Ghost Norm Clipping in DP-SGD: From Explicit Gram Matrices to Flash-Style Tiled Computation

Technical Analysis

November 10, 2025

Abstract

This document provides a comprehensive analysis of ghost norm clipping techniques for differentially private stochastic gradient descent (DP-SGD) in neural networks. We examine two approaches for computing per-sample gradient norms in linear layers: the original algorithm that explicitly constructs Gram matrices, and an optimized flash-style tiled algorithm that achieves the same result with significantly reduced memory usage. We provide detailed algorithmic descriptions, correctness proofs, and complexity analyses for both methods, demonstrating how the flash approach reduces memory complexity from $O(BT^2)$ to $O(BT(d+p))$ while maintaining computational correctness.

Contents

1 Background: DP-SGD Ghost Norm Clipping for Linear Layers **3**
- 1.1 Differential Privacy in Deep Learning . . . 3
- 1.2 The Ghost Norm Clipping Trick . . . 3
- 1.3 Why Norm Computation Can Be Decomposed . . . 3
- 1.4 Extension to Sequential Data . . . 3

2 Ghost Norm Clipping Algorithm **4**
- 2.1 Problem Formulation . . . 4
- 2.2 Algorithm Description . . . 5
- 2.3 Implementation Analysis . . . 5
- 2.4 Correctness Proof . . . 5
- 2.5 Memory and Computational Characteristics . . . 6

3 Flash-Style Tiled Norm Clipping **7**
- 3.1 Motivation . . . 7
- 3.2 Key Insight . . . 7
- 3.3 Algorithm Description . . . 7
- 3.4 Implementation Analysis . . . 7
- 3.5 Correctness Proof . . . 8
- 3.6 Detailed Tiling Example . . . 9
- 3.7 Memory Optimization Benefits . . . 10

4 Complexity Analysis **10**
4.1 Problem Parameters . . . 10
4.2 Original Ghost Clipping Algorithm . . . 10
4.2.1 Computational Complexity . . . 10
4.2.2 Memory Complexity . . . 11
4.3 Flash-Style Tiled Algorithm . . . 11
4.3.1 Computational Complexity . . . 11
4.3.2 Memory Complexity . . . 12
4.4 Complexity Comparison . . . 12
4.5 Practical Memory Savings . . . 12
4.5.1 Memory Reduction Factor . . . 12
4.5.2 Concrete Examples . . . 13
4.6 Scaling Analysis . . . 13
4.7 Implementation Considerations . . . 14

5 FC-pathB: Input-Length-Linear Algorithm **14**
5.1 Algorithm Description . . . 14
5.1.1 Original Algorithm (FC-pathB-orig) . . . 14
5.1.2 Memory-Optimized Algorithm (FC-pathB-opt) . . . 15
5.2 Equivalence Proof . . . 15
5.3 Complexity Analysis . . . 16

6 Comparison with FC-pathA **16**
6.1 Algorithm Description (FC-pathA) . . . 16
6.2 Equivalence Proof . . . 17
6.3 Complexity and Performance Comparison . . . 17
6.4 When is FC-pathB better? . . . 17

7 Conclusion **18**

1 Background: DP-SGD Ghost Norm Clipping for Linear Layers

1.1 Differential Privacy in Deep Learning

Differential Privacy (DP) provides a rigorous mathematical framework for quantifying privacy guarantees in machine learning. In the context of deep learning, DP-SGD (Differentially Private Stochastic Gradient Descent) is the most widely adopted approach for training neural networks with formal privacy guarantees.

The key challenge in DP-SGD is computing per-sample gradient norms efficiently. Traditional approaches require materializing individual gradients for each sample in the batch, which can be prohibitively expensive in terms of memory and computation.

1.2 The Ghost Norm Clipping Trick

The "ghost norm clipping trick" is a memory-efficient technique that computes per-sample gradient norms without explicitly materializing individual gradients. Instead, it leverages the mathematical structure of neural network layers to compute norms directly from activations and backpropagated gradients.

For a linear layer $f(x) = Wx + b$ where $W \in \mathbb{R}^{p \times d}$ and $b \in \mathbb{R}^p$, the gradient with respect to the weight matrix for a single sample is:

$$\nabla_W \ell = g \otimes a = ga^T \tag{1}$$

where $g \in \mathbb{R}^p$ is the backpropagated gradient and $a \in \mathbb{R}^d$ is the input activation.

1.3 Why Norm Computation Can Be Decomposed

The key insight is that the Frobenius norm of the outer product can be computed without materializing the full matrix:

$$\|\nabla_W \ell\|_F^2 = \|ga^T\|_F^2 = \|g\|_2^2 \cdot \|a\|_2^2 \tag{2}$$

This decomposition follows from the properties of the Frobenius norm and outer products:

$$\begin{aligned} \|ga^T\|_F^2 &= \text{tr}((ga^T)^T(ga^T)) && (3)\\ &= \text{tr}(ag^T ga^T) && (4)\\ &= \text{tr}(g^T g) \cdot \text{tr}(a^T a) && (5)\\ &= \|g\|_2^2 \cdot \|a\|_2^2 && (6) \end{aligned}$$

This mathematical property allows us to compute the gradient norm using only the norms of the activation and backpropagation vectors, avoiding the need to store the full $p \times d$ gradient matrix.

