분산 다중 GPU 환경에서의 생성 모델 추론 가속화

유준열°, 김진표, 이재진

서울대학교 컴퓨터공학부
junyeol, jinpyo@aces.snu.ac.kr jaejin@snu.ac.kr

A Fast and Scalable Generative Model Inference on Distributed Multi-GPU Environment

Junyeol Ryu°, Jinpyo Kim, Jaejin Lee
Department of Computer Science and Engineering, Seoul National University

요 약

Generative models (e.g., GPT-4) have recently attracted a huge demand and proposed new challenges for a fast and scalable generative model inference. We analyze the architecture and potential performance bottlenecks of the generative models and propose novel techniques to improve generative model inference latency and scalabilty. Our code is implemented and publicly available at `https://github.com/gajagajago/fastgen`.

1. Introduction

Generative models are becoming paramount in diverse creative AI domains, such as DALLE-2 [10] for image generation, ChatGPT [9] for chatbot, and Github Copilot [6] for code generation. The growing demand for generative models requires the inference results to be provided at low latency and high throughput [12].

In this paper, we analyze the architecture and potential performance bottlenecks of a generative model and propose three optimization techniques: batching, matrix multiplication kernel optimization, and kernel fusion. Our key contributions are:

- We present an analysis of generative model architecture using a prototype sequence generation model, and show that auto-regressive and compute-heavy nature of its architecture are potential limitations to the latency and scalability.
- We propose three novel optimization techniques to overcome the challenges in generative model inference, and evaluate the performance benefits on a distributed multi-GPU environment. We show that our optimization techniques significantly improves inference throughput by 430×. We also demonstrate that our optimization can exploit 80.1% compute capabilities of GPU and achieve near-linear scalability of 89.7%.
- We open-source our codes at `https://github.com/gajagajago/fastgen` for future research in generative model inference.

2. Model Architecture

In this section, we describe the general architecture of generative models using a prototype of a sequence generation model, *Namegen*, that runs an auto-regressive inference procedure to generate an English name.

Inference procedure of generative model Figure 1 illustrates the computation graph of Namegen, where nodes and edges indicate the layers and the dependencies between the layers, respectively. We define the run of all layers as a *step* and the input/output of a step as a *token*. The inference procedure of Namegen is auto-regressive in that the output (y_n) of a step is fed back to the next step as an input. This iterative procedure is started with an initial *sos* input token, and repeats until either *eos* token is generated or a predefined maximum sequence length is reached.

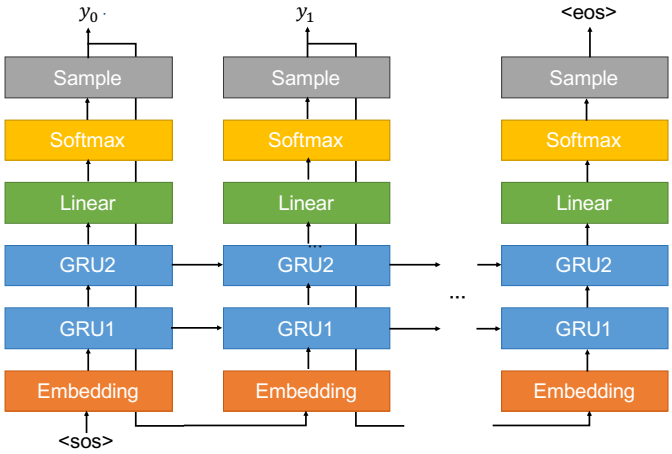

Figure 1: generative model architecture.

Layers The embedding layer maps an input token to a dense vector of real numbers that efficiently encodes the semantic information which are relevant to the model's task. In natural language processing (NLP), tokens with high semantic similarity are close with respect to their embedding vector distance. Specifically, the embedding layer first converts a token to a unique index and uses an embedding table to look-up the corresponding embedding vector.

GRU layer [3] is a special type of RNN layer that implements a gating mechanism to generate the layer output using the hidden state from the previous step. It is composed of sequential compute-intensive operators, as we thoroughly discuss in Section 3.

The linear layer shapes the GRU layer output back to the token dimension, and the softmax layer transforms the token dimension vector as a probability distribution. The sample layer selects the next token according to the token probability distribution, and feeds the output token to the next step or completes the auto-regressive process and aggregates the collection of the output tokens as a return sequence, which corresponds to a "name" for Namegen.

