AutoVM: Accelerating convolutional neural network training with actively managed GPU virtual memory

by

Luyuan Chen

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

AutoVM: Accelerating convolutional neural network training with actively managed GPU virtual memory

Luyuan Chen

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2020

The size of neural networks a GPU can train is limited by the GPU's memory capacity. Although GPU virtual memory enables training arbitrarily large neural networks, such trainings are often accompanied by severe performance penalties. Furthermore, popular frameworks for constructing machine learning applications, like TensorFlow, have disabled using GPU virtual memory by default. We propose AutoVM, a software layer that can better manage GPU virtual memory in neural network training by incorporating our understandings of neural networks. AutoVM schedules data transfers between GPU and CPU memory to relieve the memory pressure on GPU; and in turn optimizes training speed. We have integrated AutoVM into TensorFlow such that existing machine learning applications can benefit from AutoVM with minimal effort. Our tests suggest that training VGG-19 using AutoVM can be at most 2.5× faster compared to using default Nvidia virtual memory.

Contents

1 Introduction **1**

1.1 Motivating example 3

1.2 Contributions 4

1.3 Outline of the Dissertation 4

2 Background **5**

2.1 GPUs 5

2.1.1 Hardware architecture 7

2.1.2 Programming model 11

2.1.3 Memory management 14

2.1.4 GPU performance issues 16

2.1.5 Accelerated libraries 17

2.2 Deep neural network 17

2.2.1 Layers in neural networks 19

2.2.2 Inference and Training 22

2.2.3 Gradient descent 23

2.2.4 Workload in CNN training 28

2.3 TensorFlow . . . 28
2.3.1 Programming and execution model . . . 29
2.3.2 Computation graph . . . 31
2.3.3 Execution order . . . 31
2.3.4 GPU support . . . 32

3 Design **33**

3.1 Motivation and Problem Statement . . . 34
3.2 Design Overview . . . 36
3.3 Policy . . . 37
3.3.1 Identifying tensors to move . . . 37
3.3.2 Identifying when to transfer tensors . . . 40
3.4 The mechanism . . . 45
3.5 Limitations . . . 46

4 Reverse engineering Nvidia virtual memory **48**

4.1 `cudaMemPrefetchAsync()` v.s. `cudaMemAdvise()` . . . 49
4.1.1 Method . . . 50
4.1.2 Findings . . . 51
4.2 Efficient memory transfer between devices . . . 51
4.2.1 Overlapping memory transfer with computation . . . 51
4.2.2 Alternating the launch order . . . 53
4.2.3 Avoiding page faults . . . 54
4.2.4 AutoVM and pre-access . . . 55

4.3 Throughput of `cudaMemPrefetchAsync()` transfers . . . 57

5 Implementation **58**

5.1 Overview . . . 58

5.2 The policy . . . 60

5.2.1 Integrating AutoVM . . . 61

5.3 The mechanism `MemOp()` . . . 62

5.3.1 Adding support for `cudaMemPrefetchAsync()` . . . 63

5.3.2 Accessing `cudaMemPrefetchAsync()` from an operation . . . 63

5.4 Supporting pre-access . . . 64

6 Experiment **65**

6.1 Environment setup . . . 65

6.2 Experiment design . . . 65

6.2.1 Experiment code . . . 66

6.2.2 Test cases . . . 67

6.2.3 Data collection methods . . . 69

6.3 Results . . . 70

6.3.1 AlexNet . . . 70

6.3.2 VGG-19 . . . 74

6.3.3 ResNet-152 . . . 75

6.3.4 Full Training Run Experiment . . . 76

6.3.5 Summary of results . . . 78

6.4 Discussion . . . 79

6.5 Future improvements 79

6.5.1 Further optimizing AutoVM 80

7 Related work **82**

8 Conclusion **85**

Bibliography **86**

Chapter 1

Introduction

GPUs have played a major role in catapulting machine learning from largely theoretical concepts to practical solutions capable of solving a variety of non-trivial problems. Machine learning computations are dominated by matrix and vector operations that can only be executed relatively slowly on traditional CPUs. Meanwhile, GPUs were initially designed and optimized for graphical processing, which also primarily involves matrix operations. Thousands of simple processing units within GPUs allow parallel computation of such matrix and vector operations and thus execute such operations far more quickly than CPUs. Because of the similarity of computations between graphics processing and machine learning, and the speed at which they can execute these computations, GPUs have become the dominate infrastructure for machine learning.

GPUs are CPU-controlled co-processors that have their own memory system, called *global memory*, which is optimized for highly-parallel access from their compute cores. DMA (direct memory access) engines on the GPU are responsible for transferring data between CPU and global memory over a PCI-Express (or NVLink) interconnect. One key issue for many computationally intensive application running on GPUs is the fact that the size of GPU memory is fixed and typically much smaller than CPU memory (referred to as *host memory* in GPU terms) sizes.

In this dissertation, we identify methods to manage global memory to support running GPU applications efficiently with memory requirements larger than what is physically available on the GPU. We specifically target convolutional neural networks (CNNs) that have been successfully applied to numerous applications, including image classification, video analysis, and action recognition [25]. Over the last

decade, CNNs have been growing deeper with more layers and thus higher demands for compute and memory resources. For example, the winner of 2012 ImageNet Large-Scale Visual Recognition Challenge (ILSVRC), AlexNet [13], had 5 convolution layers and 2 dense layers. The winner of the 2015 contest, ResNet-152 [7], had over 150 convolution layers. The amount of available memory on GPUs, on the other hand, has not grown as fast as CNN memory requirements. For example, Google's *inception v4* [22] model has 515 layers, consuming over 80GB of global memory during training with 64-image batches.[1] To date, no commercially available one-GPU setup has enough GPU memory to train this network. Consequently, these kinds of networks can only be trained by using expensive multi-GPU setups.

Modern Nvidia GPUs support virtual memory and paging. Virtual memory support allows applications to run even if they use more memory than what is physically available. Despite the convenience provided by virtual memory, the default GPU virtual memory management policy is unaware of the workload being processed on the GPU. As a result, GPU virtual memory is managed in a suboptimal way with an attendant performance penalty.

Our work explores efficient GPU virtual memory management for CNNs. In particular, we propose AutoVM that smartly manages the GPU virtual memory in CNN training. AutoVM is a software layer that manages GPU virtual memory actively and transparently to the CNN application. Specifically, AutoVM offloads tensors without immediate reuse to host memory, and prefetches offloaded tensors back to global memory prior to their consumptions.

The computations in CNNs are often represented by a data flow graph (computation graph) in machine learning frameworks like TensorFlow [2], Torch [5] and MxNet [3]. AutoVM analyses the computation graph, identifies the data reuse patterns, and identifies the places in the computation graph to insert the offload and prefetch commands. Once AutoVM inserts those commands at appropriate locations in the computation graph, tensor offloads and prefetches are carried out automatically as the computation graph executes. We have also designed an interface that allows machine learning engineers to enable AutoVM with any existing machine learning application by changing only one line of code.

We have integrated AutoVM into TensorFlow to verify its effectiveness in a fully-functional machine learning framework. Our experiments show that our method can achieve a speedup of up to 2.7× in VGG-19 training, over Nvidia's default GPU memory management policy.

[1] Training is the iterative process where the neural network 'learns'. In each iteration, a batch of images is processed. Successful training requires moderate batch sizes — typically 32 or higher — to be used.

1.1 Motivating example

CNNs have been widely adopted in applications that perform image analysis. A CNN image classifier takes images as input and outputs categorical data that correspond to the classes of subjects present in the input images, for example, "dog", "cat", or "aeroplane". A typical CNN consists of a series of different *layers*. A layer l takes in the input image $\mathbf{x}^{(l)}$, processes it using its set of *weights*, $\mathbf{w}^{(l)}$, and outputs the resulting image[2] $\mathbf{y}^{(l)}$. The output is passed on as input to the next layer $l+1$ as $\mathbf{x}^{(l+1)}$.[3] The image-like output from the last convolution layer is flattened to a vector and is then weighted in a fashion similar to the weighted average, to produce the final categorical output. The process in which input travels from the first layer to the last is called *inference*.

Before a network can be used for inference, however, it has to be *trained*. During training, a *loss function* compares the network output with known correct values to measure the network's performance and produces a *loss value*. All layers' weights are then updated to minimize the loss value and in turn improve the network's performance. The update process of each layer's weights may require the layer's inference output, so layer l's inference output is kept in memory until layer l's weights are updated in training. These previously generated outputs typically occupy a significant amount of memory. For example, in VGG-216 [19], these outputs from a 32-image batch consume around 30GB of memory, constitute over 85% of the total memory usage. Furthermore, training is processed in the opposite order of inference, where the first layers processed in inference are visited the last in training. As such, the time gap between the processing of a layer's inference and training could be significant (hundreds to thousands of milliseconds). Consequently, a substantial amount of data are kept in global memory for an extended amount of time, despite not being actively used.

While the default virtual memory management policy available on modern GPUs can handle workloads that use more memory than what is physically available. The fact that the virtual memory system lacks the domain knowledge of the workloads often encompasses poor memory management decisions. For example, when convolution layer l's output $\mathbf{y}^{(l)}$ is no longer needed in inference, it can be paged out immediately to free up global memory. However, the fact that Least-Recently-Used (LRU) is the default eviction policy means that even if $\mathbf{y}^{(l)}$ is the optimal page out candidate, $\mathbf{y}^{(l)}$ would not be selected for page out as it was recently referenced.

[2] Image-like data to be precise. The result, like images, has three channels, but the dimensionality of the third channel might not be three.

[3] Although $\mathbf{y}^{(l)}$ and $\mathbf{x}^{(l+1)}$ refer to the same piece of data, it is referred to as layer l's output $\mathbf{y}^{(l)}$ below.

Instead, we propose AutoVM, which smartly chooses data to page out and when. This frees up physical global memory for data that is not being actively accessed. Paged out data are later prefetched into global memory before being referenced in training. Integrated into TensorFlow, our method automates the entire process of data selection for paging so that virtual memory is used more efficiently.

1.2 Contributions

This dissertation makes the following contributions:

1. We reverse engineer Nvidia's GPU virtual memory system to reveal some of its performance characteristics,
2. We design an active GPU virtual memory management policy — AutoVM, to accelerate CNN training on memory-limited GPUs, by analyzing the computation graphs of CNNs and scheduling tensor transfers and
3. We integrate AutoVM in TensorFlow, a widely used, industrial standard framework, and verified AutoVM is able to deliver non-trivial performance improvement comparing to using Nvidia's default memory management policy.

1.3 Outline of the Dissertation

The remainder of this dissertation is structured as follows. Chapter 2 provides necessary background material on GPUs, neural networks, and TensorFlow so that the remainder of the dissertation can be understood. Chapter 3 describes the design of AutoVM. Chapter 4 presents the results of experiments for the purpose of understanding how Nvidia's managed memory subsystem behaves in practice. This was necessary because Nvidia does not provide any documentation or other information that provides insight of this. The results of these experiments influenced our design and implementation of AutoVM. Chapter 5 presents the implementation of AutoVM. Chapter 6 presents the results of our experiments to evaluate AutoVM. The experiments show AutoVM is able to speed up VGG-19 training by 150.3%, compared to using default Nvidia virtual memory subsystem. We close with concluding remarks and possible future work in Chapter 8.

Chapter 2

Background

In this chapter, we present background information on GPUs, deep neural networks (DNNs) and TensorFlow so that the reader can better understand the remainder of this dissertation.

2.1 GPUs

GPUs are highly parallel co-processors, specialized and optimized for graphical processing workloads that involve an extensive amount of parallelizable and in-expensive operations. GPUs have thousands of independent but simple processing cores to process such workloads efficiently.

GPU cores are simpler than CPU cores in that they lack several architectural features that contribute to the performance of CPU cores, such as large, per core caches, branch predictors and complex instruction pipelines. Although a GPU core is weaker than a CPU core, a GPU can support a number of cores that far exceeds the number that CPUs can support. As a result, although per-core performance is lower, parallelizable programs can be executed much faster on a GPU if the program can exploit the many cores available. As co-processors, GPUs have their memory hierarchies separate from the memory hierarchy of the CPU.

GPUs are controlled by programs running on the CPU while having a distinct programming model. Nvidia GPUs use a programming model called CUDA (Compute Unified Device Architecture), which extends the C/C++ programming language to enable general-purpose GPU programming in a familiar

environment.[1] However, despite CUDA being an extension of C/C++, writing high performance GPU programs is not trivial.

In contrast to CPU architectures that tend to be backwards compatible, GPU architectures can vary quite a bit across generations, and each generation is identified by its *compute capability*, a number. Hence code optimized for one generation is usually not portable to a different generation. For example, devices with compute capability 6.0 and higher have virtual memory support built-in, while those under 6.0 do not. In this discussion we assume capability 7.5 of the *Turing* architecture [15], if not specified otherwise.

The remainder of this section will outline the details of

- GPU hardware architecture,
- software programming model,
- GPU memory hierarchy, and
- libraries for accelerated computing

The main take away is that GPUs are not trivial to program if performance is the key objective. And it is unreasonable to expect those working on machine learning applications to optimize GPU programs for performance, given the complexities of GPUs. Hence, those in the machine learning community rely on libraries that hide the intricacies of GPUs from the machine learning application developers. For the same reason, our objective in this work is to hide the complexities of memory management from the application developers in a software layer that is mostly transparent to developers.

[1] Our description here is limited to Nvidia GPUs, on which this dissertation is based.

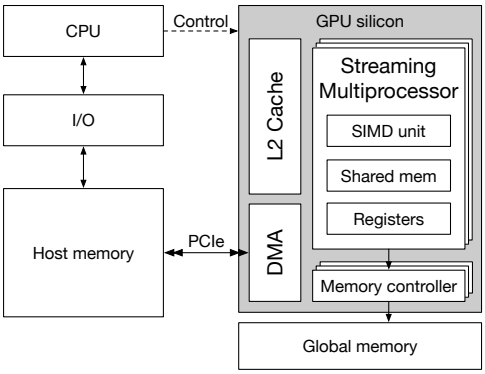

Figure 2.1: High-level GPU architecture.

2.1.1 Hardware architecture

The general architecture of a GPU is shown in Figure 2.1. CPU and GPU have separate off-chip DRAM referred to as *host* memory and *global* memory,[2] respectively. GPU also contains a shared L2 cache accessible to all processing cores. The host and the device are connected via an interconnect such as PCI-Express or NVLink, over which GPUs' DMA (direct memory access) engines transfer data between host memory and global memory. Higher-end GPUs are equipped with two DMAs to support simultaneous bi-directional transfers.

GPU cores are organized into multiple streaming multiprocessors (SMs), each containing tens to hundreds of cores. Each SM also contains a SM-local register file and a small amount of shared memory that is accessible only to the cores within the SM. Figure 2.2 depicts the schematics of Nvidia's TU102 GPU from the Turing family.[3]

2.1.1.1 Streaming multiprocessor

As shown in Figure 2.3, a Turing SM divides the following resources into four partitions:

- 64 CUDA cores, each with one `int32` unit and one `float32` unit.
- 4 groups of special function units (SFUs) that handle specific maths functions,
- a 64K 32-bit register file,
- a 96KB combined shared memory/L1 cache,
- 4 instructions schedulers.

[2]This GPUs' on-board memory is also referred to as the device memory. We will use the term global memory in this dissertation.

[3]Turing is the newest GPU architecture, as of Aug 2019.

Figure 2.2: Schematics of the TU102 GPU. All SMs share the L2 cache located in the center. Green blocks portray SIMD cores within SMs.

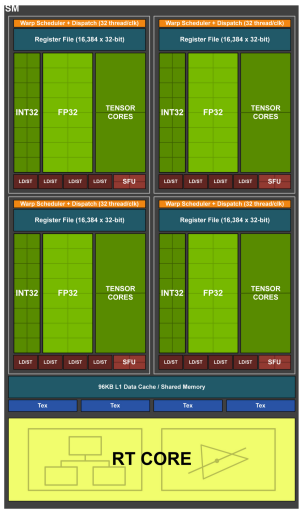

Figure 2.3: Structure of the Turing streaming multiprocessor.

These resources are local to an SM, and are not shared with other SMs in the GPU. Each SM also has one RT core that is used for ray tracing, but this is irrelevant to this project.

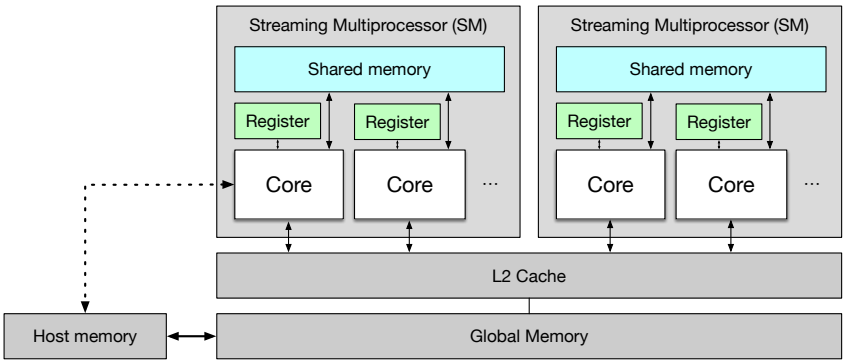

Figure 2.4: CUDA memory hierarchy.

2.1.1.2 Memory hierarchy

Data accessed by GPU cores can reside in a number of different types of memory. Table 2.1 summarizes the performance characteristics of the memories available on Nvidia Turing GPUs.

Local variables are typically stored in registers that offer very fast access. GPUs have many more registers than CPUs. Nevertheless, when a thread uses too many local variables, the register contents get spilled over into L1 and L2 caches and ultimately, global memory.

Each SM also has a fast (around 19 cycles) local scratch pad memory called *shared memory*. The key challenge with this type of memory is that it is relatively small (at most 96KB per SM) and is entirely managed by the programmer. Programmers have to decide which data resides there and when, and explicitly copy data into and out of shared memory.

Otherwise, all data reside in global memory, which can vary in size; for instance our GPU has 11GB of global memory while the highest-end GPU has 48GB.[4] Access to global memory is significantly slower than that of shared memory: an access to global memory typically takes over 250 cycles to complete.

GPU hardware uses three approaches to mitigate global memory access latencies. First, 32 words can be accessed in parallel from 32 adjacent cores, if the 32 words fall onto an aligned, continuous region. Such an access pattern is referred to as *coalesced access*. It is thus beneficial for the programmer to structure a program's data so that accesses to the data occurs in a coalesced fashion. Otherwise, access latencies can be significantly larger — in the worst case, accesses latencies to randomly located data will be 32 times higher than coalesced accesses.

[4]Quadro RTX6000 is the highest-end GPU listed for sale, as of the writing.

	Size	Performance	
		Access latency (cycles)	Bandwidth (GB/s)
L1 cache	64 KB (per SM)	32	177
L2 cache	4 MB	200	2,155
Shared memory	32 or 64 KB (per SM)	19	13,800
global memory	11 GB	300	616

Table 2.1: Access latency and bandwidth of memories available on Nvidia Turing GPUs.

Secondly, GPUs can perform context switches very quickly — i.e., typically in one cycle. As such, if a thread issues a read to global memory, a context switch occurs immediately to allow another thread to run while the first accesses global memory. This implies that the programmer will need to structure their code to use many more threads as there are cores to ensure there always are ready thread blocks to schedule.

Finally, a GPU has multiple caches for faster accesses: L1 caches, (96 KB per SM) are owned privately by each SM; a single L2 cache (4 MB), with access latency around 200 cycles, is shared between all SMs onboard.[5] The caches are managed by the hardware transparent to the applications. However, it should be pointed out that L1 caches are typically disabled for program data on modern GPUs.[6]

2.1.1.3 Hardware thread execution

GPU programs are expected to be structured to execute with many thousands of threads. Once a thread starts executing on an SM, it will remain on that SM for the duration of its life cycle; i.e., it will never be migrated onto a different SM.

The hardware executes groups of 32 threads in lock step on adjacent GPU cores. That is, threads [n **mod** 32] to [$(n+31)$ **mod** 32] execute the same set of instructions at the same time in SIMD (single instruction multiple data) fashion. Such a group of 32 threads is called a *warp*. A warp is the basic unit of scheduling and context switching. This makes the hardware more efficient because it allows 32 cores to share a single instruction stream.

Different threads in a warp may take different branches, and if they do, it is called *thread divergence*. Thread divergence causes execution to be slower because all threads in a warp share the same instruction stream. For example, in code executing `C = (cond) ? A() : B()`, when some threads in a warp

[5] The Quoted amounts are based on Turing architecture GPUs.

[6] This is because L1 cache and shared memory are implemented using the same memory device. CUDA applications typically use that entire space for shared memory, thus there is no space to support L1 caching.

execute `A`, the other threads are blocked and then continue to execute `B` when `A` has completed executing. When `B` is being executed, the `A`-executing threads will be blocked.

2.1.2 Programming model

Since GPUs are CPU-controlled co-processors, all general-purpose GPU programs consist of two parts, namely:

1. *Kernels*, which are CPU-invoked GPU functions running in parallel by multiple GPU threads,
2. CPU code that issues instructions to CUDA drivers to control GPU behaviour, such as kernel launches, memory allocations and deallocations, memory copies, and synchronization.

To illustrate, a program that does matrix multiplication, $C = A \cdot B$, involves the following steps, assuming matrices A and B reside in host memory:

1. CPU instructs memory be dynamically allocated in global memory;
2. CPU initiates the data transfer of A and B from the host memory to global memory;
3. CPU launches a matrix multiply kernel on the GPU;
4. GPU executes the matrix multiplication kernel using its thousands of cores;
5. CPU initiates the transfer of result C to the host memory and the deallocation of the memory buffer in global memory.

In the example above, only step 4 executes on the GPU. However, if step 4 takes a significant proportion of run time and the GPU executes matrix multiply much faster than the CPU, offloading the matrix multiply onto the GPU can boost performance considerably.

In the next sub-sections, we describe kernel launch, thread hierarchy, scheduling, and streaming.

