Private Training Large-scale Models with Efficient DP-SGD

Liangyu Wang
KAUST

Junxiao Wang
Guangzhou University

Jie Ren
KAUST

Zihang Xiang
KAUST

David E. Keyes
KAUST

Di Wang*
KAUST

Abstract

As large language models (LLMs) increasingly underpin technological advancements, the privacy of their training data emerges as a critical concern. Differential Privacy (DP) serves as a rigorous mechanism to protect this data, yet its integration via Differentially Private Stochastic Gradient Descent (DP-SGD) introduces substantial challenges, primarily due to the complexities of per-sample gradient clipping. Current explicit methods, such as Opacus, necessitate extensive storage for per-sample gradients, significantly inflating memory requirements. Conversely, implicit methods like GhostClip reduce storage needs by recalculating gradients multiple times, which leads to inefficiencies due to redundant computations. This paper introduces FlashDP, an innovative cache-friendly per-layer DP-SGD that consolidates necessary operations into a single task, calculating gradients only once in a fused manner. This approach not only diminishes memory movement by up to **50%** but also cuts down redundant computations by **20%**, compared to previous methods. Consequently, FlashDP does not increase memory demands and achieves a **90%** throughput compared to the Non-DP method on a four-A100 system during the pre-training of the Llama-13B model, while maintaining parity with standard per-layer clipped DP-SGD in terms of accuracy. These advancements establish FlashDP as a pivotal development for efficient and privacy-preserving training of LLMs. FlashDP's code has been open-sourced in *https://github.com/kaustpradalab/flashdp*.

1 Introduction

The transformer architecture (Vaswani et al., 2017) has revolutionized fields like natural language processing (Gao et al., 2024; Xie et al., 2023), embodied AI (Song et al., 2023; Duan et al., 2022; Xu et al., 2024), and AI-generated content (AIGC) (Cao et al., 2023; Wu et al., 2023), with Large Language Models (LLMs) demonstrating exceptional abilities in text generation, complex query responses, and various language tasks due to training on massive datasets. These models, exemplified by ChatGPT, are applied across diverse areas, including healthcare, where they enhance diagnosis and drug discovery by analyzing medical data (Toma et al., 2023; Ali et al., 2023; Sheikhalishahi et al., 2019; Sallam, 2023; Biswas, 2023). However, the extensive capabilities of LLMs raise significant privacy concerns, particularly as they can inadvertently expose or generate sensitive information, owing to their potential to memorize data from large training sets (Pang et al., 2024; Nasr et al., 2023; Carlini et al., 2023; Ippolito et al., 2022; McCoy et al., 2023; Tirumala et al., 2022; Zhang et al., 2023; Ashkboos et al., 2023).

*Corresponding Author

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

Differential Privacy (DP) ensures privacy by adding noise during data processing, such that any single data point's influence on outcomes is minimal (Dwork, 2006). As the most commonly adopted methods for ensuring DP in deep learning models, Differentially Private Stochastic Gradient Descent (DP-SGD) based methods (Abadi et al., 2016) adapt traditional stochastic gradient descent by clipping gradients per sample and adding noise. Although DP-SGD's application in LLMs is increasing, recent research (Li et al., 2022; Bu et al., 2023b; Anil et al., 2022; Hoory et al., 2021) primarily targets the fine-tuning phase, providing privacy only for fine-tuned data. While some studies (Lee & Kifer, 2021; Li et al., 2022; Bu et al., 2023b) have applied DP-SGD to pre-training, they often exhibit limited scalability or reduced training efficiency. This is primarily due to the significant computational and memory overheads inherent to per-sample gradient processing in DP-SGD, which make end-to-end pre-training of large models particularly challenging.

Integrating DP into LLM training via DP-SGD/Adam poses significant challenges, particularly due to per-sample gradient clipping. This crucial privacy technique involves adjusting each data sample's gradients to limit their influence on model updates. While critical for maintaining strict privacy standards, this approach requires computing and storing individual gradients, significantly raising computational and memory demands. Managing these gradients is especially taxing in LLMs, which are known for their large parameter spaces. Each gradient must be carefully clipped and aggregated before updating model parameters, straining computational resources, and prolonging training times. These scalability issues are particularly acute in settings with limited hardware, creating significant barriers to efficiently training privacy-aware LLMs (Li et al., 2022; Bu et al., 2023b).

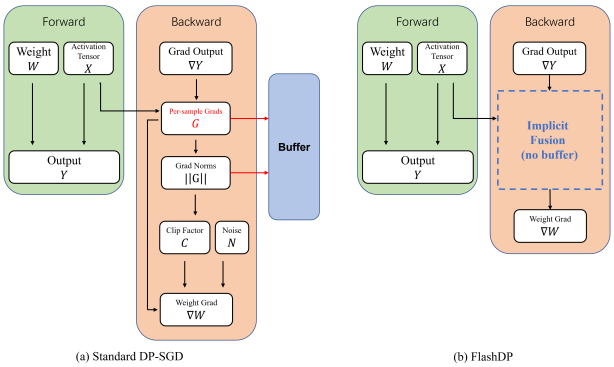


Figure 1: Comparison of different training methods. (a) Standard DP-SGD: Stores per-sample gradients $\mathbf{G}$ (red explicit cache), increasing memory usage (blue buffer). (b) FlashDP: Optimizes gradient processing by consolidating computations into a single pass, reducing redundancy and memory use.

Current research on DP-SGD for training LLMs can be categorized into two classes: explicit methods like Opacus (Yousefpour et al., 2021) stand out by directly storing per-sample gradients. This approach, while straightforward, significantly increases the memory footprint (Appendix Table 4), which becomes prohibitive for state-of-the-art LLMs characterized by billions of parameters (Touvron et al., 2023; Achiam et al., 2023). Such a substantial increase in memory requirements hampers scalability and renders these methods impractical for deployment in large-scale model training environments. The direct storage of gradients, essential for ensuring the privacy guarantees of DP, thus poses a substantial barrier to the efficient implementation of DP in LLMs.

Conversely, implicit methods, exemplified by innovations such as GhostClip (Li et al., 2021), address the memory challenge by circumventing the need for persistent storage of per-sample gradients. These methods segment the DP-SGD process into multiple discrete computational tasks, ostensibly to mitigate memory demands. However, this strategy necessitates the frequent recalculation of per-sample gradients, which introduces a high degree of computational redundancy (Table 4). This redundancy not only undermines training efficiency but also extends the duration of the training process significantly. For LLMs, which require substantial computational resources and extended training times, the inefficiencies introduced by such redundant computations become a critical bottleneck. These implicit methods, while innovative in reducing memory usage, thus struggle to deliver a practical solution for the privacy-preserving training of LLMs at scale.

To effectively tackle the challenges presented by existing methods of integrating DP into the training of LLMs, we introduce FlashDP, a novel, cache-friendly implicit algorithm designed to streamline the per-layer clipping DP-SGD process (Figure 1 (a)). We opt for per-layer clipping in our research primarily due to its efficiency in managing both memory consumption and accuracy, especially vital in the differentially private training of expansive language models (Bu et al., 2023a; He et al., 2022).

It has been shown that this type of method not only sustains commendable accuracy compared to the standard DP-SGD but also mitigates memory overhead, a critical consideration when training large-scale models under privacy constraints. FlashDP uniquely implements a unified computational strategy that performs the gradient operations required for DP-SGD in a single pass (Figure 1 (b)). This innovative approach not only eliminates the need for multiple recalculations of per-sample gradients but also consolidates the entire process into one cohesive computational task. To be specific, FlashDP's architecture, which consolidates the entire DP-SGD process into a single GPU kernel, eliminates redundant computations and optimizes data flow within the GPU. This integration results in a streamlined workflow that efficiently manages memory and processing resources. Also, FlashDP reorganizes the GPU operations to maximize data throughput and minimize latency, effectively enhancing the overall efficiency of the training process. These architectural improvements significantly reduce the volume of memory transfers and computational redundancies, thereby optimizing both the speed and resource utilization during the training of LLMs with DP.

By re-designing the gradient computation workflow, FlashDP dramatically reduces the volume of memory transfers by 50% and decreases redundant computational tasks by 20% compared to previous implicit methods. This optimization is achieved through an advanced caching mechanism that efficiently manages gradient data and computation within GPU memory, minimizing the data movement across the system. As a result, FlashDP significantly alleviates the memory overhead traditionally associated with DP-SGD, enhancing the model's scalability and training speed.

The practical impact of these improvements is substantial. On a computational platform equipped with four NVIDIA A100 GPUs, FlashDP achieves a remarkable 90% throughput compared to the non-DP method during the pre-training phase of the Llama-13B model, a state-of-the-art LLM known for its extensive data and computation demands. Crucially, this enhanced performance is attained without any degradation in the accuracy or dilution of the privacy guarantees compared to the original per-layer clipped DP-SGD. FlashDP thus not only meets but exceeds the operational requirements for effective and efficient privacy-preserving training of LLMs.

Our contributions can be summarized as follows:

- **Enhanced Throughput for LLM training with DP**: We propose FlashDP, which effectively resolves the issue of low throughput in DP-SGD/Adam with per-layer clipping during the training of LLMs. By optimizing the computational workflow and integrating more efficient handling of per-sample gradients, FlashDP significantly enhances the processing speed without compromising the model's accuracy or privacy integrity.
- **Innovative GPU I/O Optimization**: Our study pioneers the exploration of DP-SGD from the perspective of GPU input/output operations. FlashDP's architecture, which consolidates the entire DP-SGD process into a single GPU kernel, eliminates redundant computations and optimizes data flow within the GPU. This approach not only reduces the computational load but also minimizes the number of GPU memory accesses, setting a new standard for efficiency in DP implementations.
- **Experimental Validation of Efficiency and Scalability**: In practical LLM models involving Llama-13B, FlashDP matches the speed and memory usage of non-DP training methods and achieves a significant **90%** throughput compared with Non-DP methods. This performance is achieved on a computational platform equipped with four NVIDIA A100 GPUs. Importantly, it accomplishes this without any degradation in the precision or the privacy guarantees typically observed in the previous per-layer clipped DP-SGD implementations. This capability demonstrates FlashDP's effectiveness in scaling DP applications to larger and more complex LLMs without the usual trade-offs.

