Towards Efficient CNN Inference: Memory Profiling and Optimization

Akash Maji, Suraj Reddy, Utkarsh Sharma, Amandeep Nokhwal
MTech CSE, Indian Institute of Science, Bangalore, India
Emails: {akashmaji, valadrireddy, utkarsh2024, amandeepn}@iisc.ac.in

***Abstract*—Convolutional Neural Networks (CNNs) are at the heart of many modern computer vision systems. However, they often run into memory bottlenecks, especially when performing inference on GPUs. These bottlenecks can slow things down, limiting throughput, increasing latency, and consuming more power. In this project, we dug into this problem by profiling CNN inference across several well-known models (like LeNet-5, AlexNet, and various ResNets) using different batch sizes. Our goal was to pinpoint exactly where memory becomes a problem. We used GPU profiling tools to carefully track memory usage patterns. Following this analysis, we explored and tested several software-based optimization techniques, including switching to FP16 arithmetic, using Automatic Mixed Precision (AMP) combined with Automatic Memory Coalescing (AMC), and implementing memory tiling. Our findings shed light on how batch size affects performance and how effective these different optimization strategies are at cutting down the memory footprint and potentially speeding things up, ultimately aiming for more efficient large-scale deployments. We paid close attention to metrics like how long kernels took to run, the total execution time, peak memory consumption, memory transfer durations, and time spent waiting on system calls.**

***Index Terms*—CNN Inference, Memory Profiling, GPU Optimization, Deep Learning, Performance Analysis, FP16, Memory Tiling, AMP, AMC.**

I. Introduction

Deep Convolutional Neural Networks (CNNs) form the backbone of many advances in computer vision today, tackling tasks from image classification and object detection to segmentation [1], [2]. While modern GPUs boast impressive number-crunching power, the actual speed of CNN inference often gets bogged down by memory issues [3], [4]. These memory bottlenecks don't just slow things down; they can severely limit how many inferences you can run per second (throughput), increase the delay for each result (latency), and drive up power consumption [5]. Common culprits include scattered memory accesses (non-coalesced), not making the best use of caches, and simply moving too much data back and forth between different memory levels, especially relying too heavily on the slower global memory.

The need for speed in inference is crucial in many real-world applications. Think about Augmented and Virtual Reality (AR/VR) – these systems need incredibly high frame rates (often 72-120Hz), meaning each frame needs to be processed in under 15 milliseconds. Robotics rely on quick predictions for tasks like figuring out where they are or deciding what to do next. Even advanced graphics techniques, like path tracing, are increasingly using CNN-based denoisers that demand high throughput (30-100+ frames per second). To meet these tough demands, we need to look beyond just raw computational power and focus squarely on making memory access more efficient.

This project sets out to tackle these memory challenges head-on. We started by systematically profiling how memory is accessed during CNN inference. Armed with insights from detailed empirical profiling using tools like NVIDIA Nsight Compute [6], we then explored and implemented specific software-level optimizations. We looked into techniques like using reduced precision arithmetic (FP16), memory tiling, Automatic Mixed Precision (AMP), Automatic Memory Coalescing (AMC), and considered others like kernel fusion and changing data layouts. Our aim is to cut down both the latency and the overall energy used, hopefully paving the way for deploying CNN models more efficiently and at a larger scale on GPUs. We made sure to analyze how things change across different network architectures and batch sizes to get a complete picture of the trade-offs involved.

II. Related Work

Making memory operations faster for deep learning on GPUs, especially for CNN inference, has been a busy research area. Researchers have explored several promising angles:

Memory Layout and Access Optimization: One popular strategy involves rearranging how data is stored in memory (data layout transformations). The idea is to improve data locality – keeping related data close together – and enable more efficient, grouped memory reads and writes (coalesced transactions). This helps reduce stalls and makes better use of the available memory bandwidth [5]. Libraries like NVIDIA's cuDNN often have these kinds of optimizations built-in [7].