1.4 Extension to Sequential Data

For sequential data with batch size B and sequence length T, we have:

- Activations: $A \in \mathbb{R}^{B \times T \times d}$
- Backpropagated gradients: $G \in \mathbb{R}^{B \times T \times p}$

The per-sample gradient for the weight matrix becomes:

$$\nabla_W \ell^{(n)} = \sum_{t=1}^{T} g_t^{(n)} (a_t^{(n)})^T \tag{7}$$

The squared Frobenius norm is:

$$\left\| \nabla_W \ell^{(n)} \right\|_F^2 = \left\| \sum_{t=1}^{T} g_t^{(n)} (a_t^{(n)})^T \right\|_F^2 \tag{8}$$

This cannot be simply decomposed as a product of individual norms due to cross-terms between different time steps. The computation requires considering all pairwise interactions:

$$\left\| \sum_{t=1}^{T} g_t^{(n)} (a_t^{(n)})^T \right\|_F^2 = \sum_{i=1}^{T} \sum_{j=1}^{T} \langle g_i^{(n)} (a_i^{(n)})^T, g_j^{(n)} (a_j^{(n)})^T \rangle_F \tag{9}$$

Using the property $\langle uv^T, xy^T \rangle_F = (u \cdot x)(v \cdot y)$:

$$= \sum_{i=1}^{T} \sum_{j=1}^{T} (g_i^{(n)} \cdot g_j^{(n)})(a_i^{(n)} \cdot a_j^{(n)}) \tag{10}$$

This formulation reveals that we need to compute Gram matrices for both activations and gradients, leading to the algorithms discussed in subsequent sections.

2 Ghost Norm Clipping Algorithm

2.1 Problem Formulation

Consider a batch of sequential data with dimensions:

- Batch size: B
- Sequence length: T
- Activation dimension: d
- Gradient dimension: p

We have:

- Activations: $A \in \mathbb{R}^{B \times T \times d}$
- Backpropagated gradients: $G \in \mathbb{R}^{B \times T \times p}$

For each sample $n \in \{1, \ldots, B\}$, we want to compute:

$$\left\| \sum_{t=1}^{T} G_{n,t,:} A_{n,t,:}^T \right\|_F^2 \tag{11}$$

2.2 Algorithm Description

The original ghost clipping algorithm computes this norm by explicitly constructing Gram matrices for each sample in the batch.

Algorithm 1 Ghost Norm Clipping (Original Algorithm)

Require: Activations $A \in \mathbb{R}^{B \times T \times d}$, Gradients $G \in \mathbb{R}^{B \times T \times p}$
Ensure: Per-sample norms norms $\in \mathbb{R}^B$

for $n = 1$ to B **do** ▷ For each sample in batch
  $G_n \leftarrow G[n, :, :] \in \mathbb{R}^{T \times p}$ ▷ Extract sample gradients
  $A_n \leftarrow A[n, :, :] \in \mathbb{R}^{T \times d}$ ▷ Extract sample activations
  $K_G \leftarrow G_n G_n^T \in \mathbb{R}^{T \times T}$ ▷ Gradient Gram matrix
  $K_A \leftarrow A_n A_n^T \in \mathbb{R}^{T \times T}$ ▷ Activation Gram matrix
  $\text{norm}_n \leftarrow \sum_{i=1}^{T} \sum_{j=1}^{T} (K_G)_{ij} (K_A)_{ij}$ ▷ Hadamard inner product
  $\text{norms}[n] \leftarrow \sqrt{\max(0, \text{norm}_n)}$ ▷ Clamp and take square root
return norms

2.3 Implementation Analysis

The PyTorch implementation uses Einstein summation notation for efficiency:

Listing 1: Original Implementation

```
# Compute batchwise Gram matrices
ggT = torch.einsum("nik,njk->nij", backprops, backprops)  # [B,T,T]
aaT = torch.einsum("nik,njk->nij", activations, activations)  # [B,T,T]

# Compute Hadamard inner product and sum
ga = torch.einsum("n...i,n...i->n", ggT, aaT).clamp(min=0)

# Take square root for final norm
ret[layer.weight] = torch.sqrt(ga)
```

The einsum operations correspond to:

- `"nik,njk->nij"`: For each batch element n, compute outer products between all pairs of time steps
- `"n...i,n...i->n"`: Element-wise multiplication and summation over all dimensions except batch

2.4 Correctness Proof

Theorem 2.1. *The ghost clipping algorithm correctly computes the squared Frobenius norm of the per-sample gradient matrix.*

