FlashDP: Memory-Efficient and High-Throughput DP-SGD Training for Large Language Models

Liangyu Wang[1] **Junxiao Wang**[1,2] **Jie Ren**[1] **Zihang Xiang**[1]
David E. Keyes[1] **Di Wang**[1]
[1]King Abdullah University of Science and Technology, [2]Guangzhou University
{liangyu.wang, junxiao.wang, jie.ren, zihang.xiang}@kaust.edu.sa
junxiao.wang@gzhu.edu.cn
{david.keyes, di.wang}@kaust.edu.sa

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 method 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 DP-SGD in terms of accuracy. These advancements establish FlashDP as a pivotal development for efficient and privacy-preserving training of LLMs.

1 Introduction

The transformer architecture [1] has revolutionized fields like natural language processing [2, 3], embodied AI [4, 5, 6], and AI-generated content (AIGC) [7, 8], with Large Language Models (LLMs) demonstrating exceptional abilities in text generation, complex query responses, and various language tasks due to training on massive datasets.

Differential Privacy (DP) enhances data privacy by minimizing individual data points' influence through noise addition [9]. In deep learning, Differentially Private Stochastic Gradient Descent (DP-SGD) [10] is widely used, particularly in LLM fine-tuning [11, 12, 13, 14]. However, its application in LLM pre-training is limited [15, 11, 12], often constrained by shorter sequence lengths due to high computational and memory demands, thus not fully leveraging the capabilities of modern LLMs.

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

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

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 [11, 12].

Current research on DP-SGD for training LLMs can be categorized into two classes: explicit methods like Opacus [16] stand out by directly storing per-sample gradients. This approach, while straightforward, significantly increases the memory footprint (Appendix Table 3), which becomes prohibitive for state-of-the-art LLMs characterized by billions of parameters [17, 18]. 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.

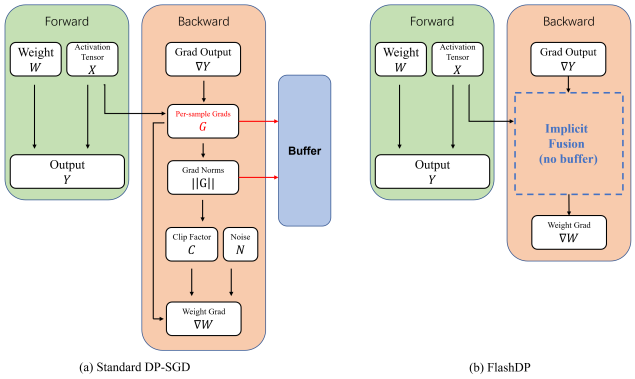


Figure 1: Comparison of different training methods. (a) Standard DP-SGD: Stores per-sample gradients $\mathbf{G}$ (red explicit cache), increasing memory usage (blue buffer). (b) FlashDP: Optimizes gradient processing by consolidating computations into a single pass, reducing redundancy and memory use.

Conversely, implicit methods, exemplified by innovations such as GhostClip [19], 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 3). 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 DP-SGD process (Figure 1 (a)). FlashDP uniquely implements a unified computational strategy that performs the gradient operations required for DP-SGD in a single pass (Figure 1 (b)). Our contributions can be summarized as follows:

- **Enhanced Throughput for Long Sequence LLM training with DP**: We propose FlashDP, which effectively resolves the issue of low throughput in DP-SGD/Adam during the training of LLMs with long sequence lengths. 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 standard 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 FlashDP Algorithm Design

This section begins with an overview of relevant literature and foundational concepts, found in Appendices A and B respectively. Subsequently, we detail our proposed FlashDP algorithm.

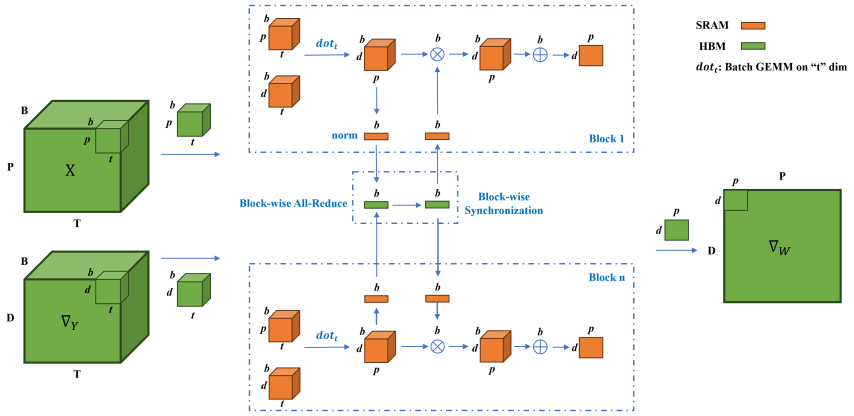


Figure 2: **Illustration of FlashDP.** It depicts the core algorithm design of FlashDP. Its features are integrated with on-chip per-sample gradient norm calculations. The workflow incorporates block-wise all-reduce and synchronization to facilitate efficient norm aggregation. SRAM (orange) and HBM (green)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

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.