GPU Profiling Tools: You can't fix what you don't understand. Tools like NVIDIA Nsight Compute [6], along with others like nvprof and nsys, are essential for getting a deep dive into memory behavior. They provide detailed numbers on memory latency, throughput, how often caches are hit, how busy the processing units (SMs) are, and how long different parts of the code (kernels) take to run. This allows developers to pinpoint exactly where the bottlenecks lie [6]. We relied heavily on these tools in our own work [8].

Reduced Precision and Mixed Precision: Using less precise number formats, like 16-bit floating point (FP16) or 8-bit integers (INT8), directly cuts down on the amount of memory needed and the bandwidth required to move data.

Plus, hardware like NVIDIA's Tensor Cores can often perform calculations much faster with these lower precisions [4]. Frameworks supporting Automatic Mixed Precision (AMP) help automate the process, smartly using FP16 for parts of the network where it's safe, while keeping more sensitive parts in standard 32-bit precision (FP32) to maintain accuracy.

Kernel Fusion and Operator Reordering: Sometimes, multiple operations are performed one after another (like a convolution followed by a ReLU activation). Kernel fusion combines these into a single GPU task (kernel). This cuts down on the overhead of launching multiple kernels and avoids writing intermediate results out to the slow global memory and then reading them back in [5]. Simply changing the order in which operations are performed can also sometimes improve data locality.

Memory Tiling: This technique involves breaking down large chunks of data (like the feature maps or weights in a CNN) into smaller blocks or 'tiles'. These smaller tiles are designed to fit into faster, smaller memory spaces on the GPU, like shared memory or the L1/L2 caches. By working on these tiles, the algorithm can drastically reduce how often it needs to access the slower global memory [5].

Distributed Inference: When dealing with extremely large models or needing very high throughput, sometimes inference is split across multiple GPUs. This introduces new challenges related to communication overhead between the GPUs. Strategies for smartly partitioning the data and optimizing how the GPUs talk to each other become important [9], although this wasn't our primary focus in this project [10].

Our work connects these threads. We combine in-depth profiling across multiple models [8] with a practical evaluation of several software optimization techniques (FP16, AMP/AMC, Tiling) specifically aimed at making CNN inference run better on commonly available GPUs.

III. Methodology

A. Profiling Setup and Tools

To get a broad view, we ran our experiments on systems using different NVIDIA GPUs: a GeForce RTX 3060 (part of the Ampere family), a GTX 1050 (Pascal architecture), and a Tesla T4 (Turing architecture). This selection gave us a sense of how things perform across a range of hardware capabilities. We used standard deep learning libraries, specifically PyTorch version 1.13 and later, to build and run our CNN models.

For digging into the performance details, our main tool was NVIDIA Nsight Compute [6]. We also used system-level tools (like the functionalities offered by nvprof, nsys, or ncu, sometimes accessed through the PyTorch profiler). We focused on collecting key metrics, including:

- DRAM Frequency and Throughput (as a percentage)
- L1/Texture Cache Throughput (as a percentage)
- L2 Cache Throughput (as a percentage)
- Compute (SM) Throughput (as a percentage)
- How many cycles the SMs were active
- Execution times for individual GPU kernels
- Time spent transferring data between the host CPU and the GPU (both ways)
- Peak GPU Memory Allocated
- Time spent waiting on system calls (like semwait or poll)

B. Models and Datasets

We chose a few representative CNN architectures to analyze:

- LeNet-5: A classic, relatively small CNN, good for baseline comparisons.
- AlexNet [1]: One of the foundational large-scale CNNs that spurred modern deep learning.
- ResNet [3]: We looked at several versions – ResNet-20, ResNet-32, ResNet-44, and ResNet-56. This let us study how network depth and overall complexity affect performance. These were custom implementations not the standard versions available in PyTorch.