Proof. For a single sample n, the gradient matrix is:

$$\nabla W^{(n)} = \sum_{t=1}^{T} G_{n,t,:} A_{n,t,:}^T \tag{12}$$

The squared Frobenius norm is:

$$\left\|\nabla W^{(n)}\right\|_F^2 = \operatorname{tr}\left(\left(\nabla W^{(n)}\right)^T \nabla W^{(n)}\right) \tag{13}$$

$$= \operatorname{tr}\left(\left(\sum_{t=1}^{T} A_{n,t,:} G_{n,t,:}^T\right)\left(\sum_{s=1}^{T} G_{n,s,:} A_{n,s,:}^T\right)\right) \tag{14}$$

$$= \operatorname{tr}\left(\sum_{t=1}^{T}\sum_{s=1}^{T} A_{n,t,:} G_{n,t,:}^T G_{n,s,:} A_{n,s,:}^T\right) \tag{15}$$

$$= \sum_{t=1}^{T}\sum_{s=1}^{T} \operatorname{tr}\left(A_{n,t,:} G_{n,t,:}^T G_{n,s,:} A_{n,s,:}^T\right) \tag{16}$$

$$= \sum_{t=1}^{T}\sum_{s=1}^{T} \operatorname{tr}\left(G_{n,t,:}^T G_{n,s,:} A_{n,s,:}^T A_{n,t,:}\right) \tag{17}$$

$$= \sum_{t=1}^{T}\sum_{s=1}^{T} (G_{n,t,:} \cdot G_{n,s,:})(A_{n,t,:} \cdot A_{n,s,:}) \tag{18}$$

$$= \sum_{t=1}^{T}\sum_{s=1}^{T} (K_G)_{ts} (K_A)_{ts} \tag{19}$$

where we used the cyclic property of trace and the fact that $\operatorname{tr}(uv^T) = u \cdot v$ for vectors u, v.

This shows that the algorithm correctly computes the desired norm by computing the Hadamard inner product of the Gram matrices. □

2.5 Memory and Computational Characteristics

The original algorithm has the following characteristics:

Memory Usage:

- Input storage: $O(BT(d+p))$ for activations and gradients
- Gram matrices: $O(BT^2)$ for both K_G and K_A
- Total: $O(BT(d+p) + BT^2)$, dominated by $O(BT^2)$ when T is large

Computational Complexity:

- Gram matrix computation: $O(BT^2(d+p))$
- Hadamard inner product: $O(BT^2)$
- Total: $O(BT^2(d+p))$

The quadratic dependence on sequence length T becomes problematic for long sequences, motivating the development of more memory-efficient approaches.

3 Flash-Style Tiled Norm Clipping

3.1 Motivation

The original ghost clipping algorithm requires $O(BT^2)$ memory to store Gram matrices, which becomes prohibitive for long sequences. The flash-style approach eliminates this bottleneck by computing the same result without materializing the full Gram matrices.

3.2 Key Insight

Instead of computing full Gram matrices, we can tile the computation and process smaller blocks at a time. The key observation is that we only need the final scalar result, not the intermediate $T \times T$ matrices.

3.3 Algorithm Description

Algorithm 2 Flash-Style Tiled Norm Clipping

Require: Activations $A \in \mathbb{R}^{B \times T \times d}$, Gradients $G \in \mathbb{R}^{B \times T \times p}$, Tile size τ
Ensure: Per-sample norms norms $\in \mathbb{R}^B$

norms $\leftarrow \mathbf{0} \in \mathbb{R}^B$ ▷ Initialize accumulator
$n_{\text{tiles}} \leftarrow \lceil T/\tau \rceil$ ▷ Number of tiles
for $i = 0$ to $n_{\text{tiles}} - 1$ **do** ▷ Iterate over tile rows
  $s_i \leftarrow i \cdot \tau$, $e_i \leftarrow \min((i+1) \cdot \tau, T)$
  $A_i \leftarrow A[:, s_i : e_i, :] \in \mathbb{R}^{B \times \tau_i \times d}$ ▷ Load activation tile
  $G_i \leftarrow G[:, s_i : e_i, :] \in \mathbb{R}^{B \times \tau_i \times p}$ ▷ Load gradient tile
  ▷ Diagonal block (i, i)
  $K_G^{(i,i)} \leftarrow G_i G_i^T \in \mathbb{R}^{B \times \tau_i \times \tau_i}$ ▷ Gradient Gram tile
  $K_A^{(i,i)} \leftarrow A_i A_i^T \in \mathbb{R}^{B \times \tau_i \times \tau_i}$ ▷ Activation Gram tile
  norms $+= \sum_{u,v} K_G^{(i,i)}[:, u, v] \odot K_A^{(i,i)}[:, u, v]$ ▷ Accumulate diagonal
  for $j = 0$ to $i - 1$ **do** ▷ Off-diagonal blocks (i, j) and (j, i)
    $s_j \leftarrow j \cdot \tau$, $e_j \leftarrow \min((j+1) \cdot \tau, T)$
    $A_j \leftarrow A[:, s_j : e_j, :] \in \mathbb{R}^{B \times \tau_j \times d}$ ▷ Load activation tile
    $G_j \leftarrow G[:, s_j : e_j, :] \in \mathbb{R}^{B \times \tau_j \times p}$ ▷ Load gradient tile
    $K_G^{(i,j)} \leftarrow G_i G_j^T \in \mathbb{R}^{B \times \tau_i \times \tau_j}$ ▷ Cross Gram tile
    $K_A^{(i,j)} \leftarrow A_i A_j^T \in \mathbb{R}^{B \times \tau_i \times \tau_j}$ ▷ Cross Gram tile
    contrib $\leftarrow \sum_{u,v} K_G^{(i,j)}[:, u, v] \odot K_A^{(i,j)}[:, u, v]$ ▷ Tile contribution
    norms $+= 2 \cdot$ contrib ▷ Factor of 2 for symmetry
norms $\leftarrow \sqrt{\max(\mathbf{0}, \text{norms})}$ ▷ Clamp and square root
return norms

