CUrator: An Efficient LLM Execution Engine with Optimized Integration of CUDA Libraries

Yoon Noh Lee
Yonsei University
Seoul, Republic of Korea
yoonnohlee@yonsei.ac.kr

Yongseung Yu
Yonsei University
Seoul, Republic of Korea
dydtmd1991@yonsei.ac.kr

Yongjun Park
Yonsei University
Seoul, Republic of Korea
yongjunpark@yonsei.ac.kr

Abstract

Large Language Models (LLMs) have recently emerged as a state-of-the-art learning model with a wide range of applications in diverse computing environments. Among the various computational operations that comprise the LLM, the GEneral Matrix Multiplication (GEMM) operation is the most frequently utilized operation within the LLM. GEMM libraries such as cuBLAS and CUTLASS provide a variety of optimization techniques to achieve optimal GEMM performance in GPU-enabled computing environments. In particular, the CUTLASS open-source library for GPUs within the CUDA programming environment provides users with the capability to optimize templates for high performance. Previous research has demonstrated the effectiveness of CUTLASS-based GEMMs in improving the performance of real-world deep neural networks on various deep learning platforms. However, these studies have not considered different model parameters for modern LLMs nor have they explored the impact of diverse GPU computing environments.

This paper presents **CUrator**, an efficient LLM execution engine that can achieve optimal end-to-end LLM performance using both cuBLAS and CUTLASS libraries on different GPUs for modern LLMs such as BERT, GPT, and Llama. CUrator first generates CUTLASS-/cuBLAS-friendly graph IRs of various LLMs on the TVM framework to maximize mapping coverage. On the CUTLASS mapping path, it performs a comprehensive search for programmable tuning parameters in the CUTLASS library with the objective of deriving optimal kernels for all GEMMs within each LLM. CUrator further introduces two optimization techniques: 1) build-time reduction key initialization support for CUTLASS Split-K GEMMs, and 2) Split-K support for CUTLASS Batch GEMMs. Finally, CUrator selects the best performing mapping path between cuBLAS and CUTLASS paths. The experimental results show that CUrator achieves inference speedups of 1.50× and 4.99×, respectively, for representative LLMs on the A100 GPU in the single and half precision, compared to the baseline. We strongly believe that the CUrator framework can provide the best direction for next-generation tuning frameworks by showing the maximum end-to-end performance of various LLMs on various GPUs.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: Large Language Model, GPU, GEMM, Compiler

ACM Reference Format:
Yoon Noh Lee, Yongseung Yu, and Yongjun Park. 2025. CUrator: An Efficient LLM Execution Engine with Optimized Integration of CUDA Libraries. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3696443.3708944

1 Introduction

In the field of deep learning, modern Large Language Models (LLMs) [34] have recently become one of the most prominent models, showing the highest quality in a variety of AI computing applications. To ensure the effective operation of LLMs on a wide range of hardware for both training and inference computations, researchers have developed various software techniques. To adequately describe and execute models, several deep learning platforms have been introduced, such as ONNX [8], PyTorch [33], and TVM [11]. To execute the deep learning models on the frameworks, GPUs are widely used, and included new computing cores or data layouts to maximize the performance of deep learning applications. To successfully achieve the excellent performance of various computations required for deep learning applications on GPUs, various SW-level libraries such as cuBLAS [1], cuDNN [14] and CUTLASS [25] have been developed. These libraries have enabled deep learning developers to effectively use GPU resources such as tensor cores and shared memory.

CUTLASS is an open-source template-based BLAS library that optimizes the GEneral Matrix Multiplication (GEMM) operation, which is one of the most commonly used operations in LLMs. CUTLASS provides several customizable features such as tiling configuration, optimization factors, and memory layout. Adjusting the features of CUTLASS directly affects the core and memory resource utilization of the target GPU, which has a significant impact on performance.

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708944

Unlike cuBLAS, CUTLASS allows users to specify options such as tile size and shared memory usage. Therefore, composing LLM computations based on well-tuned CUTLASS GEMM kernels in GPU-based computing environments can achieve excellent LLM performance.

Several previous studies have been conducted to use TVM to optimize deep learning computations [43, 50]. Zheng et al. [50] have developed a system called *Ansor* that derives the optimal values of GEMM features through evolutionary search, which reduces the huge search space that needs to be explored to find the optimal values of various features of TVM operators. Xing et al. [43] have pointed out that it is difficult for SW-only auto-tuners to achieve the performance of closed-source libraries provided by hardware vendors, and developed a framework called *BOLT* to find the optimal values of CUTLASS parameters by considering the underlying hardware characteristics as much as possible. In previous studies, researchers have tried to maximize the performance of the GEMM by tuning the features through various approaches and constructing an end-to-end optimization framework. However, there are still some remaining practical limitations, and tighter integration of deep learning frameworks and vendor-provided GPU libraries is required.

First, many deep learning frameworks translate target LLMs into layer-computation graphs for efficient execution of the models utilizing the CUDA libraries. However, the mapping coverage of kernels from the libraries over all possible layers is still low because the graph node structure is often not perfectly matched to that required by the libraries. For example, cuBLAS kernels get inputs in the variable data type, but TVM nodes contain several inputs in the constant data type, thus the cuBLAS kernels cannot be mapped to run the nodes. Second, the initialization processes of multiple kernels can be the main cause of performance degradation. In vendor-provided library kernels, several input-dependant initialization processes are required, such as split-k key allocation, and these processes often incur high performance overhead because they are typically simple but time-consuming processes such as dynamic memory allocation. However, since the input parameters of mapping kernels in model computation graphs can be known at build time, the time-consuming initialization processes can be performed once at build time and the runtime overhead can be eliminated. Last, it is difficult to estimate the full potential of using the libraries because each kernel's performance varies significantly depending on various system-related parameters such as tile sizes and split-k factors. While many previous works provide their own intelligent parameter-tuning solutions, extensive profiling to find the best parameter setting is still important to estimate the maximum performance gain for each LLM. In addition, the CUDA library should also be modified to efficiently support LLM-specific operations.

To address the limitations of previous research and to reveal the maximum performance gains when using CUTLASS/cuBLAS libraries, we propose an efficient end-to-end LLM execution framework, named **CUrator**, which integrates CUTLASS and cuBLAS GEMM kernels into the TVM deep learning compiler framework with various optimization techniques to achieve optimal computation performance for various LLMs. CUrator first constructs CUTLASS-/cuBLAS-friendly computation graphs from various LLMs on the TVM to maximize kernel mapping coverage. On the CUTLASS backend, CUrator performs an extensive search for kernel-specific parameters in the CUTLASS library to find the optimal kernels for all mapping layers in the target LLMs. CUrator also applies two key optimizations: 1) build-time reduction key initialization support for CUTLASS split-k GEMMs, and 2) split-k support for CUTLASS Batch GEMMs. Based on the performance evaluation of both CUTLASS and cuBLAS backends, CUrator finally achieves high performance model execution for various LLMs such as BERT [16, 39], GPT2 [34], and Llama [5, 17] by selecting the best mapping path.

Evaluation using CUrator demonstrates that the inference performance of LLMs at different scales on different GPU computing environments, utilizing CUTLASS GEMM and cuBLAS GEMM with the optimal optimization factors, can be significantly improved compared to the state-of-the-art solutions. Based on results on an A100 GPU, CUrator achieved average performance gains of 1.50× and 4.99× in single and half precision, respectively, over the Ansor baseline.

These findings based on CUrator can provide invaluable guidance for contemporary LLM optimizations on different GPU platforms. To provide the usability of our CUrator, we will disclose our framework as an open-release version.

This paper presents the following contributions.

- Development of a framework for extensive search of CUTLASS kernel configurations, including tile setting, split-K factor, and threadblock swizzling factor.
- Introduction of key optimizations for CUTLASS-based kernels: 1) support for batched split-k GEMM kernel and 2) support for build-time reduction key initialization for split-k GEMM kernels.
- End-to-end optimization of TVM IR-based LLMs with graph rewriting for higher kernel mapping coverage.
- Extensive evaluation of the CUrator framework across multiple LLMs on various GPU generations.
- Use of well-known auto-tuning frameworks, including ML-based tuning and OpenTuner frameworks, to demonstrate the potential effectiveness of introducing advanced auto-tuners.

2 Background and Motivation

2.1 CUTLASS

CUTLASS provides open-source template-based code for developers to optimize the operations for their target GPUs

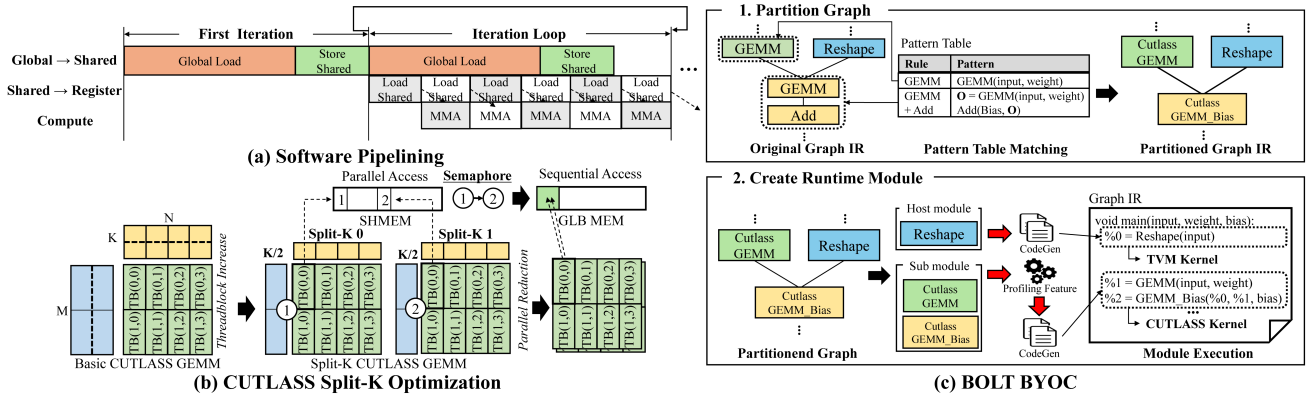

Figure 1. (a)-(b): An overview of the CUTLASS GEMM template, (c): BOLT [43] overview.

and target applications. Therefore, developers can apply various optimizations of CUTLASS to be well-matched to their environments. Basically, CUTLASS supports a wide range of tiling parameters with up to 2-levels of hierarchy and compilation options such as split-K optimization and threadblock swizzle. In addition, CUTLASS exploits the software pipelining [40] mechanism to efficiently hide the latencies of computation and memory operations. Figures 1(a)-1(b) illustrate the software pipelining technique and split-K optimization of the CUTLASS library.