2.1.2.1 Kernel launches

Launching a GPU kernel takes the form of an ordinary function call with additional *launch parameters*. Unlike a regular function invocation, CUDA kernel launches always return void, and always return immediately. That is, kernels execute asynchronously in parallel to subsequent CPU program execution. The extra launch parameters specify how the launching kernel should be executed. The two required parameters describe how many threads the kernel should execute with as well as how they are organized, something referred to as the *thread hierarchy*.

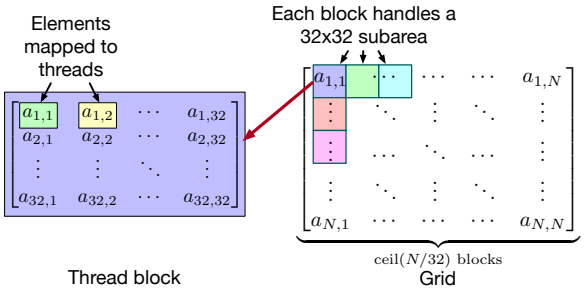

Figure 2.5: Thread hierarchy mapped to a matrix addition problem.

2.1.2.2 Thread hierarchy

Threads must be logically arranged in 1D, 2D or 3D *blocks* by the programmer. The Nvidia architectures released so far limit the maximum number of threads in one block to 1,024.

Blocks are logically organized into 1D, 2D or 3D *grids*. The limit on the number of blocks in a gird is quite high to support mapping large problems.[7] Threads can locate their location within the blocks and grids by using index values provided by the CUDA runtime. When a kernel is launched, each thread executes the same kernel and can determine what data to process based on its position within the thread hierarchy. Figure 2.5 depicts a possible thread hierarchy for matrix addition: every thread is assigned to calculate one element $A_{i,j} = B_{i,j} + C_{i,j}$ in the resulting matrix. Since up to 1,024 threads can be in one block, the programmer could appoint each block to process a 32×32 sub-matrix of A. As such, one $N \times N$ matrix is processed with N^2 threads, arranged in a $(\lceil N/32 \rceil, \lceil N/32 \rceil)$ grid of blocks.

2.1.2.3 Scheduling on GPUs

Each block is dispatched to a streaming multiprocessor that has sufficient resources available to execute the block. Once a block is dispatched to a SM, it remains on that SM until it finishes. Blocks will be queued when no SM has sufficient resources to support the execution. At every issue cycle, warp schedulers in an SM select warps for execution, whose resource or data dependencies are satisfied.

2.1.2.4 Stream interface

Abstractly, streams are GPU task queues on which CPU-submitted CUDA tasks are queued. A CUDA program can allocate multiple streams and queue tasks onto different streams.[8] But the program does not have direct control over whether tasks are executed concurrently or not. Launching

[7] $2^{32} - 1$ blocks in the x-dimension and $65,536$ blocks in the y and z dimensions.
[8] The default stream is used, if no target stream is specified when making CUDA calls.

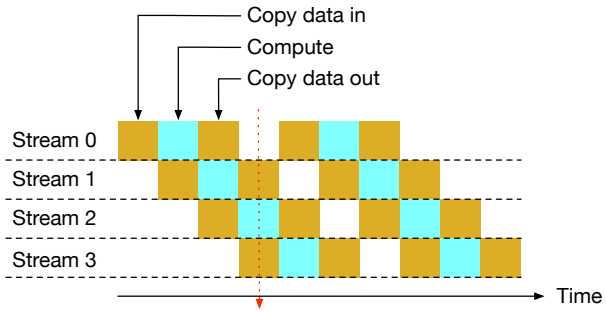

Figure 2.6: Illustration of pipelining using four streams.

tasks onto different streams is a necessary but not sufficient condition for the tasks to be executed in parallel on GPU.

Streams are mainly used to implement piplining to achieve parallel data transfer and computation. Figure 2.6 depicts a pipeline using streams: yellow blocks represent memory copies while cyan ones represent computations. Each stream repeatedly copies data in from the CPU, does the computation, and copies the data out. The memory copies and computation tasks are scheduled in a fashion so that DMA engines and processing cores are kept busy at all times. For instance at the beginning, after stream 0 copies its data to global memory and starts its computation, stream 1 starts using the DMA engine for copying its data (shown as the first yellow block on the second row). Ideally, after stream 1 has finished copying, the computation of stream 0 also finishes, so that stream 1 can start its calculation. This process is repeated such that the compute cores are continually running at full speed as if no memory transfers have taken place.

2.1.3 Memory management

Prior to CUDA 6.0, hosts and devices could only access data that are stored in their respective memory devices. For example, SMs could only access data stored in global memory but not in host memory. CUDA 6.0 introduced *unified memory*[9] that relaxed this limitation and allowed data to be accessible from any processor in the system, regardless of the data's storage location. However, if the data is not in the memory of the accessing processor, it is transferred to the memory of the accessing processor. For instance, if a GPU program accesses a piece of memory that resides in host memory, but not in global memory, CUDA runtime will copy the data to global memory via the interconnect.

Newer GPUs starting from compute capability 6.0, with CUDA 8.0, are equipped with full virtual memory support, with something Nvidia calls "*managed memory.*[10]" Nvidia also allows users to guide the virtual memory system to achieve more efficient memory management. This section provides background on GPU virtual memory system. Details on the older GPUs that do not support virtual memory are omitted from the discussion as they are not the primary concern of this work.

2.1.3.1 Virtual memory support

GPUs with compute capability 6.0 and higher, have full paging capability and support 49-bit virtual address translation. Memory pages[11] can physically reside any host memory or global memory and be migrated to any device (CPU or GPU) upon request. For example, SMs on a GPU can access data stored in host memory through demand paging. Like CPU virtual memory, CUDA applications can access a memory space that is much larger than what is physically available on the GPU, something often referred to as *memory over-subscription*. In addition to the default memory management policy, users can use *advise* and *explicit prefetching* to guide the runtime to manage GPU virtual memory better.

2.1.3.2 User control of virtual memory

As with CPU virtual memory, the keys to good memory performance include preventing page faults and keeping data local to the accessing processor. The default memory management policy often makes suboptimal paging decisions because it has no understanding of the running workload and relies only on

[9] CUDA version refers to the runtime version. Unified memory is enabled by the CUDA runtime as a software functionality that does not require extra hardware support.

[10] Unlike the unified memory, managed memory support requires extra hardware, represented by the newer compute capability.

[11] The page size is variable and cannot be set by users.

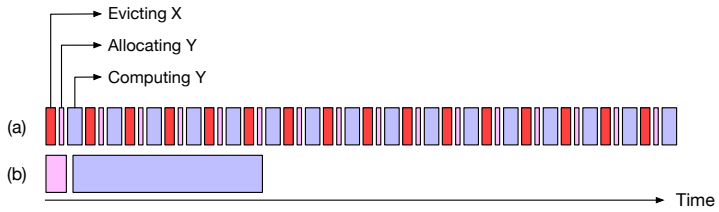

Figure 2.7: Example of computing with and without explicit memory control. (a) shows the case where X is not explicitly offloaded and no free physical page is available to accommodate the newly produced data. To allocate a physical page for Y, a page of X needs to be evicted from physical global memory. The corresponding computation cannot start until X's page is evicted and Y's page allocated. This process is repeated for every page of Y. While in case (b), if X is explicitly transferred out, then the required physical pages of Y can be allocated quickly and Y's computation can proceed non-stop. As a result, the time taken in case (b) is much shorter than (a).

a general heuristic. Nvidia provides two runtime library functions, that allow the program to guide the default memory management policy: `cudaMemAdvise()` and `cudaMemPrefetchAsync()`.[12]

- `cudaMemAdvise()` is used to provide a hint to the virtual memory system, how and from where target data will be accessed.
- `cudaMemPrefetchAsync()` prefetches data to one of the memories (host memory or global memory) by initiating an asynchronous data migration. Prefetching can be used to improve data locality and avoid page faults. Data transfers are asynchronous: memory transfers execute in parallel with computations to hide the transfer latencies. In practice, this runtime library function call can be either used to evict data from global memory or prefetch to global memory.

2.1.3.3 Benefits of user control

It is beneficial to manually control memory when the CUDA application uses more virtual memory than the physical global memory. For example consider the scenario in Figure 2.7: suppose data X was just produced in global memory, and will not be reused in the near future. Meanwhile, a computation kernel is producing data Y, but there is not enough memory to store both X and Y physically in global memory.

1. If X had been previously migrated out from global memory to host memory, the physical pages storing X in global memory can be freed to make space to store Y's pages physically in global memory allowing the computations to proceed at full speed. This scenario is shown in row (b) of Figure 2.7.

[12]Suboptimal invocations of advise and prefetch may negatively affect performance.

2. Otherwise, physical pages have to be evicted to host memory, (red-coloured blocks show the time used for evicting pages) to make space for the newly generated data, a page at a time. In particular, the GPU is performing the following steps repeatedly for every page of Y when first accessed:
   - a page fault occurs since it does not in global memory,
   - a victim page for eviction to host memory, needs to be selected because no free physical page is available,
   - the data stored in that page is transferred to host memory,
   - the evicted physical page is deallocated, and allocated a page for Y, and
   - the computation continues.

   Every step blocks the next; as a result, the performance is considerably worse compared to the first case where no eviction is needed during the production of Y.

Although X would be the best candidate for eviction since it will not be accessed in the near future, it is unlikely to be selected for eviction by the default memory management policy, as it was recently referenced. Hence, manually controlling memory by issuing an eviction request for X will benefit performance when the global memory is over-subscribed.

2.1.4 GPU performance issues

GPUs are not trivial to program if good performance is the objective, because of the complex architectures of GPUs. In particular, some of the following issues need to be taken into account:

Coalesced memory access Global memory accesses need to be coalesced to attain high memory throughput. That is, threads in a warp need to access memory addresses that fall into aligned and continuous 32, 64 or 128 bytes regions.

Thread divergence Although the thread execution model allows the threads in a warp to take different branches, divergent threads can negatively impact performance by up to a factor of 2.

Shared memory Programs often use shared memory as scratch pads, and the shared memory is managed by the programmer. The shared memory needs to be used wisely as it is limited in size. Furthermore, access latencies of shared memory will be suboptimal if threads in one warp access different addresses within the same shared memory *bank*. Such behaviour is called shared memory bank conflict and should be avoided.

Occupancy Occupancy measures the level of resource usage in SMs. As explained in §2.1.1.2, global memory access latencies of a warp can be hidden by executing other warps' computations. It is

thus often beneficial to use many threads so that when some access memory, there are other threads to schedule to hide global memory access latency. However, the more resource (registers or shared memory) a thread uses, the fewer threads can reside in a SM. This means the programmer has to be using the resources wisely so that there are enough threads in a SM to hide global memory access latencies.

The bottom line: a systematic understanding of the underlying GPU architecture is crucial for writing high-performance GPU programs. As a result, libraries that hide the system's intricacies are typically preferred. Machine learning engineers are typically not system experts, and hence they cannot be expected to write well-performing CUDA applications. As such, they rely on libraries that were written by system experts.

2.1.5 Accelerated libraries

Nvidia has released accelerated libraries that contain highly optimized functions useful in a variety of domains including signal processing, scientific computing and deep learning. These library functions do not require the machine learning engineers to launch CUDA kernels (that requires the additional launch parameters) but instead offer wrapper functions that invoke appropriate kernels. In fact, every library function is backed by multiple implementations that are optimized for specific configurations and the appropriate implementation is invoked. For example, the function for matrix multiplication has multiple implementations, optimized for different matrix shape configurations. On invocation, the library runtime applies heuristics to determine the implementation that best suit the matrix shape configuration.

2.2 Deep neural network

Neural networks, inspired by brain neurology, are a subset of machine learning algorithms that can "learn" from examples and perform specific tasks on new data. For example, a classifier can learn to categorize architectural styles of houses by analyzing example images that have been manually labelled; e.g., *Russian*, *gothic* or *baroque*. The classifier learns the traits specific to each of the architectural styles without requiring any prior knowledge of architecture or manual instructions.

Neural networks are built by arranging *layers* in specific orders. A *"layer"* is an abstraction of a math operation that transforms its input in a specific way. Each layer uses *weights*[13] to affect the

[13] A few types of layer do not use weights.

Symbol	Usage	Symbol	Usage
x	Input	k	Current output channel
y	Output	K	Total output channels
w	Weight	p	Current output height position
d or ∂	Derivative/Gradient	q	Current output width position
$f^{(l)}$	Layer l's math function	$\mathcal{L}$	Loss function
α	Learning rate	$\overline{x}$	Gradient $\partial\mathcal{L}/\partial x$
N	Batch size	i, j	Indices
H	Input image height	l	Layer number
W	Input image width	$\mathbf{\Theta}$	Set of all trainable weights
C	Total input channels	**Bold type**	N-dimensional array (or tensor)

Table 2.2: Mathematical symbols used in the dissertation.

transformation. Inputs and outputs of neural networks are also task-specific. For example, an image classifier takes in images and outputs the classes of objects in the images that were input. Specifically, the house style classifier outputs (for each image input) a 3-element vector, with each element equal to the predicted probability that the input image contains Russian, gothic or baroque architecture.

A neural network has to be trained before it can be used for *inference*. Inference is the process where inputs propagate through the layers to produce the network output. *Training* refers to the process in which the network corrects its weights by taking into account mistakes it has made during the previous inferences.

Deep neural networks (DNNs) refer to neural networks that have many layers. Our work focuses on convolutional neural networks (CNNs), a subset of DNNs, that have been widely applied in analyses of visual imagery.

In this section, we describe the mechanics and workload of CNNs. Specifically, the following subsections provide details on

- types of layers used in CNN and their performance characteristics,
- the inference and training process, and
- data flow in CNN training.

Table 2.2 summarizes the symbols used in our formulas[14] in the text that follows.

[14]In the dissertation, bias terms are often omitted for simplicity, we assume that the data and weight tensors are augmented to include the effect of bias terms.

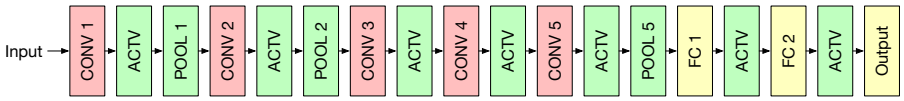

Figure 2.8: Layer structure of AlexNet. The colours in the figure represent the estimated run times of layers: red:long, yellow:medium, green:short.

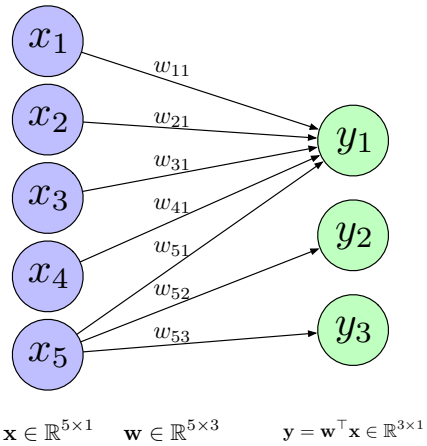

Figure 2.9: Illustration of a linear layer that takes five input and outputs three weighted results.

2.2.1 Layers in neural networks

A layer abstracts a mathematical operation performed on its inputs; its output is passed to subsequent layers as inputs, for further processing. Figure 2.8 depicts a typical CNN with four types of layers that are used in CNNs. The following subsections describes these layer types, as well as their performance characteristics.

2.2.1.1 Fully connected layers

A fully connected (FC), or linear layer, takes an input and applies a weight function to produce an output. Figure 2.9 illustrates a fully connected layer that takes five inputs and produces three outputs (some edges are omitted for cleanliness). In the example, the layer's input is a 5×1 vector $[x_1, \ldots, x_5]^\top$, which is weighted by a 5×3 *weight matrix* $\begin{pmatrix} w_{11} & & \\ & \ddots & \\ & & w_{53} \end{pmatrix}$, and the output is a 3×1 vector $[y_1, y_2, y_3]^\top$. Generally, a fully connected layer that takes in a n-element vector and outputs a m-element vector $\mathbf{y}$, uses a $m \times n$ *weight* matrix $\mathbf{w}$. The math operation, written in its matrix form, is

$$\mathbf{y} = \mathbf{w}^\top \cdot \mathbf{x}$$

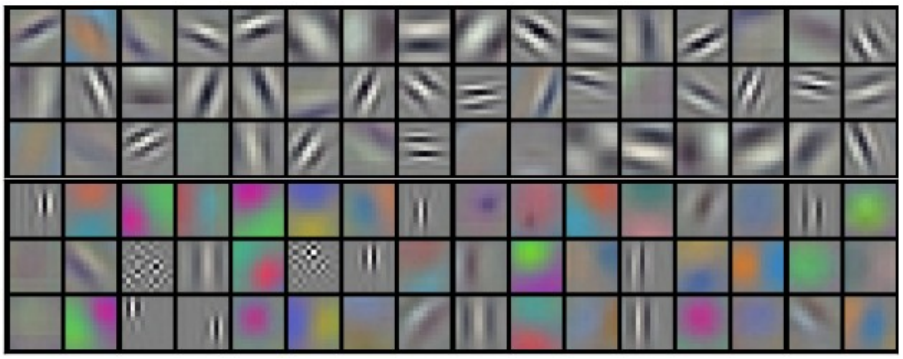

Figure 2.10: Example filters learned by the first convolution layer in Alexnet [13]. Each of the 96 filters is sized $11 \times 11 \times 3$. Note that some filters focus on discovering lines and some extract shape and colour information.

In practice, multiple inputs are stacked together to form a matrix, hence, the computation of a fully connected layer is a matrix multiplication, an operation that can be parallelized in a straight forward way. Matrix multiplication has a time complexity of $O(n^3)$.

2.2.1.2 Convolution layer

Convolution layers are used to extract local features, like lines in a particular direction or a blotch of some colour, by convolving the input with its weights (also called filters). Input images to a convolution layer are typically represented by three-dimensional arrays, where the first two dimensions are the width and height, and the third dimension if often referred to as *channel*. Colour images typically contain three channels, namely red, green and blue. The value at any given point (p, q) of channel k, in the convolution result, is defined as

$$y_{p,q,k} = \sum_{c=1}^{C} \sum_{s=-a}^{a} \sum_{t=-b}^{b} \mathbf{w}_{s,t,k} \cdot x_{p-s,q-t,c}$$

where x is the image input, $\mathbf{w}$ is the 3D filter that is spatially sized as $2a \times 2b \times K$, and y is the convolution result, also called the *feature map*. The parameter c in the formula refers to *channel* in the input image.

The workload of a convolution computation is demanding on both compute and memory resources. Many parallel algorithms, including implicit GEMM, Winograd method [24] and FFT, have been developed to accelerate convolution computations. Implicit GEMM, for example, reorders the image inputs and filters in a way such that the convolution can be computed with matrix multiplications. Some parallel algorithms, like FFT, have large memory requirements that can be in the gigabyte range. Convolution

Figure 2.11: Feature maps produced by later convolution layers. Yosinski et al. [26].

results (i.e., feature maps) can also consume a significant amount of memory. For example, in network VGG-16 [19], the size of all feature maps generated from one $224 \times 224 \times 3$ image is around 100MB, while all weights in the network consume only about 50MB. It is for this reason we target feature maps when optimizing the memory management.

2.2.1.3 Activation layer

Activation layers apply a non-linear transformation to their inputs without using weights. Non-linear activation is crucial to CNNs, as according to universal approximation theorem [8], non-linear neural networks can approximate any function. Without activation, neural networks can only represent linear systems. Activation is applied to every element in the input, that is

$$\mathbf{y} = \sigma(\mathbf{x})$$

where $\mathbf{x}$ is the input, $\sigma(\cdot)$ is an element-wise non-linear function and $\mathbf{y}$ is the output. In many CNNs, $\text{ReLU}(x) = \max(0, x)$ is used as the activation function because both the function and the function gradient are relatively inexpensive to compute compared to other activation functions like tanh().

The computation of ReLU activation is trivial and embarrassingly parallelizable. Furthermore, since the output shape is the same as that of the input, activation can be implemented as an in-place operation to save space.

2.2.1.4 Pooling layer

Pooling is used to progressively reduce the size of the feature maps generated by convolution, and in turn reduces the number of weights needed in the network. The most commonly used configuration is *max* pooling with a filter size of 2×2 and a stride of 2. This means that the input is partitioned into 2×2 sub-matrices, and for every sub-matrix, only the largest element is kept. As illustrated in Figure 2.12,

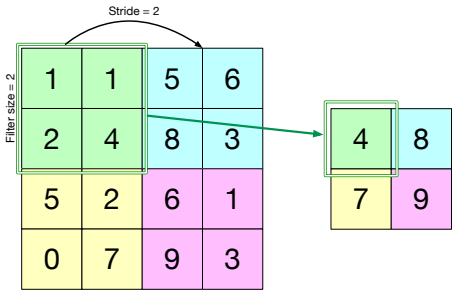

Figure 2.12: Illustration of pooling with filter size 2 and stride 2.

pooling under this configuration effectively removes 75% of the input data. The computation required for pooling is trivial and easily parallelizable.

2.2.1.5 CNN structure

Convolutional neural networks contain chains of layers arranged in specific sequences. For example, AlexNet [13] chains five instances of convolutional segments with three fully connected layers, as illustrated in Figure 2.8 (on page 19). A network, in which layer l's output is only taken as input by layer $l+1$, is called a *linear network*. In non-linear networks, however, layer l's output is used by multiple other layers. Our work focuses on linear CNNs.

2.2.2 Inference and Training

Neural networks have to be trained before they can be used for inference. Below we describe how both inference and training work.

2.2.2.1 Inference

Inference, also known as *forward propagation*, refers to the process of propagating the network input through the layers. The input is transformed by each layer from the first to the last, where the inference output is produced. The inference process can be abstracted as compounding the layers' corresponding mathematical functions:

$$\text{inference result} = \mathcal{F}\left(\text{input}, \boldsymbol{\Theta}\right) = f^{(l)} \circ f^{(l-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\text{input}, \boldsymbol{\Theta})$$

where $f^{(l)}$ represents the math function of layer l, and $\boldsymbol{\Theta}$ is the set of all trainable weights, $\boldsymbol{\Theta} = \left\{\mathbf{w}^{(i)} | \forall\, i = 1 \ldots l\right\}$.