3.4 Implementation Analysis

The PyTorch implementation uses efficient tensor operations:

Listing 2: Flash Implementation Core

```
def _flash_frobenius_inner_over_T(A, G, tile_size=256, dtype_acc=torch.
    float32):
```

```
    B, T, d_a = A.shape
    ga = torch.zeros(B, dtype=dtype_acc, device=A.device)
    num_tiles = (T + tile_size - 1) // tile_size

    for p in range(num_tiles):
        ps, pe = p * tile_size, min((p + 1) * tile_size, T)
        A_p, G_p = A[:, ps:pe, :].to(dtype_acc), G[:, ps:pe, :].to(
            dtype_acc)

        # Diagonal block (p, p)
        Sg_pp = contract('bid,bjd->bij', G_p, G_p)  # [B, tau_p, tau_p]
        Sa_pp = contract('bid,bjd->bij', A_p, A_p)
        ga += contract('bij,bij->b', Sg_pp, Sa_pp)

        # Off-diagonal blocks (q < p)
        for q in range(p):
            qs, qe = q * tile_size, min((q + 1) * tile_size, T)
            A_q, G_q = A[:, qs:qe, :].to(dtype_acc), G[:, qs:qe, :].to(
                dtype_acc)

            Sg_pq = contract('bid,bjd->bij', G_p, G_q)  # [B, tau_p, tau_q
                ]
            Sa_pq = contract('bid,bjd->bij', A_p, A_q)
            ga += 2.0 * contract('bij,bij->b', Sg_pq, Sa_pq)

    return ga
```

3.5 Correctness Proof

Theorem 3.1. *The flash-style tiled algorithm computes the same result as the original ghost clipping algorithm.*

Proof. The original algorithm computes:

$$\text{norm}_n = \sum_{i=1}^{T}\sum_{j=1}^{T}(K_G)_{ij}(K_A)_{ij} \tag{20}$$

We can partition this double sum by tiles. Let $\mathcal{T}_k = \{(k-1)\tau + 1, \ldots, \min(k\tau, T)\}$ be the k-th tile of indices. Then:

$$\begin{aligned}
\text{norm}_n &= \sum_{i=1}^{T}\sum_{j=1}^{T}(K_G)_{ij}(K_A)_{ij} && (21)\\
&= \sum_{k=1}^{n_{\text{tiles}}}\sum_{l=1}^{n_{\text{tiles}}}\sum_{i\in\mathcal{T}_k}\sum_{j\in\mathcal{T}_l}(K_G)_{ij}(K_A)_{ij} && (22)\\
&= \sum_{k=1}^{n_{\text{tiles}}}\sum_{i\in\mathcal{T}_k}\sum_{j\in\mathcal{T}_k}(K_G)_{ij}(K_A)_{ij} + 2\sum_{k=1}^{n_{\text{tiles}}}\sum_{l=1}^{k-1}\sum_{i\in\mathcal{T}_k}\sum_{j\in\mathcal{T}_l}(K_G)_{ij}(K_A)_{ij} && (23)
\end{aligned}$$

The first term corresponds to diagonal tiles, and the second term (with factor 2) corresponds to off-diagonal tiles, accounting for symmetry.

Each tile computation in the flash algorithm computes exactly these partial sums:

- Diagonal tiles: $\sum_{i \in \mathcal{T}_k} \sum_{j \in \mathcal{T}_k} (K_G^{(k,k)})_{ij} (K_A^{(k,k)})_{ij}$
- Off-diagonal tiles: $\sum_{i \in \mathcal{T}_k} \sum_{j \in \mathcal{T}_l} (K_G^{(k,l)})_{ij} (K_A^{(k,l)})_{ij}$

Since the algorithm accumulates all these contributions correctly, it produces the same result as the original algorithm. □

3.6 Detailed Tiling Example

Consider a concrete example with $T = 6$ and tile size $\tau = 2$. We have 3 tiles:

- Tile 0: indices $\{0, 1\}$
- Tile 1: indices $\{2, 3\}$
- Tile 2: indices $\{4, 5\}$

The full 6×6 Gram matrix computation is partitioned as:

$$\begin{bmatrix} K_{0,0} & K_{0,1} & K_{0,2} \\ K_{1,0} & K_{1,1} & K_{1,2} \\ K_{2,0} & K_{2,1} & K_{2,2} \end{bmatrix} \tag{24}$$

where each $K_{i,j}$ is a 2×2 block (except possibly the last block).