3. Optimization Techniques

3.1 Batching

Batching [1, 4, 13] is an optimization technique to bundle multiple requests together and run inference procedures concurrently. To exploit the computational capabilities of distributed multi-GPU environment, we implement dynamic batching to decide the efficient

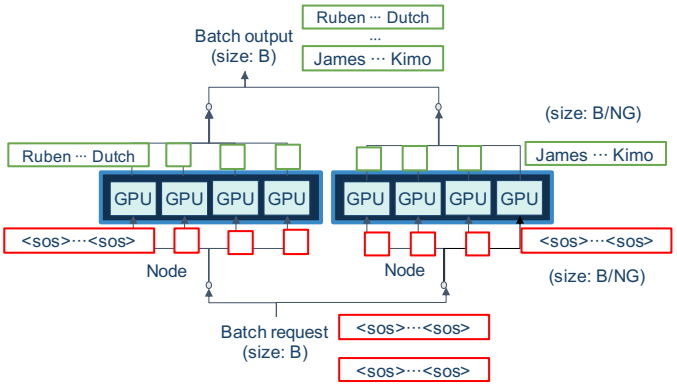

Figure 2: Batching.

batch size B (i.e., the number of requests to run concurrently) on runtime, and configure per-GPU batch size ($\frac{B}{NG}$ where N is the number of nodes and G is the number of GPUs per node) accordingly. To enable each GPU to perform inference on its $\frac{B}{NG}$ batch concurrently, we implement embedding table lookup kernel and other layer kernels to operate on matrix-size inputs. Figure 2 illustrates how the batch inference is distributed and aggregated over multiple GPUs. Each GPU initializes a chunk of $\frac{B}{NG}$ *sos* tokens and independently runs the auto-regressive procedure with replicated model parameters. The batch request finalizes as each GPU generates $\frac{B}{NG}$ sequences and the sequences are aggregated over the interconnected network.

3.2 Matrix Multiplication Kernel Optimization

We observe that matrix multiplication kernels account for 91.5% of the total GPU time by profiling Namegen using NVIDIA Nsight Systems [8]. The asymptotic complexity is $O(M^3)$ assuming a square $M \times M$ matrix multiplication, and it increases proportionally as we increase the batch size to fully exploit the GPU computational capabilities. However, optimizing the performance of matrix multiplication kernel requires thorough understandings of the performance characteristics of the GPUs. We optimize matrix multiplication starting from a naive kernel and step-by-step applying optimizations including global memory coalescing, shared memory caching, 1D/2D block tiling and its variants in consideration with memory access pattern, and loop unrolling.

Figure 3 illustrates the matrix multiplication kernel process with 2D blocktiling. Each thread loads $RPT \times CPT$ column-wise elements from each A and B into shared memory tiles in order to coalesce global memory access within a warp (①). Then, each thread computes and accumulates the partial result in the buffer (②). After iterating over tiles to the dimension boundary (③), the threads store the result in the buffer to C (④).

3.3 Kernel Fusion

GPU kernel fusion [11] is an optimization technique to reduce the kernel launch overhead and redundant global memory (GMEM) access by fusing sequential kernels into a single larger one.

GRU layer computation can be divided into 4 equations as in Figure 4, where x_t is the input vector, h_t is the output vector, z_t, r_t, $\hat{h}_t$ are activations, W, U, b are parameters, and σ_g, ϕ_h are activation functions. We observe that the common part of the activation equations can be identified with $W\square + U\square$ pattern, which involves two matrix multiplications followed by a matrix addition. Assuming a square $M \times M$ matrix for all matrices involved in the pattern, a standalone matrix multiplication/addition kernel requires $2M^2$, M^2 global memory loads and stores, respectively, which sum up to $9M^2$ GMEM accesses for the pattern. In contrast, we implement a fused matrix multiplication-addition kernel that eliminates the redundant store/load of intermediary results, finishing the pattern computation with only $5M^2$ GMEM accesses. Furthermore, our fused kernel utilizes fused multiply-add (FMA) API to contract multiply and add/subtract hardware operations into a single FMA operation.

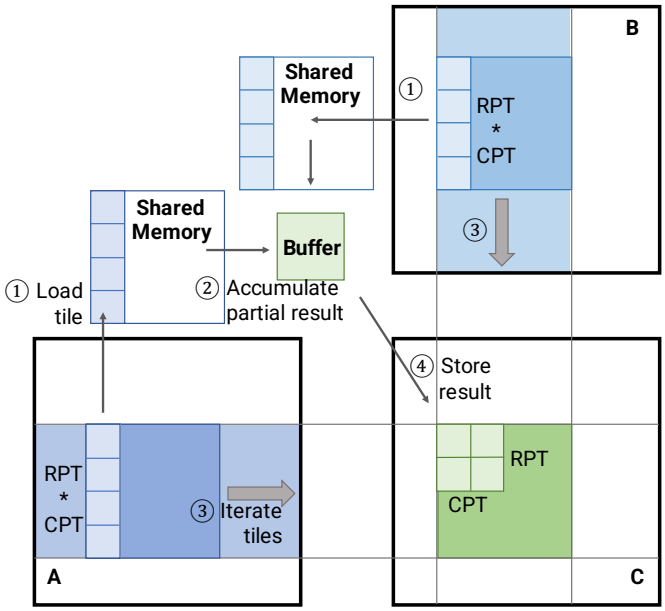

Figure 3: Matrix multiplication with 2D blocktiling. Each thread has an accumulation buffer and computes RPT (rows per thread) $\times$ CPT (columns per thread) elements of C. Two tiles for A and B are shared among threads in the same thread block.

$$
\begin{aligned}
r_t &= \sigma_g \left(W_r x_t + U_r h_{t-1} + b_r \right) \\
z_t &= \sigma_g \left(W_z x_t + U_z h_{t-1} + b_z \right) \\
\hat{h}_t &= \phi_h \left(W_h x_t + U_h \left(r_t \odot h_{t-1} \right) + b_h \right) \\
h_t &= \left(1 - z_t \right) \odot h_{t-1} + z_t \odot \hat{h}_t
\end{aligned}
$$

Figure 4: GRU layer expressions.

