bigl(\text{No\_pairs} \times M\bigr) \times (S \times M)$.

Since pairwise difference operations can be quite large, using a sparse representation significantly reduces storage overhead and speeds up matrix multiplies.

5 ADMM Subproblems on the GPU

5.1 p-Update via Local Quadratic Approximation

We approximate $-\ell(\mathbf{p})$ by a local quadratic function:

$$-\ell(\mathbf{p}) \;\approx\; \tfrac{1}{2}\,\mathbf{p}^\top(\mathbf{B}^\top\mathbf{B})\,\mathbf{p} \;-\; \mathbf{p}^\top(\mathbf{B}^\top\mathbf{A}),$$

where $\mathbf{B}$ is effectively a diagonal matrix capturing IRLS weights, and $\mathbf{A}$ is a vector capturing offsets. Therefore, the relevant subproblem in the ADMM Lagrangian is:

$$\min_{\mathbf{p}}\Big\{\tfrac{1}{2}\,\mathbf{p}^\top(\mathbf{B}^\top\mathbf{B})\,\mathbf{p} \;-\; \mathbf{p}^\top(\mathbf{B}^\top\mathbf{A}) \;+\; \frac{\alpha}{2}\,\|\Delta\mathbf{p}-\boldsymbol{\eta}^{(k)}\|_2^2 - \sum_{i<j}\langle\boldsymbol{\tau}_{ij},\;(\mathbf{p}_i-\mathbf{p}_j)-\boldsymbol{\eta}_{ij}^{(k)}\rangle\Big\}.$$

Matrix form. On the GPU, we flatten $\mathbf{p}$ to dimension $(S\times M)$. We let:

$$\mathbf{H} = \;\mathbf{B}^\top\mathbf{B} \;+\; \alpha\,\Delta^\top\Delta.$$

Hence we solve

$$\mathbf{H}\,\mathbf{p}^{(k+1)} = \;\mathbf{r.h.s.},$$

where $\mathbf{r.h.s.}$ includes $\alpha\,\Delta^\top(\boldsymbol{\eta}^{(k)}-\boldsymbol{\tau}^{(k)}/\alpha)$ and $-\,\mathbf{B}^\top\mathbf{A}$.

Conjugate Gradient Implementation

Since $\mathbf{H}$ is too large for direct factorization, we use a GPU-based iterative conjugate gradient method:

1. *Matvec:* $\mathbf{x}\mapsto\mathbf{B}^\top\mathbf{B}\,\mathbf{x}+\alpha(\Delta^\top\Delta)\mathbf{x}$. The diagonal part $\mathbf{B}^\top\mathbf{B}$ is an elementwise multiply, and $\Delta^\top\Delta$ is stored in sparse format for GPU multiplication.
2. *RHS:* $\mathbf{r.h.s.} = \alpha\,\Delta^\top(\boldsymbol{\eta}-\boldsymbol{\tau}/\alpha)-\mathbf{B}^\top\mathbf{A}$.
3. *Loop:* standard CG iteration with vector additions, dot products, and one sparse matvec each iteration, all on GPU.

This approach scales up to thousands of SNVs. If the matrix becomes extremely large, one may need advanced preconditioning or partial sub-block solves, but the essence remains the same.

5.2 Group SCAD Threshold Update for $\boldsymbol{\eta}$

After $\mathbf{p}$ is updated, we solve:

$$\boldsymbol{\eta}_{ij}^{(k+1)} = \;\arg\min_{\boldsymbol{\eta}_{ij}}\Big\{\frac{\alpha}{2}\,\|\mathbf{p}_i^{(k+1)}-\mathbf{p}_j^{(k+1)}-\boldsymbol{\eta}_{ij}\|^2 \;-\; \langle\boldsymbol{\tau}_{ij}^{(k)},\;(\mathbf{p}_i-\mathbf{p}_j)-\boldsymbol{\eta}_{ij}\rangle \;+\; \mathrm{SCAD}\big(\|\boldsymbol{\eta}_{ij}\|_2\big)\Big\}.$$

Define

$$\delta_{ij} = \;(\mathbf{p}_i^{(k+1)}-\mathbf{p}_j^{(k+1)}) \;-\; \tfrac{1}{\alpha}\,\boldsymbol{\tau}_{ij}^{(k)}.$$