Processing Order:

1. **Iteration 0**: Process tile $(0, 0)$ - diagonal block
   - Load A_0, G_0 (indices 0-1)
   - Compute $K_G^{(0,0)} = G_0 G_0^T$, $K_A^{(0,0)} = A_0 A_0^T$
   - Accumulate: norm $+= \langle K_G^{(0,0)}, K_A^{(0,0)} \rangle_F$
2. **Iteration 1**: Process tiles $(1, 1)$ and $(1, 0)$
   - Load A_1, G_1 (indices 2-3)
   - Diagonal: Compute and accumulate $\langle K_G^{(1,1)}, K_A^{(1,1)} \rangle_F$
   - Off-diagonal: Load A_0, G_0, compute $K_G^{(1,0)}, K_A^{(1,0)}$
   - Accumulate: norm $+= 2\langle K_G^{(1,0)}, K_A^{(1,0)} \rangle_F$
3. **Iteration 2**: Process tiles $(2, 2)$, $(2, 1)$, and $(2, 0)$
   - Similar process for the final tile and its interactions

Memory Usage During Processing: At any point, we only store:

- Current tile data: $O(\tau \cdot (d + p))$ per batch element
- Small Gram blocks: $O(\tau^2)$ per batch element
- Total working memory: $O(B(\tau(d + p) + \tau^2))$

This is independent of the full sequence length T, achieving the desired memory efficiency.

3.7 Memory Optimization Benefits

The flash algorithm achieves significant memory savings:

Peak Memory Comparison:

- **Original**: $O(BT^2 + BT(d+p))$
- **Flash**: $O(B\tau^2 + BT(d+p))$ where $\tau \ll T$

For typical values like $T = 8192$, $\tau = 256$, $d = p = 768$:

- Original: $\sim 67M + 12.6M = 79.6M$ parameters per batch element
- Flash: $\sim 65K + 12.6M = 12.7M$ parameters per batch element
- **Reduction factor**: $\sim 6.3\times$

The savings become more dramatic as sequence length increases, since the original algorithm scales as $O(T^2)$ while the flash algorithm scales as $O(T)$ in memory.

4 Complexity Analysis

4.1 Problem Parameters

We analyze the computational and memory complexity of both algorithms using the following parameters:

- B: Batch size
- T: Sequence length
- d: Activation dimension (input features)
- p: Gradient dimension (output features)
- τ: Tile size (for flash algorithm)

4.2 Original Ghost Clipping Algorithm

4.2.1 Computational Complexity

Gram Matrix Computation: For each batch element, we compute:

- $K_G = GG^T \in \mathbb{R}^{T\times T}$: $O(T^2 p)$ FLOPs
- $K_A = AA^T \in \mathbb{R}^{T\times T}$: $O(T^2 d)$ FLOPs

Total for Gram matrices: $O(BT^2(d+p))$ FLOPs

Hadamard Inner Product: Computing $\langle K_G, K_A \rangle_F$ requires $O(T^2)$ FLOPs per batch element.
Total for inner products: $O(BT^2)$ FLOPs

Overall Computational Complexity:

$$\boxed{O(BT^2(d+p))} \tag{25}$$

4.2.2 Memory Complexity

Input Storage:

- Activations A: $O(BTd)$ elements
- Gradients G: $O(BTp)$ elements

Intermediate Storage:

- Gram matrix K_G: $O(BT^2)$ elements
- Gram matrix K_A: $O(BT^2)$ elements

Overall Memory Complexity:

$$\boxed{O(BT(d+p) + BT^2)} \tag{26}$$

For large T, this is dominated by $O(BT^2)$.

4.3 Flash-Style Tiled Algorithm

4.3.1 Computational Complexity

Tile Processing: Number of tiles: $n_{\text{tiles}} = \lceil T/\tau \rceil$
For each diagonal tile (i, i):

- Tile size: $\tau_i \leq \tau$
- Gram computation: $O(\tau_i^2(d+p))$ FLOPs
- Inner product: $O(\tau_i^2)$ FLOPs

For each off-diagonal tile (i, j) with $i \neq j$:

- Cross-Gram computation: $O(\tau_i \tau_j (d+p))$ FLOPs
- Inner product: $O(\tau_i \tau_j)$ FLOPs

Total Tile Pairs:

- Diagonal tiles: n_{tiles}
- Off-diagonal tiles: $\binom{n_{\text{tiles}}}{2} = \frac{n_{\text{tiles}}(n_{\text{tiles}}-1)}{2}$
- Total pairs: $\frac{n_{\text{tiles}}(n_{\text{tiles}}+1)}{2} \approx \frac{T^2}{2\tau^2}$