2.1.1 Tiling Configuration.

The CUTLASS GEMM kernel divides the entire GEMM output into threadblock-tiles (TBTs), and each divided threadblock-tile is further divided into several warp-tiles (WTs). Each tile is 3-dimensional, and the size and number of threadblocks and warps are determined from the tile configuration. In particular, changing the size and shape of each tile has a significant impact on GEMM performance, as it changes the resource utilization of the target GPU.

2.1.2 Software Pipelining.

CUTLASS GEMM templates use the software pipelining technique as a basic kernel optimization. According to Figure 1(a), the software pipeline used in CUTLASS GEMM consists of three streams based on various resources of the GPU. The three streams are composed of the first stream from global memory to shared memory located within the core cluster of the GPU, the second stream from shared memory to register files, and the last computation stream. CUTLASS GEMM separates the workflow of each thread into global-shared, shared-register, and computation stages to maximize resource utilization by ensuring all pipelined streams are active.

2.1.3 Split-K Optimization.

Figure 1(b) shows a simple description of the split-K optimization of CUTLASS GEMM. As illustrated in Figure 1(b), the K dimension of the GEMM is divided by the split-K factor when split-K optimization is applied. The split-K factor represents the number by which

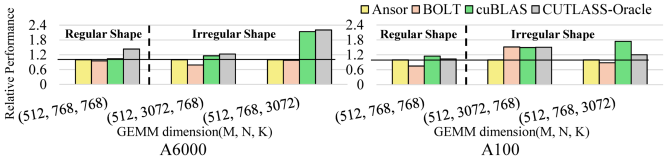

Figure 2. A comparison of the GEMM kernel performance in the single-precision datatype between previous works (Ansor, BOLT frameworks), cuBLAS, and CUTLASS-Oracle (optimal performance of CUTLASS GEMM).

the K dimension of the GEMM is divided, and the number of threadblock tiles is increased by a multiple of the factor value. Once the GEMM computation of all thread block tiles is complete, CUTLASS initiates a parallel reduction kernel to perform reduction operations on the divided threadblock tiles, thereby obtaining the final GEMM result. The split-K optimization technique allows the user to adjust the number of threadblock tiles to control the utilization of the GPU's computing core resources.

In addition, when applying the split-K optimization, CUTLASS performs semaphore-based data movement control. Through the semaphore mechanism, CUTLASS regulates the order of memory access operations from different threadblock tiles targeting the same memory space. Figure 1(b) shows that threadblock-tiles 1 and 2 store the results of GEMM operations in different shared memory areas and access the same global memory area when performing parallel reduction operations. In this case, CUTLASS GEMM uses the semaphore mechanism to prevent the global memory access of threadblock tile 2 until the access of threadblock tile 1 has been finished. The CUTLASS split-K GEMM coordinates the order of global memory accesses of parallel threadblock tiles by the aforementioned process to prevent the race condition.

2.2 TVM BYOC Example

TVM BYOC [13] is a part of the backend infrastructure of TVM, which allows users to transform the TVM graph IR

into the user-desirable graph IR. Figure 1(c) illustrates a simple example of the *BOLT* [43] framework, which uses TVM BYOC to modify the TVM graph IR to use CUTLASS GEMM kernels and to tune the CUTLASS GEMMs. As shown in Figure 1(c), TVM BYOC replaces each node in the input graph IR with the user-desired graph IR based on the user-defined pattern table. Furthermore, TVM BYOC allows the use of external third-party libraries to replace nodes within the input graph. Consequently, TVM BYOC allows nodes of input TVM graph IRs to be converted to third-party libraries, such as cuBLAS and CUTLASS, and the profiling of the computational kernels of the converted graph nodes.

2.3 Limitations of Prior Research and Motivation

There have been several attempts to optimize deep learning models on TVM [43, 50]. The developers of the framework *Ansor* [50] profiled a subset of the search space and trained the cost model to predict the performance of the entire search space to find the best tiling setting. The *BOLT* [43] framework utilizes the GEMM and batched GEMM operations from the CUTLASS library and simply tunes each GEMM operation. Figure 2 presents a comparison of FP32 GEMM kernel performance among *Ansor*, *BOLT*, cuBLAS and CUTLASS-Oracle (optimal performance of the CUTLASS GEMM) on both regular and irregular shaped GEMMs within the BERT-large model with the sequence length of 512 on A6000 and A100 GPUs. Each performance is normalized to the performance of the *Ansor* framework. As illustrated in Figure 2, with the exception of the GEMM [512, 768, 768] operation on the A6000 GPU, the *Ansor* framework demonstrates lower performance than cuBLAS on both the A6000 and A100 GPUs. In the case of the other recent work, the *BOLT* framework, shows a similar performance to that of CUTLASS-Oracle for the GEMM [512, 3072, 768] on the A100 GPU, however, in the remaining GEMMs, *BOLT* shows a lower performance than that of CUTLASS-Oracle. Because the *Ansor* framework does not consider the hardware resources of the target GPU to the same extent as cuBLAS, cuBLAS achieves better results compared to the *Ansor*. Also, because the *BOLT* framework has a narrow search space for CUTLASS GEMM tuning, the *BOLT* usually misses the optimal tuning parameter for the target GEMM. Based on the results indicated in Figure 2, it is obvious that there is a significant potential for improving GEMM performance through the use of a highly tuned CUTLASS GEMM kernel or cuBLAS GEMM kernel depending on the target GPU and target GEMM shape.

Even putting aside the question of expected performance, there are some limitations that have not been addressed in previous studies. First, previous research has not focused on the end-to-end LLM execution as a primary target workload. The LLM is one of the most intensively researched deep learning models in recent years. Therefore, it is essential to consider the end-to-end performance of the LLM. Next, it is necessary to avoid unwanted memory allocation overhead during LLM execution. It is possible for redundant runtime memory allocation to occur when a well-tuned CUTLASS split-K GEMM is being performed multiple times on the TVM framework. It is therefore important to reduce the unnecessary waste of memory allocation time at inference time. Finally, to achieve the greatest possible utilization of high-end GPU resources, it is important to implement efficient GEMM computation across the various GEMMs. Therefore, fine-tuning of the CUTLASS GEMM kernel is required to fully exploit the hardware resources of various GPUs.

Considering the above limitations, we present **CUrator**, a novel end-to-end LLM execution engine that can execute modern LLMs in various GPU computing environments. CUrator modifies the LLM graph based on the TVM IR into CUTLASS/cuBLAS friendly IR, efficiently compiles GEMM kernels constituting the target LLM, finds the best compilation parameters based on full search, and applies the best parameters for the end-to-end LLM execution with maximum performance. During LLM compilation, CUrator considers the allocation of memory resources for GEMMs not to harm the overall LLM performance. Furthermore, CUrator employs the batched split-K CUTLASS GEMM to enable a more comprehensive utilization of the target GPU cores.

3 CUrator: An Efficient LLM Execution Engine Using Multiple CUDA Libraries

The CUrator framework consists of two main components: 1) Graph Re-writer and 2) Runtime Engine. The Graph Re-writer transforms the input ONNX graph of a target LLM into CUTLASS- and cuBLAS-enabled TVM graphs by detecting graph node patterns. The Runtime Engine then compiles each modified graph into executable binaries via either BOLT or TVM BYOC backends. During the compilation of the CUTLASS module, CUrator performs a brute-force search to find the optimal tiling configuration for the target LLM and GPU. According to Figure 3(a), CUrator first converts a target LLM stored in PyTorch format into the TVM Graph IR by ONNX graph transformation. The graph IR generated from ONNX stores weight and bias data in a `constant` format. Since cuBLAS kernels only receive input data of the `variable` type, the weight and bias data formats of cuBLAS do not match with the generated graph IR. Therefore, at the frontend stage of CUrator, the `constant` data format is modified into a `variable` data format to enable the cuBLAS BYOC in TVM. Then, as shown in Figure 3(a), TVM BYOC partitions the graph IR with a predefined pattern table. At the graph partitioning stage, CUrator registers CUTLASS and cuBLAS pattern tables that match with the graph IR generated from ONNX. In order for the CUrator framework to take advantage of advanced GEMM operation fusion techniques for LLMs, several patterns for the Fused Multi-Head Attention (FMHA) technique have also been integrated into the CUrator pattern table. As shown in the CUrator pattern table of