2.2.2.2 Training

Training calibrates the network weights using the inference result. During training, labelled data is fed into the network for inference. The inference result is then compared against the known correct values by a *loss function* $\mathcal{L}$, which outputs a numerical *loss value* that measures "how accurate" the inference results are. For example, if the house style classifier correctly identified the style, then the loss will be small; whereas if it categorized Russian design to be baroque, the loss would be larger to reflect the error.

$$\begin{aligned} \text{loss} &= \mathcal{L}(\text{inference result}, \text{data labels}) \\ &= \mathcal{L}\left[\mathcal{F}\left(\text{input}, \boldsymbol{\Theta}\right), \text{data labels}\right] \end{aligned}$$

The training objective is to minimize the loss of a neural network by appropriately adjusting the network weights. Ideally, the trained weight is the one, among all possible weights, that leads to the least loss value:

$$\boldsymbol{\Theta}_{opt} = \underset{\boldsymbol{\Theta} \in \mathbb{R}}{\operatorname{argmin}} \; \mathcal{L}\left[\mathcal{F}\left(\text{input}, \boldsymbol{\Theta}\right), \text{data labels}\right]$$

Thus, loss functions quantify the network accuracy and turn training into a numerical optimization process.

2.2.3 Gradient descent

Although it is unclear whether a neural network's loss function is convex; convex optimization techniques have been successfully applied to minimize the loss. Gradient descent is an optimizing technique for finding minima in convex problems and is widely used in neural network training.

The algorithm employs the fact that the global minima of a convex function can be approached iteratively by moving along the function's curve, in the negative direction of its gradient. The gradient of a function at a point indicates the direction of the fastest descent along the function at that point.

Consider the quadratic function $\mathcal{L}(w)$ depicted in Figure 2.13, and assume weight w is initially set to -2. Intuitively, w's value needs to be increased to reach $\mathcal{L}$'s minima. The gradient of $\mathcal{L}$, at -2 is -6, denoted by the slope of the thick red line. The value of w should be increased to be closer to $\mathcal{L}$'s minima. A common approach is to adjust the value of w by $\mathcal{L}$'s gradient scaled with $-\alpha$. If $\alpha = 1/6$,

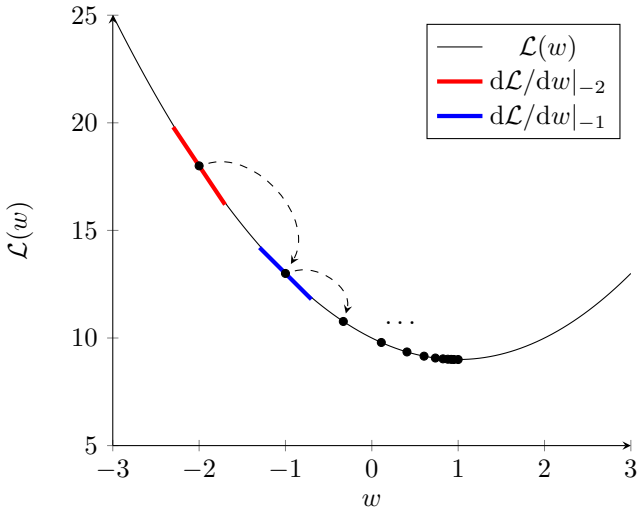

Figure 2.13: Applying gradient descent to a quadratic function (convex). The amount moved is $\alpha \cdot \mathrm{d}L/\mathrm{d}w$.

then the value of w is increased by 1 to -1 in the first iteration. The process is then repeated until the $\mathcal{L}(w)$ converges to its minima, as depicted by the black dots.[15]

The update in every iteration, written formally, is:

$$w_{new} \leftarrow w_{old} - \alpha \cdot \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}w}\Big|_{w_{old}} \tag{2.1}$$

where α is called the *learning rate*, a *hyper-parameter* that is manually set before training and will not be tuned by the gradient descent process. Finding optimal α's remains as an active research topic [28] [27]. Some optimization technique, like *ADAM* [12] that extends gradient descent, adjusts the learning rate during the iterative descent process.

Equation (2.1) adjusts only one weight. In practice, weights have more elements and every element is updated based on the partial derivative of the loss function with regard to itself:

$$w_i \leftarrow w_i - \alpha \cdot \frac{\partial \mathcal{L}}{\partial w_i}, \ \forall w_i \in \mathbf{\Theta} \tag{2.2}$$

[15] In practice, the training process is stopped when the loss converges to a small value.

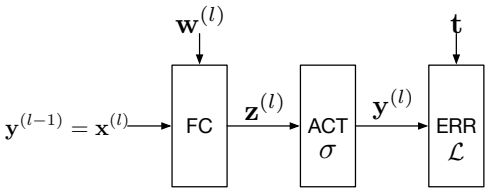

Figure 2.14: Example dataflow of the last two layers of a CNN: a fully connected and an activation layer. $\mathbf{z}^{(l)} = (\mathbf{x}^{(l)})^\top \cdot \mathbf{w}^{(l)}, \mathbf{y}^{(l)} = \sigma(\mathbf{z}^{(l)})$. $\mathcal{L}$ represents the loss function that measures the activation result $\mathbf{y}^{(l)}$ against known data label $\mathbf{t}$.

2.2.3.1 Gradient calculation

Training adjusts all the weights in the network to minimize the loss, improving inference performance. As mentioned in §2.2.2.1, inference can be abstracted as compounding layers' corresponding math functions, allowing us to apply the chain rule to find the gradient of each weight element.[16]

The rules for gradient calculation is illustrated using a simple example below; we will also generalize the rules in the next subsection. Figure 2.14 (on page 25) shows the last two layers of a CNN, consisting of a fully connected layer and an activation layer. The fully connected layer takes in $\mathbf{x}^{(l)}$, and outputs $\mathbf{z}^{(l)} = \mathbf{w}^{(l)}\mathbf{x}^{(l)}$. The activation layer then applies the non-linear function σ to its input, $\mathbf{z}^{(l)}$; its output $\mathbf{y}^{(l)}$ is then assessed by a square loss function $\mathcal{L} = \frac{1}{2}\sum_{i=1}^{N}(t_i - y_i)^2$, a typical error function used in regression tasks.[17]

In the deduction that follows, $\overline{\mathbf{a}}$ is used to denote the gradient of $\mathcal{L}$ with regarding to $\mathbf{a}$, i.e., $\overline{\mathbf{a}} = \{\partial\mathcal{L}/\partial a_i \,|\, \forall a_i \in \mathbf{a}\}$. $\mathbf{t}$ is the vector of known values of data labels. $\|\mathbf{a}\|_2^2$ denotes the square of $\mathbf{a}$'s Euclidean norm, $\|\mathbf{a}\|_2^2 = \left(\sqrt{\sum_i a_i^2}\right)^2 = \sum_i a_i^2, \forall a_i \in \mathbf{a}$.

To summarize, the dataflow in Figure 2.14 is:

$$\mathbf{z}^{(l)} = (\mathbf{x}^{(l)})^\top \cdot \mathbf{w}^{(l)};$$
$$\mathbf{y}^{(l)} = \sigma(\mathbf{z}^{(l)})$$
$$\mathcal{L} = \frac{1}{2}\sum_{i=1}^{N}(t_i - y_i^{(l)})^2 = \frac{1}{2}\|\mathbf{t} - \mathbf{y}^{(l)}\|_2^2$$

[16] Chain rule is used for finding derivatives of compound functions. Specifically, $\frac{\mathrm{d}}{\mathrm{d}x} f \circ g(x) = \frac{\mathrm{d}f}{\mathrm{d}g}\frac{\mathrm{d}g}{\mathrm{d}x}$. In multi-variable functions, the chain rule is applied to each constituent function. For example, $\frac{\mathrm{d}}{\mathrm{d}t}h(f(x), g(x)) = \frac{\partial h}{\partial f}\frac{\mathrm{d}f}{\mathrm{d}x} + \frac{\partial h}{\partial g}\frac{\mathrm{d}g}{\mathrm{d}x}$.

[17] The fraction in front cancels out when taking the gradient of the loss function.

The goal of gradient descent is to find the gradient of $\mathcal{L}$ with regard to weight $\mathbf{w}^{(l)}$. Specifically, the chain rule gives us

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(l)}} \cdot \frac{\partial \mathbf{y}^{(l)}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{w}^{(l)}}$$

The calculation of weight gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(l)}}$ can be split into the following steps. For better readability, we colour the steps following the derivation steps of, $\frac{\partial \mathbf{y}^{(l)}}{\partial \mathbf{z}^{(l)}}$, $\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{w}^{(l)}}$ and $\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{x}^{(l)}}$:

$$\overline{\mathbf{y}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(l)}} = \mathbf{1} \cdot \left(\mathbf{y}^{(l)} - \mathbf{t}\right) \tag{2.3}$$

$$\overline{\mathbf{z}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(l)}} \cdot \frac{\partial \mathbf{y}^{(l)}}{\partial \mathbf{z}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(l)}} \cdot \frac{\partial \sigma(\mathbf{z}^{(l)})}{\partial \mathbf{z}^{(l)}} = \overline{\mathbf{y}^{(l)}} \cdot \sigma'\left(\mathbf{z}^{(l)}\right) \tag{2.4}$$

$$\overline{\mathbf{w}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{w}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \left[(\mathbf{x}^{(l)})^\top \cdot \mathbf{w}^{(l)}\right]}{\partial \mathbf{w}^{(l)}} = \overline{\mathbf{z}^{(l)}} \cdot (\mathbf{x}^{(l)})^\top \tag{2.5}$$

Equation (2.5) shows that computing the weight gradient of layer l requires $\overline{\mathbf{y}^{(l)}}$ and $\mathbf{x}^{(l)}$. Similarly, updating the weight gradient of layer $l-1$ will demand $\overline{\mathbf{y}^{(l-1)}}$, as presented below:

$$\overline{\mathbf{y}^{(l-1)}} = \overline{\mathbf{x}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{x}^{(l)}} = (\mathbf{w}^{(l)})^\top \cdot \overline{\mathbf{z}^{(l)}} \tag{2.6}$$

$$\tag{2.7}$$

The resulting gradient $\overline{\mathbf{x}^{(l)}}$ is often referred to as the *data gradient* or $\mathrm{d}\mathbf{x}$ because it is calculated based on the input data. It is propagated backwards to layer $l-1$ as $\overline{\mathbf{y}^{(l-1)}}$ (or $\mathrm{d}\mathbf{y}^{(l-1)}$).

In summary Equation (2.3) and Equation (2.5) show that the following operands are required to update the weights of a fully connected layer l:[18]

- $\overline{\mathbf{y}^{(l)}}$: The gradient of error w.r.t. $\mathbf{y}^{(l)}$, or else known as $\mathrm{d}\mathbf{y}$,
- $\mathbf{y}^{(l)}$: Layer l's input during inference
- $\mathbf{w}^{(l)}$: Layer l's weight
- $\mathbf{x}^{(l)}$: Layer l's input during inference

[18] Not all types of layers require all the data listed. For example, the gradient calculation of convolutions does not require y.

2.2.3.2 Generalized gradient calculation

In general, if x_j has contributed to output element y_k in inference, y_k will influence the gradient of x_j during the gradient computation. The gradient of $\mathcal{L}$ w.r.t. x_j is the sum of all y_ks that x_j has influenced during inference, i.e.,

$$\overline{x_j} = \sum_k \overline{y_k} \cdot \frac{\partial y_k}{\partial x_j}$$

written in vector form,

$$\overline{\mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}}^\top \overline{\mathbf{y}} \tag{2.8}$$

where $\partial \mathbf{y}/\partial \mathbf{x}$ is often denoted as the *Jacobian matrix* where $J_{i,j} = \partial y_i/\partial x_j$. The calculation of multiplying the Jacobian with vector d$\mathbf{y}$ is often referred to as taking a *Vector-Jacobian product* or VJP.

The two examples below show the gradient computation methods of an activation layer and a fully connected layer.

- In activation layers, where x_i contributes only to y_i by $y_i = \sigma(x_i)$, the Jacobian matrix is diagonal since $\partial y_i/\partial x_j = 0$, if $i \neq j$. The Jacobian matrix can be thus flattened into one vector. Accordingly, the matrix multiplication between the Jacobian and $\overline{\mathbf{y}}$ becomes an element-wise product between two vectors. It is for this reason that Equation (2.4) is optimized to a vector-element-wise product.
- In fully connected layers, every input element x contributes to all output elements during inference; in turn, every x element is influenced by all y elements during gradient computation. In other words, the Jacobian matrix is dense.

The gradient of compounded function can be calculated by chaining VJP calculations. Specifically, Equation (2.3), Equation (2.4) and Equation (2.5) are three VJPs that together calculated the gradient of the function $\mathcal{L} = \frac{1}{2}\|\mathbf{t} - \sigma(\mathbf{w} \cdot \mathbf{x})\|_2^2$.

2.2.3.3 Automatic differentiation

Machine learning frameworks typically provide the functionality to compute gradients for any computation graph. The gradients are not calculated using finite difference or symbolic differentiation.

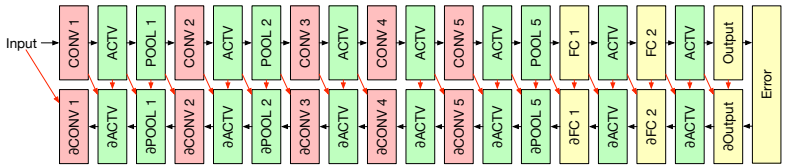

Figure 2.15: AlexNet's forward (on the top) and back propagation (on the bottom). Notice the ∂ sign in front of the layer names in the back propagation stage. The black arrows show the data flow between layers; the red arrows show the data produced in inference and reused in the backpropagation stage.

Rather, the steps for automatic differentiation are generated based on the same rule introduced in §2.2.3.2 and Equation (2.8).

Generically, for any tensor $\mathbf{x}$, all of its consumers $\mathcal{Y}$ in inference contribute to $\mathbf{x}$'s gradient during backpropagation. Two values are needed to calculate $\overline{\mathbf{x}}$ for each consumer $\mathbf{y}_i \in \mathcal{Y}$:

- the error gradient $\overline{\mathbf{y}_i}$, and
- the Jacobian matrix $\mathbf{J}_i = \frac{\partial \mathbf{y}_i}{\partial \mathbf{x}}$.[19]

With those values, the gradient of any $\mathbf{x}$ in a computation graph can be easily found by

$$\overline{\mathbf{x}} = \sum_i \mathbf{J}_i \cdot \overline{\mathbf{y}_i}, \ \forall \mathbf{y}_i \in \mathcal{Y}$$

2.2.4 Workload in CNN training

In summary, every layer that participated in inference has to conduct gradient computations and weight updates[20] in backpropagation. Figure 2.15 portrays forward and backpropagation for AlexNet [13], in which arrows depict the directions of data flow.

During forward propagation, layer l will receive input $\mathbf{x}^{(l)}$ from the previous layer, transforms it with its set of weights $\mathbf{w}^{(l)}$ and outputs the result $\mathbf{y}^{(l)}$ to the next layer. In backpropagation, a layer takes $\overline{\mathbf{y}^{(l+1)}}$ as gradient input and produces $\overline{\mathbf{x}^{(l)}}$, whose gradient calculations often require $\mathbf{w}^{(l)}$, $\mathbf{x}^{(l)}$ and $\mathbf{y}^{(l)}$.[21]

2.3 TensorFlow

Google's TensorFlow is one of the most widely adopted machine learning frameworks. It supports a vast range of machine learning models, and it supports the execution of machine learning applications

[19] Machine learning frameworks often have defined the Jacobian matrices for commonly used functions.
[20] Weight update is only required for the layers that use weights.
[21] Some types of layers require either $\mathbf{x}^{(l)}$ or $\mathbf{y}^{(l)}$.

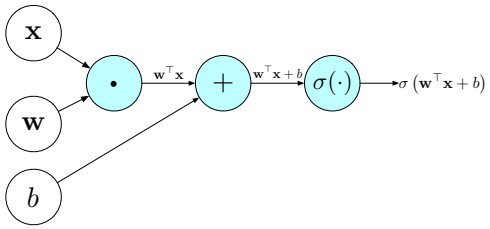

Figure 2.16: Computation graph representing $\mathbf{y} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$. Dataflow within the graph is represented by the directed edges, math operations are represented by circular nodes. The values shown on the edges are the outputs from the operations. For instance, the $\odot$ node takes the transposed $\mathbf{w}$ and $\mathbf{x}$ and calculates their inner product $\mathbf{w}^\top \cdot \mathbf{x}$.

on a variety of different types of processors and accelerators (CPU, GPU and TPU) in both local and distributed environments.

TensorFlow, like many other frameworks supporting machine learning such as Torch [5] and MxNet [3], employs direct acyclic graphs (DAGs), called *computation graphs*, to represent computations and dataflow. Figure 2.16 depicts a simple computation graph that represent the calculations of a fully connected layer, namely $\sigma(y = \mathbf{w}^\top \mathbf{x} + b)$. The n-dimensional data flowing in the computation graph are called *tensors*. For example, a 1D tensor is a vector; a 2D tensor is a matrix. This section presents a background of TensorFlow's local execution environments.[22]

2.3.1 Programming and execution model

The TensorFlow framework contains two major components, a user-facing frontend, called the *client*, and a backend runtime, called the *master*. Users interface with the client to define machine learning models; the client then constructs the computation graphs accordingly. The master is responsible for optimizing, scheduling and running the computation graph on physical processors.

Client

The TensorFlow client, available in many programming languages including Python and C, is an interface that machine learning engineers use to create machine learning models by defining the data flow. For example, the TensorFlow API calls in Listing 2.1[23] implement a typical TensorFlow application performing linear regression, whose computation graph is shown in Figure 2.17. In the first step, the input tensors `x` and `t`, and weight tensor `w` are defined. Next, `y` is defined as the inner product of `x` and `w`, they are used as symbols in creating the computation graph. Loss, gradient calculations and

[22] The distributed runtime of TensorFlow is omitted from the discussion.
[23] Some APIs in the listing are renamed for better readability.

```
# 1. Defining input/output/weights
x = tf.input(shape=[BATCH_SIZE, FEATURE_SIZE])
t = tf.input(shape=[BATCH_SIZE, OUTPUT_SIZE])
w = tf.weight(shape=[FEATURE_SIZE, OUTPUT_SIZE])
# 2. Defining inference model
y = tf.matmul(w.transpose(), x)
# 3. Defining the loss
loss = tf.mean_squared_error(y, t)
# 4.a. Add gradient calculation steps
gradients = tf.gradients(loss)
# 4.b. Update weights
opt = tf.GradientDescent(learning_rate).apply_gradients(gradients)
# Run the model with master
with tf.Session():  # opens session with master
    tf.run(opt)
```

Listing 2.1: Example TensorFlow application

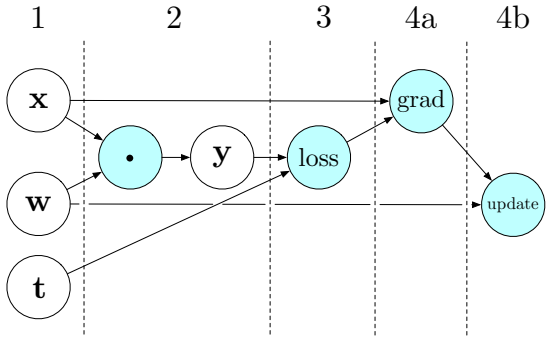

Figure 2.17: The steps involved in building the computation graph from the API calls in Listing 2.1. The step markers at the top of the figure correspond to the step numbers in the comments of the listing.

weight updates are then defined in the next steps. Lastly, the `run` method is called to invoke the TensorFlow master for execution.

The TensorFlow client is unaware of the implementation details required for graph execution. Instead, the TensorFlow client invokes the *master* and passes down the computation graph in a text format similar to XML.[24]

Master

The master first reconstructs the computation graph passed to it in text format, into its own data format. The master then applies a series of optimizations to the graph, including graph pruning, common expression elimination and constant folding. After the optimization stage, the master starts executing the graph. Execution of the optimized computation graph follows a breadth-first order. More specifically, once a node finishes execution, the scheduler enqueues operations whose inputs are computed and available into the *ready queue*.

[24]The format is called *ProtocolBuffer*, an interface that serializes structured data [6].

2.3.2 Computation graph

The computation graph is constructed with nodes (or vertices) and edges.

Nodes Nodes, also called *operations*, abstract actions that produce or transform tensors in the graph. An operation can take zero or more inputs and produces zero or more outputs. As such, operations can represent arithmetic computations, constants or even data transfers tasks. Particularly in Figure 2.16 the '$\top$' node is a unary operation that transposes its input; and the '$\cdot$' node takes two matrix-inputs and calculates inner product; data nodes like '$\mathbf{x}$' produce data tensors without taking any input.

Edges Edges set up producer-consumer relationships between operations. In computation graph speak, tensors are represented by edges.

Using a computation graph allows TensorFlow to decouple data and the computations. That is, the same computation graph is reused multiple times with different data. TensorFlow also supports *partial execution* so that if the user requests the result of $\mathbf{w}^\top\mathbf{x}$ in Figure 2.16 (the result from the dot node), TensorFlow executes only the subgraph that produces $\mathbf{w}^\top\mathbf{x}$ and will not execute the subgraph that involves the '$+$' and 'σ' operations.

One key difference between the client and master graphs is that edges in the client graph do not have data storage associated. For example, in Figure 2.16, the buffer storing $\mathbf{w}^\top\mathbf{x}$, could be used to store $\mathbf{w}^\top\mathbf{x} + b$ and y because they have the same shapes and in turn require the same amount of space. However, the client is unaware of the actual implementation of the math operations (whether they support in-place updates or not), and hence does not concern itself with buffers. It is the backend master that assigns buffers to tensors.

2.3.3 Execution order

Producer-consumer relationships, or more specifically data dependencies, determine execution orders in TensorFlow. For instance, operation A runs before operation B if B requires as input, the data that A produces. Another type of dependency, called *control dependency*, enforces execution order between operations that do not have data dependencies. By setting C as A's control dependency, C is guaranteed to run before A, even if there is no data dependency between them. If the user sets another dependency that requires A to run before C, a dependency cycle is created and TensorFlow will stop and report an error.