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 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. 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. 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. 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.

Gradient Clipping and Noise Addition in SRAM. 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. 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.

3 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 (Section 3.2), its consistency across different sequence lengths (Appendix D.2), and its scalability when employing Distributed Data Parallel (DDP) and Pipeline Parallel (PP) techniques (Appendix D.3, including Llama-13B experiments).

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 FlashDPâĂŤacross 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	-	-	-	-	-	-	-	-	-	-

3.1 Experimental Setup

Our experiments utilize the Wikitext dataset [20] and are conducted on NVIDIA A100 (80GB) GPUs using the PyTorch framework [21]. We assess the performance of FlashDP across various configurations by comparing it with established explicit methods Opacus [16], and implicit method GhostClip [19] and BK [12], under different training paradigms. The tested models include GPT-2 [22] with a sequence length of 1024 and the TinyLlama [23] and Llama [17] models, both with a sequence length of 2048. 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. More experimental settings can be found in Appendix C.

3.2 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 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 D.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.

4 Conclusion

In this paper, we introduce FlashDP, a novel approach for integrating differentially private SGD (DP-SGD) into the training of large language models (LLMs) while enhancing memory efficiency and computational throughput. By optimizing GPU input/output operations, FlashDP significantly reduces the memory transaction overhead, allowing it to achieve near-non-private throughput levels while maintaining strict privacy standards. Central to FlashDP's strategy is the Block-wise All-Reduce algorithm, which integrates several critical operations into a unified kernel execution. To achieve this, we propose a Hierarchical Reduction Architecture (HRA), which encompasses more than just reduction operations. Moreover, we employ an adaptive kernel approach to implement HRA, which addresses the limitations of CUDA's programming model in facilitating block synchronization. Our experiments demonstrate that FlashDP reduces memory usage to levels comparable with non-private methods and increases throughput, making the efficient training of substantial models like the Llama 13B feasible on modern hardware. The minimal interference in the training process and the maintenance of computational precision suggest that FlashDP could significantly advance the adoption of DP in sectors where privacy is crucial, making secure and efficient machine learning more accessible for a wider range of applications.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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*, pages 2998–3009, 2023.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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*, pages 308–318, 2016.

[11] 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.

[12] 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*, pages 3192–3218. PMLR, 2023.

[13] 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*, pages 6481–6491, 2022.

[14] 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*, pages 1178–1189, 2021.

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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. Tinyllama: An open-source small language model. *arXiv preprint arXiv:2401.02385*, 2024.

[24] 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. Software available from tensorflow.org.

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

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

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

[28] 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.

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

[30] 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.

[31] 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.

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

[33] 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)*, pages 4118–4122. IEEE, 2022.

[34] 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*, pages 265–284. Springer, 2006.

[35] 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.

[36] 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.

Appendix

A 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 [24]. Opacus [16] and [25] enhance the training efficiency by employing the outer product method [26], albeit at the cost of increased memory usage needed to store per-sample gradients. This memory overhead is mitigated in FastGradClip [27] by distributing the space complexity across two stages of backpropagation, effectively doubling the time complexity. Additionally, ghost clipping techniques [26], [19], [28] allow for clipping per-sample gradients without full instantiation, optimizing both time and space, particularly when feature dimensions are constrained. Furthermore, [12] introduces a 'book-keeping' (BK) method that achieves high throughput and memory efficiency while falling short in handling the long sequence lengths typical in LLM 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 long-sequence LLMs. FlashDP aims to enhance the efficiency and feasibility of training LLMs with long sequences 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. [29] demonstrated that public pretraining could facilitate downstream DP fine-tuning, although they did not explore fine-tuning large pre-trained models using DP-SGD. [30] explored the fine-tuning of BERT for language understanding tasks under local DP. [31] suggested the potential for cost-effective private learning through fine-tuning large pre-trained language models. [32] and [33] extended these studies to BERT, pretraining and fine-tuning under global DP, respectively, with [32] addressing datasets comprising hundreds of millions of examples, and [33] 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.

B Preliminaries

B.1 Differential Privacy

Definition 1. *(Differential Privacy [34]) 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) [10]. 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 Differentially Private Adam (DP-Adam) instead of DP-SGD. 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 1-3.

Algorithm 1 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 2 DP-SGD Specific Steps

Require: θ: Model parameters
Require: η: Learning rate
1: **for** each training step **do**
2: Perform common gradient processing as in Algorithm 1
3: Update model parameters: $\theta \leftarrow \theta - \eta\tilde{g}$
4: **end for**

Algorithm 3 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 1
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**

B.2 Transformers

The transformer architecture, proposed by Vaswani et al. [1], 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.

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 modelâĂŹs 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 GPUâĂŹs 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 GPUâĂŹs shared memory constraints and the size of the SMsâĂŹ 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 Additional Experiments Settings

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.