Then $\boldsymbol{\eta}_{ij}$ is updated by the group SCAD threshold in the ℓ_2-norm sense, done piecewise as:

$$\|\delta_{ij}\|_2 = D_{ij}, \quad \lambda_\alpha = \frac{\lambda}{\alpha}.$$

We then choose among:

$$\boldsymbol{\eta}_{ij} = \mathbf{0}, \quad \text{or} \quad \boldsymbol{\eta}_{ij} = \text{soft\_threshold}(\delta_{ij}, \lambda_\alpha), \quad \text{or} \quad \boldsymbol{\eta}_{ij} = \text{SCAD midregion formula}, \quad \text{or} \quad \boldsymbol{\eta}_{ij} = \delta_{ij}.$$

GPU vectorization. On the GPU, we gather all pairs in a single matrix $\delta \in \mathbb{R}^{(\text{No\_pairs} \times M)}$. We compute row norms and create boolean masks for the piecewise conditions. Then we assign the updated $\boldsymbol{\eta}$ for each row in one pass. This is crucial for performance: we avoid explicit Python loops over No_pairs.

5.3 Dual Update

Finally, for each pair,

$$\boldsymbol{\tau}_{ij}^{(k+1)} = \boldsymbol{\tau}_{ij}^{(k)} - \alpha\Big[(\mathbf{p}_i^{(k+1)} - \mathbf{p}_j^{(k+1)}) - \boldsymbol{\eta}_{ij}^{(k+1)}\Big].$$

Again, we do this vectorized on the GPU: $\boldsymbol{\tau} \leftarrow \boldsymbol{\tau} - \alpha\big[\Delta\mathbf{p} - \boldsymbol{\eta}\big]$. Because $\Delta\mathbf{p}$ is a sparse-matrix multiply, we can flatten $\mathbf{p}$ and do a single $\Delta\mathbf{p}$ operation in one shot.

6 Performance Considerations and Advanced GPU Techniques

6.1 Data Dimensions

$\mathbf{p}$ is shaped (S, M). The number of pairwise differences is $\frac{S(S-1)}{2} \times M$. If S is in the thousands, we easily reach millions of differences. Because each ADMM iteration involves at least one big sparse multiplication with Δ or $\Delta^\top\Delta$, GPU memory usage can be large.

6.2 Sparse Format and Coalescing

Δ is stored in COO form. Using `.coalesce()` merges duplicates and sorts indices, which can be beneficial for faster `torch.sparse.mm` performance. Alternative formats (CSR, CSC) might yield further speed-ups if PyTorch extends support.

6.3 Batching IRLS Updates

Because the IRLS approximation can change $\mathbf{B}$ each iteration, we might recast $\mathbf{B}$ only every few ADMM sub-iterations, reducing overhead. Although that modifies the theoretical iteration slightly, in practice it often converges reliably.

6.4 Scaling to Very Large S

The main bottleneck is the memory needed to store Δ. For extremely large S, we might avoid a full Δ (which has $\approx 2\binom{S}{2} \times M$ nonzeros) by imposing a *structured* or *local* difference operator (e.g., only penalizing neighbors on a given tree or graph). This drastically reduces the number of edges relative to the complete graph of all pairs.

6.5 Precision and Numerical Stability

We typically use `float32`. If numeric range is extreme, we can clamp logistic parameters. Some HPC configurations might allow `float16` or `bfloat16` for partial speed-ups, but be aware of potential underflows in exponentials or logistic transforms.

7 Post-Processing and Clustering

After ADMM converges, pairs with $\|\boldsymbol{\eta}_{ij}\| \approx 0$ are identified. This implies $\mathbf{p}_i = \mathbf{p}_j$. A final pass merges clusters accordingly. On the GPU, we can check norms rowwise. However, the actual merging of clusters is often done with a union-find data structure or BFS on the CPU, because it involves integer indexing. Once all merges are done, we finalize the cluster labels.

For multi-sample data, we typically interpret each cluster $\mathcal{C} \subset \{1, \ldots, S\}$ as a subclone with logistic vector $\mathbf{p}_{\mathcal{C}}$. Converting $\mathbf{p}_{\mathcal{C}}$ back to frequencies across the M samples gives a subclone signature that can be biologically interpreted.