The datasets we used were:

- CIFAR-10: A common benchmark dataset with 10 object classes and 60,000 small images (50k for training, 10k for testing), each 3x28x28 pixels. We used this for most of our batch size analysis and initial optimization tests.
- Mini ImageNet (a subset): A more challenging dataset with around 1000 classes, roughly 38,000 images, and larger image dimensions (3x224x224). This was particularly useful for our memory tiling experiments because the larger images make memory access patterns more critical.

In our experiments, we typically varied the batch size for inference, trying out a range of values (e.g., 1, 2, 4, and doubling up to 512 or 1024). This helped us see how performance scales and where bottlenecks might appear or shift as we process more images at once.

C. Evaluation Metrics

To judge performance and efficiency, we focused on these primary metrics:

- **Latency:** How long does it take, on average, to process one batch of images? We also looked at the execution time of individual kernels.
- **Throughput:** How many images can the model process in a given amount of time? (We inferred this from the total execution time for a set number of images).
- **Peak GPU Memory Allocated:** The maximum amount of GPU memory allocated at any point during inference.
- **GPU Utilization:** What percentage of the time were the GPU's main processing units (SMs) busy? We also looked at the utilization of specialized units like Tensor Cores when applicable.
- **System Overheads:** How much time was spent waiting for the operating system or synchronization mechanisms (like the 'semwait' system call)?
- **Accuracy:** When we applied optimizations like FP16 that might affect the results, we checked the test accuracy and loss to make sure the model was still performing correctly.

D. Optimization Techniques Explored

Based on what we learned from profiling, we implemented and tested the following software-based optimizations:

- **FP16 Arithmetic:** We used PyTorch's 'torch.cuda.amp.autocast()' feature. This enables Automatic Mixed Precision, which cleverly uses faster FP16 calculations where possible while keeping critical parts in FP32 to maintain accuracy.
- **AMP + AMC:** We tried combining Automatic Mixed Precision (AMP) with Automatic Memory Coalescing (AMC). The idea here is to optimize memory access patterns at the same time as changing precision, which can be particularly helpful on architectures like the Turing GPU (Tesla T4).
- **Memory Tiling:** We experimented with implementing tiling strategies for convolution operations. This was especially relevant when working with the larger images from the ImageNet dataset. By breaking the work into smaller tiles, we aimed to reduce trips to the slow global memory by keeping more data in faster shared memory or caches. We tried different tile sizes to see how it affected performance.

We recognized that other techniques, like explicitly changing data layouts or fusing kernels together, could also be beneficial, but we decided to focus on the ones listed above for this phase of our work.

IV. Results and Analysis

We carried out extensive profiling runs and applied various optimizations to better understand and enhance the performance of CNN inference. The Phase 2 report has all the details about profiling which are not included in this report [8], [10].

A. Impact of Batch Size (ResNet on CIFAR-10)

When we analyzed different ResNet variants on the CIFAR-10 dataset, changing the batch size revealed some clear patterns:

- **Execution Time:** Initially, the total time to run inference generally dropped as we increased the batch size (up to around 64). This makes sense, as we're better utilizing the GPU's parallel processing capabilities. However, for mid-range sizes (like 128 or 256), we sometimes saw the time increase again. This might be due to overheads or less efficient partitioning of resources. At very large sizes (512, 1024), the time might drop again as the GPU becomes fully saturated (see the top graph in Fig. 1).
- **Peak GPU Memory Allocated:** Increased monotonically and significantly with batch size, as expected due to larger activation maps (See Fig. 1, bottom). This highlights the memory constraints of using large batches.
- **GPU Utilization (%):** Similarly, the overall GPU utilization generally went up with both batch size and the complexity of the model (e.g., ResNet-56 was busier than ResNet-20). This trend usually plateaued at the larger batch sizes (see the bottom graph in Fig. 2).
- **Average Inference Time:** The time taken per batch naturally increased with both the batch size and the model's complexity (see the top graph in Fig. 2).

