Generative AI

English

Optimizing llama.cpp AI Inference with CUDA Graphs

Aug 07, 2024

By Alan Gray

The open-source llama.cpp code base was originally released in 2023 as a lightweight but efficient framework for performing inference on Meta Llama models. Built on the GGML library released the previous year, llama.cpp quickly became attractive to many users and developers (particularly for use on personal workstations) due to its focus on C/C++ without the need for complex dependencies.

the time of writing, llama.cpp sits at #123 in the star ranking of all GitHub repos, and #11 of all C++ GitHub repos.

Performing AI inference with llama.cpp on NVIDIA GPUs already offers significant benefits, due to their ability to perform the computations underlying AI inference with extreme performance and energy efficiency, paired with their prevalence in consumer devices and data centers. NVIDIA and the llama.cpp developer community continue to collaborate to further enhance performance. This post describes recent improvements achieved through introducing CUDA graph functionality to llama.cpp.

CUDA Graphs

GPUs continue to speed up with each new generation, and it is often the case that each activity on the GPU (such as a kernel or memory copy) completes very quickly. In the past, each activity had to be separately scheduled (launched) by the CPU, and associated overheads could accumulate to become a performance bottleneck.

The CUDA Graphs facility addresses this problem by enabling multiple GPU activities to be scheduled as a single computational graph. In a previous post, Getting Started with CUDA Graphs, I introduce CUDA Graphs and demonstrate how to get started. In a subsequent post, A Guide to CUDA Graphs in GROMACS 2023, I describe how CUDA Graphs were successfully applied to the GROMACS biomolecular simulation scientific software package.

When using the traditional stream model, each GPU activity is scheduled separately, whereas CUDA graphs enable multiple GPU activities to be scheduled in unison. This reduces scheduling overheads. It is relatively straightforward to adapt an existing stream-based code to use graphs. The functionality “captures” the stream execution into a graph, through a few extra CUDA API calls.

This post explains how to exploit this facility to enable the pre-existing llama.cpp code to be executed using graphs instead of streams.

This section highlights the overheads in the pre-existing code, and describes how CUDA Graphs have been introduced to reduce these overheads.

Overheads in pre-existing code

Figure 1 shows snippets of the profile of the pre-existing code, before the introduction of CUDA Graphs, performing Llama 7B Q4 inference on an NVIDIA A100 GPU using Linux. It was obtained using NVIDIA Nsight Systems. Each chunk of GPU activities in the top profile in the figure corresponds to the evaluation of a single token, where the zoom is set to show two full tokens being evaluated. It can be seen that there are gaps in the profile between the evaluation of each token, corresponding to CPU activities related to sampling and preparation of the compute graph. (I will return to this point at the end of the post.)