2.3.4 GPU support

A specialized TensorFlow interface called `StreamExecutor` is used for abstracting co-processors like GPUs. All GPU related tasks, such as performing math operations, and copying memory, are requested through this interface. However, `StreamExecutor` abstracts only a subset of the available GPU functions, such as launching compute kernels or calling device driver functions. Some functions are not natively supported; for instance, `cudaMemPrefetchAsync()`, a function that is widely used in our work. As a result, we had to modify `StreamExecutor` to also support the other GPU functions needed to control the virtual memory.

Memory management The TensorFlow runtime allocates all free global memory and manages the memory internally. As such, tensor allocations and deallocations will not invoke high overhead memory control directives like `cudaMalloc()` and `cudaFree()`. Tensors are reference counted, such that a tensor can be freed automatically when its last consuming operation finishes.

Chapter 3

Design

In the previous sections, we established the following:

- Computation and dataflow of machine learning applications are often represented by computation graphs in machine learning frameworks like TensorFlow.
- Nvidia's virtual memory subsystem on the recent GPU hardware enables running GPU programs that require more memory than what is physically available in global memory.
- Nvidia provides APIs for applications to issue memory management hints to aid the virtual memory subsystem in making virtual memory management decisions such as offloading and prefetching data from and to global memory. This allows the application to issue virtual memory hints based on its knowledge of the workload.

In this chapter, we first define the problem that we try to solve, and then present the specific requirements for supporting machine learning applications. In §3.2 we present our design, leading to a new system we call *"AutoVM."*

3.1 Motivation and Problem Statement

Motivation

Training state-of-the-art convolutional neural networks is time intensive. For example, training ResNet-50 [7] on the ImageNet-1K dataset [17] for 90 epochs takes 14 days using a single M40 GPU and takes 29 hours on a machine with eight Nvidia Tesla P100 GPUs [21]. Our goal is to improve training speed by optimizing memory access locality. Improved training speed allows machine learning engineers to apply larger networks to throughput critical applications whose latency requirements would have previously been attainable only by using simpler networks. Furthermore, performance of different neural network structures can be evaluated in a more timely manner, leading to faster time to market.

Training state-of-the-art convolutional neural networks is memory intensive. For example, training Inception v4 [22] with 64-image batches requires over 80GB of memory. Generally, two factors affect the amount of memory needed:

1. the complexity of the network, e.g., the more layers the more memory is needed, and
2. training batch sizes, i.e., the number of images used in one training iteration.

Using virtual memory in machine learning frameworks allows for more complex networks and larger batch sizes, because virtual memory size can be significantly larger than the amount of physical memory available. Although using virtual memory permits running problems that do not fit in physical memory, the default virtual memory management policy is likely to make suboptimal paging decisions since it is unaware of the running applications' memory access patterns.

Specifically for neural network training, the default memory management policy will tend to make particularly poor paging decisions for problem sets that do not fit in physical global memory, leading to poor response times. For example, in neural network inference, just consumed outputs are rarely referenced again for a prolonged period of time. These outputs are thus ideal candidates for paging out to host memory, but they are not likely to be selected for page-out under the default LRU scheme as they were just referenced.

As a result, the default mechanism does not page out data promptly. Not paging tensors out in a timely way exhausts physical global memory and causes memory thrashing. Furthermore, the default memory management policy cannot predict the data requirements of the subsequent computations.

Consequently, if those operations' operands were previously paged out, they will be demand-paged in, page-by-page, as they are being accessed, decreasing memory throughput.

It is perhaps for this reason that virtual memory is rarely used in frameworks like TensorFlow and Torch. In fact these frameworks make it exceedingly difficult to exploit virtual memory. In the case of TensorFlow, for example, parameter `per_gpu_memory_fraction` is typically set to below one by machine learning programmers to limit the amount of global memory used. If set to a number higher than one, TensorFlow would use virtual memory. Interestingly, no publicly available document reveals this behaviour. In practice, a bug in TensorFlow r1.14 prevents the users from using virtual memory even if `per_gpu_memory_fraction` was set correctly.

Problem statement

We aimed to design a software layer, AutoVM, that interfaces with the Nvidia driver and machine learning frameworks, that allows for faster training and more complex network structures, by optimizing memory locality when virtual memory is used.

Requirements

Our mechanism AutoVM must:

- work within machine learning frameworks like TensorFlow; and
- be integrated into existing machine learning applications with minimal effort, in a way that is as transparent to the machine learning engineers as possible.

Limitations

The scope of this project is limited in a number of ways however:

- The solution is only optimized for linear CNN networks. We argue in §3.5.1 that our method is generalizable to a larger class of neural networks.
- We have chosen to integrate and test our method with TensorFlow. While the design principle applies to any machine learning framework, TensorFlow's functionalities constrain the actual mechanisms we use in our implementation. In particular, we assume that machine learning frameworks represent the machine learning workloads in the forms of computation graphs.
- Our implementation uses Nvidia's virtual memory control APIs so that it operates on Nvidia GPUs only at the moment. We argue in §3.2 that our solution is partially applicable to older

GPUs without virtual memory support. Specifically, one could use the traditional memory copy APIs like `cudaMemcpy()` to trigger data transfer between host memory and global memory, instead of using APIs that controls virtual memory only (like `cudaMemPrefetchAsync()`), as is done by vDNN [16].

3.2 Design Overview

Our primary objective is to design a solution that decides which tensors to transparently transfer between host memory and global memory and when; and to initiate these transfers automatically, all without requiring machine learning engineers's programming effort. Fundamentally we use host memory as a backing store for global memory because host memory tends to (and can be) much larger than global memory. Our goal is to:

- have tensors being consumed by currently running operations on GPUs, reside in global memory without experiencing page faults;
- have tensors residing in global memory that will not be accessed in the near future, be transferred to host memory, and
- have memory transfers overlap with computations so that tensor transfer latencies are hidden and do not slow down GPU computations.

To achieve these goals, we split the solution into a policy and a mechanism:

- **the policy** determines which tensors to move and when,
- **the mechanism** triggers the tensor transfers as instructed by the policy,

AutoVM is designed to be a software layer that conceals from machine learning engineers the process of selecting tensors and initiating the tensor transfers. AutoVM first uses the policy to gather tensor transfer decisions; then, AutoVM triggers the mechanism to inject memory transfer operations into the TensorFlow computation graph. As such, tensors will be transferred between host memory and global memory automatically as the computation graph executes. Correspondingly in our TensorFlow implementation,

- The policy that analyzes the computation graphs is implemented in the TensorFlow client. The policy uses the TensorFlow graph editor module to analyze the computation graph.
- The mechanism is implemented as a TensorFlow operation, called a *`MemOp()`*. `MemOp()` contains appropriate calls to Nvidia virtual memory API that facilitates the tensor transfers. AutoVM

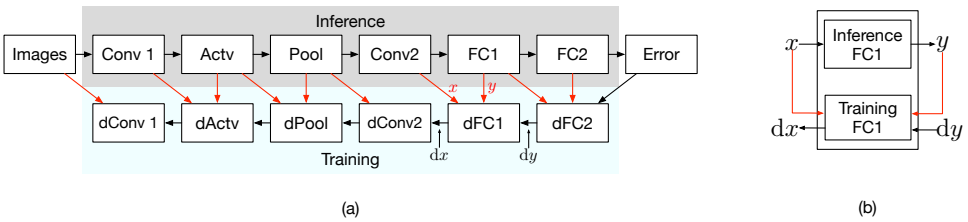

Figure 3.1: (a) depicts a simple linear CNN, in which arrows represent dataflow. Red arrows symbolize data that are produced in inference and reused in training. Weights and layer parameters are not shown in the figure. (b) on the right shows an example dataflow in layer FC1's inference and training.

inserts multiple instances of `MemOp()` into the computation graph at appropriate locations, as dictated by the policy. Control dependencies are inserted before and after each `MemOp()` to enforce the execution order of computation operations and `MemOp()`s. TensorFlow's `StreamExecutor` interface is modified to allow `MemOp()` to issue the appropriate calls to initiate tensor transfers.

- AutoVM fuses the policy and the mechanism, and is packed as a function that machine learning engineers can call after model definition to enable memory optimization.

The separation of policy and mechanism allows the design to be flexible and extensible. For example, to enable AutoVM in TensorFlow with older GPUs that do not support virtual memory, the policy remains as is, while the mechanism that controls global memory needs modification. Specifically, we will only need to create another implementation of `MemOp()` that uses memory controlling APIs like `cudaMemcpy()` to issue the memory transfer commands manually, instead of managed memory hints.

3.3 Policy

3.3.1 Identifying tensors to move

Here, we consider rules that identify tensors for *offloading* (transfer from global memory to host memory) and *prefetching* (transfer from host memory to global memory). More specifically,

- tensors to offload during inference,
- tensors to prefetch during inference,
- tensors to offload during training, and
- tensors to prefetch during training.

3.3.1.1 Offloading during inference

In a linear CNN, layer l transforms its input $\mathbf{x}^{(l)}$, optionally using its weight $\mathbf{w}^{(l)}$ and produces the result $\mathbf{y}^{(l)}$. The output $\mathbf{y}^{(l)}$ is consumed only by the subsequent layer $l+1$. In effect, after layer l completes execution, only $\mathbf{w}^{(l)}$ and $\mathbf{y}^{(l-1)}$ are candidates for offloading.[1]

The different outputs produced by the different layers, combined, consume a significant amount of memory space. For example, in training VGG16, all the outputs generated from a batch of 128 images, take up over 12GB of memory in aggregate. On the other hand, all the weights combined consume about 138MB of memory, about 1% of the total space used. As a result, we have decided not to select the weights as offloading candidates.

Only a subset of all y's produced are selected for offloading. Deciding which tensors to offload is based on two rules: tensor size and reuse distance. Listing 3.1 captures the rules in the algorithmic form.

Tensor size If an output $y^{(i)}$ is small, it is not selected for offloading. The size threshold parameter is tuned empirically.

Reuse distance If the expected duration between a tensor's last reference in inference and its first consumption in training is short, the tensor is not selected for offloading. However, since it is difficult to estimate an operation's runtime, we establish a tensor's reuse distance as the number of operations between its consumers in inference and training. A short reuse distance means that the tensor is likely to be used soon, so offloading it may incur additional performance penalties.

The reuse distance threshold is a configurable parameter, that dictates the minimal reuse distance needed for a tensor to be selected for offloading. Its value can be determined algorithmically if the exact runtimes of operations are known. However, as the exact runtimes are unknown to AutoVM, this value needs to be hand-tuned for the best performance, although the default value that skips the outputs of the last two layers in inference worked well for the CNNs we tested in our experiments.

3.3.1.2 Prefetching in inference

Layer l's inference process uses input $\mathbf{x}^{(l)}$and weight $\mathbf{w}^{(l)}$ to produce $\mathbf{y}^{(l)}$. The input will always reside in global memory before layer l starts executing, because it was just produced by the previous

[1] Recall that $\mathbf{x}^{(l)}$ and $\mathbf{y}^{(l-1)}$ refer to the same tensor.

```
def get_offload_candidates(inference_layers, distance_threshold):
  offload_candidates = []
  for layer in inference_layers:
      tensor = layer.output
      if tensor.size < size_threshold: continue
      if tensor.reuse_distance < distance_threshold: continue
      offload_candidates += [tensor]
  return offload_candidates
```

Listing 3.1: Algorithm for identifying offload candidates.

layer. The weight could be in host memory if it was evicted by the runtime. In this case, we rely on demand-paging to transfer it to global memory given its relatively small size. Therefore, there is no need to prefetch any tensor in inference.

3.3.1.3 Offloading in training

Layer l computes its gradient using the loss gradient $\mathrm{d}\mathbf{y}^{(l)}$, and its inference input $\mathbf{x}^{(l)}$, and output $\mathbf{y}^{(l)}$ to produces $\mathrm{d}\mathbf{x}^{(l)}$. Layers with weights also uses their weights to compute the weight gradients $\mathrm{d}\mathbf{w}$.

$\mathrm{d}\mathbf{x}^{(l)}$ and $\overline{\mathbf{w}^{(l)}}$ are allocated right before being produced and are consumed only by the next training layer $l-1$. $\mathbf{x}^{(l)}$ and $\mathbf{y}^{(l)}$ are usually last consumed in their inference layers' corresponding gradient computations. Tensors that are not further referenced are deallocated automatically by TensorFlow, after their last references, so there is no need to offload them manually. For example, in Figure 3.1, after processing layer dFC1,

- $\mathrm{d}\mathbf{y}$ from layer dFC2 is deallocated;
- $\mathbf{y}$ from layer FC1 is deallocated;
- $\mathbf{x}$ from layer Conv2 is deallocated;
- $\mathrm{d}\mathbf{x}$ is passed to layer dConv2, and it is deallocated once dConv2 finishes.

In other words, no tensors in training will be selected for offloading.[2]

3.3.1.4 Prefetching in training

Any tensor that is expected to be accessed in the near future and is not already in global memory should be prefetched. Prefetching in training has the following constraints:

1. multiple prefetch-transfers cannot overlap, as there is only one DMA engine that transfers tensor from host memory to global memory,

[2]This assumption might not be in the case of non-linear networks.

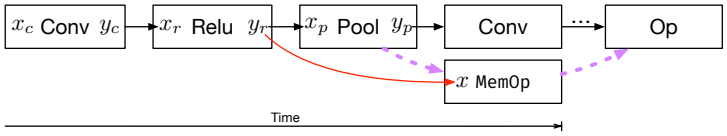

Figure 3.2: Example of offloading y_r in inference. y_r is input as x in the operation that handles offloading (`MemOp()`). The red arrow shows the data dependency. The dashed purple arrows represent 'control dependencies', a TensorFlow mechanism that enforces execution order, such that `MemOp()` runs after layer Pool completes, and before an operation in the future ('Op' in this case). The control dependency to a future operation is necessary or TensorFlow will remove `MemOp()` in the optimization stage, because no operation uses `MemOp()`'s output. The control dependency is necessary to enforce execution order, as there is no data dependency that forces `MemOp()` to start with operation `Conv`.

2. prefetched tensor should be available in global memory prior to when they are needed by their computations, and
3. prefetching begins at the end of some operation.

If a tensor X's size, combined with the size of the working set of operations between the start of X's prefetch and X's consumption does not fit in global memory, X should not be prefetched to prevent thrashing. Otherwise, prefetching X will cause the eviction of some pages belonging to the working sets of the operations in between, only to be later paged back in as they are being accessed; and some pages of X will be paged out again to make space for those working sets. This results in an excessive amount of superfluous paging activity and will negatively impact performance. Not prefetching X simply implies that X is demand-paged in as it is being accessed.

3.3.2 Identifying when to transfer tensors

This subsection describes how to time the tensor transfers, The listings in this subsection use Python syntax and resemble AutoVM's TensorFlow implementation.

3.3.2.1 Offloading during inference

Layer l's input tensor $\mathbf{x}^{(l)}$ will be transferred out to host memory immediately after layer l's computation completes, if $\mathbf{x}^{(l)}$ is not expected to be referenced in the near future. We analyze the computation graph to identify this. For example, in Figure 3.2, y_r is last consumed in the pooling layer during inference, so y_r can be safely offloaded as soon as the pooling layer finishes executing.

Listing 3.2 illustrates the algorithm that locates a tensor's last referencing operation during inference. It iterates through all operations that reference the tensor in inference and locates the last one based

```
def get_offload_timing(tensor) -> operation:
  last_inference_ref = tensor.producer
  for operation in tensor.consumers:
      if operation is not inference: continue
      # the operation with larger id runs first
      if operation.id > last_inference_ref.id:
          last_inference_ref = operation
  # offload after the last reference during inference
  return last_inference_ref
```

Listing 3.2: Algorithm for identifying when to offload. The algorithm returns an operation, meaning that the tensor can be offloaded after this operation finishes.

on the operation's `id` and name. We have found experimentally that linear CNNs' operations' `id`s are ordered chronologically, i.e., an operation with a larger `id` runs later than an operation with a lower `id`. TensorFlow follows a specific naming convention that we use to determine whether an operation is executed in inference or training (we did not include this check in Listing 3.2 for simplicity.)

3.3.2.2 Prefetching during training

In the common case, offloaded tensors are prefetched before they are needed in training. Determining the prefetch timing is more difficult because (i) the prefetch operation takes time; (ii) should be in memory before the operation that needs the tensor starts, but (iii) should not be prefetched too early. Figure 3.3 shows five different prefetch timing scenarios for prefetching the tensor needed by layer dConv1:

(**A**) shows that if no prefetching is done, demand paging transfers the accessed pages as they are accessed. This slows down the execution of layer dConv1 because pages are transferred in on page faults. After the computation consumes one page, a page fault is triggered to transfer the next page into global memory, and the computation is blocked until the prefetch completes.

(**B**) shows that if the prefetching is initiated at the beginning of the previous layer dRelu, and dRelu executes relatively quickly, then dConv1 blocks until the transfer completes. Although prefetches are asynchronous, CUDA runtime will not schedule calls to CUDA library functions that use the data being prefetched, until the transfers complete.[3]

(**C**) exemplifies the ideal case[4] where the prefetch completes just prior to the start of dConv2. In this scenario, the tensor transfer latency of the prefetch is completely hidden and the computing cores are able to run continuously with no delays being caused by tensor transfers.

[3] CUDA runtime is aware of every tensor used in every call to CUDA library functions, because such functions' operands have to be described in detail in the function calls.

[4] Assuming that layer dConv2's working set is small and will not cause thrashing as exemplified in case (**E**).

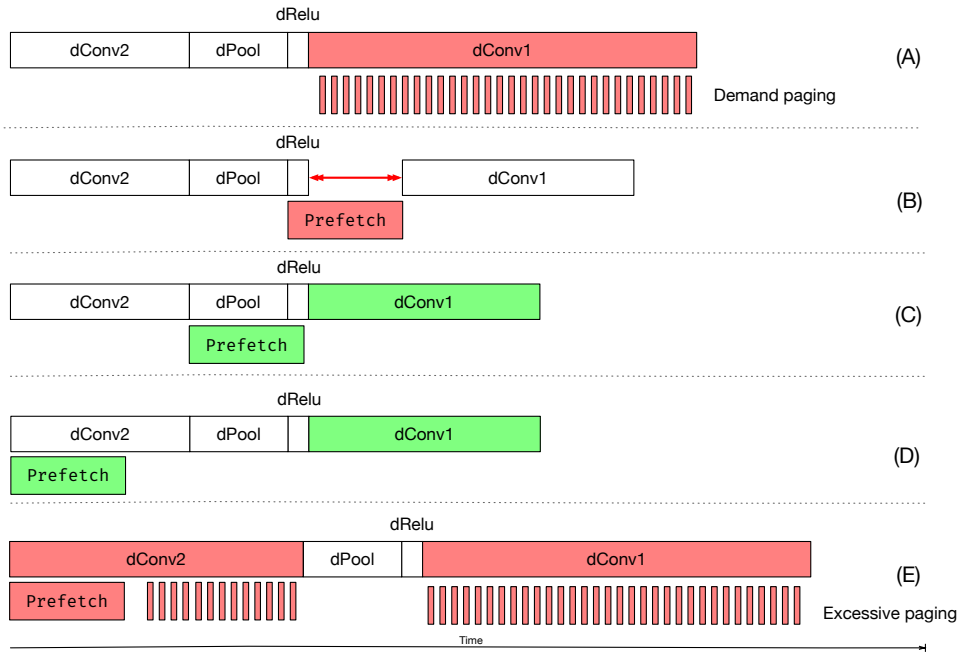

Figure 3.3: Examples showing impact of prefetching timing on performance. Lengths of blocks surrounding layer names depict the runtime of the respective layers. The tensor size and runtime in each case is independent of each other, that is, C shows the case that dConv2 requires too much memory and prefetching have caused thrashing, while dConv2 in case D do not require as much thus no thrashing is caused.

(**D**) is similar to case (**B**), but where prefetching is scheduled too early, concurrent to the execution of dConv2. This works well only if the working sets of layers dConv2, dPool, dRelu, and dConv1 all fit in global memory. If they do not, then excessive paging may occur as in case (**E**).

(**E**) illustrates the case where prefetching starts too early. Layer dConv2's working set does not fit in global memory together with the tensor being prefetched, so the prefetch causes thrashing (as depicted by the first set of page faults) because there are no physical pages available to store both dConv2's output and the prefetched content.

Such thrashing leads to severe performance degradation as discussed in §3.3.1.4. Additionally, the demand paging involved in moving the tensors required for dConv1 negatively impacts performance for the same reason described in case (**A**).

Based on the observations above, we establish two strategies for determining the when to prefetch tensors.

If exact runtimes of operations are known then one viable strategy is to prefetch tensors as late as possible but guarantee that the prefetched tensor arrives in global memory before its consumption and under the constraint that the prefetch does not induce trashing. More specifically, assuming the

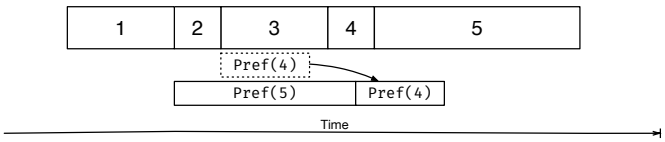

Figure 3.4: A case that cannot be handled properly by the scheduling strategy. Here, the prefetch for operation 4 is scheduled and executed after the prefetch for operation 5. The order of the prefetches for operation 4 and 5 does not match the execution order of the operations. In such cases, the prefetch for operation 4 will be the only scheduled prefetch, since operation 4 runs before operation 5.

transfer starts with operation m, the constraint states that the combined size of the prefetched tensor and the working set of m must fit in global memory so that no excessive thrashing occurs. In case (**C**) listed above, the prefetch for dConv1 is scheduled at the start of operation dPool, because

- the combined runtimes of dPool and dRelu are enough to cover the prefetch latency, and
- the working set of dPool and dRelu, together with the prefetch size, fit in global memory.