4. Evaluation

In this section, we evaluate the performance of our matrix multiplication kernel and the end-to-end performance of optimized Namegen that applies our proposed optimization techniques. Namegen and our optimizations are implemented with around 2.6K lines of C/C++ and CUDA. We use OpenMPI [5] to connect inter-node processes and OpenMP [2] to spawn threads for creating one-to-one mappings to the intra-node GPUs.

Environment. We run our evaluation using a server of 4 nodes, each equipped with 4 NVIDIA V100 32GB GPUs connected over PCIe. Each node has 1 Mellanox MT28908 family ConnectX-6 NIC and is interconnected with NVIDIA QUANTUM HDR Switch QM8700, providing an 200Gb/s of interconnect bandwidth between the nodes.

4.1 Matrix Multiplication Performance

We demonstrate the relative performance of our step-by-step optimization of matrix multiplication kernel and compare it to NVIDIA cuBLAS [7] performance using a matrix multiplication of $65536 \times$

	Kernel	GFLOPS	Perf.
1	Naive	214.5	1.5%
2	GMEM coalescing	2084.1	14.8%
3	Shared MEM	3809.3	27.2%
4	1D Blocktiling (naive)	4950.2	35.3%
5	1D Blocktiling (row)	5005.3	35.7%
6	1D Blocktiling (row, vector type)	5093.1	36.3%
7	1D Blocktiling (column)	5754.0	41.1%
8	2D Blocktiling	10440.7	74.5%
9	2D Blocktiling + Loop unrolling	11344.2	80.9%
0	cuBLAS	14015.0	100.0%

Table 1: Matrix multiplication kernel performance.

4096 and 4096×4096 matrices. Our final version of the kernel implements 2D blocktiling that optimizes load/store GMEM accesses and shared memory (SMEM) accesses, achieving 80.9% of cuBLAS peak FLOPS.

4.2 End-to-end Performance

The inference throughput of our optimization is 580201 names/sec, which outperforms the naive Namegen by 430×. To demonstrate the scalability of the optimized Namegen, we profile the throughput as we scale both the number of nodes and the number of GPUs per node to 1×2, 1×4, 2×4, to 4×4, utilizing the entire evaluation environment. Optimized Namegen shows 89.7% scaling efficiency, achieving near-linear scalability.

5. Conclusion

We analyze the architecture and the inference procedure of generative models and implement three optimization techniques, batching, matrix multiplication kernel optimization, and kernel fusion to improve the latency and scalability. We evaluate our optimization using Namegen on a distributed multi-GPU environment and show that our optimization techniques can exploit the hardware capabilities and achieve improved latency and scalability. We open-source our implementation of the optimizations and its application to Namegen at `https://github.com/gajagajago/fastgen`.

6. Acknowledgments

This work was supported in part by the Institute for Information communications Technology Promotion (IITP) grant (No. 2018-0-00581, CUDA Programming Environment for FPGA Clusters) and by the BK21 Plus program for BK21 FOUR Intelligence Computing (Dept. of Computer Science and Engineering, SNU, No. 4199990214639) through National Research Foundation of Korea (NRF), all funded by the Ministry of Science and ICT (MSIT) of Korea. ICT at Seoul National University provided research facilities for this study.

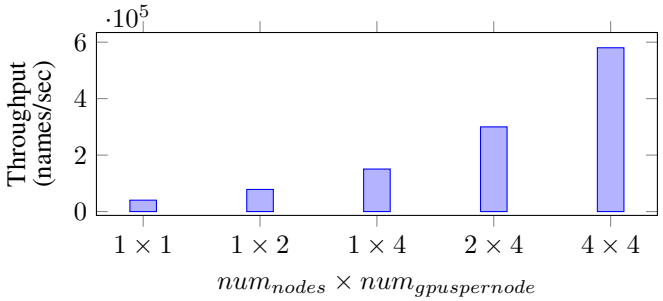

Figure 5: Namegen performance.

References

[1] A. Ali, R. Pinciroli, F. Yan, and E. Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[2] R. Chandra, L. Dagum, R. Menon, D. Kohr, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[3] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[4] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, volume 17, pages 613–627, 2017.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[6] Github. Github copilot. `https://github.com/features/copilot`.

[7] NVIDIA. cublas. `https://developer.nvidia.com/cublas/`.

[8] NVIDIA. Nsight systems. `https://docs.nvidia.com/nsight-systems/index.html`.

[9] OpenAI. Chatgpt. `https://chat.openai.com/chat`.

[10] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 2022.

[11] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350. IEEE, 2010.

[12] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[13] C. Zhang, M. Yu, W. Wang, and F. Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *USENIX Annual Technical Conference*, pages 1049–1062, 2019.