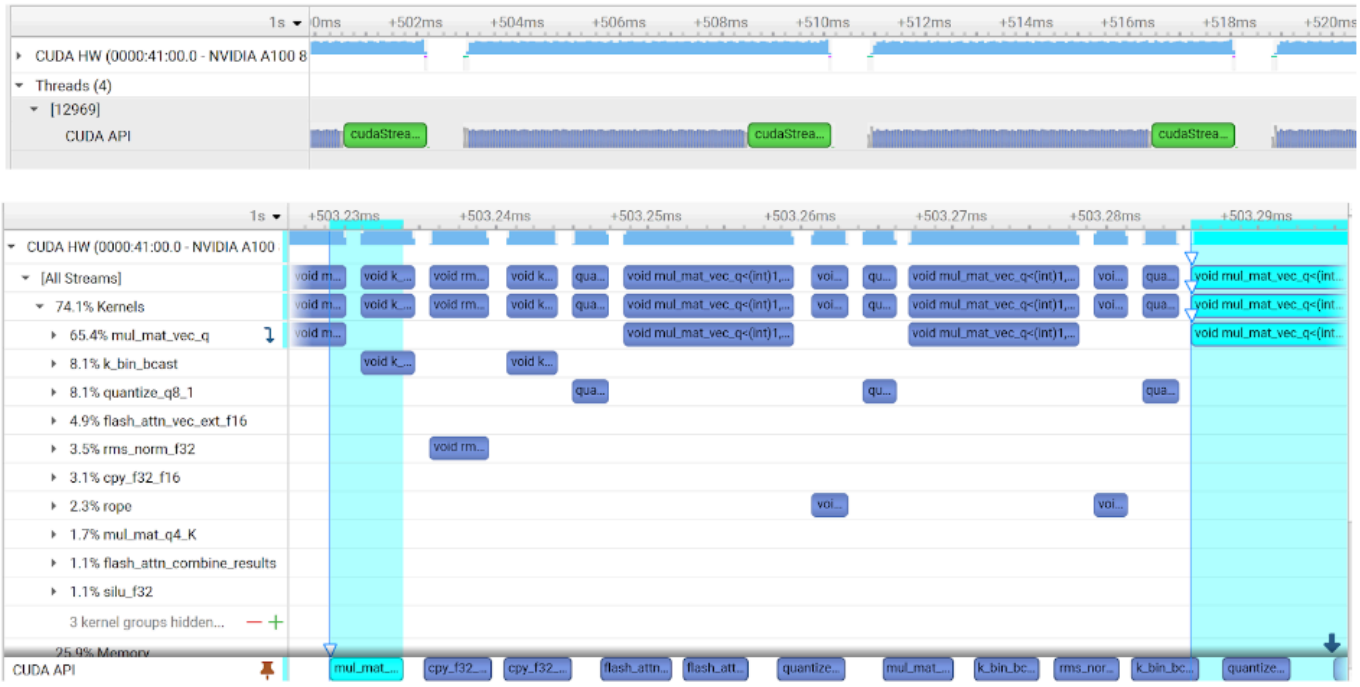


Figure 1. Overheads associated with the GPU activities involved in GPU inference, observed through sections of the profiling timeline where the GPU is idle

The bottom of Figure 1 shows the same profile but zoomed in to show several activities within the evaluation of a token. The gaps seen between kernels within token evaluation are due to launch overheads. As this post will show, the removal of these overheads with CUDA Graphs

right): the CPU is able to successfully launch well in advance of execution on the GPU.

Therefore, CPU-side launch overheads are not on critical path here. Instead, the overheads are due to GPU-side activities associated with each kernel launch. This behavior may vary across different models and hardware, but CUDA Graphs are applicable for reducing CPU and/or GPU launch overheads.

Introducing CUDA Graphs to reduce overheads

llama.cpp already uses the concept of a "graph" in GGML format. The generation of each token involves the following steps:

- Preparation of the GGML graph structure based on the model in use.
- Evaluation of the structure on the backend in use (in this case an NVIDIA GPU) to get "logits," log probability distribution across vocabulary for next token.
- Sampling is performed on the CPU to select a token from the vocabulary using the logits.

CUDA Graphs were introduced by intercepting the GPU graph evaluation stage. Code was added to capture the existing stream into a graph, instantiate the captured graph into an executable graph, and launch that to the GPU to perform the evaluation of a single token.

To be suitably efficient, it is necessary to re-use the same graph multiple times; otherwise, newly introduced overheads involved in capture and instantiation outweigh the benefits. However, the graph dynamically evolves as inference proceeds. The challenge was to develop a mechanism to enable minor (low-overhead) adjustments to the graph across tokens, to reach an overall benefit.

As inference progresses, the length of operations steps up with context size, resulting in substantial (but infrequent) changes to the compute graph. The GGML graph is inspected and only recaptured when required. cudaGraphExecUpdate is used to update the previously instantiated executable graph with much lower overhead than full re-instantiation.

