Reducing data path from storage to GPUs for Deep Learning

Filippos Petros Lanaras
Abstract

Reducing data path from storage to GPUs for Deep Learning

Filippos Petros Lanaras

Deep learning is an emerging workload in the field of HPC. This powerful method of resolution is able to tackle problems which have been out of reach for traditional algorithmic approaches. However, before being able to solve an instance of the problem, Deep Learning has to go through a learning phase involving a huge volume of data. From the computational standpoint the profile of Deep Learning applications is specific enough to be run almost exclusively on dedicated architectures such as GPUs. Most of the research in the field of deep learning has been focused in the numerical side and architectural aspects of the GPUs. Implicitly, the data access was taken as granted. However, the ever increasing amount of data to ingest for the learning phase has shifted the bottleneck from the compute to the storage part.

The contribution of this thesis is an analysis of existing Deep Learning solutions, detailed performance measurements, identifying the bottlenecks in the system and a proposition of a new scheme to shorten the data path thus accelerating the Deep Learning process as a whole.
Acknowledgements

I would like to thank my supervisors Andra Hugo and Jean-Thomas Acquaviva from DDN Storage, for the extraordinary support and insights, as well as my reviewer Alexandra Jimborean for the academic support and insights and Justin Pearson for the administration parts of the thesis. I would like to thank them for the excellent communication we had, and the understanding they showed during this master thesis work.

I would like to thank ATOS for providing all the computational resources, which were necessary for this thesis.

I would like both KaiGai Kohei (HeteroDB, Inc.) and NVIDIA Corporation for allowing me to use their images in this master thesis.

Lastly, I would like to thank my family and friends that supported me all along.
Contents

Abstract I

Acknowledgements III

1 Introduction 1

2 Background 2

2.1 Deep Learning ................................................. 2
2.2 Caffe .......................................................... 3
2.3 Dataset ......................................................... 3
2.4 Benchmarks .................................................... 4
2.5 Data format: LMDB ........................................... 4
2.6 NVIDIA GPU .................................................. 5
2.6.1 Tesla P100 and Tesla V100 hardware design ............... 5
2.6.2 NVLink ....................................................... 7
2.6.3 Unified memory on Tesla P100 and Tesla V100 ............. 9
2.7 GPUDirect ...................................................... 10
2.8 Access to the storage system through the file system .......... 13

3 Problem: how to have direct storage - GPUs communication? 14

4 Analysis 17

4.1 Target points of analysis and the corresponding tools .......... 18
4.2 Caffe Parameters used for the analysis ........................ 19
4.3 Hardware platform evaluation ................................... 19
4.4 Evaluation ...................................................... 19
4.5 Conclusions of the analysis ..................................... 25

5 Solution to improve the GPU access to the storage system 26

5.1 Step one: Delegating data loading to DTT (Data Tracking Tool) .................................................. 26
5.1.1 Limitations .................................................. 28
5.2 Step two: GPUDirect RDMA model ............................. 32
5.2.1 Model current state of Caffe ............................... 33
5.2.2 Model worst case nfs (wc - nfs) ........................... 34
5.2.3 Model worst case - lustre (wc - lustre) ...................... 34
5.2.4 Model Best Case (BC) ...................................... 35
5.2.5 Model 1: Removing LMDB (No LMDB) ....................... 35
5.2.6 Model 2: Moving the preprocessing to the GPU without LMDB (preproc GPU) .......................... 36
5.2.7 Model 3a PCIe: Moving the preprocessing to the GPU with direct data path (DD PCI) .......................... 36
5.2.8 Model 3b NVLink: Moving the preprocessing to the GPU with direct data path (DD NVLink) .................... 37
5.2.9 Discussion ................................................... 37

6 Related work 38

6.1 Tensor Comprehensions ......................................... 38
6.2 PG-Strom ....................................................... 38
6.3 GPUsfs ......................................................... 40
1 Introduction

Machine learning has progressed dramatically over the last few years, a lot of work has been done to build applications able to train systems specifically with supervised learning models where data are annotated with expected input/output behaviour. Deep Learning has been developed as a branch of Machine Learning through applications like image recognition, speech recognition, self driving cars, etc. Such applications require more data to yield better results in terms of accuracy and precision. However, processing a large amount of data takes a significant amount of time. It is then critical to improve this training phase in order to obtain faster results. Deep Learning applications are built upon a neural network mechanism, describing the computations on the input as a tree-like structure able to expose a significant amount of parallelism. Therefore, GPUs are commonly used for such a task as they are very efficient in doing many computations simultaneously.