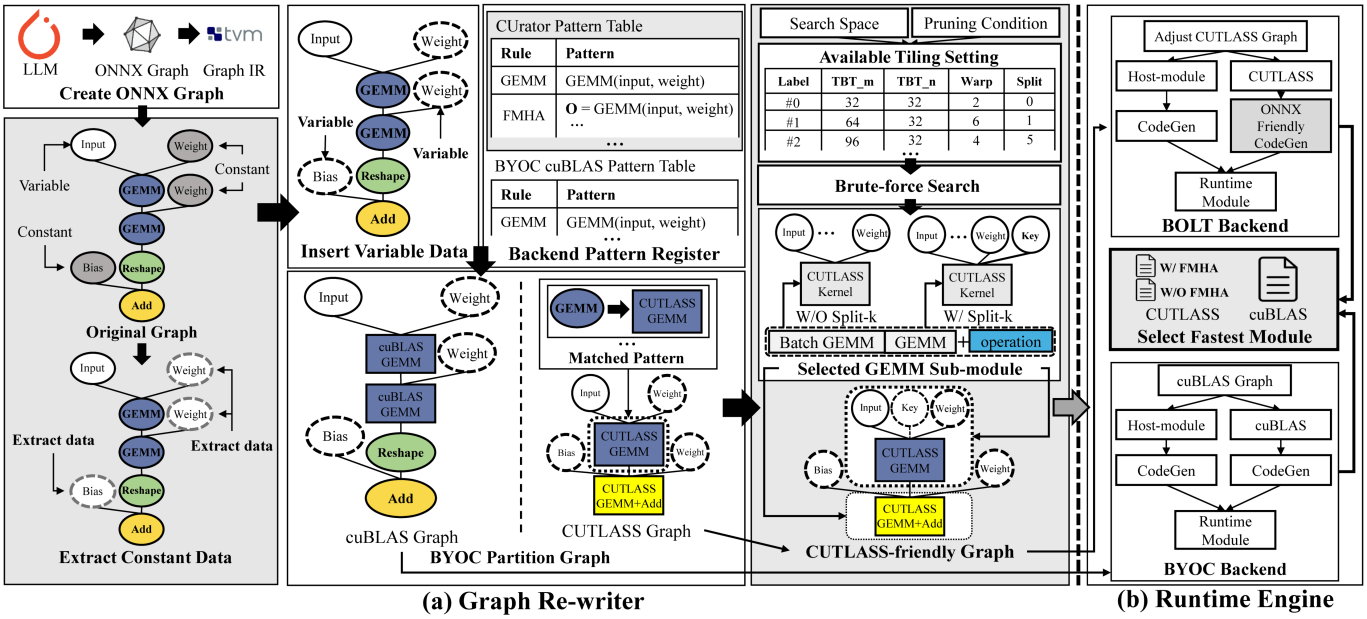

Figure 3. An overview of the CUrator framework.

Figure 3(a), CUrator can find patterns suitable for FMHA at the graph re-writing stage.

Following the CUTLASS and cuBLAS tables stored in the backend pattern register, CUrator merges multiple operation nodes of the graph IR into a single node without violating the ONNX graph IR rules. Figure 3(a) illustrates the search phase for the optimal tiling parameter for CUTLASS GEMM templates. In the search phase, CUrator lists the available tiling settings and exhaustively searches for the best tiling parameter. Finally, at the runtime module build stage in Figure 3(b), CUrator runs GEMMs from both CUTLASS and cuBLAS and derives the better performing kernels. During the search phase, CUrator also includes the CUTLASS split-K enabled batch GEMM template to fully exploit the resources of the target GPU. In addition, CUrator modifies the semaphore key allocation time from inference time to build time for the CUTLASS split-K GEMM template to reduce the end-to-end inference time.

3.1 Efficient TVM Integration of CUDA Libraries

To maximize the mapping coverage and efficiency of each mapped kernel, CUrator converts the generic TVM graph IRs into both cuBLAS/CUTLASS friendly graph IRs to realize the full potential of both cuBLAS and CUTLASS libraries.

3.1.1 Construction of cuBLAS-Enabled Graph IR. Figure 3(a) mainly illustrates the modification of the data formats of the generic TVM IR graph. As previously discussed, since the data format of the graph IR from TVM BYOC differs from that of the cuBLAS, CUrator translates the cuBLAS parameter syntax to align with the TVM BYOC data format. The transformation of each data format consists of the following steps. First, CUrator traverses the input graph IR and extracts all data information if the data format is `constant`. In order to preserve the integrity of the `constant` data, all extracted data is stored in a separate memory space. The `constant`-typed data in the target graph IR is then replaced with the `variable`-typed data. After that process, CUrator generates the modified graph IR according to the BYOC cuBLAS table as illustrated in Figure 3(a).

3.1.2 Best CUTLASS Tiling Setting Exploration Support. In order to exploit the operation fusion supported by the CUTLASS library, CUrator first merges several graph IR nodes into a fused GEMM operation based on the patterns described in the CUTLASS table of Figure 3(a), before searching for the best tiling parameter. CUrator performs the operation fusion of a GEMM operation and its epilogue function, and thus several different fused kernels can be generated to have the same GEMM operation and different epilogue functions. After modifying the graph IR for the CUTLASS GEMM template, CUrator performs a brute-force search to find the best tiling setting for the target fused GEMM kernels by considering only a single main GEMM operation. This is because the optimal tile settings for the fused GEMM kernels are the same as the optimal tile setting of their main GEMM operation, because 1) each epilogue function is an element-wise operation, and 2) its execution time is significantly smaller than that of the main GEMM.

Figure 4 provides an overview of the exploration process for finding the optimal tiling setting for the CUTLASS GEMM kernel. As shown in Figures 4(a) and (b), this process consists

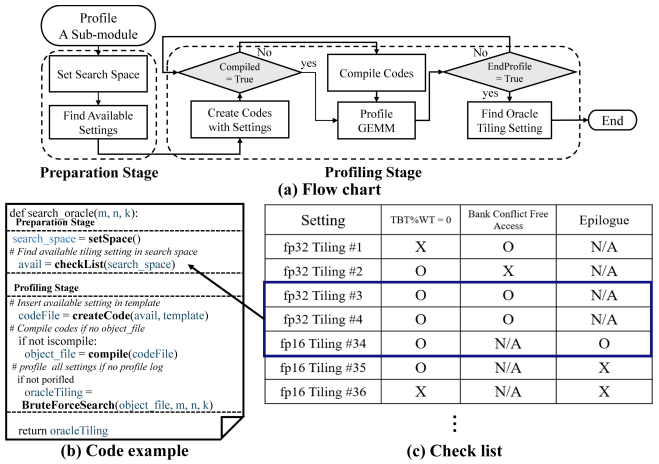

Figure 4. CUTLASS oracle tiling setting exploration process.

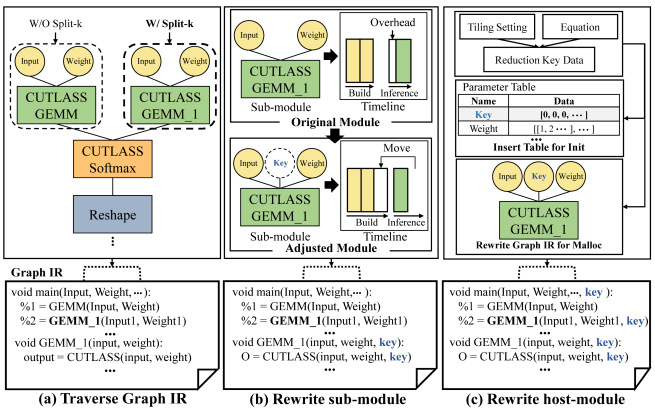

Figure 5. CUTLASS-friendly graph IR modification: (a) Graph IR traversal to identify CUTLASS GEMMs, (b)-(c): Module rewriting processes for efficient use of the split-K algorithm.

of two stages: the Preparation Stage and the Profiling Stage. In the Preparation Stage, CUrator collects the available tiling settings for the target CUTLASS GEMMs based on the search space description and the pruning conditions. In the Profiling Stage, CUrator finds the optimal tiling setting that performs the GEMM fastest among the available tiling settings.

Preparation Stage In the Preparation Stage, CUrator identifies the available CUTLASS tiling settings for the target GPU and target precision. The availability of CUTLASS tiling parameters is different depending on the specifications of the GPU hardware and the constraints of the GPU compiler. Therefore, CUrator first determines feasible tiling parameter values and enumerates available CUTLASS tiling parameter combinations by considering static assertions and CUDA-specific constraints according to the target GPU architecture and target precision. Based on the detailed checklist illustrated in Figure 4(c), CUrator verifies the available tiling settings in the search space and sends them to the Profiling Stage. Note that some configurations may not be executable depending on the resource constraints of the target GPU, such as the number of threads per threadblock or dynamic shared memory usage, and these are also excluded.

Profiling Stage In the Profiling Stage, CUrator finds the optimal tiling parameters from the available tiling parameters provided by the Preparation Stage. The available tiling parameters are first inserted as input parameters for the CUTLASS GEMM kernel and then compiled as an object file. Since the kernel compilation for every available tiling parameter incurs a high overhead, CUrator reuses previously compiled GEMM binaries when profiling is required. To minimize the overhead incurred during the compilation process, CUrator employs the non-fused version of the GEMM nodes instead of the fused version, when searching for the optimal tiling parameter. Finally, CUrator executes the compiled binaries and records the performance results in a text file.

3.1.3 CUTLASS-Friendly Graph IR Modification.

End-to-end model inference can be performed after a one-time model build (compilation) process. At build time, Curator loads parameters into GPU memory and prepares kernels to be executed during inference. For the inference process, CUrator executes the kernels based on the graph information in the runtime module. Figure 5 represents the modification process of the input graph IR into the CUTLASS-friendly IR. As shown in Figure 5(a), in the original graph IR, a detailed implementation scheme for the CUTLASS split-K GEMM, such as the allocation time of the reduction key, is not considered, therefore an unnecessary memory allocation is performed at the LLM inference time instead of allocating it at the build time. In the LLM with large hidden layers (K dimension of GEMMs), it is necessary to allocate numerous reduction keys at inference time to fully utilize all SMs. Consequently, CUrator relocates the reduction key allocation and initialization process in the CUTLASS split-K GEMM from inference time to build time, allowing the split-K algorithm to be used without runtime overhead for initializing the reduction key at inference time.