2 Related Work

Improving Time and Memory Complexities of DP-SGD. The transition from standard stochastic gradient descent to DP-SGD introduces substantial modifications in memory and computational demands. In conventional settings, parameter updates are efficiently computed by aggregating gradients across all samples within a batch. This approach is both memory-efficient and computationally straightforward. In contrast, DP-SGD mandates that each sample's gradients be preserved, clipped, and subsequently aggregated to uphold privacy guarantees. Recent innovations in DP-SGD have

primarily concentrated on ameliorating its computational and memory inefficiencies. TF-Privacy vectorizes the loss to calculate per-sample gradients through backpropagation, which is efficient in terms of memory but slow in execution (Abadi et al., 2015). Opacus (Yousefpour et al., 2021) and (Rochette et al., 2019) enhance the training efficiency by employing the outer product method (Goodfellow, 2015), albeit at the cost of increased memory usage needed to store per-sample gradients. This memory overhead is mitigated in FastGradClip (Lee & Kifer, 2020) by distributing the space complexity across two stages of backpropagation, effectively doubling the time complexity. Additionally, ghost clipping techniques (Goodfellow, 2015), (Li et al., 2021), (Bu et al., 2022) allow for clipping per-sample gradients without full instantiation, optimizing both time and space, particularly when feature dimensions are constrained. Furthermore, (Bu et al., 2023b) introduces a 'book-keeping' (BK) method that achieves high throughput and memory efficiency, but still leaves room for improvement in fully addressing the computational and memory bottlenecks inherent in large-scale DP training.

While these methodologies have made significant strides in mitigating the extensive computational and memory demands typically associated with managing per-sample gradients in DP-SGD, they have not addressed the optimization of DP training from the perspective of GPU architecture and memory access. Additionally, the approaches detailed thus far do not cater effectively to the training of today's LLMs. FlashDP aims to enhance the efficiency and feasibility of training LLMs under the constraints of differential privacy, ensuring both high performance and adherence to privacy standards.

DP for Large Language Models. The field of privacy-preserving LLMs is characterized by the use or exclusion of DP and its extensions. (He et al., 2022) evaluated the precision equivalence of per-layer clipping with flat clipping on LLMs. (Kerrigan et al., 2020) demonstrated that public pretraining could facilitate downstream DP fine-tuning, although they did not explore fine-tuning large pre-trained models using DP-SGD. (Qu et al., 2021) explored the fine-tuning of BERT for language understanding tasks under local DP. (Bommasani et al., 2021) suggested the potential for cost-effective private learning through fine-tuning large pre-trained language models. (Anil et al., 2021) and (Dupuy et al., 2022) extended these studies to BERT, pretraining and fine-tuning under global DP, respectively, with (Anil et al., 2021) addressing datasets comprising hundreds of millions of examples, and (Dupuy et al., 2022) reporting on datasets of utterances with relatively high ϵ values. Our research distinguishes itself by focusing on pre-training and fine-tuning large language models with high throughput and low memory usage.

3 Understanding the Limitations of Previous Methods

In this section, we introduce the previous non-DP, explicit, and implicate methods of DP-SGD from the GPU I/O perspective to see their weakness, which motivates our framework. Due to the space limit, please refer to Appendix B for the background on DP, Transformers, GPU architecture, and CUDA programming. As discussed in Section B.2, the linear operation is crucial in the architecture of LLMs, particularly within Multi-Head Attention (MHA) and Feedforward Network (FFN) modules. Given its significance, we utilize the linear operation as an exemplar to elucidate the training workflow on GPUs, as shown in Figure 6. See Appendix C for details.

In the standard non-private training workflow of a linear layer, the forward pass involves a matrix multiplication $Y = XW^\mathsf{T}$ between the activation tensor $X \in \mathbb{R}^{B\times T\times P}$ and the weight matrix $W \in \mathbb{R}^{D\times P}$, resulting in the output $Y \in \mathbb{R}^{B\times T\times D}$, where B, T, P, and D denote the batch size, sequence length, input feature dimension, and output feature dimension, respectively. The backward pass calculates the output gradient $\nabla_Y \in \mathbb{R}^{B\times T\times D}$ and the weight gradient $\nabla_W \in \mathbb{R}^{D\times P}$ via $\nabla_W = \sum_B \sum_T (\nabla_Y)^\mathsf{T} X$. Figure 6 (a) illustrates this process, showing that the activation tensor X and weights W are stored in HBM for efficient access during computations, while intermediate operations utilize SRAM to enhance memory access time and throughput.

The explicit DP-SGD workflow, as depicted in Figure 6 (b), categorizes the process into four stages to ensure privacy adherence by explicitly managing per-sample gradients. **Stage 1** involves computing per-sample gradients $\mathbf{G} = \sum_T \nabla_Y^T X$ using batched GEMM operations on SRAM to minimize latency, with subsequent storage of the gradients back to HBM. **Stage 2** requires reloading these gradients to compute their norm $\|\mathbf{G}\| = \sqrt{\sum_D \sum_P \mathbf{G}^2}$, then storing the results back in HBM. **Stage 3** includes loading the gradients and their norms for the per-layer clipping operations, ensuring

that no gradient norm exceeds the predefined threshold C, with the clipped gradients $\mathbf{G}'$ written back to HBM. **Stage 4** focuses on adding Gaussian noise to the clipped gradients in SRAM for privacy preservation, followed by their aggregation for model updates, and storing the final noisy gradient ∇_W back in HBM. This explicit handling of per-sample gradients not only increases memory usage but also complicates processing due to frequent memory swaps and disrupts efficient GPU utilization by breaking down kernel fusion strategies, becoming notably impractical for LLMs with their extensive parameter and gradient sizes, severely impacting training efficiency.

The implicit DP-SGD workflow, illustrated in Figure 6 (c), employs a method such as GhostClip to recalculate gradients in a fused manner, thus circumventing the need for explicit storage of per-sample gradients. **Stage 1** consolidates the first three stages of the explicit method into a single fused computational step, where the activation tensor X and output gradient tensor ∇_Y are loaded into SRAM. Per-sample gradient tensor $\mathbf{G}$ recalculations, norm calculations, and the per-layer clipping are integrated into one operation, minimizing latency and avoiding repeated data transfers to HBM. **Stage 2** mirrors the explicit method's final stage, where the recalculated and clipped gradients $\mathbf{G}'$ undergo Gaussian noise addition in SRAM, followed by aggregation and storage in HBM for model updates. This approach reduces memory usage but increases computational load due to the redundancy of multiple gradient recalculations, which can significantly extend training times, rendering the method less practical due to the increased time complexity proportional to T.

To address the previous limitations, the subsequent section will introduce FlashDP, a novel strategy designed to address these inefficiencies by rethinking the execution pipeline of DP-SGD. Without delving into specifics here, FlashDP's architecture will streamline the integration of per-sample gradient computation and clipping, potentially reducing the operational bottlenecks observed in existing methods.

4 FlashDP Algorithm Design

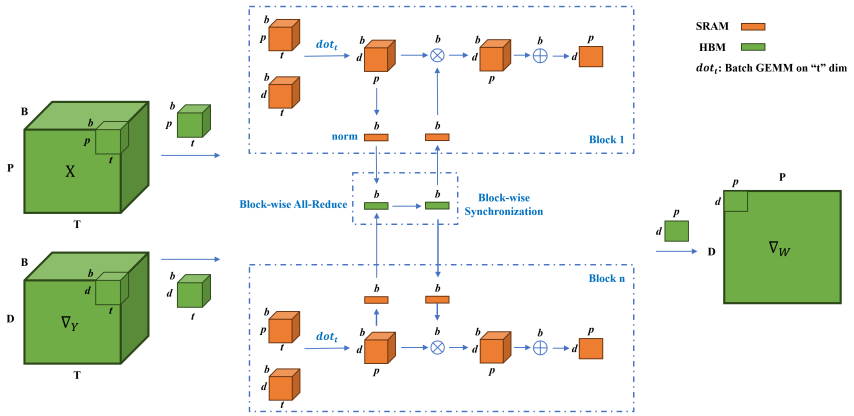


Figure 2: **Illustration of FlashDP.** It depicts the core algorithm design of FlashDP. Its features are integrated with on-chip per-sample gradient norm calculations. The workflow incorporates block-wise all-reduce and synchronization to facilitate efficient norm aggregation. SRAM (orange) and HBM (green) are optimally utilized to manage memory efficiently, addressing the kernel fusion challenges and reducing computational redundancy inherent in traditional DP-SGD implementations.

4.1 Algorithmic Enhancements in FlashDP

FlashDP introduces a suite of algorithmic enhancements designed to reconcile the computational demands and memory constraints associated with DP-SGD. At the heart of these enhancements is the Block-wise All-Reduce algorithm, which integrates several critical operations into a unified kernel execution, thereby optimizing on-chip memory utilization and enhancing computational throughput.

Efficient Kernel Fusion through Block-wise All-Reduce. Central to FlashDP's strategy is our proposed Hierarchical Reduction Architecture (HRA), which encompasses more than just reduction operations. HRA is a structured approach that manages the computation and synchronization of data across various stages, beginning with intra-block reduction of gradient norms within individual GPU blocks. This phase employs an HRA-based reduction strategy executed in shared memory,

culminating in a single norm scaler per block. Such a design significantly reduces the data footprint necessary for subsequent inter-block communications, optimizing the efficiency of the all-reduce operation across the GPU grid.

Following the compact intra-block reduction, FlashDP coordinates a global all-reduce operation across blocks, which computes a global gradient norm crucial for consistent gradient clipping across the entire mini-batch. Efficiently handled in HBM thanks to the minimized data size from earlier reductions, this step avoids the common memory bottlenecks typically associated with large-scale data operations in HBM, thus maintaining high computational throughput.

Algorithm 1 Algorithm: FlashDP with Block-wise All-Reduce on GPUs

Require: Input activation tensor $X \in \mathbb{R}^{B\times T\times P}$ and output gradient tensor $\nabla_Y \in \mathbb{R}^{B\times T\times D}$ in GPU HBM
Require: Clipping threshold C, noise scale σ
Require: Block dimensions b, t, d, and p for batch size, sequence length, output features, and input features, respectively.
Split block for output gradient tensor $B_{\nabla_Y} \in \mathbb{R}^{b\times t\times d}$, input activation tensor $B_X \in \mathbb{R}^{b\times t\times p}$ based on GPU on-chip SRAM size M.
for each training backward iteration **do**
  for each block input index $i_p = 1, 2, \ldots, \frac{P}{p}$ in parallel **do**
    for each block output feature $i_d = 1, 2, \ldots, \frac{D}{d}$ in parallel **do**
      for each block batch size $i_b = 1, 2, \ldots, \frac{B}{b}$ in parallel **do**
        Load output gradient block B_{∇_Y} and input activation block B_X from HBM to SRAM.
        Compute per-sample gradients block $B_G = \sum_T B_{\nabla_Y}^T B_X$ on-chip SRAM.
        Intra-block Reduce: Compute per-sample gradients norm square block $\|B_G\|^2 = \sum_d \sum_p {B_\mathbf{G}}^2$ on-chip SRAM.
        Inter-block Reduce: Offload all per-sample gradients norm square blocks $\|B_G\|^2$ from SRAM to HBM, and perform block-wise all-reduce.
        Block-wise synchronization: Wait until all blocks finish the all-reduce operation to get all-reduced per-sample gradients norm square blocks $\|B_G\|^{2'}$.
        Upload $\|B_G\|^{2'}$ from HBM to SRAM.
        Compute clipped per-sample gradients block $B'_G = B_G / \max\left(1, \frac{\sqrt{\|B_G\|^{2'}}}{C}\right)$ on-chip SRAM.
        Add noise to clipped per-sample gradients block and aggregate to compute parameter gradient block $B_{\nabla_W} = \sum_b B'_G + \mathcal{N}(0, \sigma^2 C^2 \mathbf{I})$ on-chip SRAM.
        Offload parameter gradient block B_{∇_W} from SRAM to HBM.
      end for
    end for
  end for
end for
Return entire parameter gradient ∇_W.