There are also frequent, but very minor, changes to the compute graph, where kernel parameters for certain nodes (related to the KV cache) change for each token. NVIDIA developed a mechanism to update only these parameters in a re-usable CUDA Graph. Before each graph is launched, we leverage CUDA Graph API functionality to identify the part of the graph that requires updating, and to manually replace the relevant parameters.

developments and ongoing work to address issues and restrictions, see the GitHub issue, new optimization from NVIDIA to use CUDA Graphs in llama.cpp, and pull requests linked therein.

Impact of CUDA Graphs in reducing overheads

Before the introduction of CUDA Graphs there existed significant gaps between kernels due to GPU-side launch overhead, as shown in the bottom profile in Figure 1.

Figure 2 shows the equivalent with CUDA Graphs. All the kernels are submitted to the GPU as part of the same computational graph (with a single CUDA API launch call). This vastly reduces the overheads: the gap between each kernel within the graph is now very small.

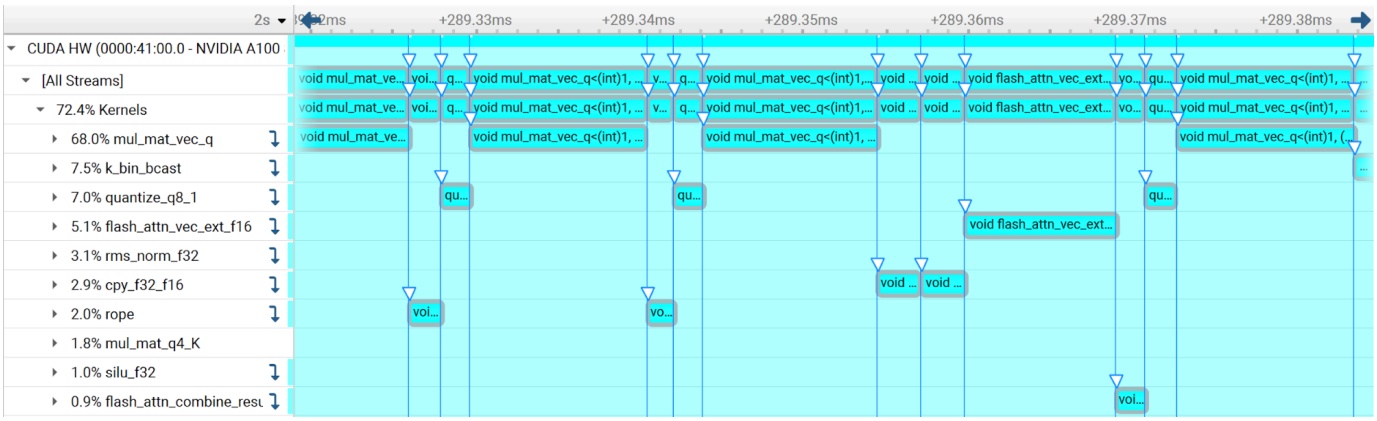


Figure 2. Overheads associated with the GPU activities involved in GPU inference are vastly reduced with CUDA Graphs

Performance results

Figure 3 shows the benefit of the new CUDA Graphs functionality in llama.cpp. The measured speedup varies across model sizes and GPU variants, with increasing benefits as model size decreases and GPU capability increases. This is in line with expectations, as using CUDA Graphs reduces the overheads most relevant for small problems on fast GPUs. The highest achieved speedup is 1.2x for the smallest Llama 7B model on the fastest NVIDIA H100 GPUs. Linux systems were used for all results.

CUDA Graphs are now enabled by default for batch size 1 inference on NVIDIA GPUs in the main branch of llama.cpp.