Listing 3.3 on page 44 presents the scheduling strategy which determines when to prefetch each tensor. The following is performed on each of the offloaded tensors:

1. Find the operation m that first references tensor t during training, based on the operation `id`.
2. Calculate when the prefetch should be scheduled, based on the time of copying t over the PCIe bus. Note we can estimate how long it takes to copy t, since we know the size of t.
3. Check whether the combined working set sizes of all operations between the start of prefetching t and the start of executing m exceeds the physical global memory limit.

However, there are cases that the algorithm cannot handle properly. For instance, in Figure 3.4, operation 5's prefetch requires a long transfer time and is thus scheduled at the start of operation 2. On the other hand, operation 4's prefetch needs less time and is scheduled at the start of operation 3. As a result, operation 4's prefetch is started after that of operation 5 completes. When the order of prefetch does not match the order of operation execution, we reverse the order of the prefetches if the working set sizes permit, as shown in Figure 3.5. This way, the prefetch for operation 4 can complete before operation 4 starts; while the prefetch for operation 5 is partially done. The execution of operation 5 will not start until the prefetch completes, but as our tests on GPU virtual memory show (see §4.2.4), that blocking an operation until all of its tensors are prefetched is faster than depending purely on demand-paging.

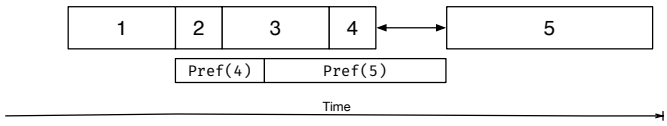

Figure 3.5: A fix for the case where the order of prefetch does not match the order of operation execution. The prefetch for operation 4 is scheduled before the prefetch for operation 5. The side effect is that, the execution of operation 5 will not start until the prefetch completes.

```
def get_prefetch_timing(tensor) -> operation:
  # 1. finding the first operation that references the tensor during training
    based on id
  first_training_ref = tensor.last_referencing_operation
  for operation in tensor.consumers:
      if operation is not training: continue
      if operation.id < first_training_ref.id:
          first_training_ref = operation

  # 2. find the operations such that transfer latency is covered
  op_runtime = 0, operations = []
  for op in first_training_ref.predecessors:
    if op_runtime >= tensor.transfer_time: break
    op_runtime += op.runtime
    operations.append(op)

  # 3. make sure working set size fit in global memory
  for op in operations:
    if op.working_set + tensor.size > GLOBAL_MEMORY_SIZE: return None
  # prefetch at the latest operation
  return operations.first_element
```

Listing 3.3: Algorithm for identifying when to prefetch tensors, assuming exact operation runtimes are known. The algorithm returns an opration, meaning the `tensor`'s prefetch should start with that operation.

If exact runtimes of operations are not known a different strategy is needed. We schedule the prefetching of a tensor as early as possible so that the tensor arrives in global memory prior to being consumed, under the constraint that

- The size of all prefetched tensors, combined with the working sets' sizes of operations between the start of prefetch and beginning of consumption fit in global memory.

In practice, the exact runtimes of operations are not known to AutoVM, so the method assuming known operation runtimes is not an option. Instead of prefetching as late as possible (when the exact runtimes are known), we prefetch the tensors as early as possible to maximize the likelihood that a tensor is made available in global memory before it is accessed. Listing 3.4 on page 45 shows the algorithm for identifying the prefetch timing. Similar to the previous strategy, it searches the prefetch timing for each of the tensors offloaded during inference individually, using the following steps:

1. Find the operation m that first references tensor t during training, based on operation `id`,

```
def get_prefetch_timing(tensor) -> operation:
  # finding the first operation that references the tensor during training based
     on id
  first_training_ref = tensor.last_referencing_operation
  for operation in tensor.consumers:
      if operation is not training: continue
      if operation.id < first_training_ref.id:
          first_training_ref = operation

  # find the operations such that the working set size plus tensor size fit in
    global memory
  operations = []
  for op in first_training_ref.predecessors:
    if op.working_set + tensor.size > GLOBAL_MEMORY_SIZE: break
    operations.append(op)

  if operations:
    # the size of the prefetched tensor is added to the operations' working set
    for op in operations:
      op.add_to_working_set(tensor)

    # return the earliest element in the list so that prefetch occurs as early
    as possible
    return operations.last_element
  return None
```

Listing 3.4: Algorithm for identifying the prefetch location, assuming exact operation runtimes are unknown. The algorithm returns an opration, meaning the `tensor`'s prefetch should start with that operation.

2. Iterate through the operations that executes before m. For each such operation, check if that operation's working set together with the size of prefetch fits in global memory.
3. Add the prefetch size to the working sets of the operations from prefetch to t's final consumption. This is done to prevent prefetching too many tensors that lead to thrashing.

The disadvantage of this strategy is that, the prefetch timings are not precise. Imprecise timings can lead to computations being blocked until all of their required tensors are transferred into global memory. Moreover, this greedy method favours the first few tensors processed, because after the prefetch timings are determined for those tensors, it might become hard to locate prefetch locations for the other tensors. The scenario depicted in Figure 3.4 cannot be handled correctly by this strategy either. Again, if such cases arises, the prefetches are scheduled according to the execution order of the operations.

3.4 The mechanism

Given a policy that identifies which tensors to transfer and when, we have designed a mechanism to trigger the memory transfers accordingly. Specifically, we designed `MemOp()`, a TensorFlow operation that wraps calls to Nvidia APIs to transfers specified tensors, as shown in Listing 3.5. Instances of `MemOp()` are inserted into the computation graph of the target machine learning application.

```
def MemOp(tensors_to_offload, tensors_to_prefetch):
    for tensor in tensors_to_offload:
        offload(tensor)
    for tensor in tensors_to_prefetch:
        prefetch(tensor)
```

Listing 3.5: The overview of `MemOp`.

The function calls in Listing 3.5, `offload()` and `prefetch()`, interface with the Nvidia's memory controlling function `cudaMemPrefetchAsync()` to issue memory transfer commands to the Nvidia runtime. In TensorFlow, we have modified the `StreamExecutor` interface to enable the `MemOp()`'s access to virtual memory control. The implementation details of the mechanism are described in Chapter 5.

3.5 Limitations

Our current design has the following limitations:

1. The policy described in §3.3 may not work well under all types of neural networks.
   - It can correctly identify the consumers of tensors in inference and training stages if the networks are layer-based and data flows from one layer to the next. However, the tensor transfer decisions may not be optimal as we have not optimized our design for network types beyond linear CNNs.
   - If in a non-linear CNN, a tensor is used twice during inference, and the time gap between the two references is long, offloading the tensor during inference could improve performance. The current design only allows offloading in inference, so it cannot exploit this potential performance improvement. The same argument applies to training.
2. AutoVM cannot identify the operations that perform in-place updates because such information is not provided by TensorFlow. However, knowing whether an operation does in-place update is important when making tensor transfer decisions. The current solution is to manually examine the implementations of the operations used in linear CNNs and generate a list of operations that do in-place update.
3. Our design works sub-optimally when the memory transfer time is longer than most of the computation runtimes. The calls to the CUDA runtime are invoked from operations, so, memory transfers can only be initiated at the start of a layer's computation. Furthermore, the control dependency method[5] we use will block the subsequent computations until the memory transfer completes. If the computation takes longer, there will be no problem; but if the computation time is less than

[5] As introduced in §2.3.3, control dependencies are used to enforce execution order of operations when there are no data dependencies between them.

the memory transfer time, the memory transfer will block the next computation. As a result, if most computations run faster than the memory transfers, many computations block for memory transfer, negatively impacting performance.

Chapter 4

Reverse engineering Nvidia virtual memory

Nvidia does not provide detailed documentation that describes the behaviour of the GPU virtual memory subsystem. However, for our work it is essential that we understand precisely how the virtual memory system behaves, since our work primarily targets managing GPU virtual memory to speed up CNN applications. We have thus designed and conducted a series of experiments to "reverse engineer" the virtual memory behaviours of Nvidia GPUs.

Our experiments are designed to reveal the following aspects of Nvidia's virtual memory system:

1. how to launch immediate data transfer between global and host memory, given the fact that Nvidia provides two APIs for initiating data transfer, i.e., `cudaMemAdvise()` and `cudaMemPrefetchAsync();`
2. how to transfer data between global and host memory efficiently, in TensorFlow-based machine learning applications, and
3. what is the page migration throughput over PCI-Express.

Experimental setup

Table 4.1 summarizes information on the system we performed our experiments on. We installed 96GB of host memory in the system, about 9 times the amount of global memory, to ensure there is enough space on the host side to accommodate the data paged out from global memory.

Item	Value	Specification
	Hardware setup	
CPU	Intel i9-9820x	10 cores @ 3.30 GHz
Memory	96GB	DDR4-2666
	GPU information	
GPU	Nvidia RTX 2080Ti	Turing TU102 architecture
Compute capability	7.5	
Memory size	11 GB	~9.5 GB usable
Memory type	GDDR6	
Memory bus	352 bit	
Memory throughput	616.0 GB/s	
Host interface	PCI-Express	@3.0x16
Measured Interface throughput	13.0 GB/s	
	Software setup	
CUDA Driver version	418.56	
CUDA Runtime version	10.1	
CUDA cuDNN version	7.5	
TensorFlow version	r1.14	
Operating system version	Ubuntu 18.04.1	Linux kernel v5.0.0

Table 4.1: Environment setup.

```
__global__ void kernel(float *input, float *output, size_t num_elem) {
    int threadId = ... // omitted for simplicity
    if (threadId < num_elem)
        output[threadId] = input[threadId] + 1;
}
```

Listing 4.1: CUDA kernel used as a simple computation workload.

Data collection method

We have used C++'s high precision timer to measure kernel run times in our experiments. In this chapter's listings, `time()` marks the places at which time measurements are taken. The amount of data transferred and the transfer throughput are measured using Nvidia's *Visual profiler*.

4.1 `cudaMemPrefetchAsync()` v.s. `cudaMemAdvise()`

Nvidia provides two methods, namely `cudaMemPrefetchAsync()` and `cudaMemAdvise()`, for managing data locality. Intuitively, both methods can transfer data between global and host memory, however, Nvidia documentation does not state clearly as to when the data transfers start after either method's invocations. This experiment aims to discover the transfer characteristics of both methods, and to help us decide which one to use in implementing AutoVM.

```
size_t num_elem = 1 << 29, bytes = num_elem * sizeof(float);
float *buf1 = cudaMallocManaged(bytes);  // a 2GB buffer
float *buf2 = cudaMallocManaged(bytes);  // a 2GB buffer
kernel<<<>>>(buf1, buf2, num_elem);  // launch the kernel
cudaDeviceSynchronize();  // wait until the kernel finishes
// start data transfer
cudaMemPrefetchAsync(buf1, bytes, CPU_DEVICE, stream);
cudaMemAdvise(buf2, bytes, cudaMemAdviseSetPreferredLocation, CPU_DEVICE);
```

Listing 4.2: Test to reveal the difference between `cudaMemPrefetchAsync()` and `cudaMemAdvise()` in transfers from global memory to host memory.

```
size_t num_elem = 1 << 29, bytes = num_elem * sizeof(float);
float *buf1 = cudaMallocManaged(bytes);  // a 2GB buffer
float *buf2 = cudaMallocManaged(bytes);  // a 2GB buffer
// make sure buf1 and buf2 reside in host memory
for (int i = 0; i < num_elem, i++)
    buf1[i] = i; buf2[i] = i;
// start data transfer
cudaMemPrefetchAsync(buf1, bytes, GPU_DEVICE, stream);
cudaMemAdvise(buf2, bytes, cudaMemAdviseSetPreferredLocation, GPU_DEVICE);
```

Listing 4.3: Test to reveal the difference between `cudaMemPrefetchAsync()` and `cudaMemAdvise()` in transfers from host memory to global memory.

4.1.1 Method

Listing 4.2 shows the code we use to test the differences in transferring data between the two Nvidia-provided methods. The kernel that is invoked to represent a simple computation workload is shown in Listing 4.1.

We allocate two 2GB buffers in GPU virtual memory, launch `kernel` that accessed these buffers, and wait until `kernel` finishes. Then, we use `cudaMemPrefetchAsync()` to transfer `buf1`; and use `cudaMemAdvise()` to transfer `buf2` to host memory. The two buffers will reside in global memory after having been accessed in `kernel`. If both methods start transferring data immediately after invocation, we would observe in the profiler two separate memory transfers after kernel finishes and 4GB of data will have been transferred in total.

In a separate experiment we test the two methods on data transfers from host memory to global memory. The test code is shown in Listing 4.3: we ensure the two 2GB buffers are resident in host memory by accessing all of their elements from the CPU before the transfers start. Next, we call both Nvidia methods in a similar fashion as Listing 4.2, but change the transfer destination to `GPU`. If both methods transfer data to global memory after immediately invocation, we would observe two memory transfers and 4GB of data being transferred in the profiler.

```
size_t num_elem = 1 << 29;
size_t bytes = num_elem * sizeof(float);
float *buf1 = cudaMallocManaged(bytes);  // 2GB
float *buf2 = cudaMallocManaged(bytes);  // 2GB
float *buf3 = cudaMallocManaged(bytes);  // 2GB
// transfer buf1 and buf2 to GPU
cudaMemPrefetchAsync(buf1, bytes, GPU_DEVICE, stream1);
cudaMemPrefetchAsync(buf2, bytes, GPU_DEVICE, stream1);
cudaDeviceSynchronize();  // wait until data transfers finish
kernel<<<>>>(buf1, buf2, num_elem, stream1); // K1
cudaEventRecord(event, stream1); // records an event on stream 1
cudaMemPrefetchAsync(buf1, bytes, CPU_DEVICE, stream1);
cudaEventSynchronize(event);     // blocks CPU until K1 finishes
time();  // Measure K2 run time
kernel<<<>>>(buf2, buf3, num_elem, stream2); // K2
cudaDeviceSynchronize();
time();  // Measure K2 run time
```

Listing 4.4: Test of overlapping memory transfer with computation.

4.1.2 Findings

We only observed the transfers initiated by `cudaMemPrefetchAsync()` in the profiler, in other words, calling `cudaMemAdvise()` did not incur any data transfer in either of the two experiments. We have thus confirmed that calling `cudaMemAdvise()` will not incur immediate data transfers, and decided that we use `cudaMemPrefetchAsync()` in our implementation of AutoVM.

4.2 Efficient memory transfer between devices

This section presents our experiments that explore how to achieve efficient data transfer between global and host memory, as well as ways to accelerate `kernel` execution when global memory is over-subscribed.

4.2.1 Overlapping memory transfer with computation

In this experiment, we aim to ascertain that we can achieve have computation and data transfer overlap, by using GPU streams as suggested by Nvidia [18]. If they overlap, then the communication can be hidden from a performance point of view. This experiment also aims to verify that no page fault will be generated during the execution of a kernel, if all the data accessed by that kernel reside in global memory.

Methods

Listing 4.4 shows the code we use in this experiment. We create three 2GB buffers and ensure `buf1` and `buf2` reside in global memory. Then we launch `kernel` K1 on `stream1` using `buf1` and `buf2`. Next, the event that marks the finish of K1 is recorded on `stream1`. We then launch the transfer of `buf1`

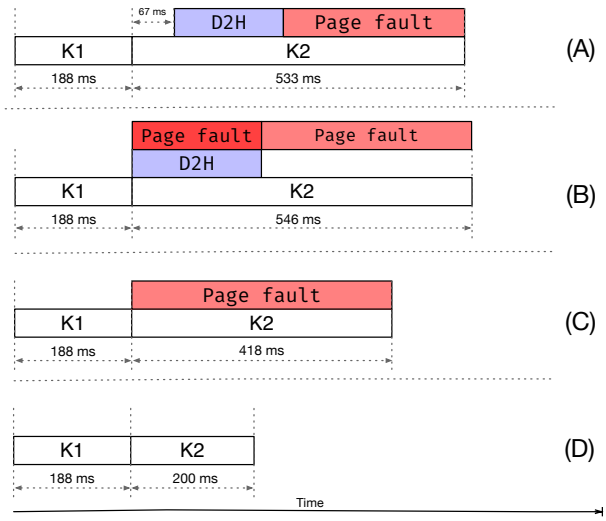

Figure 4.1: Profiling results of experiments on Nvidia virtual memory. The lengths of the blocks are drawn to scale to represent the kernel run times. Buffer allocation and transferring `buf1` and `buf2` to global memory are not shown.

to host memory on `stream1`, so that the transfer starts after K1 finishes. `cudaEventSynchronize()` is used to block the CPU, until K1 finishes, and we launch kernel K2 on `stream2`. In theory, K2 and the transfer of `buf1` should overlap, since they should both start after K1 finishes.

Findings

Figure 4.1 line (A) shows the results of the experiments using the code in Listing 4.4. In the figure, `D2H` represents the transfer of `buf1` to host memory; the `Page Faults` block shows the time range where page faults are generated as K2 accesses `buf3`. We have observed that no page faults are generated during K1's execution, because all the data it accesses reside in global memory.

There were a few unexpected behaviours that we observed however.

1. The start of `D2H` is delayed by around 70ms to after K1 finishes. However, according to Nvidia's documentation on streams, `D2H` should start right after K1 finishes.
2. K2's computation does not start until `buf1`'s transfer completes. This is implied from the fact the page faults for `buf3` starts after `buf1`'s transfer completes, and the kernel's only computation task that generates page faults is when accessing `buf3`. In other words, if K2's computation started with `D2H`, the page faults should have overlapped with `D2H`.
3. The profiler reported that the amount of data transferred is around 900MB, which is much smaller than the requested amount of 2GB. Furthermore, the transferred amount varies across experiments even though the same code is used.

```
1 // omitted the first 9 lines, which are identical to Listing 4.4
2 kernel<<<>>>(buf1, buf2, num_elem, stream1);  // K1
3 // blocks CPU until the previously submitted tasks are done
4 cudaDeviceSynchronize();
5 kernel<<<>>>(buf2, buf3, num_elem, stream2);  // K2
6 cudaMemPrefetchAsync(buf1, bytes, CPU_DEVICE, stream1);
```

Listing 4.5: The code with the launch order changed to try to overlap K2 and data transfer. The first 9 lines of code are identical with Listing 4.4 and are omitted here. Red font highlights the major code changes relative to Listing 4.4.

This experiment shows that (i) launching kernels and transfers in the order in Listing 4.4 does not overlap computation and data transfers; (ii) if all the data accessed by a kernel resides in global memory, running the kernel does not generate any page fault.

4.2.2 Alternating the launch order

As the previous methods produce unexpected results, we then tried another order of launching kernels and data transfers, namely the one shown in Listing 4.5.

Methods

The differences between this method and the previous one are: (i) we do not rely on implicit synchronization within a stream and we use `cudaDeviceSynchronize()` explicitly[1]; and (ii) we launch `buf1`'s transfer on `stream1` after launching kernel K2 on `stream2`. As the CPU is blocked by `cudaDeviceSynchronize()` until K1 finishes, both data transfer and K2 are launched on idle streams. K2's run time is measured from the end of K1 till the end of K2.

Findings

The experimental results are shown on line (B) in Figure 4.1 on page 52. Although the page faults (and hence the computation) do overlap with the data transfer, the run time of K2 increases relative to case A. From the profiler, we determined that the global memory throughput is 67.4MB/s when the red page fault block overlaps with `D2H`. The throughput increased to 4,295MB/s when page faults are processed, and there are no ongoing data transfers, shown as the second page fault box in pink.

We thus conclude that page fault handling is the main reason that K2 ran slowly, because global memory throughput is much lower when page fault handling is running in parallel with a data transfer. If data transfers and page fault handling could run at full speed independently, K2 would have finished in a much shorter period of time.

[1] In Listing 4.4 line 10 and 12, we have assumed launching both K1 and `cudaMemPrefetchAsync()` on `stream1` guarantees that the data transfer starts immediately after K1 finishes.

```
1 // omitted the first 9 lines, which are identical to Listing 4.4
2 kernel<<<>>>(buf1, buf2, num_elem, stream1);  // K1
3 // blocks CPU until previously submitted tasks are done
4 cudaDeviceSynchronize();
5 // pre-access, so that buf3 resides in global memory
6 cudaMemPrefetchAsync(buf3, bytes, GPU_DEVICE, stream1);
7 kernel<<<>>>(buf2, buf3, num_elem, stream2);  // K2
```

Listing 4.6: The code used to avoid page faults generated during executing the second kernel. The first 9 lines of code are identical with Listing 4.4 and are omitted here. Red font highlights the major code changes relative to Listing 4.4.

4.2.3 Avoiding page faults

The result from the previous subsection suggests that page fault handling interferes with data transfers between global and host memory. Therefore, it makes sense to eliminate page faults prior to starting a computation.

We have found that `cudaMemPrefetchAsync()` can be used to force the pages in a newly allocated buffer to reside physically in global memory, without having to access the buffer manually. We call this *pre-accessing* in the later parts of this dissertation. In this experiment, we aim to ascertain that pre-accessing can help avoid page faults and in turn reduce page fault handling overhead to speed up execution.

Methods

We ran the experiment using the code in Listing 4.6. We set up the buffers the same way as in Listing 4.5; then we launch kernel K1 and wait until it completes. Before launching kernel K2 on `stream2`, we pre-access `buf3` to global memory on `stream1`. For comparison, we have also run an experiment using the same code, but without pre-accessing `buf3`, so that K2 relies on demand paging to transfer `buf3` to global memory.

Findings

Line C of Figure 4.1 on page 52 shows the result of our experiment where `buf3` was not pre-accessed, but demand paged into global memory; line D shows the result where `buf3` was pre-accessed. By comparing the results from both experiments, we see that pre-accessing `buf3` has indeed eliminated the page faults generated during executing K2; and allowed K2 to complete in a much shorter time. Relying on demand paging to transfer in `buf3`, on the other hand, is only half as fast. We conclude that using pre-accessing yields better run time than relying on demand-paging.

```
1  // omitted the first 9 lines, which are identical to Listing 4.4
2  kernel<<<>>>(buf1, buf2, num_elem, stream1);  // K1
3  // blocks CPU until previously submitted tasks are done
4  cudaDeviceSynchronize();
5  time()
6  // a. transfer buf1 to host memory
7  cudaMemPrefetchAsync(buf1, bytes, CPU_DEVICE, stream1);
8  // b. pre-access, so that buf3 resides in global memory
9  cudaMemPrefetchAsync(buf3, bytes, GPU_DEVICE, stream2);
10 kernel<<<>>>(buf2, buf3, num_elem, stream2);  // K2
11 cudaDeviceSynchronize();
12 time()
```

Listing 4.7: The code used to test transferring `buf1` to host memory and avoid page faults generated during executing the second kernel. The first 9 lines of code are identical with Listing 4.4 and are omitted here. Red font highlights the major code changes relative to Listing 4.6.