For clarification, the CUTLASS algorithm employs the semaphore algorithm in the split-K implementation to prevent race conditions when multiple thread blocks attempt to store data in global memory at the same location. The semaphore algorithm allows only a thread block that has acquired the key to access data in global memory while preventing accesses from other thread blocks. The CUTLASS split-K GEMM creates multiple threadblocks for each output matrix tile based on the split-K parameter, and the threadblocks corresponding to other output matrix tiles can write results to global memory concurrently. Therefore, the number of keys is equal to the number of output matrix tiles, and it is determined by the TBT size ($\#key = (output_m/TBT_m) \times (output_n/TBT_n)$). Allocating and initializing the space for

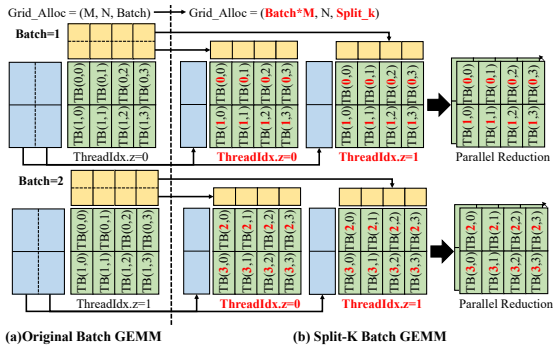

Figure 6. Split-K support on batch GEMM: (a): original batch GEMM, (b): modified batch GEMM with split-K support.

the semaphore keys at inference time incurs latency overhead. To reduce the reduction key allocation overhead at inference time, the size of the TBT should be increased to reduce the number of threadblocks allocated in the output matrix tile. Increasing the size of the TBT reduces the memory allocation and initialization overhead of the keys at inference time, but may reduce efficiency in threadblocks. Allocating and initializing the reduction key at build time can eliminate the latency overhead at inference time and maximize efficiency in threadblocks. Therefore, CUrator modifies the original Graph IR to a split-K friendly Graph IR to help the CUTLASS kernel find the best tiling setting without the reduction key allocation overhead of split-K GEMMs.

Figure 5 demonstrates the modification process of CUrator from the original graph IR to the CUTLASS-friendly graph IR. CUrator first traverses the partitioned Graph IR(Figure 5(a)) to find sub-modules using CUTLASS as a backend. CUrator then checks that the oracle tiling setting of the sub-module uses the split-K algorithm. Second, for sub-modules where the best tiling setting is to use the split-K algorithm, CUrator modifies the partitioned graph IR as shown in Figure 5(b)-Figure 5(c). According to Figure 5(b), CUrator adds the reduction key to the sub-module as an input parameter to reduce the allocation and initialization overheads of the reduction key in the inference time.

By registering reduction keys in the input table and adding the parameter to the graph IR in the host-module, the reduction key can be allocated and initialized at build time (Figure 5(c)). Consequently, modifying the graph IR results in a reduction in memory allocation overhead at inference time and an increase in the efficiency of threadblocks.

3.2 Split-K Support on Batch GEMM

In general, when utilizing a GPU to execute LLM operations, the size of the GEMMs is generally sufficient to occupy the GPU's compute cores. However, as a result of the enhanced processing capabilities of GPUs, the execution of LLM computations on modern GPUs often results in underutilization of GPU cores due to insufficient target GEMM size. In order

Table 1. System configuration

	V100[24]	RTX3090[27]	A6000[28]	A100[26]	RTX4090[30]
TVM[11]	0.12.0				
LLVM[21]	10.0.0				
CUDA[31]	11.4.4		12.0.0		
Pytorch[33]	1.12.1		2.1.2		
cuDNN[14]	8.9.7		8.9.5		
CUTLASS[25]	3.0.0 (w/o FMHA) or 3.5.1 (w/ FMHA)				
TensorRT-LLM[15]	0.11.0				

Table 2. Ratio of profiled GEMMs on LLMs [5, 16, 17, 34, 39]

Model	Profiled/Total	Model	Profiled/Total
BERT-tiny	6/17	BERT-mini	6/33
BERT-small	6/33	BERT-medium	6/65
BERT-base	6/97	BERT-large	6/193
GPT2	6/72	GPT2-medium	6/144
OpenLlama-3B	8/235	MetaLlama3-8B	8/289

to address this issue, CUrator exploits a batch GEMM integrated with the split-K mechanism to compensate for low core occupancy on the GPU during LLM operations. GEMM batching and split-K techniques are common optimization strategies, but the batched GEMM kernel with split-K optimization is not yet supported as a regular template in the latest version of the CUTLASS library.

With the basic split-K GEMM template, multiple threadblocks are assigned to the same output matrix tile. Therefore, three-dimensional threadblocks are assigned to a two-dimensional output matrix tile. Each output matrix tile is allocated to threadblocks with the same *blockIdx.x* and *blockIdx.y* but different *blockIdx.z*. Calculated data from multiple threadblocks with the same *blockIdx.x* and *blockIdx.y* but different *blockIdx.z* are accumulated sequentially in global memory using the semaphore algorithm after performing MMA operations in parallel.

As shown in Figure 6(a), the original Batch GEMM allocates threadblocks in three dimensions in the output matrix using the *blockIdx.z* index as the batch index, making it difficult to use the *blockIdx.z* index as the semaphore key index. For this reason, it is difficult to apply the semaphore algorithm directly to batch GEMM in the same way as the semaphore algorithm used in Basic GEMM. Therefore, CUrator applies the split-K algorithm to Batch GEMM by changing the threadblock organization by modifying the CUTLASS template. As shown in Figure 6(b), the three-dimensional threadblock structure in the original Batch GEMM is converted to a two-dimensional structure by updating *blockIdx.x* to further consider the batch index. By allocating threadblocks in only two dimensions, the semaphore algorithm of the Basic GEMM can be applied to the Batch GEMM so that it can use the *blockIdx.z* index as the semaphore key index.

4 Evaluation

The CUrator derives the maximum potential performance gain of GEMM operations for various target LLMs on a wide

range of GPU computing environments. Typically, LLMs include GEMM as well as important computational kernels such as GEMV and Softmax. However, in a GPU computing environment, the time required for GEMM operations of several LLMs is reported to be more than 70% of the total processing time [49]. The time required to derive the first token, referred to as the *Time To First Token (TTFT)*, has also become an important metric in evaluating the performance of recent LLMs [2, 3, 53]. We therefore focus on optimizing the performance of GEMM operations, given the high proportion of execution time and the importance of TTFT.

4.1 Evaluation Setup

System Configuration The CUrator was implemented based on the BOLT [43] on top of TVM BYOC [13] infrastructure, including the CUTLASS [25] and cuBLAS [1] libraries. The CUrator was evaluated on five different GPUs: V100 [24], RTX3090 [27], A6000 [28], RTX4090 [30], and A100 [26]. The system configuration for the evaluation of CUrator is described in Table 1. When compiling the cuBLAS runtime module, CUrator employed the cuDNN [14] library in order to utilize the Softmax kernel. For the compilation of CUTLASS runtime module, CUrator used a custom tuning-enabled Softmax kernel based on the OneFlow [46] framework. Note that we updated the BOLT profiling infrastructure which currently has a narrow search space, into the newer version to support optimal CUTLASS tiling parameter exploration, allowing the sub-modules to be mapped to best-fit kernels with optimal tiling parameters. By leveraging the BOLT's caching capabilities that store the best tiling setting for each GEMM, we ensure that GEMMs that have previously been profiled are not profiled again. Therefore, we only profiled the subset of GEMMs from each model, as detailed in Table 2, rather than profiling all of them. We integrated the cuBLAS and CUTLASS runtime modules into the CUrator. In terms of compilation time, it takes 13 hours to complete an extensive search of the GEMM subset for a BERT-base model (batch size 8, sequence length 512) on the RTX4090. As a case study, we employed multiple auto-tuners to determine the optimal tiling settings, aiming to reduce the time required for exhaustive searches (see Section 4.4).

Baselines and CUrator Versions We compared the CUrator with several other deep learning compiler frameworks, such as Ansor [50], BOLT [43] and TensorRT-LLM [15]. For CUTLASS-Oracle, we focused on identifying the maximum potential performance gain of the CUTLASS GEMM kernel. Therefore, an operation fusion for the TVM graph was applied to the CUrator at the same level as in the previous frameworks [43, 50]. For the BOLT framework, we used the default split-K parameter for all GEMM kernels, since the possibility of varying the split-K parameters according to each GEMM kernel was not considered. For the Ansor framework, we set the number of total tuning trials to 900, following the official example of the Ansor evaluation. We also evaluated all LLMs on the TensorRT-LLM [15] framework. For a fair comparison, we applied all optimizations on TensorRT-LLM except the precision downscaling technique. As explained in Section 3, the Fused Multi-Head Attention (FMHA) technique was also used in the ONNX graph rewriting process of CUrator. Therefore, we evaluated the end-to-end inference performance for each model both without and with the FMHA technique applied in CUrator (CUrator w/o FMHA and CUrator w/ FMHA).

DataSet To perform an evaluation, we obtained target LLMs from HuggingFace [41] and imported them into PyTorch [33] to transform the models into ONNX [8] graph IRs. The transformed ONNX graph IRs are then converted to the Relay graph IR in TVM. To evaluate the end-to-end performance of LLM on each available GPU computing core, both single-precision LLM for CUDA cores and half-precision LLM for tensor cores were evaluated. Half-precision LLM models were constructed by converting the data type of the model parameters from single to half precision using TVM APIs. We evaluated three LLMs, including BERT [16, 39] models (mini, tiny, small, medium, base, and large), GPT-2 [34] models (default, and medium), and Llama models (openLlama-3B [17], MetaLlama3-8B [5]). [1]. In the evaluation, the sequence length of the LLMs was set to 512, with reference to other research projects [6, 9, 10, 32, 38, 42, 47, 52]. Furthermore, we evaluated the end-to-end LLM performance with batch sizes of 1, 4, and 8, to illustrate the effectiveness of CUrator over a wide range of GEMM dimensions.