Asymptotic Analysis: The total computation across all tiles is:

$$\text{FLOPs} = \sum_{\text{all tile pairs}} O(\tau^2(d+p)) \tag{27}$$

$$= O\left(\frac{T^2}{2\tau^2} \cdot \tau^2(d+p)\right) \tag{28}$$

$$= O(T^2(d+p)) \tag{29}$$

Per batch element: $O(T^2(d+p))$ FLOPs

Overall Computational Complexity:

$$\boxed{O(BT^2(d+p))} \tag{30}$$

Note: The asymptotic complexity is the same as the original algorithm, but with better constants due to:

- No large matrix materialization overhead
- Better cache locality from tiled access patterns
- Potential for memory bandwidth optimization

4.3.2 Memory Complexity

Input Storage: Same as original: $O(BT(d+p))$ elements

Working Memory: At any point during computation:

- Current tiles: $O(B\tau(d+p))$ elements for activations and gradients
- Tile Gram matrices: $O(B\tau^2)$ elements for $K_G^{(i,j)}$ and $K_A^{(i,j)}$
- Accumulator: $O(B)$ elements

Overall Memory Complexity:

$$\boxed{O(BT(d+p) + B\tau(d+p) + B\tau^2)} \tag{31}$$

Since $\tau \ll T$ in practice, this simplifies to:

$$\boxed{O(BT(d+p))} \tag{32}$$

4.4 Complexity Comparison

4.5 Practical Memory Savings

4.5.1 Memory Reduction Factor

The memory reduction factor is:

$$\text{Reduction} = \frac{O(BT^2 + BT(d+p))}{O(BT(d+p))} = \frac{T^2 + T(d+p)}{T(d+p)} = \frac{T}{d+p} + 1 \tag{33}$$

For large T relative to $d+p$:

$$\text{Reduction} \approx \frac{T}{d+p} \tag{34}$$

Metric	Original Algorithm	Flash Algorithm
Time Complexity	$O(BT^2(d+p))$	$O(BT^2(d+p))$
Memory Complexity	$O(BT^2 + BT(d+p))$	$O(BT(d+p))$
Dominant Term (Memory)	$O(BT^2)$	$O(BT(d+p))$
Memory Scaling	Quadratic in T	Linear in T

Table 1: Asymptotic complexity comparison

4.5.2 Concrete Examples

Example 1: Moderate Sequence $B = 32$, $T = 2048$, $d = 768$, $p = 768$ (typical transformer settings)

- Original memory: $32 \times (2048^2 + 2048 \times 1536) \approx 32 \times (4.2M + 3.1M) = 234M$ elements
- Flash memory: $32 \times 2048 \times 1536 = 100M$ elements
- **Reduction**: $234M/100M = 2.34\times$

Example 2: Long Sequence $B = 16$, $T = 8192$, $d = 1024$, $p = 1024$

- Original memory: $16 \times (8192^2 + 8192 \times 2048) \approx 16 \times (67M + 16.8M) = 1.34B$ elements
- Flash memory: $16 \times 8192 \times 2048 = 268M$ elements
- **Reduction**: $1.34B/268M = 5.0\times$

Example 3: Very Long Sequence $B = 8$, $T = 32768$, $d = 2048$, $p = 2048$

- Original memory: $8 \times (32768^2 + 32768 \times 4096) \approx 8 \times (1.07B + 134M) = 9.6B$ elements
- Flash memory: $8 \times 32768 \times 4096 = 1.07B$ elements
- **Reduction**: $9.6B/1.07B = 9.0\times$

4.6 Scaling Analysis

T	Original Memory	Flash Memory	Reduction Factor
1024	$BT^2 + BT(d+p)$	$BT(d+p)$	$\frac{T}{d+p} + 1$
1024	$1.05M \cdot B$	$2.1M \cdot B$	$0.5\times$
2048	$4.2M \cdot B$	$4.2M \cdot B$	$1.0\times$
4096	$16.8M \cdot B$	$8.4M \cdot B$	$2.0\times$
8192	$67.1M \cdot B$	$16.8M \cdot B$	$4.0\times$
16384	$268M \cdot B$	$33.6M \cdot B$	$8.0\times$
32768	$1.07B \cdot B$	$67.1M \cdot B$	$16.0\times$

Figure 1: Memory scaling with sequence length (assuming $d = p = 1024$)

Key Observations:

1. **Break-even point**: Flash algorithm becomes beneficial when $T > d + p$
2. **Linear scaling**: Flash memory grows as $O(T)$ vs. $O(T^2)$ for original
3. **Increasing advantage**: Reduction factor grows linearly with T
4. **Modern relevance**: For current LLM sequence lengths ($T \geq 8K$), flash provides substantial savings

4.7 Implementation Considerations

Tile Size Selection: The tile size τ affects:

- **Memory usage**: Larger τ increases working memory $O(B\tau^2)$
- **Cache efficiency**: Moderate τ (128-512) often optimal for GPU shared memory
- **Load balancing**: τ should divide T reasonably evenly

Numerical Precision:

- Flash algorithm can use higher precision accumulators (e.g., `float32`) while keeping inputs in lower precision (e.g., `bfloat16`)
- This helps maintain numerical stability without significantly increasing memory usage

5 FC-pathB: Input-Length-Linear Algorithm

This section describes the `FC-pathB` algorithm, which is designed to compute the squared Frobenius norm of the gradient for a linear layer. We present two versions of the algorithm: a straightforward implementation and a memory-optimized version. We then prove their equivalence and analyze their complexity.