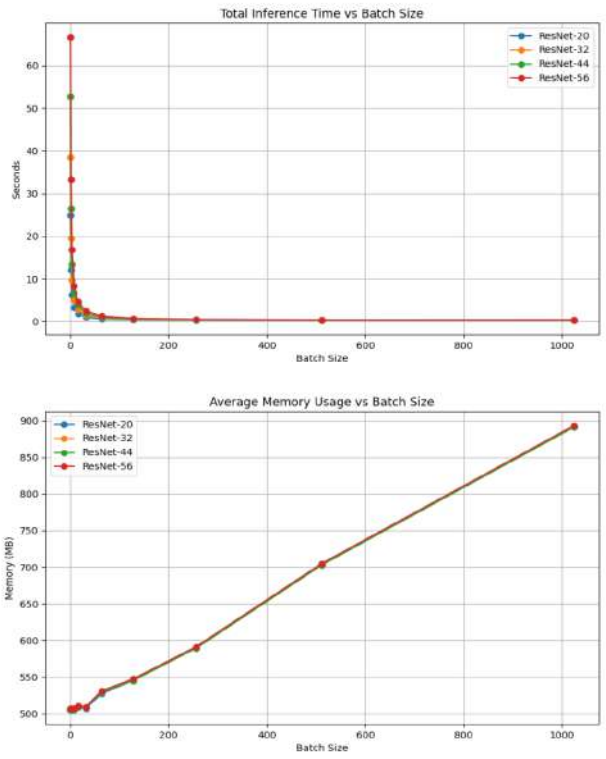

Fig. 1. Total inference time and peak GPU memory allocated for various ResNet models on CIFAR-10 across different batch sizes. Notice the initial drop in time and the steady increase in memory usage.

These results highlight a fundamental trade-off: larger batches make better use of the GPU's compute power but put more pressure on memory and can sometimes introduce unexpected scheduling overheads.

B. FP16 Arithmetic Optimization

We tested using FP16 precision by enabling PyTorch's 'autocast()' feature for a ResNet-110 model (which was originally trained with standard FP32 weights) doing inference on CIFAR-10.

- **Inference Time:** We actually observed a slight *increase* in the inference time at smaller batch sizes compared to using only FP32. This was a bit counterintuitive. It's likely because the overhead involved in converting data types and managing the mixed precision outweighed the computational speedup for these relatively small workloads on this dataset (see the top graph in Fig. 3). We'd expect the benefits of FP16 to be more significant for tasks that are heavily compute-bound or when running on hardware with strong FP16 support (like GPUs with Tensor Cores).
- **Memory Usage:** On the plus side, we saw a noticeable *reduction* in the average memory used, especially at larger

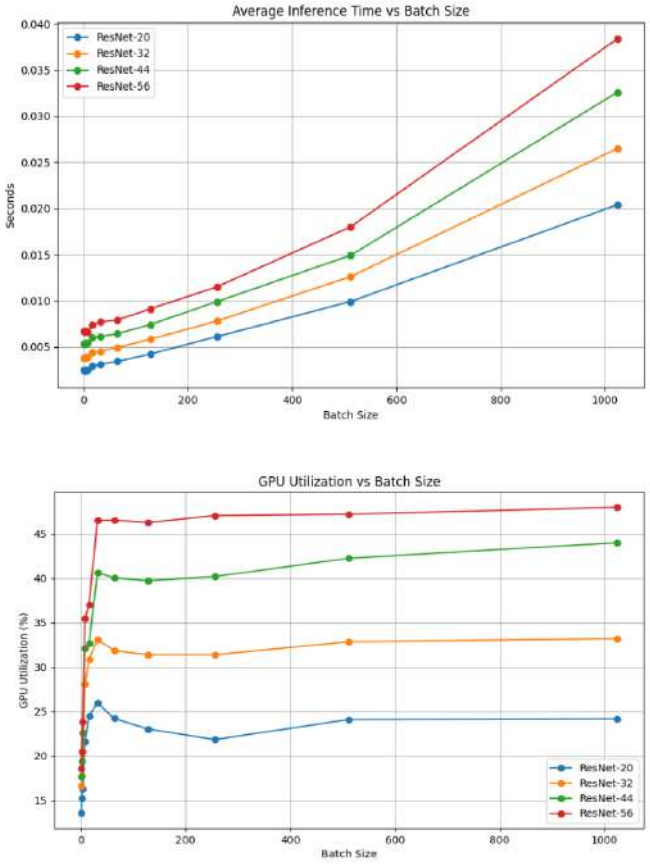

Fig. 2. Average inference time per batch and GPU utilization for ResNet variants on CIFAR-10. This shows how latency and utilization increase with batch size and model depth.