The strategic implementation of HRA not only facilitates these reductions but also orchestrates synchronized updates and data consistency across the GPU architecture. By managing data flow from the point of loading through to final computation and storage, HRA ensures that the most intensive computations are confined to the faster, on-chip memory. This methodical approach leverages the GPU's capabilities to facilitate high-performance differentially private training, minimizing memory and bandwidth overhead.

The practical implementation and operational dynamics of the FlashDP approach are thoroughly illustrated in Algorithm 1 and visually depicted in Figure 2. FlashDP innovatively reduces the four distinct stages typically involved in explicit DP-SGD into a **single streamlined stage**. This consolidation is achieved without adding any extra computational steps, thereby enhancing the overall efficiency of the process. Here is a detailed breakdown of this single streamlined stage:

Optimized Block Processing and Memory Management (Line 1-6). Initially, FlashDP partitions the input activation tensor X and the output gradient tensor ∇_Y into blocks based on the SRAM capacity. This strategic partitioning is crucial for managing the limited on-chip memory more effectively and ensuring that data transfers between the HBM and SRAM are minimized.

Fused Computation of Gradients and Norms (Line 7-8). Within the GPU's SRAM, FlashDP simultaneously computes the per-sample gradients block and their norms square (intra-block reduce) for each block. This computation leverages the GPU's powerful batched GEMM operations, enabling it to handle large data sets efficiently.

Block-wise All-Reduce (Line 9-11). After computing the gradient norms, FlashDP performs a Block-wise All-Reduce operation in parallel to aggregate these norms across all blocks (inter-block reduce). This all-reduce operation is crucial for obtaining a global view of gradient norms square, which is necessary for consistent gradient clipping across the entire batch. This step is executed efficiently within the SRAM, reducing the latency and memory bandwidth requirements typically associated with inter-GPU communications.

Per-layer Gradient Clipping and Noise Addition in SRAM (Line 12-13). Following the gradient and norm calculations, clipping is performed directly on the chip. Each gradient is scaled according to the computed norms and a predefined clipping threshold C, ensuring compliance with DP standards. Immediately after clipping, Gaussian noise based on the noise scale σ and the clipping threshold is added to each gradient block.

Efficient Parameter Aggregation (Line 14-19). The final step in the FlashDP algorithm involves aggregating the noisy, clipped gradients across all blocks and batches directly within SRAM. This aggregation is optimized to minimize memory accesses, ensuring that only the final gradient used for the model update is transferred back to HBM.

4.2 Adaptive Kernel Implementation

The implementation of the FlashDP algorithm leverages the robust and versatile capabilities of the PyTorch framework (Paszke et al., 2019), which is renowned for its intuitive handling of automatic differentiation and dynamic computational graphs. One of the critical features of our implementation involves customizing PyTorch's autograd functionality to accommodate the specific needs of differential privacy during the training of deep neural networks. To this end, operators that necessitate trainable parameters are intricately defined by wrapping them within PyTorch's autograd function.

However, implementing the Block-wise All-Reduce algorithm has presented unique challenges, primarily due to the limitations of CUDA's programming model in facilitating block-wise synchronization. Block-wise synchronization is essential in our algorithm; without it, clip operations might be executed prematurely, while the inter-block reduce operation is still incomplete, leading to numerical inaccuracies in the computation of per-sample gradients' norm squares. There are two primary methods to implement synchronization: 1. cooperative groups (CG) [*] and 2. adaptive kernel. We opted for the second method because the grid synchronization required by CG necessitates launching all blocks simultaneously, which is impractical for DP applications. Although CG supports global synchronization, its requirement to launch all blocks concurrently imposes hardware constraints, making it unsuitable for large-scale DP training. We therefore opted for the adaptive kernel method.

To address this limitation, FlashDP's implementation employs an adaptive approach. Instead of relying on a monolithic kernel to perform the entire Block-wise All-Reduce operation, the process is split across different kernels, which are executed iteratively over the batch dimension. This iterative approach allows for synchronization points between the execution of kernels, using the inherent block synchronization that occurs at kernel launch and completion.

The execution flow in FlashDP is as follows: (1) **Intra-block Reduction**: Each block computes the norms of its gradients and performs an HRA-based reduction within the shared memory. This step employs a shuffle-reduce mechanism, optimizing intra-block operations by minimizing memory footprint and synchronization overhead. This results in a single norm value per block. (2) **Inter-block Reduction**: Each block transfers the outcome of its intra-block reduction to the HBM. This transfer is facilitated through atomic operations for several reasons. Firstly, the result of the intra-block reduction comprises only a single element, and each block elects only one thread to perform the

[*] https://developer.nvidia.com/blog/cooperative-groups/

atomic operation on this element. This approach minimizes potential bottlenecks, as the differing execution speeds across blocks prevent serious serialization issues. Secondly, atomic operations benefit from acceleration by the hardware instruction set, ensuring that these operations are executed swiftly and efficiently. (3) **Inter-kernel Synchronization**: After the completion of the inter-block reduction, FlashDP leverages the termination of the kernel as a natural synchronization point. At this juncture, all blocks have finished their individual reductions. (4) **Iterative Kernel Launch**: For each batch element, a new kernel is launched serially, maintaining synchronization across kernels. This approach involves broadcasting operations where source operands are dimensionally disparate, ensuring uniform data handling across computational units.

This implementation strategy, while divergent from the ideal single-kernel solution, allows FlashDP to function effectively within the current constraints of CUDA. It underscores FlashDP's adaptability and represents a practical solution to the block synchronization challenge, ensuring accurate gradient norm calculations essential for maintaining the model's differential privacy.

5 Experiments

Our experimental suite is methodically designed to assess the robustness and efficiency of FlashDP across a range of training paradigms and hardware configurations. We explore FlashDP's performance in terms of memory efficiency and throughput under varying batch sizes, its adaptability to Automatic Mixed Precision (AMP) training (Appendix Section E.2), its scalability when employing Distributed Data Parallel (DDP) and Pipeline Parallel (PP) techniques (Appendix Section E.3), and utility evaluation (Appendix Section E.4).

Table 1: **Differential Batch-size Analysis.** The table displays a multi-panel comparison of memory usage and throughput for four differential privacy methods–NonDP, Opacus, GhostClip, BK, and FlashDP–across different batch sizes B (1, 2, 4, and 8) when applied to GPT-2 models of varying sizes (small, medium, and large). Instances of '-' in the table indicate scenarios where the corresponding method failed to execute due to memory constraints.

Model	B	Memory Usage (MB x1e4)					Throughput (tokens/sec x1e4)				
		NonDP	Opacus	GhostClip	BK	FlashDP	NonDP	Opacus	GhostClip	BK	FlashDP
GPT2-small	1	0.50	0.75(x1.50)	**0.46(x0.92)**	0.53(x1.06)	0.50(x1.00)	2.84	0.91(x0.32)	0.57(x0.20)	1.56(x0.54)	**1.83(x0.64)**
GPT2-medium	1	1.26	1.53(x1.21)	**1.12(x0.89)**	1.68(x1.33)	1.26(x1.00)	1.10	0.42(x0.38)	0.39(x0.35)	0.75(x0.68)	**0.86(x0.78)**
GPT2-large	1	2.48	3.99(x1.61)	**2.17(x0.88)**	2.73(x1.18)	2.48(x1.00)	0.58	0.25(x0.43)	0.27(x0.46)	0.40(x0.69)	**0.51(x0.89)**
GPT2-small	2	0.87	1.30(x1.49)	**0.79(x0.91)**	1.01(x1.16)	0.87(x1.00)	3.22	1.68(x0.52)	0.92(x0.29)	1.91(x0.59)	**2.32(x0.72)**
GPT2-medium	2	2.07	2.89(x1.39)	**1.87(x0.90)**	2.44(x1.18)	2.07(x1.00)	1.28	0.74(x0.58)	0.59(x0.46)	0.81(x0.63)	**1.02(x0.80)**
GPT2-large	2	3.91	4.79(x1.23)	**3.53(x0.90)**	4.81(x1.23)	3.91(x1.00)	0.68	0.38(x0.56)	0.38(x0.56)	0.45(x0.66)	**0.59(x0.87)**
GPT2-small	4	1.53	2.07(x1.35)	**1.44(x0.94)**	1.68(x1.09)	1.53(x1.00)	3.60	2.42(x0.67)	1.42(x0.39)	2.24(x0.62)	**2.59(x0.72)**
GPT2-medium	4	3.58	4.26(x1.19)	**3.33(x0.93)**	4.00(x1.12)	3.58(x1.00)	1.42	0.90(x0.63)	0.81(x0.57)	0.95(x0.67)	**1.13(x0.80)**
GPT2-large	4	6.60	-	**6.15(x0.93)**	6.60(x1.00)	6.60(x1.00)	0.76	-	0.50(x0.66)	0.53(x0.70)	**0.64(x0.84)**
GPT2-small	8	2.86	3.44(x1.20)	**2.72(x0.95)**	2.86(x1.00)	2.86(x1.00)	3.80	2.64(x0.69)	1.92(x0.51)	2.40(x0.63)	**2.72(x0.72)**
GPT2-medium	8	6.60	-	**6.24(x0.95)**	6.60(x1.00)	6.60(x1.00)	1.52	-	0.99(x0.65)	1.03(x0.68)	**1.19(x0.78)**
GPT2-large	8	-	-	-	-	-	-	-	-	-	-

5.1 Experimental Setup

Our experiments utilize the Wikitext dataset (Merity, 2016) and are conducted on NVIDIA A100 (80GB) GPUs using the PyTorch framework (Paszke et al., 2019). We assess the performance of FlashDP across various configurations by comparing it with established explicit methods Opacus (Yousefpour et al., 2021), and implicit method GhostClip (Li et al., 2021) and BK (Bu et al., 2023a), all in the per-layer clipping mode, under different training paradigms.* The tested models include GPT-2 (Radford et al., 2019) with a sequence length of 1024 and Llama (Touvron et al., 2023) models, both with a sequence length of 2048. More experimental settings and explanations can be found in Appendix D.