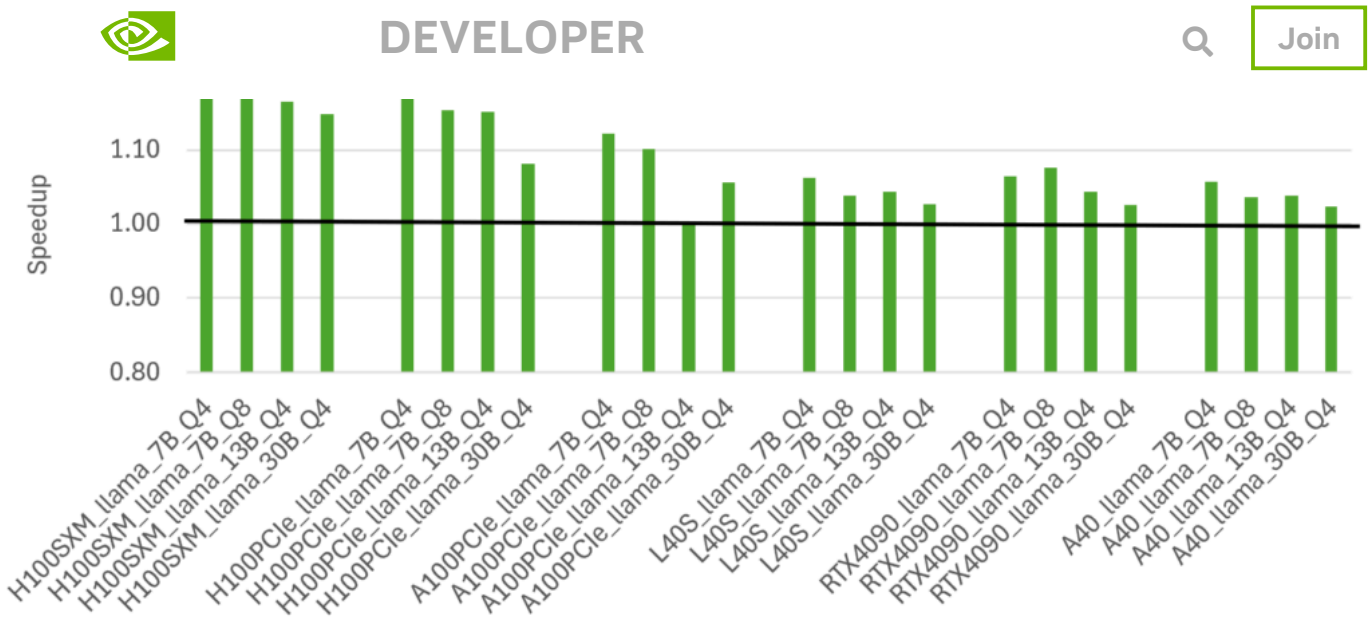


Figure 3. The speedup achieved with CUDA Graphs against traditional streams, for several Llama models of varying sizes (all with batch size 1), including results across several variants of NVIDIA GPUs

Ongoing work to reduce CPU overheads

The top profile in Figure 1 shows gaps (where the GPU is idle) between token evaluation in the timeline. These are due to CPU activities associated with preparation of the GGML graph and with sampling. Work to reduce these overheads is at an advanced stage, as described in this GitHub issue and the pull requests linked therein. This work is expected to offer up to ~10% further improvement.

Summary

In this post, I showed how the introduction of CUDA Graphs to the popular llama.cpp code base has substantially improved AI inference performance on NVIDIA GPUs, with ongoing work promising further enhancements. To take advantage of this work for your own AI-enabled workflow, follow the Usage Instructions.

Related resources

- **GTC session:** Accelerate Inference on NVIDIA GPUs
- **GTC session:** Fast and Secure LLM Inference: GPU-Optimized CKKS Homomorphic Encryption
- **GTC session:** Structure From Chaos: Accelerate GraphRAG With cuGraph and NVIDIA NIM

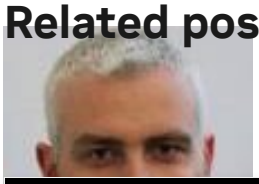
Discuss (0) +13 Like

Tags

Generative AI | General | CUDA | Nsight Tools - Compute | AI Inference / Inference Microservices | CUDA Graphs | Featured | Llama.Cpp | LLMs