4.2 End-to-End Inference Performance Evaluation

4.2.1 Single Precision. Figure 7 shows the inference performance of the models in single-precision. Each end-to-end inference performance is normalized to the performance of the Ansor framework. In addition, the absence of bars in the figure indicates that the model inference was terminated due to an out-of-memory error, or that the model is currently unavailable for use with the framework. Each table at the bottom of each graph in Figures 7(a)-7(e) represents the absolute latency of end-to-end model inference for Ansor (baseline), TensorRT-LLM, and CUrator w/ FMHA. As shown in Figure 7, CUrator shows the best performance because CUTLASS-Oracle outperforms the previous frameworks in most cases, and cuBLAS sometimes shows the best performance. More specifically, a comparison of geomean performance across all evaluated models for each batch size indicates that CUTLASS-Oracle outperforms cuBLAS in most cases, except for batches of 4 and 8 on the V100 GPU. CUrator can show further performance gains when applying the FMHA technique. According to Figure 7, evaluations on diverse GPU environments with various LLMs show that

[1] Since TensorRT-LLM does not yet support the specific LLM configuration that is predominantly used in openLlama-3B, we have not evaluated the openLlama-3B model with the TensorRT-LLM framework [15].

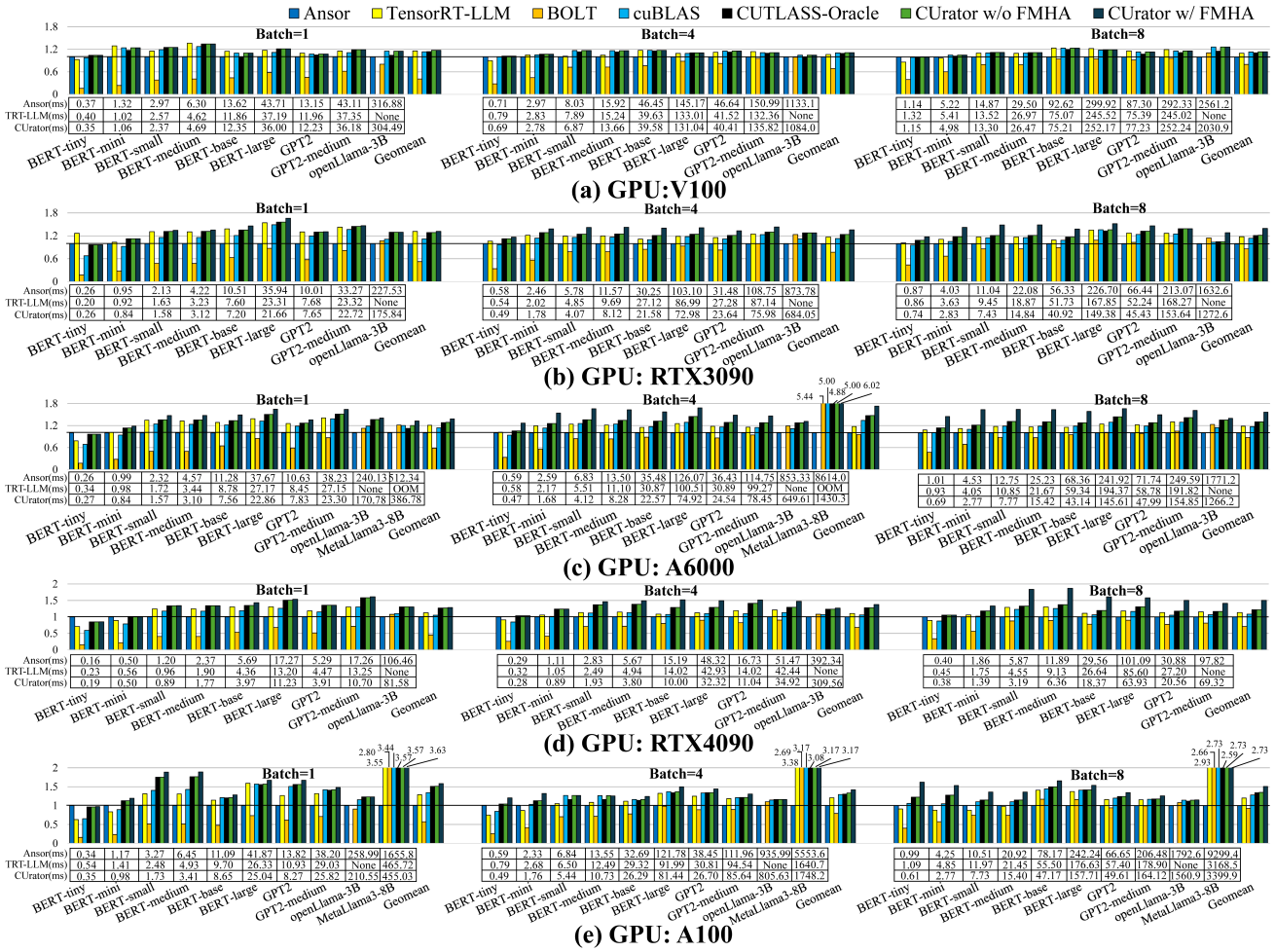

Figure 7. A comparison of the end-to-end single-precision LLM inference performance of CUrator and other frameworks on five GPUs. Each performance is normalized to the performance of the Ansor framework. Absolute performance numbers are given for Ansor, TensorRT-LLM (TRT-LLM), and CUrator w/ FMHA (CUrator). (None: not supported, OOM: Out-Of-Memory)

CUrator w/ FMHA achieves an average speedup of up to 1.18×, 1.40×, 1.73×, 1.50×, and 1.58× over the Ansor framework on the V100, RTX 3090, A6000, RTX 4090, and A100 GPUs, respectively. In comparison to TensorRT-LLM, CUrator w/ FMHA achieves a considerable average speedup across all GPUs. Based on the above result that CUrator w/ FMHA outperforms other frameworks, it is apparent that there is a significant potential for performance improvement when using CUTLASS GEMM in single-precision LLM inference. While CUrator may show slightly lower end-to-end inference performance than TensorRT-LLM in certain cases because CUrator does not perform target-GPU-specific optimizations, CUrator generally shows better performance than TensorRT-LLM on the target GPUs by maximizing resource utilization of GEMM, which is the core operation on LLMs. Therefore, CUrator can find the most efficient GEMM version for the target GPU and GEMM.

According to Figure 7(a)-Figure 7(e), Ansor can perform similarly or better than other frameworks when the model input is a single batch and the model size is small because GEMMs for small sized matrices on Ansor are more efficient than other techniques. However, as shown in Figure 7, Ansor cannot outperform other frameworks when the size of the GEMMs configuring the models increases because Ansor does not understand GPU resources well. Therefore, it is very important to tune the CUTLASS GEMM parameters for the best resource utilization, as the performance of CUTLASS-Oracle generally outperforms Ansor in single precision if the size of the GEMM is large enough.

As shown in Figure 7(a)-Figure 7(e), BOLT performs worse than the other frameworks. Since BOLT also uses the same CUTLASS kernels as CUTLASS-Oracle, it can understand GPU resources better than Ansor. However, BOLT performs worse than CUTLASS-Oracle because it often chooses the best tiling setting from a narrow search space (this can be

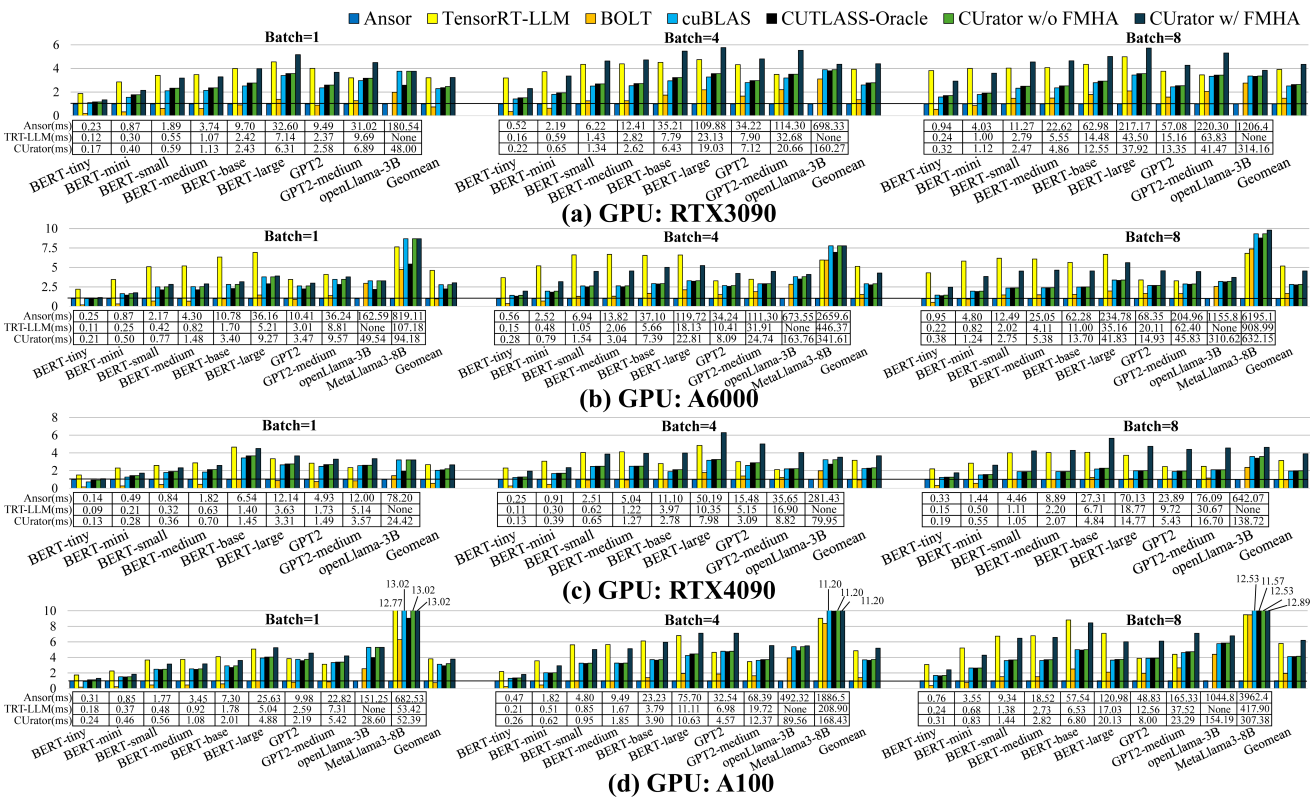

Figure 8. A comparison of the end-to-end half-precision LLM inference performance of CUrator and other frameworks on four GPUs. Each performance is normalized to the performance of the Ansor framework. Absolute performance numbers are given for Ansor, TensorRT-LLM (TRT-LLM), and CUrator w/ FMHA (CUrator). (None: not supported)