This result is important for accelerating machine learning applications in TensorFlow, because the output tensor of a TensorFlow operation is always allocated right before the computation starts. As a result, the computation can only accesses its output using demand paging. Pre-accessing the output tensor before starting the computation helps reduce the number of page faults generated during the computation, and in turn speeds up the execution of the operation.

4.2.4 AutoVM and pre-access

This experiment explores the performance characteristics of running `kernels` when global memory is oversubscribed.

Methods

Listing 4.7 shows the code we use for this experiment. The code is similar to the previous tests, but all buffers are now 4GB instead of 2GB. kernel K1 runs after `buf1` and `buf2` have been manually transferred to global memory. After K1 finishes, `buf1` and `buf3` are managed in three different ways in a set of three experiments:

1. without any memory optimization, i.e. without pre-accessing `buf3` or evicting `buf1` to host memory, on lines 7 and 9,
2. with pre-accessing only; i.e. with line 9 that pre-accesses `buf3` on `stream1`, and
3. with transferring `buf1` to host memory on `stream1`, and pre-accessing `buf3` on `stream2`.

Then, we launch kernel K2 on `stream3` using `buf2` and `buf3`. K2's run time is measured from the end of K1 till the end of K2.

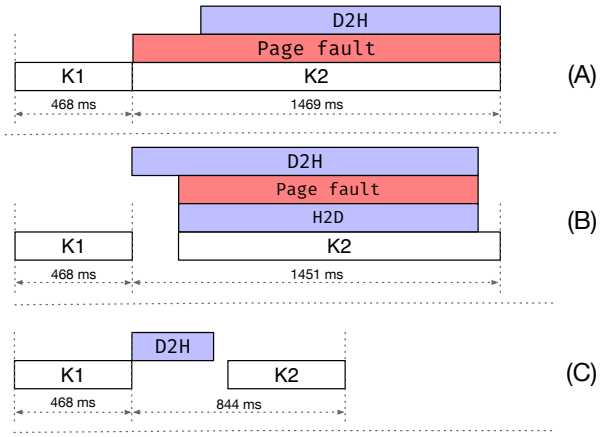

Figure 4.2: Profiling result of using three 4GB buffers with three different memory optimization strategies.

Findings

The run times of K2 using the three configurations were 1,469ms, 1,451ms, and 844ms respectively; Figure 4.2 shows their profiling results. In particular:

1. Line A shows the result of running K2, without using pre-access or data transfers. First, K2 runs without having to evict pages. But after a while, as global memory become fully populated, some pages have to be evicted to host memory to accommodate `buf3`. These evictions are manifested by the `D2H` block. Since `buf3` is only mapped but no physical frames are allocated, it does not incur transfers between global and host memory.
2. Line B shows the result of running K2 after pre-accessing `buf3`. Before K2 starts, CUDA runtime has evicted some pages to host memory to make space for `buf3` in global memory, shown as the `D2H` block. However during this process, Nvidia's memory management subsystem has evicted pages that belong to `buf2` to host memory. As a result, page faults are generated when K2 accesses those evicted pages. The demand-paging process is manifested by the `H2D` block.

   In this case, pre-access does not eliminate the page faults generated; and K2's run time is only negligibly better than running K2 without pre-accessing or evicting `buf1`.
3. Line C shows the result of running K2 after transferring `buf1` to host memory, and pre-accessing `buf3` to global memory. Transferring `buf1` to host memory has made 4GB of space available in global memory so that pre-accessing `buf3` can complete without having to evict pages. Consequently, running K2 does not generate any page fault, allowing K2 to execute over 74% faster, compared to the previous two cases.

Using pre-access alone is not sufficient to eliminate K2's page faults when global memory is oversubscribed. In fact, transferring `buf1` to host memory before pre-accessing new buffers resembles AutoVM's behaviour in TensorFlow applications.

4.3 Throughput of `cudaMemPrefetchAsync()` transfers

We measured the throughput of data transfers between global and host memory initiated by `cudaMemPrefetchAsync()` using Nvidia's visual profiler. The throughput was around 12.0GB/s, almost fully utilizing the available PCI-Express bandwidth of our GPU. However, we found that if the transfers are accompanied by page fault handling, the throughput could drop to around 9.0GB/s, using only 72.0% of the PCI-Express bandwidth. Our results differed from those obtained by Rhu et al. on an earlier version of CUDA runtime system [16]. They claimed that the page migration throughput was only around 80 to 200MB/s, which is much lower than our measured figures.

Chapter 5

Implementation

As outlined in §3.2, AutoVM is split into a policy part and a mechanism part. The policy part analyzes a machine learning application's computation graph and decides which tensors to transfer and when. The policy's decision then informs the mechanism to initiate the corresponding memory transfers by inserting instances of our `MemOp()` operation, into the computation graph at appropriate locations. As such, memory transfers introduced by using AutoVM's policy are carried out as TensorFlow executes the computation graph.

We implement the policy part in the TensorFlow client using its graph editor module; the mechanism, `MemOp()`, is implemented in the TensorFlow master. We have also created an interface that allows existing machine learning applications to easily integrate AutoVM. This chapter discusses the details in implementing AutoVM's policy, the interface for integration and the mechanism.

5.1 Overview

Listing 5.1 shows the structure of a simplified TensorFlow application that performs linear regression (written with TensorFlow's Python client). There are four typical steps:

1. **Model definition:** lines 2 and 3 define the input/output and weight variables. Line 4 sets up the dataflow, in this example, $y = \mathbf{xw}$; and line 5 defines the square loss function. TensorFlow translates these definitions into a computation graph as shown in the portion to the left of the dashed line in Figure 5.1.

```
1  # 1. model definition
2  x = tf.input(), t = tf.input()
3  w = tf.trainable_variable()
4  y = tf.matmul(x, w) # model definition
5  loss = tf.square_loss(y, t) # a loss function definition
6
7  # 2. gradient optimizer definition
8  gradients = tf.gradients(loss)  # generate gradient calculation steps
9  opt = tf.GradientDescentOptimizer(...).apply_gradients(gradients)
10
11 # 3. connect to the backend runtime
12 session = tf.Session()
13
14 # 4. run the application
15 session.run(opt)
```

Listing 5.1: A simplified TensorFlow application structure

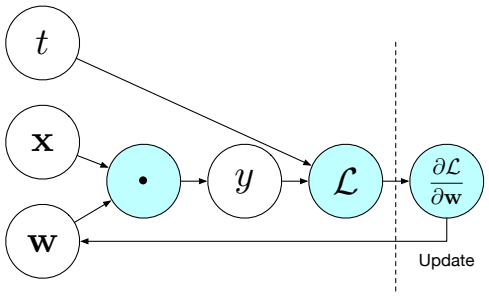

Figure 5.1: The computation graph built with the code in Listing 5.1. The portion on the left of the dashed line represents the graph corresponding to the inference dataflow; the portion on the right represents the gradient calculation and weight update process.

2. **Gradient definition:** lines 8 and 9 add necessary gradient computation and weight update steps into the dataflow by using TensorFlow's automatic gradient calculation API. The right portion of Figure 5.1 shows the added nodes for gradient computation ($\partial\mathcal{L}/\partial\mathbf{w}$) and the weight update.
3. **Connect to backend runtime:** on line 12, we instantiate a session with TensorFlow's backend runtime (TensorFlow master).
4. **Run the application:** run the computation graph on the hardware through the session (line 15).

AutoVM analyzes the computation graph with the gradient calculation and weight update steps, because AutoVM operates on machine learning applications that have both inference and training dataflow defined. AutoVM inserts instances of `MemOp()` — the mechanism — into the computation graph to trigger memory transfers during execution.

`MemOp()` is implemented as a user-defined operation in the TensorFlow master.[1] A key challenge we faced is that, `MemOp()` needs to invoke the appropriate Nvidia APIs to transfer tensors to and from global memory, while by default TensorFlow does not allow operations to control memory directly.

[1] TensorFlow requires all operations be implemented in the master.

The following sections present the implementation details of the policy and the mechanism.

5.2 The policy

The policy is implemented entirely in the TensorFlow's Python client[2] for two reasons, even though both TensorFlow's client and master operate on the computation graph that the user defined:

1. Only the client provides intuitive APIs for graph editing, which allows us to add each operation's control dependencies and redirect operation inputs and outputs.
2. Most TensorFlow-based machine learning applications are implemented in TensorFlow's Python client. Implementing AutoVM in the Python client makes the integration of AutoVM easier.

The computation graph is a directed acyclic graph represented in TensorFlow client's graph format where graph nodes are `operations` and edges are `tensors`. Each operation (node) keeps lists of its input and output tensors; each tensor (edge) keeps records of its producer and consumers. Given any node in the computation graph, TensorFlow can traverse the graph and determine an execution path that will produce that node's result.[3] Adding an operation into the computation graph is similar to adding a node to a linked list: instantiate an operation (node) and route its corresponding input/output tensors.

AutoVM takes in the computation graph generated by TensorFlow's client, defined in the first two steps shown in Listing 5.1 and analyzes it to produces a mapping for every tensor, $\{t : op\}$, with the tensors selected for transfer as keys, and the corresponding timings for offloading/prefetching as values. Here timing is represented by an operation, indicating the transfer should start when the operation starts. In particular,

- AutoVM gets all the operations used in inference and determines the offload timing for each operation's output tensor t by analyzing t's consumer list. Specifically, t is offloaded after its last consuming operation op_l finishes during inference. So the op in t's mapping is the next operation after op_l.
- For every tensor t selected for offload, AutoVM identifies when to schedule its prefetch so that the transfer finishes before t is accessed. AutoVM locates t's first consumer in training by analyzing t's consumers. Then, t's prefetch is scheduled as early as possible to ensure that t arrives in global memory before it is consumed, since the exact runtimes of the operations are unknown.

[2]Python client refers to the Python implementation of TensorFlow's client.
[3]This is the reason that on line 15 of Listing 5.1, the `run()` method's parameter only includes the `opt` operation.

```
# Method 1
loss = ... # model definition
gradients = tf.gradients(loss)  # generate gradient calculation steps
opt = tf.GradientDescentOptimizer(...).apply_gradients(gradients)

# Method 2
loss = ... # model definition
opt = tf.GradientDescentOptimizer(...).optimize(loss)
```

Listing 5.2: Two different methods for adding gradient calculation and weight update steps in TensorFlow.

AutoVM then uses this mapping produced by the policy stage to insert instances of `MemOp()` into the computation graph using control dependencies, a mechanism TensorFlow uses to ensure execution order between operations that do not have direct data dependencies. For instance, setting *op* as the control dependency of an instance of `MemOp()` m ensures *op* executes before m does. In our implementation, *op*'s computation kernels and m's memory transfer requests are submitted to different GPU streams, so that the computation and memory transfer can run in parallel. The computation graph with `MemOp()`s inserted is ready to be transferred to the master and start execution.

5.2.1 Integrating AutoVM

This subsection focuses on the interface we provide to integrate AutoVM into existing machine learning applications.

As described in §5.1, machine learning engineers define the inference dataflow and use a TensorFlow API to generate the steps for gradient descent and weight update. There are mainly two ways to generate those steps as shown in Listing 5.2:

1. Manually call `tf.gradients()` to add the gradient calculation steps and then instantiate an optimizer to apply the gradients to update the network weights. This method is often used in machine learning applications where some additional computation is performed on the gradients, before the gradient is applied to weight update. For example, in some machine learning applications, the weights are updated using squared gradient instead of the raw gradient computed by `tf.gradient()`. So an additional step that squares the gradient needs to be inserted after calling `tf.gradient()`.
2. Alternatively, if no extra precessing is needed, machine learning engineers can instantiate an optimizer and let it generate the necessary steps. The `optimize()` method effectively wraps the call to `tf.gradient()` and `apply_gradients()` to provide a simpler interface.

```
def avm_gradient(loss, **kwargs):
    gradients = tf.gradients(loss, **kwargs)
    avm.optimize(tf.get_computation_graph())
    return gradients
loss = ... # model definition
gradients = avm_gradients(loss)  # use the new gradient function
opt = tf.GradientDescent(learning_rate).apply_gradients(gradients)
```

Listing 5.3: Example of using the first method to integrate AutoVM.

```
loss = ... # model definition
opt = tf.GradientDescentOptimizer(...).optimize(loss)
avm.optimize(tf.get_computation_graph())
```

Listing 5.4: Example of using the second method to integrate AutoVM.

Correspondingly, we have designed two ways to integrate AutoVM into TensorFlow-based machine learning applications, to accommodate the two methods shown in Listing 5.2.

1. As shown in Listing 5.3, we have created a function `avm_gradient()` that wraps the calls to `tf.gradient()` and `avm.optimize()`,[4] such that calling `avm_gradient()` will incur both TensorFlow's gradient and AutoVM's optimization functions. Effectively, machine learning engineers can integrate AutoVM by changing only the gradient function. This method is tailored for the first method shown in Listing 5.2.
2. Alternatively, machine learning engineers can invoke AutoVM in a separate function call. This is a universal solution that works in both listed methods, but it requires inserting one line of code that calls `avm.optimize()` after instantiating the `GradientDescentOptimizer`, as shown in Listing 5.4.

We have implemented both integration methods so that machine learning engineers can choose either one that better suits their applications. It is important to note that AutoVM performs the exact same tasks i.e., analyzes the graph and inserts the necessary nodes of `MemOp()`, regardless of how it is invoked.

5.3 The mechanism `MemOp()`

The `MemOp()` method is inserted into the computation graph at appropriate locations identified by the policy. It transfers its input tensors to and from global memory by calling Nvidia's managed memory API, `cudaMemPrefetchAsync()`.

As mentioned earlier, by default TensorFlow does not provide operations for controlling memory. In fact, an operation does not even have access to tensors' actual storage locations. Thus, TensorFlow does not have a way to call `cudaMemPrefetchAsync()` even though it is essential for AutoVM.

[4]The function `avm.optimize()` takes in a computation graph and applies AutoVM's optimization steps.

```
1 Stream &ThenOffloadGPU(const DeviceMemoryBase &gpu_src, uint64 size,
2                        Stream *offload_stream);
3 Stream &ThenPrefetchGPU(const DeviceMemoryBase &gpu_src, uint64 size,
4                         Stream *prefetch_stream);
```

Listing 5.5: The new `StreamExecutor` function signatures we have added to support Nvidia GPU's virtual memory.

`MemOp()` must be able to call `cudaMemPrefetchAsync()` from within an operation. In the next subsections, we present how we achieve this in two parts:

1. Adding the support for calling `cudaMemPrefetchAsync()` to TensorFlow, and
2. Enabling calling `cudaMemPrefetchAsync()` from a TensorFlow operation.

5.3.1 Adding support for `cudaMemPrefetchAsync()`

`StreamExecutor` is TensorFlow's interface that abstracts and standardizes the interfaces of streaming processor related functions, including

- invoking functions in high performance libraries like BLAS,
- managing devices functions, for instance, memory allocation, memory copy and synchronization.

`StreamExecutor` only defines an interface, which standardizes implementations from multiple different vendors.[5] The CUDA-specific implementation of `StreamExecutor` makes calls to the CUDA runtime, driver, and libraries like cuBLAS and cuDNN. It is thus natural to add to the interface function signatures for virtual memory prefetching/offloading and add the corresponding functions that invoke `cudaMemPrefetchAsync()` to the CUDA-specific `StreamExecutor`.

We have added two function signatures for virtual memory control to `StreamExecutor` as shown in Listing 5.5. `ThenOffloadGPU()` is used to offload data, identified by its starting address and length, to host memory on a specified stream; `ThenPrefetchGPU()` on the other hand, prefetches data to global memory. In their respective implementations, we include calls to `cudaMemPrefetchAsync()` to transfer the data stored at the address from/to global memory, on the specified stream.

5.3.2 Accessing `cudaMemPrefetchAsync()` from an operation

Operations are not given direct access to `StreamExecutor` by TensorFlow. This subsection describes how we enable an operation to use the `StreamExecutor` methods that we defined in Listing 5.5.

[5] The only publicly available and officially supported implementation is for Nvidia CUDA, however.

```
1 Tensor *do_CUDA_GEMM(Tensor *opA, Tensor *opB) {
2   Tensor *output(...);  // allocate output.
3   ThenPrefetchGPU(output, size, stream); // pre-access
4   cuda_GEMM(opA, opB, output); // call CUDA function
5   return output;
6 }
```

Listing 5.6: Simplified example of adding pre-access to an existing operation's `StreamExecutor` implementation. We add the code segment in red to pre-access the output buffer.

Internally, TensorFlow provides a mechanism to transfer tensors between devices. TensorFlow guarantees that all of the operands of an operation *op* are present on the device that executes *op*. To achieve this, TensorFlow inserts special operations, `SendOp` and `RecvOp`, that transfer tensors between devices. For example, transfers from host memory to global memory is scheduled when TensorFlow identifies (during the graph optimization stage) a tensor is not allocated in global memory while the operation that needs that tensor executes on GPU. The transfers are initiated through an interface called *`Rendezvous`*. After examining the implementation of `Rendezvous`, we found that a module called `DeviceContext` uses `StreamExecutor` methods to transfer tensors between devices. We have also found that the `DeviceContext` module can be accessed via `OpKernelContext`, an object that is available to all operations.[6] As a result, we have `MemOp()` access `cudaMemPrefetchAsync()` through several levels of indirection: from `OpKernelContext` to `DeviceContext` to `StreamExecutor`, and eventually, `cudaMemPrefetchAsync()`.

5.4 Supporting pre-access

As mentioned in Chapter 4, using pre-access to force the pages of a newly allocated buffer to reside physically in global memory can improve the run time of the operations using that buffer. Pre-accessing is achieved using `cudaMemPrefetchAsync()`. When processing machine learning workloads on Nvidia GPUs, TensorFlow calls CUDA library (cuDNN) functions via the `StreamExecutor` interface, where our support for `cudaMemPrefetchAsync()` is implemented. As such, the pre-accessing for a TensorFlow operation that calls CUDA library function M and produces output O, is implemented as inserting a call to `cudaMemPrefetchAsync()` of buffer O after its allocation, prior to invoking function M. Listing 5.6 shows an example of pre-accessing the output of a matrix multiplication operation's CUDA implementation. The output buffer is allocated right before invoking the corresponding CUDA library function. We add the code segment in red to pre-access the output buffer.

[6] `OpKernelContext` stores the necessary context for operation execution, including the operation's inputs, outputs, handle to `Rendezvous`, and `DeviceContext`.

Chapter 6

Experiment

In this chapter, we present the results of the experiments conducted to test and verify the effectiveness of AutoVM. In the next sections, we introduce our environment settings, our experimental design, our experimental results, and the discussion on the results.

6.1 Environment setup

Table 6.1 summarizes information on the system we performed our experiments on. We installed 96GB of host memory in the system, about 9 times the amount of global memory, to ensure there is enough space on the host side to accommodate the data paged out from global memory. We use an Intel i9 processor in the setup because it is readily available, but the GPU virtual memory performance would be higher in an IBM Power9 system because Power9 CPU supports NVLink that offers almost twice the bandwidth of PCI-Express 3.0x16.

6.2 Experiment design

We have designed experiments to verify that using AutoVM improves the training speed over default Nvidia virtual memory. We used three iconic CNN structures to test AutoVM's effectiveness, namely AlexNet [13], VGG-19 [19] and ResNet-152 [7], and we used the images in the validation image set of the ILSVRC2012 dataset [17]. In the next subsections, we present the code used for our experiments, our test cases, and out data collection methods in detail.

Item	Value	Specification
	Hardware setup	
CPU	Intel i9-9820x	10 cores @ 3.30 GHz
Memory	96GB	DDR4-2666
	GPU information	
GPU	Nvidia RTX 2080Ti	Turing TU102 architecture
Compute capability	7.5	
Memory size	11 GB	around 9.5 GB usable
Memory type	GDDR6	
Memory bus	352 bit	
Memory throughput	616.0 GB/s	
Host interface	PCI-Express	@3.0x16
Measured Interface throughput	13.0 GB/s	
	Software setup	
CUDA Driver version	418.56	
CUDA Runtime version	10.1	
CUDA cuDNN version	7.5	
TensorFlow version	r1.14	
Operating system version	Ubuntu 18.04.1	Linux kernel v5.0.0

Table 6.1: Environment setup.

6.2.1 Experiment code

Listing 6.1 on page 67 outlines the code we used in our experiments. It consists of the following four parts:

1. **Specify session configuration:** First we set up the session configuration on lines 2 to 6. The parameter `per_process_gpu_memory_fraction` controls the amount of memory to allocate per GPU. We set this value to 4 so that TensorFlow allocates 43,952MB of virtual global memory. In experiments that do not use virtual memory, this value is set to allocate all available physical global memory, which is 0.9.[1]
2. **Define inference dataflow:** On lines 9 and 10 we set up the inference data flow. The input `images` and `labels` are loaded into host memory prior to when it is needed.[2] `network()` then calls appropriate TensorFlow methods to construct the inference dataflow for the specified CNN structure (shown in the example is an instantiation of VGG-19). On line 11, we use the built-in *softmax cross entropy* function to calculate the loss.[3]

[1] TensorFlow will allocate the 90% of physically available global memory, not 90% of free global memory.

[2] The images are resized to proper size (224×224) before running the experiments, so no extra time would be spent on image processing during our experiments. We used TensorFlow's `DataSet` module for data loading, which prefetches the next image batches to host memory, while the GPU trains the current batch.