5.2 Results of Batch Size & Micro Batch Size

Efficient batch processing is crucial in LLM training due to its high computational and memory demands. By examining both batch and micro-batch sizes, we assess FlashDP's ability to manage

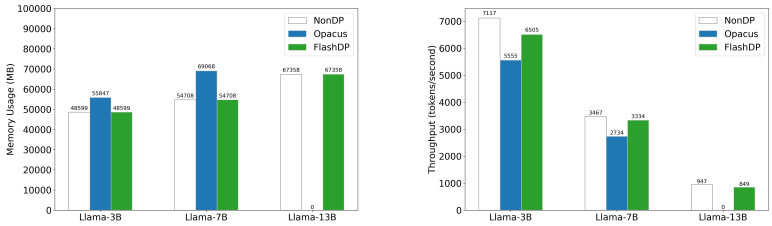
*As (Bu et al., 2023a; He et al., 2022) demonstrated that for LLMs, compared to global clipping, per-layer clipping is more memory-efficient and time-efficient while achieving comparable performance in terms of both privacy preservation and accuracy. Here, we only consider per-layer clipping baselines.

memory more effectively and maintain high throughput. This also tests the practicality of gradient accumulation (GA), which allows larger effective batch sizes by splitting them into smaller, manageable micro-batches. The experiment results of different micro batch sizes can be seen in Appendix E.1.

In Table 1, FlashDP was benchmarked against traditional DP-SGD methods like Opacus, GhostClip, and BK, as well as a non-DP (NonDP) configuration, demonstrating superior memory efficiency and throughput. FlashDP utilized approximately 38% less memory than Opacus and nearly matched the NonDP configuration while processing the GPT-2 large model at a batch size of 1. It achieved a throughput nearly double that of Opacus and only slightly lower than NonDP, showcasing its effective balance between privacy preservation and computational efficiency. Opacus exhibited the highest memory usage, which escalated with batch size, leading to failure at a batch size of 8. GhostClip, while more memory-efficient than Opacus, suffered from reduced throughput at higher batch sizes due to gradient re-computation. BK's performance was intermediate, lacking distinct advantages. Overall, FlashDP not only maintained lower memory usage and higher throughput than the DP methods across all batch sizes but also approached the efficiency of NonDP configurations.

5.3 Results of Distributed Training

Distributed Data Parallel (DDP) (Li et al., 2020) and Pipeline Parallel (PP) (Kim et al., 2020) are two advanced techniques crucial for scaling the training of LLMs efficiently across multiple GPUs or nodes.



(a) Memory Usage (b) Throughput

Figure 3: **Memory and Throughput for Llama Models Using Pipeline Parallel Training.** (a) Memory usage for Llama-3B, Llama-7B, and Llama-13B models. (b) Throughput in tokens per second across these model sizes. A value of 0 indicates out of memory.

Distributed Data Parallel (DDP). Figure 7 in Appendix illustrates the performance of different methods in a DDP setting across GPT-2 models of varying sizes. FlashDP showcases superior memory usage efficiency and higher throughput across all model sizes when compared to Opacus and BK. Notably, even as the model size increases, FlashDP maintains a competitive edge close to the NonDP benchmarks, highlighting its effective parameter distribution and gradient computation across multiple GPUs. This is crucial in scenarios where training speed and model scalability are priorities.

Pipeline Parallel (PP). In the PP scenario depicted in Figure 5, FlashDP was tested with Llama models varying from 3 billion to 13 billion parameters. The results indicate that FlashDP not only scales efficiently with increasing model size but also demonstrates significant throughput improvements compared to Opacus and BK. Particularly, FlashDP's ability to handle the largest model (Llama-13B) with minimal throughput degradation illustrates its robustness in managing extensive computational loads, characteristic of PP environments.

6 Conclusion

In this paper, we introduce FlashDP, a cache-friendly approach to per-layer DP-SGD that improves memory efficiency and computational throughput for large language model (LLM) training. By optimizing GPU I/O through a unified Block-wise All-Reduce algorithm and a Hierarchical Reduction Architecture (HRA), FlashDP significantly reduces memory transactions and eliminates redundant computations. We also adopt an adaptive kernel design to overcome CUDA's synchronization limitations. Experiments show that FlashDP achieves memory usage close to non-private baselines and maintains 90% throughput during Llama-13B training on four A100s, without compromising privacy or accuracy. FlashDP may enable the deployment of privacy-preserving LLMs in sensitive

domains such as healthcare, education, and finance, where data protection is critical. At the same time, it highlights the need for responsible release practices to mitigate potential misuse under the guise of privacy.

Acknowledgments and Disclosure of Funding

Di Wang and Liangyu Wang are supported in part by the funding BAS/1/1689-01-01 and funding from KAUST - Center of Excellence for Generative AI, under award number 5940.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 308–318, 2016.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Stephen R Ali, Thomas D Dobbs, Hayley A Hutchings, and Iain S Whitaker. Using chatgpt to write patient clinic letters. *The Lancet Digital Health*, 5(4):e179–e181, 2023.

Rohan Anil, Badih Ghazi, Vineet Gupta, Ravi Kumar, and Pasin Manurangsi. Large-scale differentially private bert. *arXiv preprint arXiv:2108.01624*, 2021.

Rohan Anil, Badih Ghazi, Vineet Gupta, Ravi Kumar, and Pasin Manurangsi. Large-scale differentially private bert. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 6481–6491, 2022.

Saleh Ashkboos, Ilia Markov, Elias Frantar, Tingxuan Zhong, Xincheng Wang, Jie Ren, Torsten Hoefler, and Dan Alistarh. Towards end-to-end 4-bit inference on generative large language models. *arXiv preprint arXiv:2310.09259*, 2023.

Som S Biswas. Role of chat gpt in public health. *Annals of biomedical engineering*, 51(5):868–869, 2023.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Zhiqi Bu, Jialin Mao, and Shiyun Xu. Scalable and efficient training of large convolutional neural networks with differential privacy. *Advances in Neural Information Processing Systems*, 35: 38305–38318, 2022.

Zhiqi Bu, Ruixuan Liu, Yu-Xiang Wang, Sheng Zha, and George Karypis. On the accuracy and efficiency of group-wise clipping in differentially private optimization. *arXiv preprint arXiv:2310.19215*, 2023a.

Zhiqi Bu, Yu-Xiang Wang, Sheng Zha, and George Karypis. Differentially private optimization on large model at small cost. In *International Conference on Machine Learning*, pp. 3192–3218. PMLR, 2023b.

Yihan Cao, Siyu Li, Yixin Liu, Zhiling Yan, Yutong Dai, Philip S Yu, and Lichao Sun. A comprehensive survey of ai-generated content (aigc): A history of generative ai from gan to chatgpt. *arXiv preprint arXiv:2303.04226*, 2023.

Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. Quantifying memorization across neural language models. In *International Conference on Learning Representations*, 2023.

Jiafei Duan, Samson Yu, Hui Li Tan, Hongyuan Zhu, and Cheston Tan. A survey of embodied ai: From simulators to research tasks. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 6(2):230–244, 2022.

Christophe Dupuy, Radhika Arava, Rahul Gupta, and Anna Rumshisky. An efficient dp-sgd mechanism for large scale nlu models. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4118–4122. IEEE, 2022.

Cynthia Dwork. Differential privacy. In *International colloquium on automata, languages, and programming*, pp. 1–12. Springer, 2006.

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*, pp. 265–284. Springer, 2006.

Mingqi Gao, Xinyu Hu, Jie Ruan, Xiao Pu, and Xiaojun Wan. Llm-based nlg evaluation: Current status and challenges. *arXiv preprint arXiv:2402.01383*, 2024.

Ian Goodfellow. Efficient per-example gradient computations. *arXiv preprint arXiv:1510.01799*, 2015.

Jiyan He, Xuechen Li, Da Yu, Huishuai Zhang, Janardhan Kulkarni, Yin Tat Lee, Arturs Backurs, Nenghai Yu, and Jiang Bian. Exploring the limits of differentially private deep learning with group-wise clipping. *arXiv preprint arXiv:2212.01539*, 2022.

Shlomo Hoory, Amir Feder, Avichai Tendler, Sofia Erell, Alon Peled-Cohen, Itay Laish, Hootan Nakhost, Uri Stemmer, Ayelet Benjamini, Avinatan Hassidim, et al. Learning and evaluating a differentially private pre-trained language model. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 1178–1189, 2021.

Daphne Ippolito, Florian Tramèr, Milad Nasr, Chiyuan Zhang, Matthew Jagielski, Katherine Lee, Christopher A Choquette-Choo, and Nicholas Carlini. Preventing verbatim memorization in language models gives a false sense of privacy. *arXiv preprint arXiv:2210.17546*, 2022.

Gavin Kerrigan, Dylan Slack, and Jens Tuyls. Differentially private language models benefit from public pre-training. *arXiv preprint arXiv:2009.05886*, 2020.

Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. torchgpipe: On-the-fly pipeline parallelism for training giant models. *arXiv preprint arXiv:2004.09910*, 2020.

Jaewoo Lee and Daniel Kifer. Scaling up differentially private deep learning with fast per-example gradient clipping. *arXiv preprint arXiv:2009.03106*, 2020.

Jaewoo Lee and Daniel Kifer. Scaling up differentially private deep learning with fast per-example gradient clipping. *Proceedings on Privacy Enhancing Technologies*, 2021.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Xuechen Li, Florian Tramer, Percy Liang, and Tatsunori Hashimoto. Large language models can be strong differentially private learners. *arXiv preprint arXiv:2110.05679*, 2021.

Xuechen Li, Florian Tramer, Percy Liang, and Tatsunori Hashimoto. Large language models can be strong differentially private learners. In *International Conference on Learning Representations*, 2022.

Anton Lozhkov, Loubna Ben Allal, Leandro von Werra, and Thomas Wolf. Fineweb-edu: the finest collection of educational content, 2024. URL https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu.

R Thomas McCoy, Paul Smolensky, Tal Linzen, Jianfeng Gao, and Asli Celikyilmaz. How much do language models copy from their training data? evaluating linguistic novelty in text generation using raven. *Transactions of the Association for Computational Linguistics*, 11:652–670, 2023.

Stephen Merity. The wikitext long term dependency language modeling dataset. *Salesforce Metamind*, 9, 2016.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Milad Nasr, Nicholas Carlini, Jonathan Hayase, Matthew Jagielski, A Feder Cooper, Daphne Ippolito, Christopher A Choquette-Choo, Eric Wallace, Florian Tramèr, and Katherine Lee. Scalable extraction of training data from (production) language models. *arXiv preprint arXiv:2311.17035*, 2023.

Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 130–130. IEEE Computer Society, 2024.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Chen Qu, Weize Kong, Liu Yang, Mingyang Zhang, Michael Bendersky, and Marc Najork. Privacy-adaptive bert for natural language understanding. *arXiv preprint arXiv:2104.07504*, 190, 2021.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Gaspar Rochette, Andre Manoel, and Eric W Tramel. Efficient per-example gradient computations in convolutional neural networks. *arXiv preprint arXiv:1912.06015*, 2019.