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	-	-	-	-	-	-	-	-	-	-

D More Experimental Results

D.1 Results of Micro Batch Size

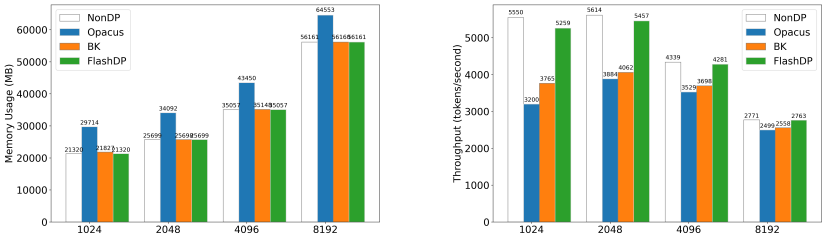
Table 2 further explores the impact of varying micro batch sizes, a crucial factor for managing memory in constrained environments and optimizing the use of gradient accumulation steps. FlashDP consistently displayed minimal memory footprint increases and maintained high throughput efficiency, even as micro batch sizes increased. For example, at a micro batch size of 8 for the GPT-2 medium model, FlashDPâĂŹs memory usage was 6.49×10^4 MBâĂŤmarginally higher than its usage at smaller micro batch sizes and significantly lower than Opacus at the same size. This robust performance underscores FlashDPâĂŹs 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.

D.2 Results of Sequence Length

In the training of LLMs, the ability to process long sequences of data is crucial for enhancing the model's capability to understand and generate coherent, contextually rich text.

Memory Usage Analysis. As illustrated in Figure 3 (a), there is a clear trend of increasing memory usage with longer sequence lengths across all methods, which is



(a) Memory Usage (b) Throughput

Figure 3: **Memory and Throughput Comparison for TinyLlama with Varied Sequence Lengths Using Flash Attention.** (a) Memory usage across sequence lengths of 1024, 2048, 4096, and 8192. (b) Throughput measured in tokens per second across the same sequence lengths.

expected due to the larger computational requirements. However, FlashDP always maintains the same GPU memory usage as NonDP, especially at the highest sequence length of 8192. This indicates that FlashDPâĂŹs method is particularly effective at managing the increased memory demands, thus facilitating the scalability of models trained with long sequences.

Throughput Performance. Figure 3 (b) highlights throughput in terms of tokens per second at varying sequence lengths. FlashDP consistently maintains higher throughput compared to Opacus and BK across all sequence lengths, with its performance closely approaching that of the NonDP method. This efficiency in throughput underlines FlashDP's capability to handle larger sequence lengths without significant compromises in processing speed, a critical factor for training usable and responsive LLMs.

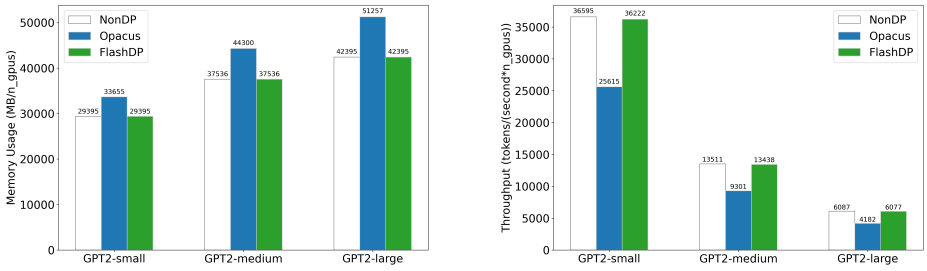
The experimental data clearly demonstrates FlashDPâĂŹs superior memory management and throughput efficiency across a range of sequence lengths. The ability of FlashDP to handle longer sequences with minimal increase in memory usage and only slight reductions in throughput is particularly impressive.

D.3 Results of Distributed Training

Distributed Data Parallel (DDP) [35] and Pipeline Parallel (PP) [36] are two advanced techniques crucial for scaling the training of LLMs efficiently across multiple GPUs or nodes.

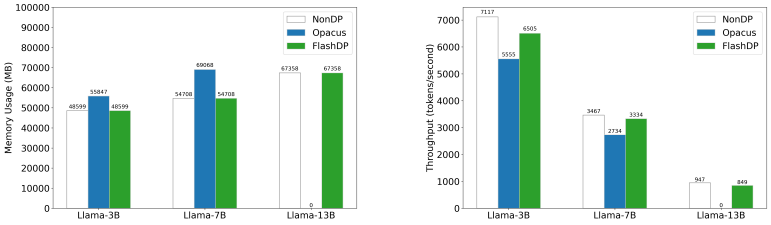
Distributed Data Parallel (DDP). Figure 4 in Appendix illustrates the performance of different methods in a DDP setting across GPT-2 models of varying sizes. FlashDP showcases superior memory usage efficiency and higher throughput across all model sizes when compared to Opacus and BK. Notably, even as 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



(a) Memory Usage (b) Throughput

Figure 4: **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.



(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.

indicate that FlashDP not only scales efficiently with increasing model size but also demonstrates significant throughput improvements compared to Opacus and BK. Particularly, FlashDPâĂŹs 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 Additional Tables

Table 3: Comparison of Backward Propagation Methods.

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