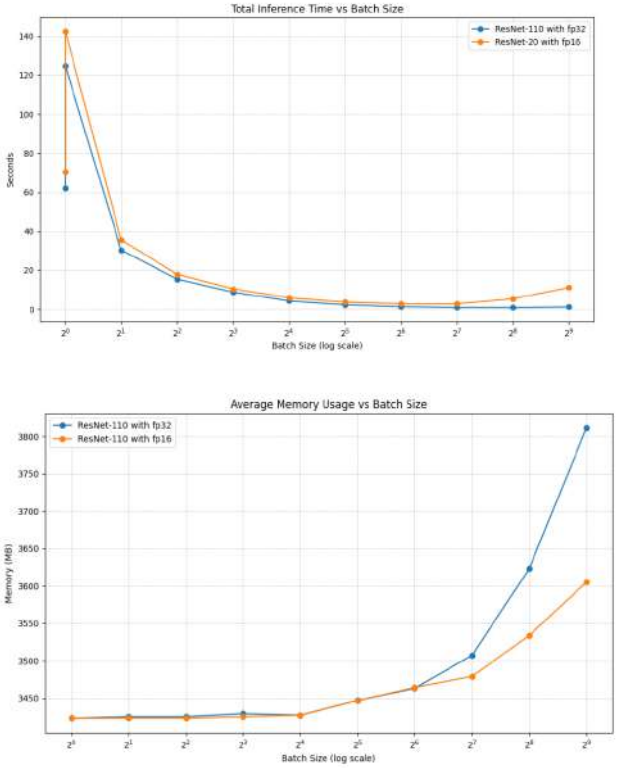

Fig. 3. Comparing inference time and memory usage for standard FP32 and FP16 arithmetic on CIFAR-10.

batch sizes. This is because the activation maps could be stored using half the precision (FP16 instead of FP32) (see the bottom graph in Fig. 3).

- **Accuracy:** Importantly, we didn't find any significant drop in test accuracy or increase in test loss compared to the FP32 baseline. This confirms that using FP16 was a viable option for this specific model and dataset without hurting the results (see Fig. 4).

C. AMP + AMC Optimization

Using a Tesla T4 GPU (which has the Turing architecture), we tested combining Automatic Mixed Precision (AMP) with Automatic Memory Coalescing (AMC). We ran this on several ResNet variants using a batch size of 1024.

- **Inference Time:** Similar to our pure FP16 experiment, we again saw a slight *increase* in the average time taken per batch compared to the FP32 baseline (see the top graph in Fig. 5). It seems that, for these specific models and this batch size on this GPU, the overhead of managing mixed precision dynamically, and perhaps less-than-perfect benefits from memory coalescing, still outweighed the gains from faster computation.
- **Peak Memory Allocated:** However, the benefit in memory usage was striking. We achieved a *drastic reduction* – around 45-50% – in the peak amount of GPU memory allocated across all the ResNet models we tested (see the bottom graph in Fig. 5). This is a huge win, as it frees up a significant chunk of GPU memory that could be used for other tasks or allow for running even larger models or bigger batches.

D. Memory Tiling Optimization

We explored memory tiling using a ResNet-20 model on the Mini ImageNet dataset. Since ImageNet has larger images (3x224x224), memory access patterns become more important. We used a fixed batch size of 8 and varied the tile size (trying 8x8, 16x16, 32x32, and 64x64 tiles).