customized). In addition, unlike CUrator, BOLT maps an untuned native TVM kernel to the Softmax operation, which is another performance bottleneck in LLM inference. Furthermore, BOLT's Graph IR does not consider the performance overhead of reduction key allocation for split-K GEMMs at runtime. According to Figure 7, CUrator outperforms TensorRT-LLM in every case when comparing geomean performance. This clearly indicates that for single-precision LLM operations, the potential performance gains from maximally optimized GEMM kernels are significantly higher than the gains from LLM-specific graph level optimizations.

4.2.2 Half Precision. Figure 8 shows the performance of end-to-end LLM inference in half-precision. The evaluation is performed on only four GPUs, excluding V100 because the tensor cores in the V100 GPU are used differently compared to the newer GPUs [36] (Ampere and later generations). Similar to Figure 7, Figure 8 also shows the absolute performance of end-to-end inference for each model for Ansor, TensorRT-LLM, and CUrator w/ FMHA with small tables. The reported latency information can be used to estimate the actual inference time of each model with different frameworks. Based on the results in each batch size, the experimental results in Figure 8 show that CUrator w/ FMHA achieves an average speedup of up to 4.40×, 4.24×, 3.89×, and 6.20× over the Ansor framework on the RTX 3090, A6000, RTX 4090, and A100 GPUs, respectively. In addition, CUrator generally shows a considerable average speedup compared to TensorRT-LLM on various target GPUs, except for the A6000 GPU. In particular, TensorRT-LLM often outperforms CUrator for models with a smaller hidden size, because the performance gains from various optimization techniques, such as FMHA, are higher than the performance gains from efficient GEMM operations. However, for larger models with a much larger hidden size, CUrator shows superior performance due to the greater effectiveness of GEMM optimization rather than graph-level optimizations. This indicates that CUrator is highly effective in the context of the current trend towards significantly larger model sizes, across different GPU generations.

Although Ansor supports auto tuning on multiple hardware, it does not consider resource utilization in detail for specific hardware well. Thus, as shown in Figure 8, Ansor performs worse than cuBLAS and CUTLASS-Oracle, which can fully utilize the given GPU resources. Therefore, it is difficult for Ansor to maximize performance compared to cuBLAS and CUTLASS-Oracle on GPUs that understand

Table 3. A comparison of memory utilization and compute throughput between cuBLAS and CUTLASS-Oracle.

Library	GEMM	MEM Thro.	SM Thro.	Pipe. Stall	GMem. Access
cuBLAS	512, 3200, 3200	60.73%	72.02%	21.11%	6.65 MB
CUTLASS		36.57%	75.18%	4.93%	4.14 MB
cuBLAS	512, 3200, 8640	50.32%	66.77%	20.90%	13.80 MB
CUTLASS		37.26%	77.14%	4.55%	8.67 MB
cuBLAS	512, 8640, 3200	48.08%	66.95%	21.15%	17.53 MB
CUTLASS		45.70%	78.29%	7.53%	19.69 MB
cuBLAS	Geomean	52.76%	68.53%	21.05%	11.71 MB
CUTLASS		39.63%	76.85%	5.52%	8.90 MB
-	Normalization (CUTLASS/cuBLAS)	0.75	1.12	0.26	0.76

vendor-specific hardware well, such as tensor cores that can speed up GEMM computations. In addition, BOLT has the potential to outperform Ansor as the batch size increases in half-precision. As BOLT does not provide Softmax kernel tuning, the GEMMs that configure the model can be faster than Ansor, but BOLT often shows suboptimal performance in end-to-end inference when the Softmax computation bottleneck is a large portion of the inference time.

4.2.3 CUTLASS Kernel Performance Analysis. Table 3 briefly shows the profiling results from the NVIDIA Nsight profiler [29] for common GEMMs used in the openLlama-3B model [2] on the A6000 GPU. Since GPU kernel performance is highly affected by the computing core utilization (SM utilization) and memory resource utilization, we profiled relevant performance metrics such as memory throughput (MEM Thro.), SM throughput (SM Thro.), pipeline stalls (Pipe. Stall), and global memory accesses (GMem. Access). According to Table 3, CUTLASS kernels have only 26.22% pipeline stalls on average, compared to cuBLAS kernels, which also have only 0.76x global memory accesses on average. These results clearly support that CUTLASS kernels can achieve higher SM throughput with a reduced requirement for memory operations in comparison to their corresponding cuBLAS kernels. Therefore, GPU resource utilization can be optimized beyond that of the cuBLAS kernel by finding the most efficient tiling settings in the CUTLASS GEMM.

4.3 Evaluation on Techniques of CUrator

4.3.1 Split-K Optimization. When compiling the CUTLASS backend of the CUrator, the proportion of split-K enabled and disabled CUTLASS GEMM kernels is determined based on the GEMM dimensions of a target LLM. The results of our in-house experiments for the openLlama-3B model on the RTX 4090 GPU indicate that the proportions of split-K GEMMs out of the total GEMMs are 44% and 45% for batch sizes of 1 and 4, respectively. We further evaluated the inference performance of the openLlama-3B model in the same environment as above, with and without the split-K option. As a result, the inference performance with the split-K option

[2]Similar to Figure 2, we considered both regular([512, 3200, 3200])- and irregular([512, 3200, 8640], [512, 8640, 3200])-shaped GEMMs for profiling.

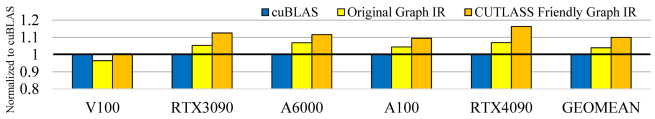

Figure 9. Performance comparison of CUTLASS-friendly Graph IR and baselines for BERT models.

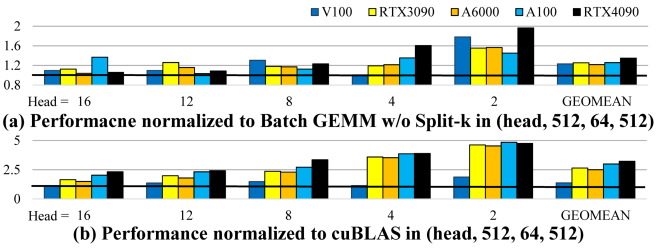

Figure 10. Performance comparison of CUTLASS GEMMs with and without the split-K Batch GEMM option in BERT models. (batch size, sequence length) of inputs are (1, 512).

outperforms the performance without the split-K option by 1.09× and 1.08× for batch sizes of 1 and 4, respectively. According to the above result, a significant number of GEMMs should be tuned with the split-K optimization since the output matrix size of GEMMs in recent LLMs is insufficient to fully utilize the entire GPU computing cores. Therefore, it is important to apply the split-K optimizations efficiently for the best LLM performance.

4.3.2 CUTLASS-Friendly Graph IR Modification. To evaluate the effectiveness of the CUTLASS-friendly graph IR, we compare the CUTLASS-friendly graph IR with the original graph IR. Figure 9 shows that the CUTLASS-friendly Graph IR generally outperforms the original Graph IR in all cases. When averaging the relative performance of BERT inference with a single-precision between Original Graph IR and CUTLASS-friendly Graph IR on each GPU, CUTLASS-friendly Graph IR outperforms Original Graph IR on average by 1.04×, 1.06×, 1.04×, 1.04×, and 1.08× on the V100, RTX3090, A6000, A100, and RTX4090 GPUs, respectively. This shows that CUTLASS-friendly Graph IR is generally effective across multiple GPUs.

According to Figure 9, the Graph IR Modification is effective as the number of cores increases for multiple GPUs. GEMMs do not require the split-K algorithm on prior GPUs consisting of a small number of SMs, but the split-K algorithm is often required on modern GPUs consisting of many SMs. Therefore, workloads that do not currently use the split-K algorithm should apply the split-K algorithm, or workloads that currently use the split-K algorithm should increase the split-K parameter to achieve maximum performance. In addition, it is highly effective to move the overhead of the reduction memory allocation from run-time to build time as the number of GPU cores increases.