Malik Sallam. Chatgpt utility in healthcare education, research, and practice: systematic review on the promising perspectives and valid concerns. In *Healthcare*, volume 11, pp. 887. MDPI, 2023.

Seyedmostafa Sheikhalishahi, Riccardo Miotto, Joel T Dudley, Alberto Lavelli, Fabio Rinaldi, Venet Osmani, et al. Natural language processing of clinical notes on chronic diseases: systematic review. *JMIR medical informatics*, 7(2):e12239, 2019.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 2998–3009, 2023.

Kushal Tirumala, Aram Markosyan, Luke Zettlemoyer, and Armen Aghajanyan. Memorization without overfitting: Analyzing the training dynamics of large language models. *Advances in Neural Information Processing Systems*, 35:38274–38290, 2022.

Augustin Toma, Patrick R Lawler, Jimmy Ba, Rahul G Krishnan, Barry B Rubin, and Bo Wang. Clinical camel: An open-source expert-level medical language model with dialogue-based knowledge encoding. *arXiv preprint arXiv:2305.12031*, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Jiayang Wu, Wensheng Gan, Zefeng Chen, Shicheng Wan, and Hong Lin. Ai-generated content (aigc): A survey. *arXiv preprint arXiv:2304.06632*, 2023.

Yaqi Xie, Chen Yu, Tongyao Zhu, Jinbin Bai, Ze Gong, and Harold Soh. Translating natural language to planning goals with large-language models. *arXiv preprint arXiv:2302.05128*, 2023.

Zhiyuan Xu, Kun Wu, Junjie Wen, Jinming Li, Ning Liu, Zhengping Che, and Jian Tang. A survey on robotics with foundation models: toward embodied ai. *arXiv preprint arXiv:2402.02385*, 2024.

Ashkan Yousefpour, Igor Shilov, Alexandre Sablayrolles, Davide Testuggine, Karthik Prasad, Mani Malek, John Nguyen, Sayan Ghosh, Akash Bharadwaj, Jessica Zhao, et al. Opacus: User-friendly differential privacy library in pytorch. *arXiv preprint arXiv:2109.12298*, 2021.

Chiyuan Zhang, Daphne Ippolito, Katherine Lee, Matthew Jagielski, Florian Tramèr, and Nicholas Carlini. Counterfactual memorization in neural language models. *Advances in Neural Information Processing Systems*, 36:39321–39362, 2023.

A Explanation of FlashDP

Note that FlashDP does not interfere with any sampling strategy, since these decisions are made during the data loading stage.

B Preliminaries

B.1 Differential Privacy

Definition 1. *(Differential Privacy (Dwork et al., 2006)) Given a data universe $\mathcal{X}$, two datasets $X, X' \subseteq \mathcal{X}$ are adjacent if they differ by one data example. A randomized algorithm $\mathcal{M}$ is (ε, δ)-differentially private if for all adjacent datasets $X,\ X'$ and for all events S in the output space of $\mathcal{M}$, we have* $\Pr(\mathcal{M}(X) \in S) \leq e^{\varepsilon} \Pr(\mathcal{M}(X') \in S) + \delta$.

Differentially Private Stochastic Gradient Descent (DP-SGD) (Abadi et al., 2016). DP-SGD is an adaptation of this principle for machine learning models, where privacy is preserved during the training process by modifying the gradient computation.

In the context of a model parameterized by weights θ for loss $\mathcal{L}$, the standard SGD update is modified in DP-SGD to include a mechanism for privacy preservation. Specifically, the gradient $\nabla\mathcal{L}(\theta, x_i)$ for each training example x_i is first computed, and then processed as follows to incorporate privacy:

1. **Clipping:** Each gradient is clipped to a maximum norm C, defined as: $g_i' = g_i \min(1, \frac{C}{\|g_i\|_2})$, where $g_i = \nabla\mathcal{L}(\theta, x_i)$.
2. **Noise Addition:** Gaussian noise is added to the aggregated clipped gradients to ensure differential privacy:

$$\tilde{g} = \frac{1}{B}\sum_{i=1}^{B} g_i' + \mathcal{N}(0, \sigma^2 C^2 I)$$

   where B is the batch size, and σ is the noise scale, determined by the privacy budget, subsampling rate, and iteration number.

The model parameters are then updated using the noisy, aggregated gradient: $\theta \leftarrow \theta - \eta\tilde{g}$, where η is the learning rate. This approach to privacy-preserving training addresses the fundamental trade-off between accuracy and privacy by controlling the granularity of the updates through the parameters C and σ.

In this work, we actually use Per-layer clipping and Differentially Private Adam (DP-Adam) instead of standard DP-SGD. The key distinction of the per-layer DP-SGD compared to standard DP-SGD lies in its approach to clipping gradients layer by layer and incorporating noise accordingly. (While there are various adaptations of per-layer DP-SGD, we focus on the simplest format that directly extends from the standard DP-SGD.) (Bu et al., 2023a) have demonstrated that per-layer clipping not

only matches the accuracy of global clipping but also significantly enhances memory and throughput efficiency. While DP-Adam incorporates the same mechanisms for gradient clipping and noise addition as described for DP-SGD, it also leverages the adaptive learning rates characteristic of Adam. The detailed algorithms can be found in Algorithm 2-3-5.

Algorithm 2 Common Gradient Processing in DP-SGD and DP-Adam

Require: $\mathcal{L}(\theta, x_i)$: Loss function for parameter θ and input x_i
Require: C: Clipping threshold
Require: σ: Noise scale
Require: B: Batch size
for $i = 1$ to B **do**
  Compute gradient: $g_i = \nabla\mathcal{L}(\theta, x_i)$
  Clip gradient: $g'_i = g_i \min(1, \frac{C}{\|g_i\|_2})$
end for
Aggregate clipped gradients and add Gaussian noise: $\tilde{g} = \frac{1}{B}\sum_{i=1}^{B} g'_i + \mathcal{N}(0, \sigma^2 C^2 I)$

Algorithm 3 Per-Layer Gradient Processing in DP-SGD

Require: $\mathcal{L}(\theta^{(l)}, x_i)$: Loss function for layer parameters $\theta^{(l)}$ and input x_i
Require: $C^{(l)}$: Clipping threshold for layer l
Require: $\sigma^{(l)}$: Noise scale for layer l
Require: L: Total number of layers
for each layer $l = 1$ to L **do**
  for $i = 1$ to B **do**
    Compute gradient for layer l: $g_i^{(l)} = \nabla\mathcal{L}(\theta^{(l)}, x_i)$
    Clip gradient for layer l: $g_i'^{(l)} = g_i^{(l)} \min\left(1, \frac{C^{(l)}}{\|g_i^{(l)}\|_2}\right)$
  end for
  Aggregate clipped gradients for layer l and add Gaussian noise: $\tilde{g}^{(l)} = \frac{1}{B}\sum_{i=1}^{B} g_i'^{(l)} + \mathcal{N}(0, (\sigma^{(l)} C^{(l)})^2 I)$
end for

Algorithm 4 DP-SGD Specific Steps

Require: θ: Model parameters
Require: η: Learning rate
for each training step **do**
  Perform common gradient processing as in Algorithm 2
  Update model parameters: $\theta \leftarrow \theta - \eta\tilde{g}$
end for

B.2 Transformers

The transformer architecture, proposed by Vaswani et al. (Vaswani et al., 2017), is predicated on self-attention mechanisms that process input tokens in parallel, significantly improving the performance and training efficiency of sequence-to-sequence tasks. This architecture has become the backbone of LLMs.

In a transformer model, the input tensor $\mathbf{X}$ of size $B \times T \times P$ (since we are considering LLM, so we only focus on text data as the input), where B is the batch size, T is the sequence length (number of tokens), and P is the embedding size of a token, undergoes a series of transformations through multi-head self-attention and feedforward neural network blocks. For each token in the sequence, the transformer computes a weighted sum of all tokens in the input, where the weights are determined through the self-attention mechanism.

Algorithm 5 DP-Adam Specific Steps

Require: m, v: Estimates of the first and second moments (initially 0)
1: **for** each training step **do**
2: Perform common gradient processing as in Algorithm 2
3: Update moment estimates: $m \leftarrow \beta_1 m + (1 - \beta_1)\tilde{g}$
4: $v \leftarrow \beta_2 v + (1 - \beta_2)\tilde{g}^2$
5: Compute adaptive learning rate: $\hat{\eta} = \eta/(\sqrt{v} + \epsilon)$
6: Update parameters: $\theta \leftarrow \theta - \hat{\eta} m$
7: **end for**

Multi-Head Attention (MHA). The attention mechanism is primarily built upon linear transformations where the query $\mathbf{Q}$, key $\mathbf{K}$, and value $\mathbf{V}$ matrices are obtained as follows:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V \tag{1}$$

where $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ are the weight matrices that are subject to training.

Feedforward Network (FFN). The FFN in the transformer consists of two linear transformations with a ReLU activation in between:

$$\text{FFN}(\mathbf{x}) = \text{ReLU}(\mathbf{x}\mathbf{W}_1)\mathbf{W}_2 \tag{2}$$

Here, $\mathbf{W}_1$ and $\mathbf{W}_2$ are the weight matrices, all of which are trainable parameters of the linear layers within the FFN.

Layer Normalization (LN). LN is applied post-attention and FFN in each layer of the transformer. It normalizes the output of each neuron to have a mean of zero and a variance of one, which are then scaled and shifted by the trainable parameter vectors $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$, respectively:

$$\text{LayerNorm}(\mathbf{x}) = \boldsymbol{\gamma} \odot \left(\frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \boldsymbol{\beta} \tag{3}$$

where μ and σ^2 are the mean and variance calculated over the last dimension of the input tensor $\mathbf{x}$, ϵ is a small constant added for numerical stability, and $\odot$ denotes element-wise multiplication. The layer normalization parameters $\boldsymbol{\gamma}$ (scale) and $\boldsymbol{\beta}$ (shift) are learned to optimally scale and shift the normalized data.

The key trainable parameters in the transformer model are:

1. Weights of the WHA mechanism, including query $\mathbf{W}_Q$, key $\mathbf{W}_K$, and value $\mathbf{W}_V$ matrices, each of size $P \times P$.
2. Position-wise FFN weights $\mathbf{W}_1$ of size $P \times H$ and $\mathbf{W}_2$ of size $H \times P$, where H is the hidden layer size.
3. LN parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$, which are vectors of size P.

It is important to highlight that the bulk of the trainable parameters in the transformer model stems from MHA and FFN modules, both of which consist of linear transformations. These linear parameters are responsible for the vast majority of transformations within the transformer and significantly contribute to its parameter count. In contrast, the trainable parameters in LN represent a relatively smaller portion of the model's total parameters. Therefore, we focus on the linear parameters gradient computation.