[3] Softmax cross entropy is a loss function commonly used in multi-class classification problems.

```
1  # 1. specifying session configuration
2  config = tf.ConfigProto(
3    gpu_options=tf.GPUOptions(
4      per_process_gpu_memory_fraction=MEMORY_SCALE_COEF,
5      experimental=tf.GPUOptions.Experimental(use_unified_memory=True)
6    ))
7
8  # 2. defining inference dataflow
9  input, label = load_batch(batch_size)
10 inference = network(inputs, VGG19)
11 loss = tf.losses.softmax_cross_entropy(inference, onehot_labels=labels)
12
13 # 3. adding gradient calculation steps
14 vars = tf.trainable_variables()
15 grads = avm_gradient(loss, vars)
16
17 # 4. defining optimizer
18 optimizer = tf.train.AdamOptimizer()
19 training = optimizer.apply_gradients(zip(grad, vars))
20
21 # 5. connect to master and run
22 with tf.Session(config=config) as sess:
23   sess.run(tf.global_variables_initializer())
24   for i in range(ITERATIONS):
25     start = time()
26     sess.run(training)
27     batch_train_time = time() - start
```

Listing 6.1: TensorFlow code used for testing AutoVM. The code segments marked in red are the parameters we alter across experiments. They are discussed in detail in §6.2.2

3. **Add gradient computation steps:** Lines 14 and 15 are used to generate the steps for gradient computations. In the experiments that do not use AutoVM, `tf.gradient()` is called on line 15 instead of `avm_gradient()`.
4. **Define optimizer:** On lines 18 and 19, an Adam optimizer [12] is defined to apply the gradient results to update weights.
5. **Connect and run:** The last step instantiates a session of TensorFlow master using the configurations defined in the first step, and run the machine learning application iteratively. Every iteration trains one image batch, the run time is measured using Python timing APIs on lines 25 and 27. We call the time taken to train each batch, *batch training time*.

6.2.2 Test cases

The three parameters that we control in our experiments are memory management policy, batch size, and number of training iterations (marked red in Listing 6.1). The specific configuration of each parameter is presented in detail below.

1. **Memory management policy:** We run experiments with the following three memory management policies individually:

- virtual memory with the default virtual memory management that Nvidia provides,
- virtual memory with pre-accessing and AutoVM optimizations, and
- traditional memory without enabling virtual memory.

2. **Batch size:** The amount of memory required in machine learning application training is mostly linearly proportional to the batch size used. For each CNN structure, we select the batch size that leads to the maximum memory allocation in training (N_M) from around 1GB to around 24GB.
3. **Number of iterations:** Training a neural network until its accuracy converges typically requires a very large number of iterations. Since our primary goal is to test the effectiveness of a memory management policy that theoretically does not interfere with the training process, we ran all our experiments for 10 training iterations to verify that using AutoVM offers a performance advantage. However, to verify that AutoVM can improve end-to-end training time and AutoVM does not disturb training, for example, requiring more iterations to achieve a certain accuracy compared to using Nvidia virtual memory, we ran full training experiments using VGG-19 under all three memory management policies with batch sizes 64, 128, 192, and 256.

6.2.2.1 Training environment

The following methods were used to provide a variable-controlled environment that allowed for fair comparisons between experiments when using different memory management policies and batch sizes.

- **Image batches** We generated the image batches using images from the ILSVRC2012 validation dataset (50,000 images in total) in sequential order without shuffling. In particular, the i-th batch included the images numbered bi to $b(i+1)$ (non-inclusive), where b is the batch size. As such, the images in the i-th batch were always identical across all experiments that used the same batch size, for all values of i.
- **Weight** In the experiments with VGG-19, we initialized all the network weights using a pre-trained VGG-19 model[4] [1]. Initializing network weights to the same pre-trained model not only reduced the required training time, but also ensured an identical starting state of training, across experiments. In the experiments with AlexNet and ResNet-152, weights are randomly initialized with the same seed to ensure an identical starting state.
- **Optimizer** We used an *Adam optimizer* with learning rate 0.0001 [12].[5]

[4]The pre-trained VGG-19 model contains weight values of a VGG-19 model that has been trained previously. Although the pre-trained model might not have been trained with our dataset, it gives our model prior knowledge of common image features, like lines and colours.

[5]Adam Optimizer is a widely used gradient descent optimizer.

We verified that when training with the same batch size, the loss and accuracy results from every iteration were identical across multiple experiment runs, regardless of the memory management policy used. This allowed us to conclude two things. First, AutoVM does not interfere with the training result as expected. Second, it allows us to compare the end-to-end training times under different memory management policies using the same batch size.

6.2.3 Data collection methods

For each training iteration, we collected its accuracy, loss, and run time using the same way as shown in Listing 6.1. End-to-end run time was computed by summing all iteration run times. Further, we collected the following metrics in our experiments.

- **Batch training time** was measured from when TensorFlow started the training iteration to when TensorFlow signalled the completion of the iteration, using Python's `time` module. We report the truncated mean (calculated after removing the max and min values) and the standard deviation of the measured values.
- **Layer-wise memory usage** was recorded by using TensorFlow's application trace. For every test case, we ran the experiment once with tracing on to collect the memory usage information. The memory usage information did not vary between different iterations of the same experiment. Tracing was turned off when measuring training run times.

  We primarily focus on the maximum amount of memory allocated during the training process (N_M), which is mostly dictated by the CNN structure and the batch size used. Instead of reporting the absolute values as is, we report the normalized amount N_m in the presentations below, where N_m is the ratio between N_M and the total amount of physical global memory available (around 9.5GB in our machine).
- The following metrics were collected using Nvidia's visual profiler in experiments that were not timed. In each experiment, the following metrics were collected after training the CNN model for 10 iterations:
  - the amount of data transferred between global memory and host memory,
  - the data transfer bandwidth over PCI-Express, and

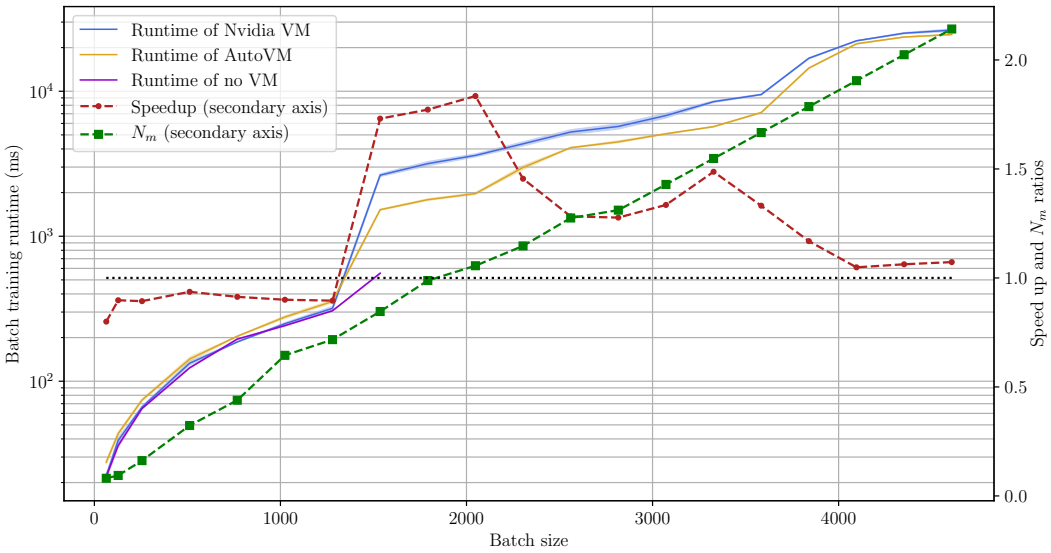

Figure 6.1: The measured run time of one AlexNet training iteration for at different batch sizes, using AutoVM, Nvidia virtual memory, and no virtual memory

6.3 Results

This section presents the results from our experiments with the three CNNs. In each graph we present:

- the x-axis plots the batch sizes used in the experiment,
- the primary y-axis plots the measured mean batch training times in logarithmic scale, with errors of plus/minus one standard deviation,
- the secondary y-axis plots the speed up of AutoVM over Nvidia virtual memory in linear scale
- the secondary y-axis plots the normalized memory usage N_m in linear scale, and
- the secondary y-axis plots a dashed line at $y = 1.0$ for visual reference.

6.3.1 AlexNet

Figure 6.1 shows the results of our experiments with AlexNet. We tested the different memory management policies while varying the amount of memory allocated, N_m from 0.08 to 2.14. The behaviour of AutoVM can be categorized into three segments based on the amount of memory required:

1. At smaller memory requirements with $N_m \leq 0.8$, it is possible to train AlexNet without using virtual memory, where the corresponding run times were the shortest among the memory management policies, although the performance advantage of training without virtual memory is insignificant, at around 3%. With smaller memory requirements, training with AutoVM was slower than with Nvidia virtual memory by around 8.7%.
2. At intermediate memory requirements where $0.8 \leq N_m \leq 1.8$, AutoVM reached a peak speed up of 83.4% at $N_m = 1.06$.
3. At high memory requirements with $N_m > 1.8$, AutoVM was around 5% faster than Nvidia virtual memory.

With lower memory requirements Training without using virtual memory is the fastest until batch size 1,536, because there is no virtual memory related overhead like page table lookup. All the tensors generated in training fit in physical global memory, so there is no need to transfer tensors between global and host memory.

Without offloading and prefetching, the only difference between training using AutoVM and training using Nvidia virtual memory was whether pre-accessing is used or not. As stated earlier, TensorFlow allocates output tensors right before the corresponding computations that produce them. Pre-accessing output tensors helps save page fault handling overhead by forcing the pages of the newly allocated tensors to be mapped physically, when global memory is not oversubscribed. On the other hand, TensorFlow manages memory internally to avoid invoking time-consuming memory functions (like `cudaMalloc()`) on every allocation. Similarly, tensor deallocation in TensorFlow only returns the used spaces to TensorFlow's internal free memory pool without using free functions like `cudaFree()`. As a result, the Nvidia driver only sees one allocation request on start-up and one free request on exit throughout the life cycle of a TensorFlow application. Consequently, when TensorFlow deallocates a tensor, although its storage space is marked free by TensorFlow, the Nvidia driver is unaware of the deallocation and will keep the corresponding page mappings. Later, if TensorFlow allocates another tensor at the same location, the Nvidia driver will not treat the tensor as newly allocated because the location's corresponding page mappings are still in the page table. That is, when everything can fit in global memory, none of the memory accesses made after the first training iteration[6] would generate a page fault.

As such, there should be no performance difference whether or not pre-accessing is used, under the assumption that pre-accessing introduces no extra overhead. However this assumption directly

[6]It should be emphasized that in first training iteration, pages of new tensors are not mapped so accesses during the first training iteration will generate page faults.

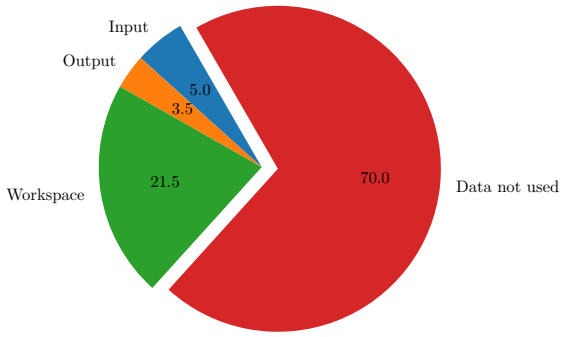

Figure 6.2: The itemized memory utilization when processing AlexNet's fourth (inference) convolution layer at batch size 3,328. The layer's computation uses only 30% of what is allocated.

contradicts our observations that using pre-accessing has negatively impact training time, as depicted in Figure 6.1 on page 70. Therefore, we conclude that using pre-accessing actually slows down training because it needs additional processing time, and the processing time is non-negligible compared to the batch training time.

With moderate memory requirements The maximum amount of memory allocated in training exceeds the amount of physical global memory, but the working set sizes of most computations fit in global memory. Training without using virtual memory is no longer possible. Under Nvidia virtual memory management, as the tensors produced by previous computations accumulate, thrashing starts once the newly allocated tensor no longer fits in physical global memory along with the tensors used in previous computations. On the other hand, AutoVM offloaded tensors to host memory in time to make space for subsequent computations. To illustrate, Figure 6.2 shows the itemized memory utilization in processing the fourth convolution layer in AlexNet using a batch size of 3,328. The total amount of memory allocated while processing the layer was 13.64GB, in which

1. the layer's input and output combined, used 1.16GB of memory,
2. the convolution workspace used 2.96GB of memory, and
3. the tensors used by previous computations, which would not be referenced in the near future, occupied 9.62GB of memory. The tensors in this category alone do not fit in physical global memory.

In this scenario, the working set of the convolution alone (items 1 and 2 above) is only around 4GB, but global memory is over-subscribed even before running the convolution. Without applying AutoVM's optimization, accessing the convolution's workspace and output would involve heavy paging activity. AutoVM, on the other hand, offloads the tensors that will not be accessed shortly (those belonging to the red region), so there is space available in physical global memory to map the workspace and output. The number of page faults incurred during processing the convolution is greatly reduced under AutoVM, thus reducing the run time.

With high memory requirements ($N_m > 2$) The overall training run time of both Nvidia virtual memory and AutoVM almost double near batch size 3,840, once the working set sizes of the last few gradient computations[7] in training do not fit in physical global memory. The run times are dominated by page fault handling during the execution of these computations, for the fact that the operations that have large memory requirements run more than ten times slower, over a 6.7% increase in batch size. In moderate memory requirement range, those operations' run times contribute to around 28.5% of the training time, but in high memory requirement range, they account for over 75.0% of the training time. This behaviour renders AutoVM's optimization less significant.

As specified in §3.3.1.4, AutoVM does not schedule prefetches when the working set size combined with prefetch size exceeds the physical memory limit. In this case, we expect the memory behaviours of AutoVM and Nvidia virtual memory to be almost identical. Therefore, the slight improvements observed while using AutoVM were the result of actively offloading tensors during inference for the same reason explained above: AutoVM offloads the tensors that are not referenced in the near future to relieve global memory pressure on for later computations.

Potential for further improvements We consider room for performance improvement mainly in the region of moderate memory requirements. The high memory utilization region is not targeted because the run time is dominated by heavy paging and will not likely be alleviated by using a better memory management policy. In the experiment using batch size 2,048 with AutoVM turned on, 91.9GB and 100.0GB of data are transferred to and from global memory at 71.3% and 89.4% of peak measured PCI-Express bandwidth. In the 24,170ms of training, AlexNet computation was running at full speed on the GPU for 82.6% of the time, while being blocked by memory operations (offloading, prefetching, pre-accessing, and page fault handling) for the other 17.4% of time. If the memory optimizations AutoVM

[7] The working set sizes of the last few gradient computations are the largest during CNN training.

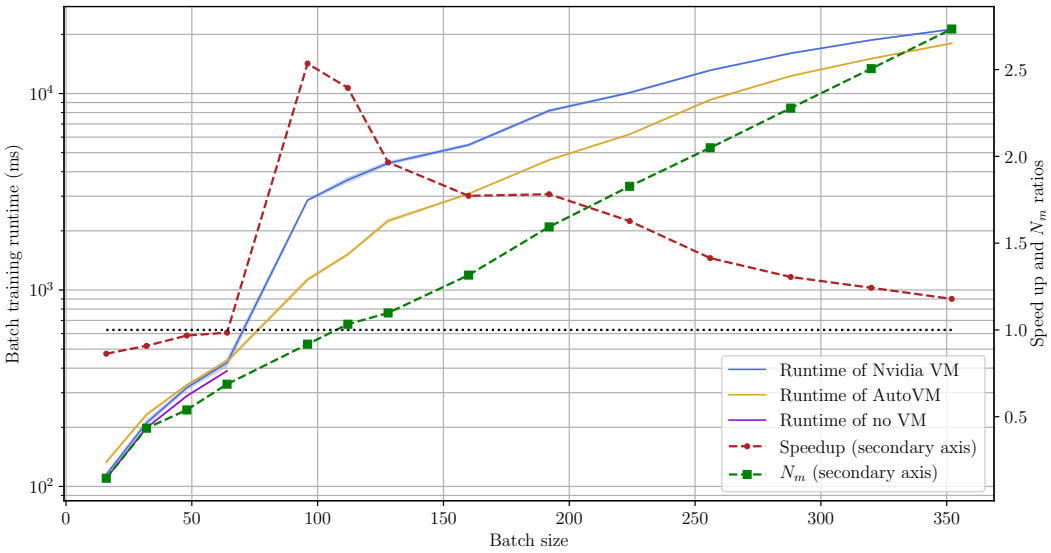

Figure 6.3: The measured run time of one VGG-19 training iteration for at different batch sizes, using AutoVM, Nvidia virtual memory, and no virtual memory.

introduced no extra overhead, the run time could theoretically be optimized further by 17.4%. The main optimization blocker here is the GPU virtual memory behaviour during inference when global memory is over-subscribed: AutoVM-initiated transfers and pre-accesses launched prior to starting an operation block the operation until they finish. The computation runs the fastest, even though the transfers and pre-accesses are almost synchronous, as demonstrated in §4.2.4.

6.3.2 VGG-19

Figure 6.3 shows the mean batch training time of VGG-19 for $N_m \in (0.14, 2.73)$. AutoVM attains its peak speedup near $N_m \approx 1.6$, at 2.53×. Similar to the result in AlexNet, AutoVM's performance can be summarized into three segments:

1. Before reaching $N_m = 0.8$, training without using virtual memory was still the fastest, but not by much (about 5%) compared to using the two other memory management policies. Training with AutoVM was around 3.24% slower than with Nvidia virtual memory.
2. In the range $0.8 \leq N_m \leq 1.8$, the speedup of AutoVM peaked near $N_m = 1.0$ (2.53×) and slowly decreases to 1.63 × at $N_m = 2.0$,
3. Above $N_m = 2.0$, AutoVM's speed up further decreased from 41.4% to 17.9% at $N_m = 2.73$.

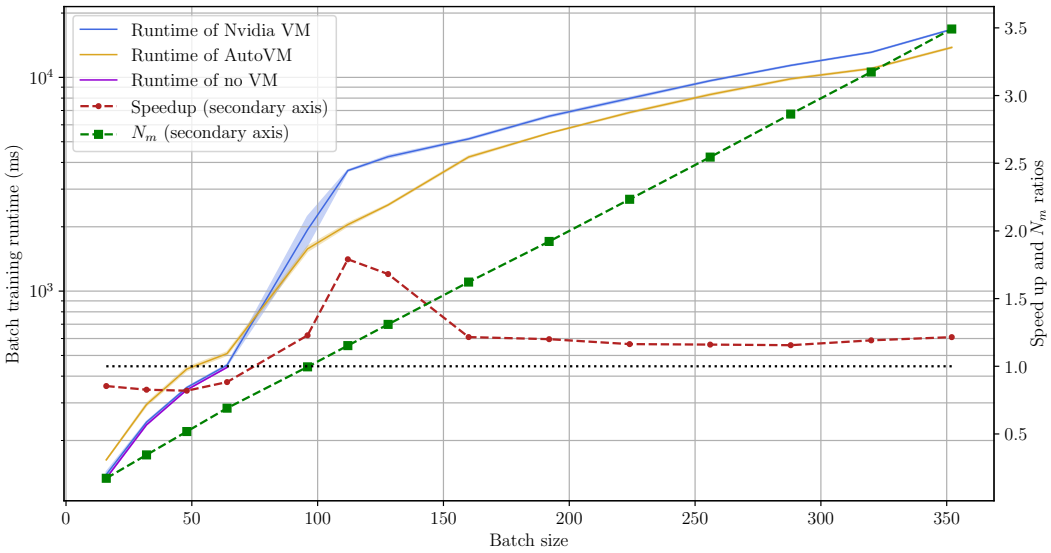

Figure 6.4: Measured one training iteration run time of ResNet-152 at different batch sizes, using AutoVM and Nvidia virtual memory.

The speedups AutoVM attained for VGG-19 with $N_m < 0.8$ is similar to that in AlexNet: using pre-accessing introduced extra performance overhead and thus slowed down training. AutoVM achieved higher speedup with $N_m > 2.0$ because there are more convolution computations in VGG-19 compared to AlexNet, and convolution computations benefit from the offloads AutoVM initiates during inference, as explained in §6.2.

At batch size 128, the transfer speeds of offload and prefetch were 86.3% and 76.9% of the measured peak PCI-Express bandwidth. The GPU compute cores were only actively running for 1,031ms out of the 1,850ms batch training time. About 58.6% of the idle time resulted from the computations being blocked by AutoVM introduced memory transfers while pre-accessing was the main cause of the other stalls. If no extra overhead was introduced by performing offload/prefetch the batch training time would be shortened to around 1,500ms, around 23.3% faster than the current solution. In theory, the run time could be shortened to 1,031ms if the computations could run at full-speed constantly and the memory optimizations introduces zero extra overhead.

6.3.3 ResNet-152

Figure 6.4 shows the results from experiments using ResNet-152, with $N_m \in (0.7, 3.5)$. Specifically,

1. Before reaching $N_m = 0.7$, training with AutoVM is around 19% slower than with Nvidia virtual memory. Training without virtual memory is still the fastest among the three memory management policies, but the advantage is insignificant, at about 3%.
2. In the region $1.0 \leq N_m \leq 1.5$, the speedup of AutoVM peaked at 79.0% and then dropped to around 20%.
3. In the region $N_m > 1.7$, the speedup of AutoVM stably converged to around 20%.

AutoVM was able to achieve moderate speed up even at higher memory utilization, because ResNet-152 is constructed with many smaller convolution computations that use less data, as compared to the convolution computations in AlexNet. ResNet-152 benefits more than AlexNet from the offloads AutoVM makes during inference and achieves higher speedups at higher memory requirements because the computations of ResNet-152 are mostly dominated by convolutions.

At batch size 112 where AutoVM achieved most speedup, 102.1GB and 98.6GB of data were transferred from and to global memory, at 99.8% and 58.1% of peak PCI-Express bandwidth, respectively. Computations were running for 89.3% of the total run time.

6.3.4 Full Training Run Experiment

We ran a set of experiments to measure the training time[8] required to reach 85% top-1 training accuracy. For this we used VGG-19 with the validation set of ILSVRC12 images, We trained the network at batch sizes 64, 128, 192, and 256 using configurations specified in §6.2.2.1. We compare the training time obtained under AutoVM, Nvidia virtual memory, and no virtual memory.