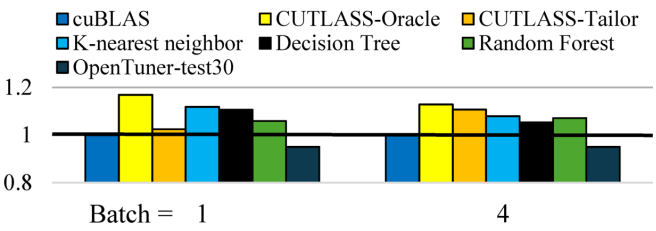

Figure 11. Case study for the introduction of auto-tuners for the CULTASS backend with the OpenLlama-3B model (single batch, sequence length 512).

4.3.3 Split-K Support on Batch GEMMs.

Figures 10(a) and (b) show the relative performance of the Batch GEMM with split-K normalized to Batch GEMM w/o split-K and the cuBLAS Batch GEMM, respectively. According to Figures 10(a) and (b), as the number of cores in GPUs increases, the Batch GEMM w/ split-K becomes more effective than other libraries for small-sized Batch GEMMs. According to Figure 7, there are kernels where Batch GEMM with split-K is effectively applied when the input size is small, resulting in improved performance over cuBLAS.

4.4 Case Study

We have performed a case study to introduce six well-known auto-tuners [7, 45] to find the CUTLASS oracle tiling setting on the CUrator. The auto-tuners used in the case study are categorized into two categories: 1) open-tuner[7] and 2) ML-based tuners [45]. Open-tuner searches for oracle tiling settings by testing 30× per GEMM on openLlama-3B. ML-based tuners are trained using GEMM profile information from BERT and GPT2 models (Section 3.1.2) and predict the oracle tiling settings of target GEMMs of openLlama-3B. As shown in Figure 11, the tiling settings predicted by ML-based tuners for GEMMs cannot exploit the GPU potential as much as CUTLASS-Orcale when applied to the OpenLlama-3B model, but they can show better performance than cuBLAS. However, open-Tuner has not found fair tiling settings for GEMM operations compared to ML-based tuners and CUrator. The experimental results indicate that if an advanced auto-tuner is developed that can identify the tiling settings of the CUTLASS oracle, it could approach the performance of CUrator within a small compilation time budget. For the development of next-generation tuning frameworks, CUrator can be a good baseline to show the best performance.

5 Related Works

In recent years, the field of deep learning computation has been in urgent need of faster computational speeds, and numerous researchers have been engaged in developing solutions to this problem. These studies are primarily divided into two categories: 1) enhancing the efficiency of BLAS computations such as GEMM, and 2) focusing on end-to-end performance optimization for various deep learning models.

Improving the performance of linear algebra computations Rahman et al. [35] and Malik [23] used a machine learning model to predict optimal tile sizes for several linear algebra computations as a replacement for a complex analytical performance model. Huang et al. [18] developed a detailed analytical performance model for Strassen's algorithm, which is an advanced algorithm of the GEMM operation, in a GPU computing environment. In terms of building an analytical model, Lym et al. [22] similarly constructed a novel analytical cost model for convolution operations in GPU computing environments considering various compiler features such as the software pipelining technique. Zhang et al. [48] also developed a detailed performance model for the SGEMM operation in GPUs based on the detailed analysis of the CUDA PTX (Parallel Thread Execution) ISA. In addition, Yu et al. [45] introduced an end-to-end GEMM performance calibration framework based on a fully-connected neural network model, which predicts the best tiling parameters and compile parameters for the CUTLASS GEMM. The research presented above has provided invaluable guidance on the tuning of linear algebra computations. However, they did not focus on the end-to-end performance of deep learning models. Consequently, the targeting workload scope of the aforementioned research is orthogonal to the CUrator, since the CUrator is designed to focus on the performance of the entire deep learning computation process.

Improving the end-to-end deep learning model performance Chen et al. [12] introduced AutoTVM, which learns various domain-specific cost models and uses them to optimize deep learning computations in order to automatically optimize deep learning models on different computing platforms. Since the introduction of the AutoTVM framework, several research teams have employed machine learning methods to enhance the efficiency of tensor programs or to accelerate the optimization of tensor programs [4, 20, 37, 51]. In order to reduce the lengthy optimization times associated with tensor programs, Li et al. [20] developed the AdaTune framework, which incorporates deep learning models in order to enhance the compilation times and performance of tensor programs. Similarly, Ahn et al. [4] presented the Chameleon framework, which employs reinforcement learning techniques to reduce the compilation time of deep learning models while enhancing their execution performance. Zheng et al. [51] introduced the FlexTensor framework, which schedules tensor programs by considering intrinsic features of various hardware, heuristic methods, and machine learning-based methods to compile optimal tensor programs on various hardware platforms, including CPUs, GPUs, and FPGAs. Ryu et al. [37] introduced OneShot, a neural network-based tensor program tuning framework that can replace the laborious process of tensor program hand-tuning, which requires the exploration of vast search

spaces. Apart from optimizing deep learning compilers with the aforementioned machine learning based techniques, Jeon and Park et al. [19] introduced Collage, a framework that enables sophisticated integration of various DL backends, so that deep learning compilers on different platforms can fully benefit from optimizations provided by third-party libraries. These studies have shown that various techniques can be used to reduce the compile time of deep learning compilers or to sufficiently improve the performance of deep learning models. However, these studies lack a detailed analysis of large language models (LLMs), which represent the latest generation of deep learning models. Furthermore, they are unable to derive the optimal end-to-end LLM execution performance through a comprehensive search based on state-of-the-art third-party libraries. From this perspective, CUrator has the advantage of achieving optimal end-to-end LLM performance in modern GPU computing environments.

6 Conclusion

In this paper, we proposed CUrator, an efficient modern LLM execution framework using maximally tuned third-party libraries on various GPUs. As a result of extensive evaluation of CUrator on the A100 GPU, the average end-to-end LLM execution speedups when maximally tuned are 1.50× and 4.99× in single and half precision, respectively, compared to the baseline. We strongly believe that the use of CUrator as a fundamental tool for studying GEMM operations across a spectrum of GPUs will inevitably lead to the construction of robust analytical cost or predictive models, informed by the insights derived from the comprehensive analysis of CUrator.

Acknowledgments

Thanks to the shepherd and reviewers for all their help and valuable feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant (RS-2024-00339187, 2021-0-00310, RS-2020-II201361, No.RS-2023-00277060), and by the National Research Foundation of Korea(NRF) grant (BK21 FOUR (Department of Computer Science and Engineering, Yonsei University)) funded by the Korea government(MSIT). Yongjun Park is the corresponding author.

Data-Availability Statement

The data that support the findings of this study are openly available in Zenodo (DOI: 10.5281/zenodo.14509993) [44].

A Artifact Appendix

A.1 Abstract

The artifact provides source codes for cuBLAS, CUTLASS (without Fused Multi-Head Attention (FMHA)), and CUrator with FMHA. It also provides the RTX 4090 CUTLASS tiling configurations used in Figure 7 and Figure 8 in Section 4.2. Getting all the oracle tiling settings on each GPU evaluated in this paper takes a long time. Therefore, we especially recommend testing on the RTX 4090 as we have provided a full CUTLASS tiling configuration in our Zenodo repository. The path to these files is `curator/LLM/cutlass_rtx4090`.

A.2 Artifact Check-List (Meta-Information)

- **Hardware:** Tesla V100-DGXS-32GB, NVIDIA GeForce RTX 3090, NVIDIA RTX A6000, NVIDIA GeForce RTX 4090, and NVIDIA A100-SXM4-80GB
- **Software:** TVM and CUTLASS are included in the Zenodo repository, while LLVM, CUDA, Pytorch, and cuDNN must be installed by the user.
- **Architecture:** x86_64
- **Output:** Profiling data (CUTLASS tiling configuration search results and end-to-end inference results for each library) and end-to-end inference results based on the CUDA library selected by CUrator
- **Experiments:** Makefile, Python, Manual Linux shell scripts
- **How much disk space is required (approximately):** Up to 32GB Maximum
- **How much time is needed to complete experiments (approximately):** On a multi-GPU system with four NVIDIA RTX A6000 GPUs, it takes about a week to find all the CUTLASS-Oracle tiling settings for all the GEMMs that configure the target LLMs used in the evaluation section, and about a day for the end-to-end inference of all the target LLMs.
- **Publicly available?** Yes.

A.3 Public Availability

Our source codes, CUTLASS tiling configuration search results, and scripts are available on Zenodo: https://doi.org/10.5281/zenodo.14509993[3]

A.4 Hardware Dependencies

In this paper, we have evaluated LLMs on various GPUs: NVIDIA V100-DGXS-32GB, RTX 3090, RTX A6000, RTX 4090, and A100-SXM4-80GB. CUrator is also compatible with any NVIDIA GPUs not included in the evaluation.

A.5 Software Dependencies

TVM v12.0.0, LLM v10.0.0, and CUTLASS v3.0.0 are required for all GPUs. To enable the latest FMHA support, CUrator with FMHA requires CUTLASS v3.5.1. NVIDIA RTX A6000, RTX 4090, and A100-SXM4-80GB GPUs require CUDA v12.0.0, Pytorch v2.1.2, and cuDNN v8.9.5. V100-DGXS-32GB, and RTX 3090 GPUs require CUDA v11.4.4, Pytorch v1.12.1, and cuDNN v8.9.7. Hugging Face's Llama models use FMHA, which is supported in PyTorch v2.0.0 and above (causing an error in PyTorch v1.12.1). Therefore, in our evaluation, we use Pytorch v2.1.2 for the end-to-end inference of the Llama model.