DP-SGD for Training Transformers. The process of adapting DP-SGD to transformers is formalized as follows: For each batch of input data X and corresponding loss function $\mathcal{L}$, compute the per-sample gradients $\mathbf{G}_\theta$ for all trainable parameters $\theta = \{\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V, \mathbf{W}_1, \mathbf{W}_2, \boldsymbol{\gamma}, \boldsymbol{\beta}\}$:

$$\mathbf{G}_\theta = \nabla_\theta \mathcal{L}(\theta, X) \in \mathbb{R}^{B \times |\theta|}. \tag{4}$$

where $\nabla_\theta \mathcal{L}(\theta, X)$ denotes the computation of gradients of the loss with respect to the parameters θ for the batch X.

B.3 GPU Architecture and CUDA Programming

High performance in deep learning, particularly in operations like General Matrix to Matrix Multiplication (GEMM), is largely attributable to the parallel processing power of modern Graphics Processing Units (GPUs). The architectural design of GPUs, with their numerous cores and hierarchical memory systems, is optimized for the parallel execution of operations, making them ideal for the matrix-intensive computations required in neural network training.

GPU Architecture. At the heart of GPU's computational efficiency are its Streaming Multiprocessors (SMs), which are essentially multiprocessor units that execute a large number of threads concurrently. Each SM is a powerhouse of performance, containing a set of processing cores and a block of on-chip memory, primarily Shared Random Access Memory (SRAM), which includes registers and shared memory. Shared memory, an ultra-fast SRAM, allows threads within the same block to exchange data without involving the slower global memory (HBM), thus acting as a crucial facilitator for matrix blocking.

CUDA and GEMM. The quintessential challenge in optimizing GEMM lies in the meticulous orchestration of data movement and computation, an endeavor where matrix blocking emerges as a pivotal strategy. Leveraging the robust architecture of GPUs and the sophisticated abstractions provided by CUDA (Compute Unified Device Architecture), matrix blocking transforms the theoretical prowess of parallel computation into a practical performance paradigm.

Principles of Matrix Blocking. Matrix blocking, also known as matrix tiling, is a technique ingeniously conceived to enhance data locality and parallelism. It systematically partitions extensive matrix operands into smaller, manageable sub-matrices or 'blocks' that can be independently dispatched to the GPU's SMs. The judicious use of shared memory within SMs for these blocks reduces the frequency and volume of global memory accesses, a common bottleneck due to its higher latency. Blocking is pivotal in minimizing the communication overhead between the slow global memory and the fast but limited on-chip shared memory. This stratagem leverages the temporal and spatial locality by reusing data within the fast-access memory hierarchies, significantly reducing the volume of data shuttled to and from the global memory, thereby enhancing the computational throughput.

Mathematical Formalization of Blocking GEMM. Consider the GEMM operation defined as $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{B} \in \mathbb{R}^{n \times p}$, and the resultant matrix $\mathbf{C} \in \mathbb{R}^{m \times p}$. Blocking decomposes this operation into smaller, tractable computations over blocks such that:

$$\mathbf{C}_{ij} = \sum_{k=1}^{N} \mathbf{A}_{ik} \times \mathbf{B}_{kj}, \tag{5}$$

where N is the number of blocks, and each $\mathbf{C}_{ij}$, $\mathbf{A}_{ik}$, and $\mathbf{B}_{kj}$ represents a sub-matrix or block within $\mathbf{C}$, $\mathbf{A}$, and $\mathbf{B}$, respectively. The indices i, j, and k denote the specific block within the partitioned matrices.

The dimensions of each block are chosen based on the GPU's shared memory constraints and the size of the SMs' thread blocks, enabling optimal utilization of resources. These dimensions are represented as $B_m \times B_n$ for $\mathbf{A}_{ik}$ and $B_n \times B_p$ for $\mathbf{B}_{kj}$, leading to a block $\mathbf{B}_C$ in size of $B_m \times B_p$ for $\mathbf{C}_{ij}$. Hence, the computational paradigm shifts to:

$$\mathbf{B}_{C_{ij}} = \sum_{k=1}^{B_n} (\mathbf{B}_{A_{ik}} \times \mathbf{B}_{B_{kj}}), \tag{6}$$

where each multiplication within the summation is an independent block-level GEMM that can be executed in parallel.

C Details of Training Workflow

C.1 Non-private Training Workflow

In the standard training regime without privacy constraints, the linear forward operation takes an activation tensor $X \in \mathbb{R}^{B \times T \times P}$ and a weight matrix $W \in \mathbb{R}^{D \times P}$, producing an output $Y \in \mathbb{R}^{B \times T \times D}$ according to the matrix multiplication $Y = XW^{\mathsf{T}}$, where B, T, P, and D indicate the batch

size, sequence length (token length), feature dimension of input activation tensor X, and feature dimension of output activation tensor Y, respectively.

During the backward pass, the gradient of the output with respect to the loss, denoted by $\nabla_Y \in \mathbb{R}^{B\times T\times D}$, is computed to be of the same dimensions as the output tensor Y. Subsequently, the gradient with respect to the weight matrix W, denoted by $\nabla_W \in \mathbb{R}^{D\times P}$, is obtained by summing the product of the transpose of the gradient tensor of each batch item and the corresponding input tensor, expressed as $\nabla_W = \sum_B \sum_T (\nabla_Y)^\mathsf{T} X$, where $\sum_B$ represents the summation along the dimension B (similar for other notations).

Figure 6 (a) illustrates the computational workflow for the forward and backward pass of a linear operation within this conventional training framework. As shown in the figure, the activation tensor X and the weights W reside in HBM, which allows for rapid parallel access and is typically used for storing larger datasets and model parameters during GPU computations. The intermediate dot products and summations are handled using SRAM, shown in orange, which is faster than HBM and suitable for storing temporary, small blocks of data during computation. This setup minimizes memory access time and maximizes throughput.

Clarification on Gradient Formulations and Reviewer Feedback. To clarify the gradient computation in our NonDP baseline and address a reviewer's concern, we now present two mathematically equivalent formulations:

Format 1 (Used in Our Implementation). The default in frameworks like PyTorch is to use batched matrix operations. Let $\nabla_Y \in \mathbb{R}^{B\times T\times D}$ be the gradient of the output and $X \in \mathbb{R}^{B\times T\times P}$ be the input activation. The weight gradient $\nabla_W \in \mathbb{R}^{D\times P}$ is computed as:

$$\nabla_W = (\nabla_Y)^\mathsf{T} \cdot X \tag{7}$$

This corresponds to the batched GEMM routine invoked during `loss.backward()` and does not require computing or storing per-sample gradients.

Format 2 (Shown in Figure 6a for Comparison Only). For structural alignment with the DP workflows, we also illustrate an equivalent formulation that computes per-sample gradients and then aggregates them:

$$G^{(b)} = \sum_{t=1}^{T} (\nabla_Y^{(b,t)})^\mathsf{T} X^{(b,t)}, \quad \nabla_W = \sum_{b=1}^{B} G^{(b)} \tag{8}$$

This version is shown in Figure 6 (a) only to highlight architectural differences across methods. We reiterate that this is not used in our actual NonDP implementation.

Summary. We emphasize that our implementation of the NonDP baseline strictly uses Format 7 (batched GEMM) and does not compute or store per-sample gradients. The inclusion of per-sample nodes in Figure 6(a) is purely illustrative and will be clarified in the revised caption and main text.

C.2 Explicit DP-SGD Workflow

Figure 6 (b) terms the explicit method (e.g., Opacus, FastClip), demonstrates the traditional DP approach where per-sample gradients are stored explicitly, resulting in increased memory usage due to the retention of individual gradient information for noise addition and clipping. The explicit DP-SGD workflow is normally organized into four distinct stages to ensure adherence to privacy constraints:

Stage 1: Per-sample Gradient Computation. At this initial stage, the activation tensor $X \in \mathbb{R}^{B\times T\times P}$ and the output gradient tensor $\nabla_Y \in \mathbb{R}^{B\times T\times D}$ are loaded in blocks from the HBM to the on-chip SRAM. The per-sample gradients tensor $\mathbf{G} \in \mathbb{R}^{B\times D\times P}$ is computed by performing the operation $\mathbf{G} = \sum_T \nabla_Y^T X$ directly on the SRAM to minimize latency, effectively implementing a batched GEMM operation, where each slice of $\mathbf{G}$ is per-sample gradient. After computation, the per-sample gradients are written back to the HBM for further processing.

Stage 2: Gradient Norm Computation. The computed per-sample gradients $\mathbf{G}$ are again loaded into SRAM in smaller blocks. The norm of per-sample gradient is then computed on-chip, $\|\mathbf{G}\| = \sqrt{\sum_D \sum_P \mathbf{G}} \in \mathbb{R}^B$. Then, this norm calculation is stored in HBM.

Stage 3: Gradient Clipping. This stage involves loading both the per-sample gradients $\mathbf{G}$ and its norm $\|\mathbf{G}\|$ from the HBM into SRAM. The clipping operation is performed by computing $\mathbf{G}' = \mathbf{G}/\max\left(1, \frac{\|\mathbf{G}\|}{C}\right)$ (this division occurs in dimension B), ensuring that each gradient's norm does not exceed the clipping threshold C. The clipped gradients $\mathbf{G}'$ are then stored back in HBM.

Stage 4: Noise Addition and Aggregation. In the final stage, the clipped per-sample gradients $\mathbf{G}'$ are loaded into SRAM, and Gaussian noise $\mathcal{N}(0, \sigma^2 C^2 \mathbf{I})$ is added to each, according to the specified noise scale σ. This process ensures differential privacy by obfuscating the contributions of individual training examples. The noisy, aggregated gradient for the weight update, $\nabla_W = \sum_B \mathbf{G}' + \mathcal{N}(0, \sigma^2 C^2 \mathbf{I})$, is computed and then written to HBM, ready for updating the model parameters.

Limitations. Standard DP-SGD requires the explicit storage of per-sample gradients in HBM, which is crucial for computing the gradient norms needed for clipping. This requirement substantially increases the memory footprint. This method becomes impractical for LLMs, which have large model parameters and gradients due to extended sequence lengths. The extensive memory needed to store these gradients often exceeds the available HBM capacity, leading to frequent data swapping between memory and processing units, which severely slows down the training process. Crucially, the computation of gradient norms breaks down standard kernel fusion strategies, preventing the efficient integration of gradient computation and subsequent processing steps into a single operation, resulting in increased latency and inefficient GPU utilization.