About the Authors

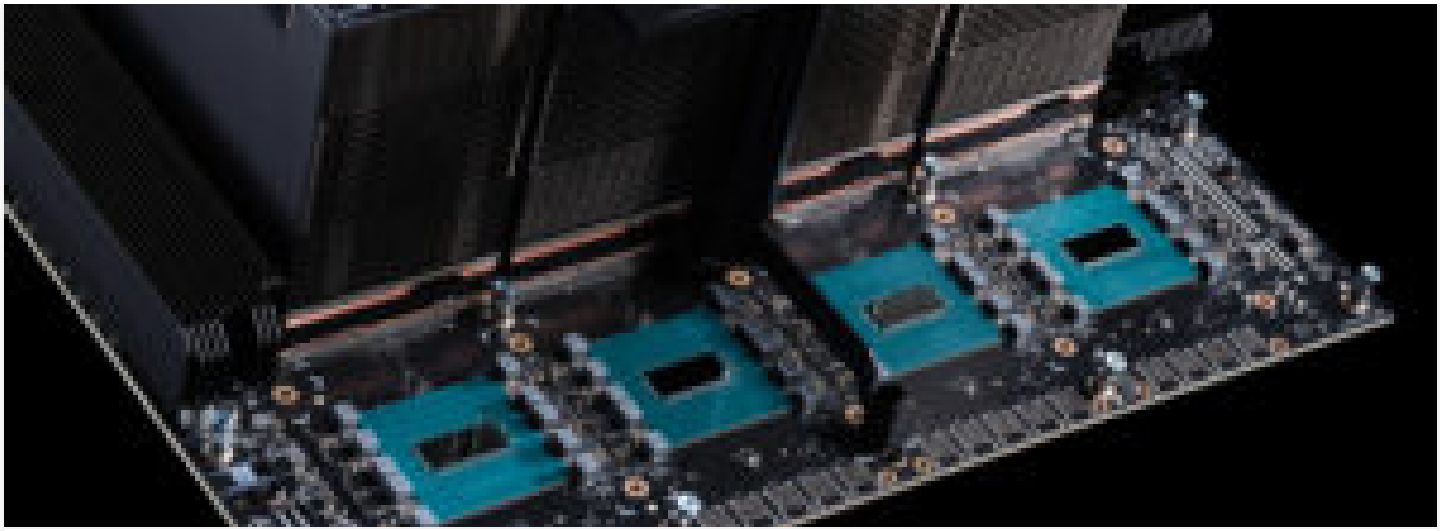
Related posts



About Alan Gray

Alan Gray is a Principal Developer Technology Engineer at NVIDIA where he specializes in application optimization, particularly on large-scale

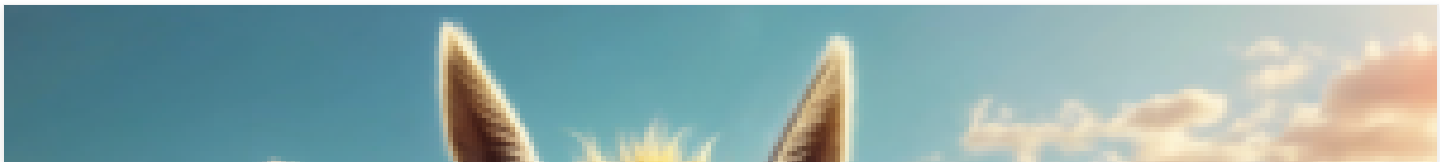
NVIDIA NIM 1.4 Ready to Deploy with 2.4x Faster Inference



Boosting Llama 3.1 405B Throughput by Another 1.5x on NVIDIA H200 Tensor Core GPUs and NVLink Switch

Boosting Llama 3.1 405B Performance up to 1.44x with NVIDIA TensorRT Model Optimizer on NVIDIA H200 GPUs

Supercharging Llama 3.1 across NVIDIA Platforms



Turbocharging Meta Llama 3 Performance with NVIDIA TensorRT-LLM and NVIDIA Triton Inference Server

DEVELOPER

Join

Sign up for NVIDIA News

Subscribe

Follow NVIDIA Developer