- **Inference Time:** The time taken for inference dropped significantly as we increased the tile size. Using larger tiles allows the GPU to process more data efficiently within its faster memory levels (like shared memory or caches), meaning fewer slow trips to the main global memory (see the red line in Fig. 6).
- **GPU Utilization:** As the inference time dropped, the GPU utilization increased with larger tile sizes. This indicates that the compute resources were being used more effectively once the memory access bottleneck was eased (see the blue dashed line in Fig. 6).

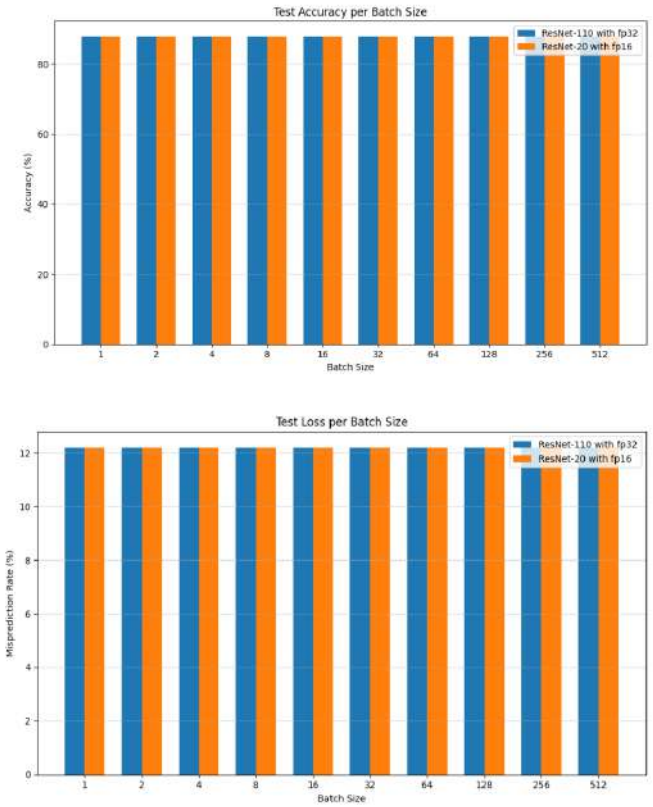

Fig. 4. Comparing test accuracy and loss for standard FP32 and FP16 arithmetic. The impact on accuracy is minimal.

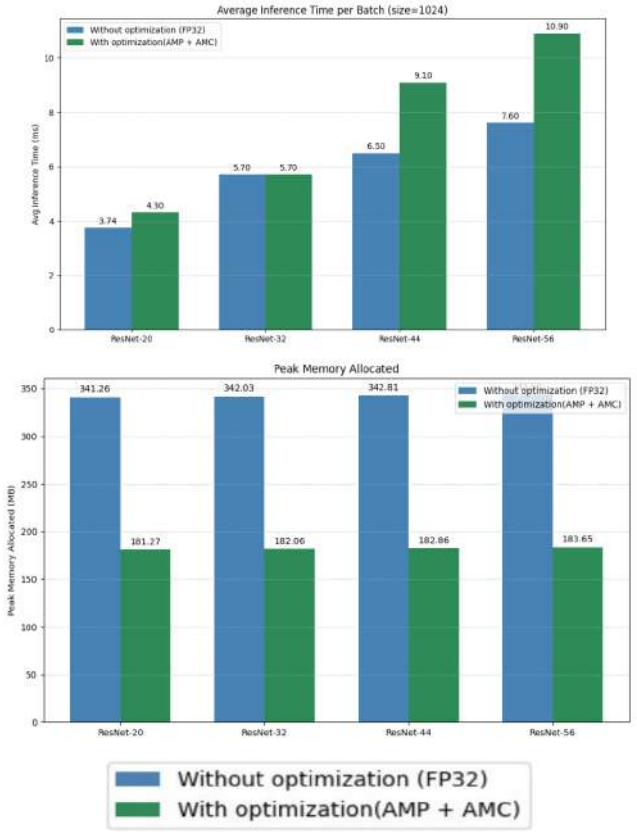

Fig. 5. Comparing inference time and peak GPU memory allocated using FP32 versus the combined AMP + AMC optimization on a Tesla T4 GPU.