Figure 6.5 shows the comparison of end-to-end training times under the three memory management policies. The y-axis plots the total training time in linear scale and the x-axis plots the batch size. When using no virtual memory, TensorFlow was only able to train with batch size 64. We have recorded the number of iterations needed: 24,000 for 64-image batches, 6,000 for 128-image batches, 3,800 for 192-image batches, and 2,200 for 256-image batches.

At batch size 64, training using no virtual memory beat Nvidia virtual memory by 6.96% and AutoVM by 13.24%. At larger batch sizes 128, 192, and 256, AutoVM is 1.98, 1.82 and 1.53 times faster than Nvidia virtual memory.

[8] In our other experiments, we trained the networks for only 10 iterations to measure the batch training time.

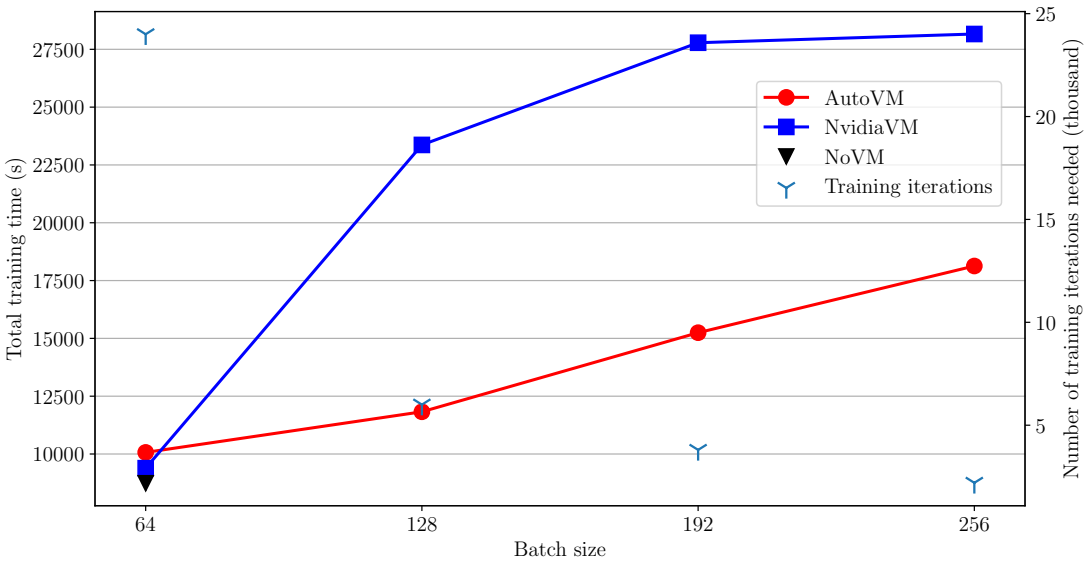

Figure 6.5: The training time of VGG-19 to achieve 85% accuracy, versus batch size, using three different memory management strategies. Only batch size 64 is trainable if no virtual memory is used. The secondary y-axis plots the number of iterations needed for each batch size, in thousand. The same number of iterations were needed regardless of the memory management policy used.

As can be observed, the end-to-end training time grows with the batch size under both AutoVM and Nvidia virtual memory. Although training using larger batch sizes generally requires fewer iterations, the longer run times of each iteration (under larger batch sizes) increases faster and causes the longer training times. Increasing the batch size from 64 to 256, triples the amount of memory required in training from $N_m = 0.68$ to 2.04. But the training time under AutoVM increases by only 81%, but triples under Nvidia virtual memory.

Although training with batch size 64 was the fastest in our experiments, AutoVM allows machine learning engineers to explore using larger batch sizes without much performance penalty. In practice, using larger batch sizes has advantages as pointed out by Smith et al. [20]. We only tested training under a very specific setting that used a pre-trained model and a restricted dataset, our results are only meant to compare the performance impact of different memory management policies.

Figure 6.6 shows the time required to attain a certain accuracy. The almost-linear curves in the logarithmic scale plot imply that the run times are exponential in linear scale. Further, the time required to acquire one percent more training accuracy is also exponential. The trends of the different run time curves, and they are similar in that they are steeper at both ends.

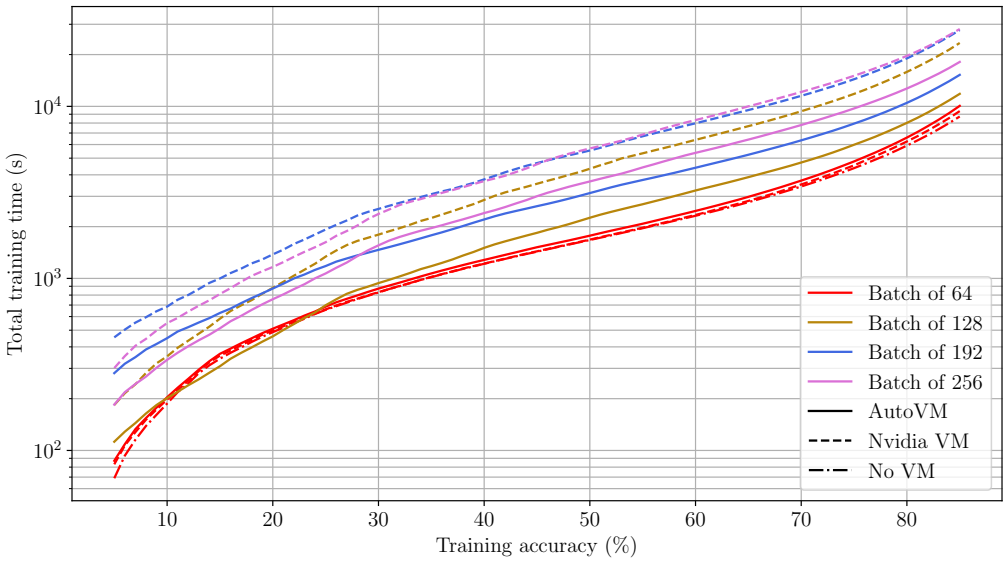

Figure 6.6: The time required to train VGG-19 to a certain accuracy in logarithmic scale. The solid lines show the results from using AutoVM; the dashed line show the results of using Nvidia virtual memory; the results using the same batch size are shown in the same colour.

		Speedup at N_m				
	Max speed up/N_m	0.5	1.0	1.5	2.0	2.5
AlexNet	83.4% @ 1.06	-8.73%	83.4%	48.7%	6.27%	
VGG-19	153.4% @ 1.03	-3.24%	153.4%	78.2%	41.4%	24.6%
ResNet-152	79.0% @ 1.15	-19.0%	22.8%	68.1%	20.0%	16.4%

Table 6.2: The speedup achieved by AutoVM, shown in percentages, at different levels of memory requirements.

6.3.5 Summary of results

Table 6.2 summarizes the speedups attained by AutoVM over Nvidia virtual memory in the three tested neural networks, at different batch sizes that lead to N_m of 0.5, 1.0, 1.5, 2.0, and 2.5. In our experiments, using AutoVM is slower compared to Nvidia virtual memory, before the maximum memory allocation reaches $N_m = 1.0$; the speedup quickly increases after $N_m = 1.0$ and decreased at higher memory utilization after $N_m > 2.0$ but remains positive.

6.4 Discussion

There are mainly two factors that contribute to the speed up with AutoVM:

1. Pre-accessing offers better performance compared to relying on demand-paging, according to our experiments with Nvidia's virtual memory system (refer to Chapter 4). Pre-accessing, however, does add extra overhead, which is more significant at lower batch sizes where the batch training time is only hundreds of milliseconds. The overhead is mostly offset by the performance advantages it brings about at higher batch sizes.
2. AutoVM's memory management policy can make space in global memory by transferring tensors that will not be used in the near future to host memory.

At lower memory utilization $N_m < 1$, there is no need to offload or prefetch tensors because the tensors generated during the training process are able to fit in physical global memory. The run time will be negatively impacted if any transfer latency is not hidden under computations. The performance degradation at lower memory utilization results from the extra overhead introduced by using pre-accessing, for the reasons explained in §6.3.1.

In the region $1.0 \leq N_m \leq 2.0$, pre-accessing and automatic data transfers for offloading and prefetching both contribute to the speedup. In the experiment with AlexNet using batch size 2,048 ($N_m = 1.04$); using pre-accessing sped up the application 53.4%. We were able to accelerate the application by an additional 26.0% after enabling automatic data transfer. Using pre-accessing did introduce extra overhead, but at this level of memory utilization, pre-accessing a tensor in whole before its computation starts is much faster than paging it in on a per-page basis during the computation.

At high memory utilization, the performance benefit from pre-accessing become less significant compared to the time required to handle heavy paging. The observed performance advantage is primarily a result of AutoVM offloading tensors during inference to make space for subsequent computations.

6.5 Future improvements

The experiments confirmed that using AutoVM can lead to performance improvements in training CNNs. Ideally, AutoVM should run without introducing any overhead by hiding all added tensor transfers concurrent other computations. However, AutoVM's potential is limited mainly by two factors:

1. AutoVM's policies are not well optimized because of the lack of operations' exact run times.

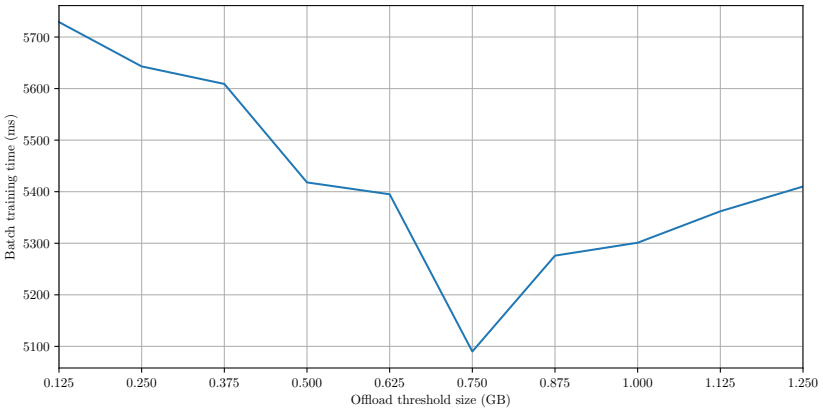

Figure 6.7: The batch training times under different offload size threshold parameter. We tuned the parameter with AlexNet and batch size 3,072 ($N_m \approx 1.4$).

2. Nvidia's virtual memory subsystem cannot hide the tensor transfer latencies under computations completely.

The following subsections identify AutoVM's possible improvements.

6.5.1 Further optimizing AutoVM

AutoVM's policies would be more robust if the exact run times of the operations are known. A possible solution is to profile the application first and use the results to guide AutoVM towards better memory management decisions. However, we were not able to implement this strategy, in a way that is easy to integrate into existing machine learning applications under the current TensorFlow programming model. One major limitation of TensorFlow is its use of a static computation graph, which means the computation graph cannot change during execution. If operations could be added dynamically at runtime, we could use the profiling results gathered by running the application once without enabling AutoVM and then use the collected information to make memory management decisions in later iterations. We thus argue that we could further improve AutoVM's performance by using a framework that supports adding operations dynamically (like Torch) during execution.

Furthermore, AutoVM relies mainly on one empirically determined parameter: the size threshold that determines whether a tensor is offloaded during inference. If the threshold is set too small, AutoVM offloads and prefetches add extra overheads and negatively affects training time. On the other hand, if this parameter is set too high, AutoVM would prevent many tensors from being offloaded, defeating the purpose of using AutoVM. In our experiments, this parameter was tuned while in the moderate

memory usage region, at $N_m \approx 1.4$, while training AlexNet. Figure 6.7 shows the batch training time at different values of the threshold parameter. The difference between the longest and shortest run time is around 11.8%.

Although the parameter was tuned to be optimal for a specific model and batch size, there is no guarantee that the same value will be optimal in other model/batch size combinations. We expect AutoVM would offer better performance if the parameter is automatically tuned for a specific network structure. The tuning can be achieved by exhaustively searching the parameter value space. We did not include this feature in AutoVM because it cannot be implemented under our current integration interface.

Chapter 7

Related work

Previous works related to our work described in this dissertation studied methods to enable more complex neural networks to train on memory-limited GPUs by reducing the required GPU memory footprint. None of these methods use GPU virtual memory.

vDNN [16] proposed to virtualize the tensors stored in global memory, by transparently transferring tensors produced in inference to host memory and prefetching those tensors back to global memory prior to being accessed during training. vDNN uses a rather naïve heuristic to decide which tensors to offload: either all output tensors, or only the ones produced by convolution operations. Another simple heuristic is used to time the prefetch: for each layer l, vDNN finds the next closet layer m, whose operands need prefetching; and initiates the prefetch for m's operands when starting l's computation. Without using hardware virtual memory support, vDNN explicitly uses the memory copy command `cudaMemcpyAsync()` to asynchronously copy data between CPU and GPU. However, it claimed that the page migration bandwidth when using GPU virtual memory is 80 to 200MB/s. But according to our experiments, the bandwidth can usually be over 8GB/s, which is over 70% of the bandwidth achievable by using explicit `cudaMemcpyAsync()`. vDNN's performance results were normalized to a baseline where no memory optimization was applied. In training VGG-16 with batch size 256, vDNN was 18% slower than their *oracular* baseline.[1] In our experiments at the same batch size with VGG-19, a slightly larger network than VGG-16, AutoVM is able to train the network 53% faster than using Nvidia virtual memory. We could not compare our results directly without vDNN's actual run time figures.

[1] Oracular, because they could not train the network on any single-GPU setup as of their writing. The baseline is derived from running each required computation separately.

Chen et al. [4] proposed a method that discards a subset of data generated in training and recomputes them before they are needed during back-propagation. In extreme cases, the GPU memory footprint could be reduced to $O(\log n)$ in a n-layered network. For example, the GPU memory footprint of training ResNet-1000 was reduced from 48GB to 7GB in their experiments. However, the paper did not include experimental results that show the impact their solution poses on runtime.

SuperNeurons [23] by Wang et al. combined the concepts of the previous two works, which selectively offloads/prefetches tensors and recomputes some others. A cost model decides whether a tensor should be offloaded to host memory, or discarded for recompute. Wang et al. also proposed two other methods to optimize global memory usage. First, *liveness analysis* analyzes tensor dependencies between computations and frees tensors that are no longer referenced. Second, *Unified Tensor Pool* manages global memory internally to reduce the overhead in handling allocation and free requests. However, they failed to realize that both methods are applied by machine learning frameworks like TensorFlow by default. Layup [10] extended SuperNeurons that it characterizes the performance implications of performing offload/prefetch and recompute for each type of layer; and then decides whether the output of a specific type of layer should be optimized with the offload/prefetch approach or the recompute approach. Zhang et al. [29] proposed two methods to optimize memory usage in GPU-based training: a memory pool that alleviates memory external fragmentation; and a memory swapper to transfer data that will not be used in the near future between global and host memory. The memory swapper uses a scheduler that implements a priority scoring system to decide which data to transfer and uses Bayesian optimization to automatically tune the scoring system. However, the work did not disclose the exact algorithms used in the scheduler and the memory swapper. Their work introduced nearly no overhead while SuperNeurons had over 40% added overhead, in training VGG-19 at 20% memory footprint reduction.

Le et al. [14] aimed at transferring the data with long reuse distances between global and host memory. The distance between any two operations is defined as the difference between their corresponding indices that are found by topological sorting the computation graph. They also integrated their design in TensorFlow, using a very different mechanism for data transfer than ours. However, they did not use virtual memory in the work. The results show that the work increased the ResNet-50 batch size trainable on memory limited GPUs; but they did not discuss the added performance penalty in depth.

Other researchers have tried to minimize the memory usage by reducing neural network sizes by removing neural connections that have near-zero weights [9], or by using lower precision data formats [11].

These works primarily consider saving memory usage by minimizing the weight sizes that only account for a small portion of memory usage in training CNNs.

Chapter 8

Conclusion

The amount of memory resources available on a GPU limits the complexity of CNNs the GPU can train. Although using GPU virtual memory allows training more complex neural networks with limited physical global memory, such trainings are frequently accompanied by heavy performance penalty because GPU's default memory management policy is unaware of CNNs' workloads and is often unable to make educated memory management decisions.

This research project aims to develop a GPU virtual memory management policy that accelerates CNN training on memory-limited GPUs. We have developed AutoVM that actively manages GPU virtual memory, in a way that is almost transparent to the machine learning applications.

AutoVM achieves its optimizations by transferring out tensors that are recently produced but will not be immediately accessed, to host memory to make space for subsequent computations (offload); the offloaded tensors are transferred back to global memory before they are accessed again (prefetch). AutoVM uses a heuristic to control the timings of offloads and prefetches and to minimize the overhead introduced by the added memory transfers. Furthermore, during reverse-engineering Nvidia's virtual memory system we have found a method called *pre-accessing* to accelerate computations, by forcing all operands of a computation to be physically resident in global memory. At runtime, AutoVM interfaces with the Nvidia driver to issue the corresponding memory transfer requests. We have implemented AutoVM in TensorFlow so existing machine learning applications can benefit from AutoVM's memory optimizations.

AutoVM has been tested in TensorFlow using three iconic CNN structures: AlexNet, VGG-19 and ResNet-152. We have observed non-trivial performance improvements of using AutoVM v.s. using

Nvidia virtual memory, in both benchmarks[1] and full-training experiments. We have also confirmed that using AutoVM improves end-to-end training time and does not interfere with the training process. In particular, VGG-19 training with batch size 192 is 1.98 times faster with AutoVM than with Nvidia virtual memory.

Although AutoVM achieved speedups over Nvidia virtual memory, its performance is limited by the following factors, and should be improved accordingly in the future.

1. TensorFlow's use of static computation graph prevents AutoVM from acquiring precise runtimes and scheduling tensor transfers accordingly. We expect AutoVM to further accelerate training by taking into account the operations' exact run times.
2. AutoVM uses tunable parameters, whose values need to be exhaustive searched specifically for every combination of CNN structure and batch size. We did not include this search function in our current AutoVM implementation.

We claim the following contributions in this dissertation.

1. We have reverse engineered Nvidia's GPU virtual memory system to reveal some of its performance characteristics,
2. We have designed an active GPU virtual memory management policy — AutoVM, to accelerate CNN training on memory-limited GPUs, by analyzing the computation graphs of CNNs and scheduling tensor transfers and
3. We have integrated AutoVM in TensorFlow, a widely used, industrial standard framework, and verified AutoVM is able to deliver non-trivial performance improvement comparing to using Nvidia's default memory management policy.

[1] We only train a CNN ten iterations in benchmark experiments.

Bibliography

[1] VGG19 and VGG16 on Tensorflow. https://github.com/machrisaa/tensorflow-vgg, 2016.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), USENIX, pp. 265–283.

[3] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems. In *Proceedings of Advances in Neural Information Processing Systems 29* (2015), NIPS.

[4] Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[5] Collobert, R., Bengio, S., and Mariéthoz, J. Torch: a modular machine learning software library. Tech. rep., Idiap, 2002.

[6] Google. Protocol buffers. https://developers.google.com/protocol-buffers/.

[7] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition* (2016), IEEE, pp. 770–778.

[8] Hornik, K., Stinchcombe, M., and White, H. Multilayer feedforward networks are universal approximators. *Neural Networks 2*, 5 (1989), 359–366.

[9] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[10] Jiang, W., Ma, Y., Liu, B., Liu, H., Zhou, B. B., Zhu, J., Wu, S., and Jin, H. Layup: layer-adaptive and multi-type intermediate-oriented memory optimization for GPU-based CNNs. *ACM Transactions on Architecture and Code Optimization (TACO) 16*, 4 (2019), 1–23.

[11] Judd, P., Albericio, J., Hetherington, T., Aamodt, T. M., Jerger, N. E., and Moshovos, A. Proteus: exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), ACM, pp. 18–23.

[12] Kingma, D. P., and Ba, J. Adam: a method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations* (2015), ICSA.

[13] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Proceedings of Advances in Neural Information Processing Systems 25* (2012), NIPS, pp. 1097–1105.

[14] Le, T. D., Imai, H., Negishi, Y., and Kawachiya, K. Automatic GPU memory management for large neural models in TensorFlow. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management* (2019), pp. 1–13.

[15] Nvidia. Nvidia Turing GPU architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018.

[16] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W. vDNN: virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture* (2016), IEEE Press, p. 18.

[17] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision 115*, 3 (2015), 211–252.

[18] Sakharnykh, N. Maximizing unified memory performance in CUDA. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/, 2017.

[19] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[20] Smith, S. L., Kindermans, P.-J., Ying, C., and Le, Q. V. Don't decay the learning rate, increase the batch size. In *Proceedings of the 6th International Conference on Learning Representations* (2018), ICSA.

[21] Sun, P., Feng, W., Han, R., Yan, S., and Wen, Y. Optimizing network performance for distributed DNN training on GPU clusters: ImageNet/AlexNet training in 1.5 minutes. *arXiv preprint arXiv:1902.06855* (2019).

[22] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. Inception-v4, Inception-ResNet and the impact of residual connections on learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence* (2017), AAAI, pp. 4278–4284.

[23] Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T. SuperNeurons: dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2018), ACM, pp. 41–53.

[24] Winograd, S. On multiplication of polynomials modulo a polynomial. *SIAM Journal on Computing 9*, 2 (1980), 225–229.

[25] Yamashita, R., Nishio, M., Do, R. K. G., and Togashi, K. Convolutional neural networks: an overview and application in radiology. *Insights Into Imaging* (2018), 611–629.

[26] Yosinski, J., Clune, J., Nguyen, A., Fuchs, T., and Lipson, H. Understanding neural networks through deep visualization. In *Proceedings of International Conference on Machine Learning, Deep Learning Workshop 15* (2015).

[27] Yu, X.-H., and Chen, G.-A. Efficient backpropagation learning using optimal learning rate and momentum. *Neural Networks 10*, 3 (1997), 517–527.

[28] Zeiler, M. D. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* (2012).

[29] Zhang, J., Yeung, S. H., Shu, Y., He, B., and Wang, W. Efficient memory management for GPU-based deep learning systems. *arXiv preprint arXiv:1903.06631* (2019).