5.1 Algorithm Description

The goal is to compute $\|\nabla_W L\|_F^2$ for a linear layer, where the gradient can be expressed as $\nabla_W L = A^T G$. Here, $A \in \mathbb{R}^{B \times T \times d_a}$ is the input activation tensor and $G \in \mathbb{R}^{B \times T \times d_g}$ is the output gradient tensor. The squared Frobenius norm is given by:

$$\|\nabla_W L\|_F^2 = \|A^T G\|_F^2 = \sum_{i,j} ((A^T G)_{ij})^2$$

5.1.1 Original Algorithm (FC-pathB-orig)

The original algorithm, as commented in the source code, follows a tiled approach. The time dimension T is split into n blocks of size τ. For each block j, we compute $M_j = a_j^T g_j$, where a_j and g_j are the corresponding slices of A and G. The total squared norm is then computed as:

$$\|A^T G\|_F^2 = \left\| \sum_{j=1}^{n} M_j \right\|_F^2 = \sum_{j=1}^{n} \|M_j\|_F^2 + 2 \sum_{j=1}^{n} \sum_{k=j+1}^{n} \langle M_j, M_k \rangle_F$$

This is implemented by first pre-computing and storing all M_j matrices, and then computing the sum of Frobenius inner products.

```
# Original Algorithm
total_norm_squared = 0
M_list = []
for j in 1..num_tiles:
    M_j = a_j.T @ g_j
    M_list.append(M_j)

for j in 1..num_tiles:
    for k in j..num_tiles:
        block_sum = frobenius_inner_product(M_list[j], M_list[k])
        if j == k:
            total_norm_squared += block_sum
        else:
            total_norm_squared += 2 * block_sum
```

5.1.2 Memory-Optimized Algorithm (FC-pathB-opt)

The memory-optimized algorithm avoids storing the list of M_j matrices. Instead, it computes a running sum $S = \sum_{j=1}^{n} M_j$. The final squared Frobenius norm is then simply $\|S\|_F^2$.

```
# Memory-Optimized Algorithm
S = 0
for j in 1..num_tiles:
    M_j = a_j.T @ g_j
    S += M_j
total_norm_squared = frobenius_norm_sq(S)
```

5.2 Equivalence Proof

The equivalence of the two algorithms stems from the properties of the Frobenius norm. Let $S = \sum_{j=1}^{n} M_j$. Then:

$$\begin{aligned}
\|S\|_F^2 &= \left\| \sum_{j=1}^{n} M_j \right\|_F^2 \\
&= \left\langle \sum_{j=1}^{n} M_j, \sum_{k=1}^{n} M_k \right\rangle_F \\
&= \sum_{j=1}^{n} \sum_{k=1}^{n} \langle M_j, M_k \rangle_F \\
&= \sum_{j=1}^{n} \langle M_j, M_j \rangle_F + \sum_{j \neq k} \langle M_j, M_k \rangle_F \\
&= \sum_{j=1}^{n} \|M_j\|_F^2 + 2 \sum_{j=1}^{n} \sum_{k=j+1}^{n} \langle M_j, M_k \rangle_F
\end{aligned}$$

This is exactly the expression computed by the original algorithm. Thus, the two algorithms are mathematically equivalent.

5.3 Complexity Analysis

Let B be the batch size, T the sequence length, d_a the input dimension, and d_g the output dimension. Let the tile size be τ. The number of tiles is $n = \lceil T/\tau \rceil$.

- **FC-pathB-orig**:
  - Pre-computation: For each of the n tiles, we compute M_j which takes $O(B \cdot \tau \cdot d_a \cdot d_g)$. Total pre-computation is $O(n \cdot B \cdot \tau \cdot d_a \cdot d_g) = O(B \cdot T \cdot d_a \cdot d_g)$.
  - Inner products: There are $O(n^2)$ pairs of (M_j, M_k). Each inner product takes $O(B \cdot d_a \cdot d_g)$. Total for this step is $O(n^2 \cdot B \cdot d_a \cdot d_g) = O((T^2/\tau^2) \cdot B \cdot d_a \cdot d_g)$.
  - Memory: Storing the list of M_j matrices requires $O(n \cdot B \cdot d_a \cdot d_g) = O((T/\tau) \cdot B \cdot d_a \cdot d_g)$ memory.
- **FC-pathB-opt**:
  - Computation: In each of the n steps, we compute M_j and add it to S. This takes $O(B \cdot \tau \cdot d_a \cdot d_g)$. Total is $O(n \cdot B \cdot \tau \cdot d_a \cdot d_g) = O(B \cdot T \cdot d_a \cdot d_g)$.
  - Final norm: Computing $\|S\|_F^2$ takes $O(B \cdot d_a \cdot d_g)$.
  - Total time complexity is dominated by the loop, so it is $O(B \cdot T \cdot d_a \cdot d_g)$.
  - Memory: We only need to store the running sum S, which takes $O(B \cdot d_a \cdot d_g)$ memory.