- **Average Memory Usage:** We observed a slight increase in the average memory usage with larger tile sizes. This reflects the fact that larger tiles require storing more intermediate data in the faster (but limited) memory regions like shared memory (see the green dash-dot line in Fig. 6).

This experiment clearly demonstrates how effective tiling can be when memory access is the limiting factor, particularly with larger input data. However, it also highlights a trade-off: reducing latency through tiling comes at the cost of using more of the GPU's limited fast memory resources. Finding the right tile size is key.

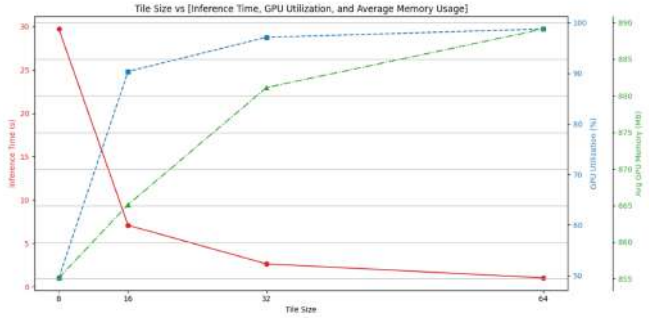

Fig. 6. How tile size affects inference time (lower is better), GPU utilization (higher is better), and average memory usage for ResNet-20 on ImageNet.

E. Validation against Published Results

Figure 7 shows the normalized execution times originally reported by Li et al. [5] for five complete CNNs—LeNet, CIFAR, AlexNet, ZFNet, and VGG—evaluated under six implementation strategies: *cuDNN-MM*, *cuDNN-FFT*, *cuDNN-FFT-T*, *cuda-convnet*, *cuDNN-Best*, and *Opt*. All bars are normalized to the *cuDNN-MM* baseline, so the vertical axis can be read directly as speed-up.

1) Legend and Axes: The horizontal axis lists the five CNN models, while the vertical axis shows speed-up relative to *cuDNN-MM*. Each color denotes one of the six schemes:

- **cuDNN-MM**: standard matrix-multiplication mode in cuDNN
- **cuDNN-FFT**: FFT-based convolution (with MM fall-back)
- **cuDNN-FFT-T**: FFT-tiling mode (with MM fallback)
- **cuda-convnet**: CHWN-based direct convolution
- **cuDNN-Best**: per-layer selection of the fastest cuDNN mode
- **Opt**: layout-aware, memory-optimized framework

2) Key Observations:

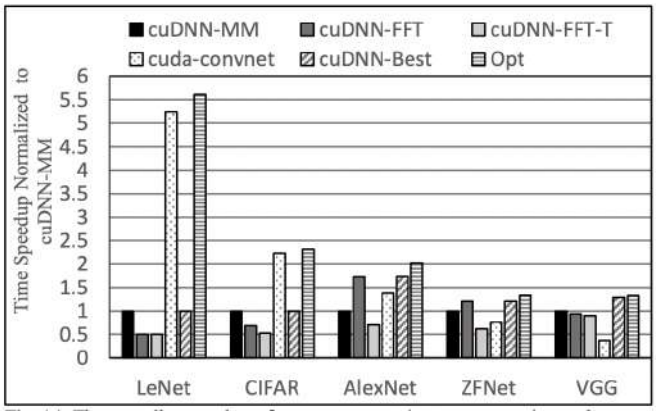

Fig. 7. Overall network-level speed-up reported in the literature.

1) *No single kernel wins everywhere. cuda-convnet* tops the tiny LeNet and CIFAR models, whereas cuDNN variants gain the edge once the networks scale up (AlexNet, ZFNet, VGG).
2) *cuDNN-Best*—which cherry-picks the fastest kernel per layer—narrows the gap, but still trails *Opt*.
3) *Opt* delivers the highest throughput across the board: **5.6**× on LeNet and roughly **2**× on AlexNet, ZFNet, and VGG, demonstrating its wide applicability.