8 Explicit Summary of GPU Techniques

In this final section, we collect all GPU-centric techniques used in solving the SCAD-penalized subclone problem. These techniques aim to maximize throughput and minimize host-device transfers:

1. **Full Data Transfer to GPU**: All major arrays $\mathbf{r}, \mathbf{n}, \mathbf{p}, \mathbf{A}, \mathbf{B}, \boldsymbol{\eta}, \boldsymbol{\tau}$ live as `torch.tensor` objects on the GPU. Intermediate values ($\Delta\mathbf{p}, \mathbf{B}\mathbf{p}, \ldots$) stay on GPU to avoid repeated CPU-GPU copies.

2. **Sparse Matrix Representation for Δ**: We define Δ once (or rebuild if S changes). Its multiplication with $\mathbf{p}$ is done by `torch.sparse.mm(...)`. Storing Δ in COO format drastically reduces memory usage since each row of Δ has only two nonzero entries ($+1$ and -1).

3. **Vectorized Group SCAD Thresholding**: Instead of for-loops over pairs, we keep $\boldsymbol{\eta}$ as a $(\text{No\_pairs}, M)$ GPU tensor, compute δ-vectors, row norms, and piecewise threshold in a single pass using PyTorch boolean masks.

4. **Iterative Conjugate Gradient on GPU**:
   - A custom `matvec_H(x)` routine multiplies a vector $\mathbf{x}$ by $\text{diag}(B^2) + \alpha\Delta^\top\Delta$.
   - We do not form that matrix in memory; we only store $\Delta^\top\Delta$ as a sparse matrix and $\text{diag}(B^2)$ as a 1D vector.
   - The entire CG loop remains on the GPU, using $\mathbf{r} = \mathbf{b} - \text{matvec\_H}(\mathbf{x})$ and standard $\langle \mathbf{r}, \mathbf{r} \rangle$ dot products in `torch`.

5. **Clamping and IRLS Expansions in GPU Tensors**: We clamp logistic parameters to $[-C, C]$ to avoid numeric extremes. IRLS expansions (computing $\mathbf{B}$ and $\mathbf{A}$) are performed as elementwise GPU transformations, e.g. $\exp(\mathbf{p})$, $\text{sigmoid}(\mathbf{p})$, etc.

6. **Adaptive α Updates in GPU**: Some ADMM schemes update $\alpha \leftarrow \rho\,\alpha$. That multiplication is trivially a GPU scalar operation, ensuring consistent concurrency with all other GPU-based steps.

7. **Post-Processing Partly on GPU, Partly on CPU**: Norm-based threshold checks remain on GPU, but final union-find or BFS merges of clusters may run on CPU for convenience. This final step typically has a smaller overhead than the main ADMM loop.

Together, these techniques exploit parallel GPU computing for the major linear algebra tasks, which dominate the runtime for large S. The SCAD penalty's non-convex threshold can also be performed in a fully parallel manner across all pairs.

9 Conclusion

This document has provided an explicit summary of the GPU-based formulation and techniques for solving a SCAD-penalized multi-sample subclone reconstruction problem. The solution approach includes:

- Building a sparse difference operator Δ on the GPU via `PyTorch` COO tensors.
- Incorporating a local (IRLS) quadratic approximation of the negative log-likelihood, turning the problem into repeated solves of $\big(\mathbf{B}^\top\mathbf{B} + \alpha\,\Delta^\top\Delta\big)\mathbf{w} = \text{rhs}$.
- Using a GPU-based conjugate gradient loop for these solves, with matvec_H referencing Δ in sparse format and $\mathbf{B}^2$ in a diagonal vector format.
- Updating the auxiliary variables $\boldsymbol{\eta}_{ij}$ via a group SCAD threshold step in one large, vectorized pass on GPU.
- Updating dual variables $\boldsymbol{\tau}$ similarly on GPU.
- Clamping logistic parameters and applying the final cluster merges.

  These GPU techniques can dramatically accelerate the ADMM algorithm, especially with thousands of SNVs or multiple tumor samples. By removing costly Python loops and repeated CPU–GPU transfers, the design leverages the massively parallel operations on modern GPUs to achieve higher throughput.

  While memory usage grows with $\binom{S}{2}$ pairs, partial or graph-based strategies can reduce the penalty structure if needed. Overall, the combination of sparse matrix multiplication, iterative solver, and vectorized thresholding constitutes a powerful approach for large-scale subclone reconstruction in cancer genomics.