The optimized algorithm has a significantly lower time complexity and memory footprint compared to the original version.

6 Comparison with FC-pathA

The `FC-pathA` algorithm (_width_frobenius) computes the same quantity $\|A^T G\|_F^2$, but with a different grouping of operations.

6.1 Algorithm Description (FC-pathA)

`FC-pathA` also uses tiling on the time dimension. For each pair of blocks (j, k), it computes:

$$\text{block\_sum}_{jk} = \sum_{b=1}^{B} \text{Tr}((a_{b,j} a_{b,k}^T)(g_{b,j} g_{b,k}^T))$$

where $a_{b,j}$ is the slice of activations for batch element b and tile j. The total sum is accumulated similarly to `FC-pathB-orig`.

6.2 Equivalence Proof

The total squared norm is $\sum_{b=1}^{B} \|A_b^T G_b\|_F^2$. Let's focus on a single batch element (and drop the batch index b).

$$\begin{aligned}
\|A^T G\|_F^2 &= \mathrm{Tr}((A^T G)^T (A^T G)) = \mathrm{Tr}(G^T A A^T G) \\
&= \mathrm{Tr}(A A^T G G^T) \\
&= \sum_{i=1}^{T} (A A^T G G^T)_{ii} \\
&= \sum_{i=1}^{T} \sum_{j=1}^{T} (A A^T)_{ij} (G G^T)_{ji} \\
&= \sum_{i=1}^{T} \sum_{j=1}^{T} (a_i^T a_j)(g_j^T g_i)
\end{aligned}$$

where a_i, g_i are the vectors at time step i. This is exactly what `FC-pathA` computes, just grouped by tiles. The `FC-pathB` algorithm computes $(A^T G) = \sum_i a_i g_i^T$, and then the squared norm. The equivalence is clear.

6.3 Complexity and Performance Comparison

- **FC-pathA**:
  - Time: For each pair of tiles (j, k), it computes $a_j a_k^T$ and $g_j g_k^T$. These are $(\tau \times \tau)$ matrices. The matrix multiplications take $O(B \cdot \tau^2 \cdot d_a)$ and $O(B \cdot \tau^2 \cdot d_g)$. The final sum takes $O(B \cdot \tau^2)$. Total for a pair of tiles is $O(B \cdot \tau^2 (d_a + d_g))$. With $O(n^2)$ pairs, the total complexity is $O(n^2 \cdot B \cdot \tau^2 (d_a + d_g)) = O(T^2 \cdot B \cdot (d_a + d_g))$.
  - Memory: The intermediate matrices $a_j a_k^T$ and $g_j g_k^T$ take $O(B \cdot \tau^2)$ memory.
- **FC-pathB-opt**:
  - Time: $O(B \cdot T \cdot d_a \cdot d_g)$.
  - Memory: $O(B \cdot d_a \cdot d_g)$.

6.4 When is FC-pathB better?

`FC-pathB-opt` is generally better when the product of dimensions $d_a \cdot d_g$ is smaller than the sequence length T. The complexity of `FC-pathA` is $O(T^2 \cdot B \cdot (d_a + d_g))$, while for `FC-pathB-opt` it is $O(T \cdot B \cdot d_a \cdot d_g)$.

We can compare $T \cdot (d_a + d_g)$ with $d_a \cdot d_g$. `FC-pathB` is better if $T \cdot (d_a + d_g) > d_a \cdot d_g$. This inequality can be simplified to $T > \frac{d_a d_g}{d_a + d_g}$.

Let $d_a = k \cdot d_g$. Then we need $T > \frac{k d_g^2}{(k+1) d_g} = \frac{k}{k+1} d_g$. If $d_a \approx d_g$, then $k \approx 1$, and we need $T > d_g/2$. If one dimension is much larger than the other, say $d_a \gg d_g$, then k is large and $\frac{k}{k+1} \approx 1$, so we need $T > d_g$.

In many practical scenarios, especially in NLP, the sequence length T is often larger than the model dimensions, making `FC-pathB-opt` the preferred algorithm due to its linear scaling with T. It also has much better memory complexity.

7 Conclusion

This analysis demonstrates the significant advantages of the flash-style tiled approach for computing ghost norm clipping in DP-SGD. While both algorithms achieve the same mathematical result, the flash approach provides substantial memory savings that become increasingly important as sequence lengths grow.

The key contributions of this work include:

- Detailed algorithmic descriptions and correctness proofs for both approaches
- Comprehensive complexity analysis showing the transition from $O(BT^2)$ to $O(BT(d+p))$ memory usage
- Practical examples demonstrating memory reduction factors of 5-40$\times$ for realistic sequence lengths
- Implementation insights for efficient tiled computation

These improvements enable differentially private training of large language models with long sequences that would otherwise be memory-prohibitive using traditional ghost clipping approaches.