3) Significance: These results emphasize that end-to-end CNN inference speed hinges on holistic memory-efficiency measures—namely adaptive data layouts and streamlined pooling/softmax accesses.

V. Conclusion

In this project, we took a close look at the memory bottlenecks that often plague CNN inference on GPUs and experimented with several software techniques to make things run better. By carefully profiling models like LeNet-5, AlexNet, and various ResNets with different batch sizes using tools like Nsight Compute, we gained valuable insights into their performance behavior. One key takeaway is that the choice of batch size has complex effects – it influences not just how busy the compute units are and how much memory is used, but also introduces system-level overheads, such as time spent waiting on semaphores.

We evaluated a few specific optimization strategies:

- **FP16 / AMP:** These techniques showed great promise for cutting down memory usage. When combined with Automatic Memory Coalescing (AMC), we saw peak memory allocation drop by as much as 50%. Although we didn't see faster inference times in our specific tests on CIFAR-10 (likely due to overheads outweighing benefits for those workloads), the model's accuracy remained intact. It is worth noting that the lack of observed speedup might be specific to the workloads and the range of hardware tested; architectures with more powerful Tensor Core implementations or more compute-intensive tasks could potentially realize significant latency reductions from FP16/AMP. This makes mixed precision a very attractive option, especially when GPU memory is tight.
- **Memory Tiling:** This proved very effective at reducing latency and boosting GPU utilization, particularly when dealing with larger input images like those in ImageNet. Tiling optimizes how data moves within the GPU's memory hierarchy. The catch is that you need to choose the tile size carefully, balancing the speed gains against the limited amount of fast memory available on the GPU.

Our results suggest that there's no single "magic bullet" optimization. The best strategy really depends on the specific situation – the model architecture, the GPU hardware being used, the chosen batch size, the characteristics of the input data, and whether the main goal is to minimize latency, reduce the memory footprint, or save energy. Furthermore, the significant system call overheads we observed remind us that optimization efforts should look beyond just the GPU kernel execution times and consider the entire system context.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems 25 (NIPS 2012)*, 2012, pp. 1097–1105. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: https://arxiv.org/abs/1409.1556
[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, June 27-30, 2016, Las Vegas, NV, USA*. IEEE Computer Society, 2016, pp. 770–778. [Online]. Available: https://doi.org/10.1109/CVPR.2016.90
[4] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," in *Proceedings of the IEEE*, vol. 105, no. 12, 2017, pp. 2295–2329.
[5] C. Li, Y. Yang, M. Feng, S. T. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on gpus," *CoRR*, vol. abs/1610.03618, 2016. [Online]. Available: https://arxiv.org/abs/1610.03618
[6] NVIDIA Corporation, "Nsight Compute: A Comprehensive Tool for Profiling GPU Applications," NVIDIA Developer Documentation, 2018-2024, version 2024.3.2.0 (build 34861637). Available: https://developer.nvidia.com/nsight-compute.
[7] NVIDIA, "cuDNN: GPU Accelerated Library for Deep Neural Networks."
[8] A. Maji, S. Reddy, U. Sharma, and A. Nokhwal, "Towards Efficient CNN Inference: Memory Profiling and Optimization - Phase 2 Report."
[9] H. Zheng, S. Oh, H. Wang, P. Briggs, J. Gai, A. Jain, Y. Liu, R. Heaton, R. Huang, and Y. Wang, "Optimizing memory-access patterns for deep learning accelerators," *CoRR*, vol. abs/2002.12798, 2020. [Online]. Available: https://arxiv.org/abs/2002.12798
[10] A. Maji, S. Reddy, U. Sharma, and A. Nokhwal, "Towards Efficient CNN Inference: Memory Profiling and Optimization - Phase 1 Report."