C.3 Implicit DP-SGD Workflow

Figure 6 (c) illustrates the implicit method (e.g., GhostClip, BK), which optimizes the DP-SGD process by recalculating gradients in a fused manner, thereby avoiding the explicit storage of per-sample gradients. This approach reduces memory demands but introduces computational redundancy due to multiple gradient recalculations. The implicit DP-SGD workflow is normally organized into two distinct stages:

Stage 1: Fused Computation (corresponds to Stage 1-3 of the explicit method). In the implicit method, stages 1 through 3 of the explicit method are executed in a fused computational process. This involves loading the activation tensor $X \in \mathbb{R}^{B\times T\times P}$ and the output gradient tensor $\nabla_Y \in \mathbb{R}^{B\times T\times D}$ into SRAM. The per-sample gradients tensor $\mathbf{G} \in \mathbb{R}^{B\times D\times P}$ is recalculated by integrating gradient computation, norm calculation, and clipping into a single pass. This minimizes latency and avoids repeated data transfers to HBM. During this fused operation, the per-sample gradient norms are calculated $\|\mathbf{G}\|$ directly on the chip. Clipping is simultaneously performed by scaling the gradients: $\mathbf{G}' = \mathbf{G}/\max\left(1, \frac{\|\mathbf{G}\|}{C}\right)$, where C is the clipping threshold. These operations are performed without storing the intermediate states, reducing the memory footprint.

Stage 2: Noise Addition and Aggregation (corresponds to stage 4 of the explicit method). The clipped gradients $\mathbf{G}'$ are recalculated and loaded into SRAM where Gaussian noise $\mathcal{N}(0, \sigma^2 C^2 \mathbf{I})$ is added, adhering to the specified noise scale σ. The final aggregate gradient is then computed and written back to HBM for the model update.

Limitations of Implicit methods: Implicit methods attempt to mitigate the high memory usage by segmenting the gradient computation and clipping it into several smaller, manageable tasks. However, these methods involve multiple recalculations of the per-sample gradients, which is computationally expensive.

D Additional Experiments Settings and Explanations

D.1 Experiments Settings

Our evaluations mainly focus on memory usage (MB) and throughput (tokens/sec) to determine the efficiency. We also show the loss of the validation data to measure the utility of private pre-training. Unless specified otherwise, the settings for each experiment use GPT-2 models with a sequence length of 1024, and Llama models with a sequence length of 2048, employing the AdamW optimizer as the base.

Batch Size & Micro Batch Size For the batch size experiment, we vary the batch sizes at 1, 2, 4, and 8, using GPT-2 models of small, medium, and large scales to test the method's scalability and efficiency. Similarly, in the micro-batch size experiment, we set the micro-batch sizes at 1, 2, 4, and 8, with a gradient accumulation step of 4.

Experiments on Testing Utility We conduct an experiment to evaluate the performance of the GPT2-small model trained from scratch using DP-SGD and FlashDP under differential privacy constraints, with epsilon values set at 0.2, 0.5, and 0.8. The model is trained on the Fineweb-edu (Lozhkov et al., 2024) dataset. Key hyperparameters include a total batch size of 524,288 tokens, a micro batch size per device of 32, and a sequence length of 1024. We use a maximum learning rate of 6×10^{-4} and a minimum learning rate of 6×10^{-5}, with weight decay set at 0.1 and gradient clipping at 1.0. The model undergoes training with a validation frequency every 250 steps and model saving every 5000 steps, using both DP-SGD and FlashDP, enabling differential privacy with delta set at 1×10^{-5} and a clipping threshold of 100. The training aims to compare utility across different privacy levels and analyze the trade-offs between privacy and utility. We use the validation loss as the evaluation metric in Table 3.

Distributed Training DDP involves distributing the model's parameters across several devices, and each device computes gradients for a subset of the data independently. This method is beneficial for managing models that fit within the memory limits of a single GPU but need faster processing through parallel execution. On the other hand, Pipeline Parallel (PP) splits the model's layers across different devices, allowing different parts of the model to be processed simultaneously. PP is particularly useful for very large models that exceed the memory capacity of individual GPUs, enabling concurrent processing of different stages of the model across the pipeline. The experiments with DDP and PP are designed to evaluate the effectiveness of FlashDP in a distributed training context, assessing its performance in terms of memory usage and throughput across various model sizes and batch sizes. These experiments are critical to demonstrate that FlashDP can maintain its efficiency and scalability when applied to state-of-the-art LLMs, which require substantial computational resources and sophisticated training mechanisms to manage their size and complexity.

In this setup, we explore the scaling capabilities of FlashDP using DDP on four A100 GPUs (80GB each) by training GPT-2 models of small, medium, and large sizes with fixed sequence lengths of 1024 and varying batch sizes of 8, 4, and 2. Additionally, PP experiments are conducted on Llama models of sizes 3B, 7B, and 13B to evaluate throughput and memory efficiency across different stages of the model pipeline. It is important to note that GhostClip and BK do not support the distributed modes we used.

D.2 Additional Explanations

GhostClip initially supports only global clipping; however, it can be easily adapted to per-layer clipping as outlined in Algorithm 6.

E More Experimental Results

E.1 Results of Micro Batch Size

Table 2: **Micro Batch Size Analysis.** Comparing memory and throughput at varying micro batch sizes B (1, 2, 4, 8) and the same gradient accumulation steps (4) for GPT-2 sizes with differential privacy methods under consistent settings with Table 1.

Model	B	Memory Usage (MB x1e4)					Throughput (tokens/sec x1e4)				
		NonDP	Opacus	GhostClip	BK	FlashDP	NonDP	Opacus	GhostClip	BK	FlashDP
GPT2-small	1	0.51	0.97(x1.90)	0.51(x1.00)	0.71(x1.39)	**0.51(x1.00)**	3.07	1.20(x0.39)	0.60(x0.20)	1.75(x0.57)	**1.86(x0.61)**
GPT2-medium	1	1.26	1.69(x1.34)	**1.25(x0.99)**	1.81(x1.44)	1.26(x1.00)	1.27	0.61(x0.48)	0.45(x0.35)	0.86(x0.68)	**0.91(x0.72)**
GPT2-large	1	2.48	3.64(x1.47)	**2.46(x0.99)**	3.21(x1.29)	2.48(x1.00)	0.67	0.39(x0.43)	0.32(x0.46)	0.47(x0.69)	**0.53(x0.89)**
GPT2-small	2	0.87	1.15(x1.32)	1.00(x1.15)	1.06(x1.22)	**0.87(x1.00)**	3.22	1.68(x0.52)	0.92(x0.29)	1.91(x0.59)	**2.32(x0.72)**
GPT2-medium	2	2.07	2.88(x1.39)	**2.01(x0.97)**	2.62(x1.27)	2.07(x1.00)	1.38	0.88(x0.64)	0.65(x0.47)	0.88(x0.64)	**1.04(x0.75)**
GPT2-large	2	3.91	6.07(x1.55)	**3.83(x0.98)**	4.43(x1.13)	3.91(x1.00)	0.74	0.46(x0.62)	0.43(0.58)	0.49(x0.66)	**0.59(x0.80)**
GPT2-small	4	1.53	2.10(x1.37)	**1.48(x0.97)**	1.73(x1.13)	1.53(x1.00)	3.72	2.49(x0.67)	1.50(x0.40)	2.30(x0.62)	**2.59(x0.70)**
GPT2-medium	4	3.58	5.51(x1.54)	**3.46(x0.97)**	4.04(x1.13)	3.58(x1.00)	1.48	0.97(x0.66)	0.86(x0.58)	0.99(x0.67)	**1.29(x0.87)**
GPT2-large	4	6.60	-	**6.45(x0.98)**	-	6.60(x1.00)	0.79	-	0.53(x0.67)	-	**0.65(x0.82)**
GPT2-small	8	2.86	4.00(x1.40)	**2.78(x0.97)**	3.06(x1.07)	2.86(x1.00)	3.87	2.60(x0.67)	1.99(x0.51)	2.44(x0.63)	**2.73(x0.71)**
GPT2-medium	8	6.60	-	**6.37(x0.97)**	7.16(x1.08)	6.60(x1.00)	1.55	-	1.03(x0.66)	1.05(x0.68)	**1.19(x0.77)**
GPT2-large	8	-	-	-	-	-	-	-	-	-	-

Algorithm 6 Per-Layer GhostClip

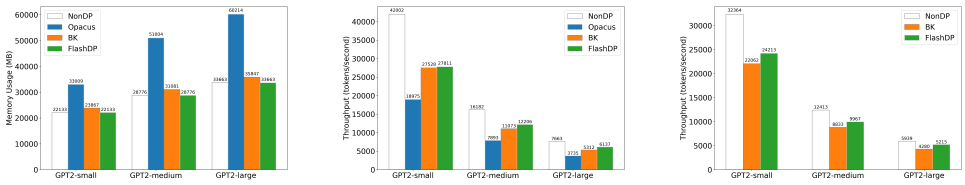
Require: $\mathcal{L}(\theta^{(l)}, x_i)$: Loss function for layer parameters $\theta^{(l)}$ and input x_i
Require: $C^{(l)}$: Clipping threshold for layer l
Require: $\sigma^{(l)}$: Noise scale for layer l
Require: L: Total number of layers

1: **for** each layer $l = 1$ to L **do**
2: **for** $i = 1$ to B **do**
3: Compute gradient norm for layer l: $\|g_i^{(l)}\| = \|\nabla\mathcal{L}(\theta^{(l)}, x_i)\|$ by first computing per-sample gradient in-place then computing per-sample norm.
4: Clip gradient for layer l: $g_i'^{(l)} = g_i^{(l)} \min\left(1, \frac{C^{(l)}}{\|g_i^{(l)}\|_2}\right)$ by re-computing per-sample gradient $g_i^{(l)}$ in-place.
5: **end for**
6: Aggregate clipped gradients for layer l and add Gaussian noise: $\tilde{g}^{(l)} = \frac{1}{B}\sum_{i=1}^{B} g_i'^{(l)} + \mathcal{N}(0, (\sigma^{(l)}C^{(l)})^2 I)$
7: **end for**

Table 2 further explores the impact of varying micro batch sizes, a crucial factor for managing memory in constrained environments and optimizing the use of gradient accumulation steps. FlashDP consistently displayed minimal memory footprint increases and maintained high throughput efficiency, even as micro batch sizes increased. For example, at a micro batch size of 8 for the GPT-2 medium model, FlashDP's memory usage was 6.49×10^4 MB–marginally higher than its usage at smaller micro batch sizes and significantly lower than Opacus at the same size. This robust performance underscores FlashDP's effective management of memory, which is essential for scaling up the training of large models without excessive hardware requirements.

To be specific, 1) Opacus showed a consistent increase in memory usage as micro batch sizes increased, which is indicative of its inefficient memory handling under fragmented gradient computations. 2) GhostClip, while better in memory usage compared to Opacus, didn't scale as well in throughput, which decreased noticeably with larger micro batches, reflecting the computational cost of gradient recalculations. 3) BK displayed trends similar to Opacus but generally used slightly less memory and provided slightly better throughput, suggesting a more optimized handling of gradient accumulation steps. 4) FlashDP maintained minimal increases in memory usage with increasing micro batch sizes and consistently provided the highest throughput, highlighting its effective integration of operations within the computational workflow. To summarize, as the micro batch size increases, FlashDP's memory usage increases only slightly and still maintains the highest throughput, demonstrating its efficient memory management techniques.