[3]The artifact is also available on the Github repository: https://github.com/yoon5862/CUrator

A.6 Installation

Navigate to your $HOME directory. Run the following commands:

```
$ git clone https://github.com/yoon5862/CUrator.git
(or download from Zenodo)
$ cd CUrator
$ conda env create -f conda.yml --name curator
$ conda activate curator
$ pip install torch==2.1.2 torchvision==0.16.2 \
 --index-url https://download.pytorch.org/whl/cu118
```

After installation, the directory $HOME/CUrator/ is considered to be the CUrator_HOME directory. To download and convert LLMs supported by the CUrator framework into ONNX graph files, run the following commands in the CUrator_HOME directory:

```
$ cd script
$ ./create_model.sh > create_model.log
```

The supported_models are listed in the Zenodo repository. Note that the GPT-2 and BERT models do not require the user to obtain a license key. However, the MetaLlama3 model requires a license key, which can be obtained from HuggingFace [41].

A.7 Experiment Workflow

The CUrator framework consists of three processes: (1) rewriting the ONNX graphs into cuBLAS-/CUTLASS-enabled graphs, (2) performing full profiling for all GEMM kernels included in the ONNX graph, using cuBLAS, CUTLASS (w/o FMHA), and CUTLASS (w/ FMHA) versions, and (3) determining the optimal kernel versions for each target GEMM. Steps (1) and (2) are performed using the following instructions. First, navigate to the CUrator_HOME directory. Then, run the following commands:

```
$ cd script
$ ./profile_single.sh 89 ./cutlass_rtx4090 \
 > profile_single.log
```

Please note that upon initial execution of the aforementioned process by CUrator for a target GPU, it should perform an exhaustive search for the full CUTLASS compilation parameter combinations, which can take from a few days to several weeks. To reduce the time needed for artifact evaluation, we provide the pre-profiled data on the RTX 4090 GPU. Once steps (1) and (2) have been completed, the process of identifying the most efficient GEMM kernels for all target GEMMs, considering the cuBLAS, CUTLASS (w/o and w/ FMHA) libraries, is performed using the following instructions. First, navigate to the CUrator_HOME directory. Then, run the following commands:

```
$ cd script
$  ./curator_single.sh 89 ./cutlass_rtx4090 \
 > curator_single.log
```

Once CUrator's kernel decision process is complete, it provides a pass that exploits the full potential of the target GPU and shows the optimal performance for end-to-end inference of LLMs. A text file is also created that records the contents in a human-readable text format.

A.8 Evaluation and Expected Result

Navigate to the CUrator_HOME/LLM/cutlass_rtx4090/ directory. In this directory, the inference results for LLMs on the RTX 4090 are stored in individual json files (for example, gaunernst-bert-small-uncased_1_512_float16.json). In this json file, it contains the end-to-end inference performance results of LLMs using cuBLAS, CUTLASS (without FMHA), CUTLASS (with FMHA), and CUrator. The performance of cuBLAS and CUTLASS (without FMHA) can be found in the cuBLAS and CUTLASS-Oracle bars in Figure 7(d) and Figure 8(c), respectively. Users can also inference LLMs with CUrator on various GPUs in addition to the RTX 4090.

References

[1] 2024. cuBlas Library. http://developer.nvidia.com/cublas.

[2] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A Large-Scale Simulation Framework For LLM Inference. *Proceedings of Machine Learning and Systems* 6 (2024), 351–366.

[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 7, 18 pages.

[4] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rygG4AVFvH

[5] AI@Meta. 2024. Llama 3 Model Card. (2024). https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[6] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed- Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41404.2022.00051

[7] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: an extensible framework for program autotuning *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[8] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. Onnx: Open neural network exchange.

[9] Alexander Borzunov, Max Ryabinin, Artem Chumachenko, Dmitry Baranchuk, Tim Dettmers, Younes Belkada, Pavel Samygin, and

Colin A Raffel. 2023. Distributed Inference and Fine-tuning of Large Language Models Over The Internet. In *Advances in Neural Information Processing Systems*, A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 12312–12331. https://proceedings.neurips.cc/paper_files/paper/2023/file/28bf1419b9a1f908c15f6195f58cb865-Paper-Conference.pdf

[10] Hongzheng Chen, Jiahao Zhang, Yixiao Du, Shaojie Xiang, Zichao Yue, Niansong Zhang, Yaohui Cai, and Zhiru Zhang. 2024. Understanding the Potential of FPGA-based Spatial Acceleration for Large Language Model Inference. 18, 1, Article 5 (Dec. 2024), 29 pages. https://doi.org/10.1145/3656177

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[12] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. 31 (2018). https://proceedings.neurips.cc/paper_files/paper/2018/file/8b5700012be65c9da25f49408d959ca0-Paper.pdf

[13] Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. 2021. Bring your own codegen to deep learning compiler. *arXiv preprint arXiv:2105.03215* (2021).

[14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759

[15] NVIDIA Corporation. 2024. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM/tree/v0.11.0.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[17] Xinyang Geng and Hao Liu. 2023. *OpenLLaMA: An Open Reproduction of LLaMA*. https://github.com/openlm-research/open_llama

[18] Jianyu Huang, Chenhan Yu, and Robert van de Geijn. 2020. Strassen's Algorithm Reloaded on GPUs. *ACM Trans. Math. Software* 46 (03 2020), 1–22. https://doi.org/10.1145/3372419

[19] Byungsoo Jeon, Sunghyun Park, Peiyuan Liao, Sheng Xu, Tianqi Chen, and Zhihao Jia. 2023. Collage: Seamless Integration of Deep Learning Backends with Automatic Placement. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 517–529. https://doi.org/10.1145/3559009.3569651

[20] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. 2020. Adatune: Adaptive tensor program compilation made efficient. *Advances in Neural Information Processing Systems* 33 (2020), 14807–14819.

[21] LLVM. 2012. libclc. http://libclc.llvm.org.

[22] Sangkug Lym, Donghyuk Lee, Mike O'Connor, Niladrish Chatterjee, and Mattan Erez. 2019. DeLTA: GPU Performance Model for Deep Learning Applications with In-Depth Memory System Traffic Analysis . In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, Los Alamitos, CA, USA, 293–303. https://doi.org/10.1109/ISPASS.2019.00041

[23] Abid M. Malik. 2012. Optimal Tile Size Selection Problem Using Machine Learning. In *2012 11th International Conference on Machine Learning and Applications*, Vol. 2. 275–280. https://doi.org/10.1109/ICMLA.2012.214

[24] NVIDIA. 2017. NVIDIA Tesla V100. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[25] NVIDIA. 2020. CUTLASS: CUDA Templates for Linear Algebra. https://github.com/NVIDIA/cutlass.

[26] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[27] NVIDIA. 2021. NVIDIA RTX 3090 Family. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090-3090ti/.

[28] NVIDIA. 2021. NVIDIA RTX A6000 Graphics Cards. https://www.nvidia.com/en-us/design-visualization/rtx-a6000/.

[29] NVIDIA. 2022. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems Retrieved September 8, 2022.

[30] NVIDIA. 2022. NVIDIA RTX 4090. https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/.

[31] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit

[32] Daon Park, Sungbin Jo, and Bernhard Egger. 2023. Improving Throughput-oriented Generative Inference with CPUs. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems* (Seoul, Republic of Korea) *(APSys '23)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/3609510.3609815

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[34] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://api.semanticscholar.org/CorpusID:160025533

[35] Mohammed Rahman, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. 2010. Neural Network Assisted Tile Size Selection. (11 2010).

[36] Md Aamir Raihan, Negar Goli, and Tor M. Aamodt. 2019. Modeling Deep Learning Accelerator Enabled GPUs. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 79–92. https://doi.org/10.1109/ISPASS.2019.00016

[37] Jaehun Ryu, Eunhyeok Park, and Hyojin Sung. 2022. One-shot tuner for deep learning compilers. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 89–103. https://doi.org/10.1145/3497776.3517774

[38] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 31094–31116. https://proceedings.mlr.press/v202/sheng23a.html

[39] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-Read Students Learn Better: On the Importance of Pre-training Compact Models. *arXiv preprint arXiv:1908.08962v2* (2019).

[40] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2009. Software Pipelined Execution of Stream Programs on GPUs. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. IEEE Computer Society, USA, 200–209. https://doi.org/10.1109/CGO.2009.20

[41] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick Platen, Clara

Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Scao, Sylvain Gugger, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[42] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 38087–38099. https://proceedings.mlr.press/v202/xiao23c.html

[43] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: Bridging the gap between auto-tuners and hardware-native performance. *Proceedings of Machine Learning and Systems* 4 (2022), 204–216.

[44] Lee Yoon Noh, Yu Yongseung, and Park Yongjun. 2024. Curator: An Efficient LLM Execution Engine with Optimized Integration of CUDA Libraries. https://doi.org/10.5281/zenodo.14509993

[45] Yongseung Yu, Donghyun Son, Younghyun Lee, Sunghyun Park, Giha Ryu, Myeongjin Cho, Jiwon Seo, and Yongjun Park. 2023. Tailoring CUTLASS GEMM using Supervised Learning. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. 465–474. https://doi.org/10.1109/ICCD58817.2023.00077

[46] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, et al. 2021. Oneflow: Redesign the distributed deep learning framework from scratch. *arXiv preprint arXiv:2110.15032* (2021).

[47] Amir Zandieh, Insu Han, Majid Daliri, and Amin Karbasi. 2023. KDEformer: Accelerating Transformers via Kernel Density Estimation. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 40605–40623. https://proceedings.mlr.press/v202/zandieh23a.html

[48] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 31–43. https://doi.org/10.1145/3018743.3018755

[49] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2024. Atom: Low-Bit Quantization for Efficient and Accurate LLM Serving. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 196–209. https://proceedings.mlsys.org/paper_files/paper/2024/file/5edb57c05c81d04beb716ef1d542fe9e-Paper-Conference.pdf

[50] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[51] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 859–873. https://doi.org/10.1145/3373376.3378508

[52] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. 2023. Response length perception and sequence scheduling: an LLM-empowered LLM inference pipeline. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2859, 14 pages.

[53] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. {DistServe}: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.

Received 2024-09-12; accepted 2024-11-04