E.2 Results of AMP Training Scalability



(a) Memory Usage - float16 (b) Throughput - float16 (c) Throughput - bfloat16

Figure 4: **Memory and Throughput Analysis of GPT-2 Models Using Automatic Mixed Precision (AMP) Training Across Float16 and BFloat16 Precision.**: (a) Demonstrates the memory usage for GPT-2 small, medium, and large models with Float16 precision. (b) shows throughput using Float16 precision, and (c) shows throughput with BFloat16 precision.

Automatic Mixed Precision (AMP) (Micikevicius et al., 2017) training involves utilizing lower precision formats like float16 and bfloat16 within a training session to reduce computational demands and memory usage. This strategy is particularly valuable for large language models (LLMs), which typically require substantial computational resources. By employing AMP, training processes can be accelerated, and larger models or batches can be managed more efficiently without proportional increases in hardware capacity. The integration of differential privacy with AMP, especially in techniques like FlashDP, is critical for exploring the practical limits of DP-SGD. This experiment assesses how FlashDP adapts to AMP settings compared to other methods, and evaluates the impact on memory efficiency and processing speed, which are crucial for the scalability of private training in constrained environments.

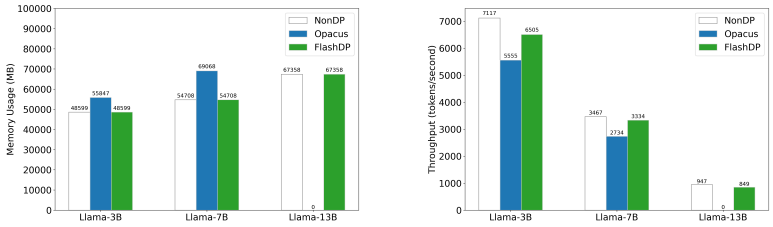
In our experiments, we analyze GPT-2 models of varying sizes using batch sizes of 8, 4, and 2 across float16 and bfloat16 precision formats to measure memory usage and throughput, examining FlashDP's performance relative to NonDP, Opacus, and BK methods. It is important to note that GhostClip does not support AMP, and Opacus does not support the bfloat16 precision format.

Memory Usage Analysis. As depicted in Figure 4 (a), the memory usage across GPT-2 models of different sizes indicates that FlashDP, when utilizing AMP in both float16 and bfloat16 formats, maintains lower memory consumption compared to Opacus and BK, and closely approximates the NonDP configuration. This showcases FlashDP's effective use of AMP to minimize memory overhead, facilitating the training of large models under stringent privacy constraints.

Throughput Analysis with Float16 and BFloat16. In terms of throughput, Figure 4 (b) and 5(c) present a comprehensive look at the advantages of using float16 and bfloat16 precision formats under AMP. FlashDP consistently outperforms Opacus and BK in throughput metrics across both precision types. This is especially notable in larger model configurations, where the differences in throughput become more pronounced, highlighting FlashDP's capability to handle extensive computational loads efficiently. As demonstrated in Figure 5(b), FlashDP exhibits significant throughput advantages over the other DP methods. This performance is indicative of the efficient computational optimizations that FlashDP leverages within the AMP framework. As shown in Figure 4 (c), while bfloat16 typically offers slightly lower computational throughput than float16 due to its numerical properties, FlashDP's implementation still ensures that it outperforms other differential privacy methods. This underscores FlashDP's robust performance across varying precision settings.

E.3 Results of Distributed Training

Distributed Data Parallel (DDP) (Li et al., 2020) and Pipeline Parallel (PP) (Kim et al., 2020) are two advanced techniques crucial for scaling the training of LLMs efficiently across multiple GPUs or nodes.



(a) Memory Usage (b) Throughput

Figure 5: **Memory and Throughput for Llama Models Using Pipeline Parallel Training.** (a) Memory usage for Llama-3B, Llama-7B, and Llama-13B models. (b) Throughput in tokens per second across these model sizes. A value of 0 indicates out of memory.

Distributed Data Parallel (DDP). Figure 7 in Appendix illustrates the performance of different methods in a DDP setting across GPT-2 models of varying sizes. FlashDP showcases superior memory usage efficiency and higher throughput across all model sizes when compared to Opacus and BK. Notably, even as the model size increases, FlashDP maintains a competitive edge close to the

NonDP benchmarks, highlighting its effective parameter distribution and gradient computation across multiple GPUs. This is crucial in scenarios where training speed and model scalability are priorities.

Pipeline Parallel (PP). In the PP scenario depicted in Figure 5, FlashDP was tested with Llama models varying from 3 billion to 13 billion parameters. The results indicate that FlashDP not only scales efficiently with increasing model size but also demonstrates significant throughput improvements compared to Opacus and BK. Particularly, FlashDP's ability to handle the largest model (Llama-13B) with minimal throughput degradation illustrates its robustness in managing extensive computational loads, characteristic of PP environments.

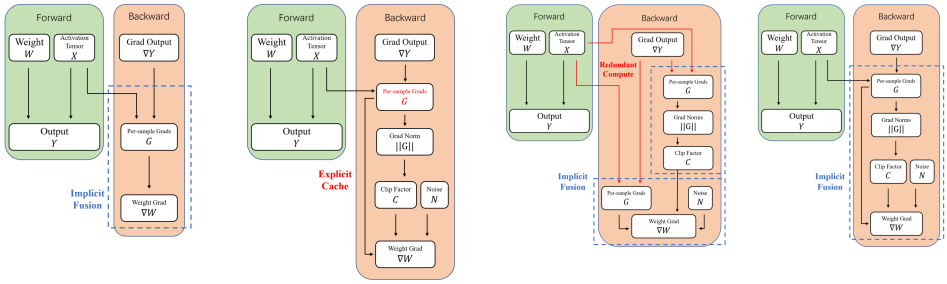
E.4 Results of Utility

Table 3: **FlashDP Pretrain Precision validation on GPT2-small with different privacy ϵ.**

Method	Validation loss		
	$\epsilon = 0.2$	$\epsilon = 0.5$	$\epsilon = 0.8$
DP-SGD	4.8082	4.8063	4.8061
FlashDP	4.8082	4.8063	4.8061

In our study, FlashDP is meticulously optimized for DP-SGD, focusing on enhancing GPU I/O and system-level efficiencies without altering the fundamental algorithmic components of per-layer DP-SGD. We conducted experiments on utility with GPT-2 small to support this, whose results are shown in Table 3. From the table, we can easily see that FlashDP demonstrates an identical validation loss to that of DP-SGD across all privacy levels.

F Additional Tables and More Figures

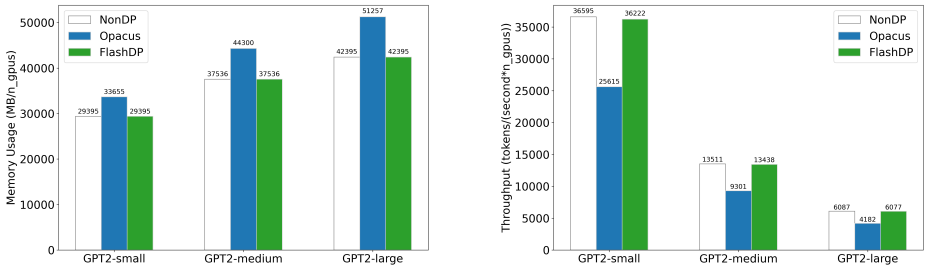


(a) Non-DP (b) Explicit Method (e.g. Opacus, FastClip) (c) Implicit Method (e.g. GhostClip, BK) (d) FlashDP

Figure 6: Comparison of different training methods. (a) Non-DP: Basic training without DP. (b) Explicit Method (e.g., Opacus, FastClip): Stores per-sample gradients $\mathbf{G}$ (red explicit cache), increasing memory usage. (c) Implicit Method (e.g., GhostClip, BK): Reduces memory by recalculating gradients in fused manners (blue dotted box) but implicitly calculating the per-sample gradient twice, causing computational redundancy. (d) FlashDP: Optimizes gradient processing by consolidating computations into a single pass, reducing redundancy and memory use.

Table 4: Comparison of Backward Propagation Methods.

Method	Per-sample Gradient		Implicit Fusion
	Cache	Recalculation	
Non-DP	✗	✗	✓
Explicit-DP	✓	✗	✗
Implicit-DP	✗	✓	✓
FlashDP	✗	✗	✓



(a) Memory Usage

(b) Throughput

Figure 7: **Memory and Throughput for GPT Models Using Distributed Data Parallel Training.** (a) Memory usage for GPT-samll, GPT-medium, and GPT-large models. (b) Throughput in tokens per second across these model sizes. A value of 0 indicates out of memory.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction clearly describe the paper's key contributions, including the design of FlashDP for efficient per-layer DP-SGD training, GPU I/O optimization via fused kernels, and scalability to large models like Llama-13B. These are substantiated by both theoretical and experimental results (see Sections 1 and 4âĂŞ6).

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The conclusion and discussions throughout the paper acknowledge the current scope of FlashDP, including its focus on per-layer clipping rather than global clipping, and reliance on fused CUDA kernels, which may require hardware-specific tuning (see Section 6 and Appendix A.3).

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.

- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: The paper is primarily systems- and engineering-oriented and does not include formal theoretical results requiring assumptions or proofs.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper provides detailed descriptions of model sizes, batch configurations, datasets, and hardware used in experiments (see Section 5.1 and Appendix C), allowing reproduction of key results even without code.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may

be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.

- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: Due to ongoing code cleanup and anonymization, code is not yet released. However, we commit to open-sourcing FlashDP with reproduction scripts upon acceptance.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so âĂIJNoâĂİ is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

Justification: The training settings, model variants, datasets (Wikitext), and hardware (A100 GPUs) are clearly described in Section 5.1 and Appendix C, with baseline comparison details provided in Table 1.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: Error bars are not reported because it would be too computationally expensive.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: Section 5.1 specifies that experiments were conducted on NVIDIA A100 GPUs (80GB), and various resource constraints (e.g., out-of-memory for Opacus) are reported in Table 1 and Figure 3.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research follows the NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: The conclusion section discusses the positive impact of enabling scalable privacy-preserving LLMs. Potential negative societal impacts (e.g., misuse of DP-trained models) are acknowledged.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: Our work does not release pretrained models or datasets with high risk of misuse.

    Guidelines:

    - The answer NA means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All datasets and tools used are publicly available and appropriately cited (e.g., Wikitext, PyTorch, Opacus). See Section 5.1 and References.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: No new datasets or models are released in this paper.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: This research does not involve human participants or crowdsourcing.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: No IRB approval is required as the study does not involve human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [NA]

    Justification: While our work targets LLM training, it does not use an LLM as a methodological component or for content generation. No LLMs were used to produce the paper content.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.