The large computation capabilities provided by the GPUs allowed scientists today to improve their machine learning applications by providing more input data. However, all this input data needs to be accessed and stored without introducing a performance bottleneck. Until now typical data sets were small enough to be loaded in RAM, seldom exceeding by a small factor. An architecture based on the assumption of having in-memory data set is no longer feasible with PetaByte scale data sets.

The community has invested a significant amount of work to build faster GPUs and design larger storage systems. However, this is not enough, in an architecture it is critical to have good articulation between the components and the communication speed is critical. Currently, the data flow in the training phase requires data to first be preprocessed by the host CPU and then to be shipped to the GPU to do the computation. Thus, the training halts while waiting for data, leaving the GPU idle for a long time. The system has a hard time to deliver the processed data at a fast enough pace to keep the GPU busy. This imbalance in the system leads to a loss of efficiency, idle time for the GPU and more importantly longer time to result. This reveals the fact that the CPU is not adapted to do such a task and should be removed from the data flow as hardware nowadays provides the possibility to enable the communication between the GPUs and the storage systems directly.

This thesis tackles this problem and provides a software solution to reduce the attributes of the CPU in such applications, and allow it to just have a host role, that is dispatching computation to the GPU. The required data transfer is thus done directly from the storage system to GPU, providing this later with a data tracking system able to coordinate the data available on the device with new requests for data.

The main contributions of this thesis are the following:

• Analyse and understand the data flow of one of the most used Deep Learning frameworks Caffe,
• Implement a tracking system of available data on the GPU in order to coordinate with future data requests from the storage to the GPU and
• Model the expected performance of a system able to use the hardware capabilities connecting the GPU and the storage system directly

This document is thus divided in 7 sections: after Section 1 introducing the topic of the thesis, Section 2 describes a deep understanding of the background of the topic related to Deep Learning frameworks, GPU hardware design and storage system capabilities, Section 3 describes the problem in detail together with the reasons for which this thesis’ topic has not been tackled before, Section 4 presents the analysis done on the performance and work flow of the Caffe
framework, Section 5 describes the implementation and the model provided as a solution to the problem, Section 6 will present the related work and finally Section 7 will conclude and propose future work.

2 Background

Deep Learning frameworks have massively improved recently, both universities and industry contribute to different frameworks providing different software layers. While this leads to a diversity of frameworks, some components or third party libraries remain shared across different frameworks. As an example LMDB is the main file interface for the two major Deep learning framework, TensorFlow and Caffe. Different training datasets and benchmarks, like AlexNet, are widely used as reference to compare the accuracy of the learning and the performance of different frameworks. GPUs are the most commonly used processor to run these kind of tools due to their high computational capabilities. High computational capabilities means large amount of data have to be processed which need in turn to be stored and accessed efficiently. To address this matter, storage systems have done significant progress in recent years to allow faster access to data and larger capacity. This section will describe these points in detail, starting first with the Deep Learning frameworks, GPUs and Storage Systems.

2.1 Deep Learning

Many people are using Machine Learning and more specifically Deep Learning nowadays. To distinguish between the two: [1] Machine Learning consist of Artificial Intelligence solutions able to extract patterns from raw data. Deep Learning is the approach where the knowledge is represented as a hierarchy of concepts, building more complex concepts from simpler ones. Creating a graph showing these relations would come with many layers built on the previous ones. Such a graph would be deep, hence the name Deep Learning. Artificial Neural Networks (ANNs) also know as Neural Networks (NNs) is a class of Machine Learning algorithms that are inspired by the biological brain. Some types of NN are Deep Neural Networks (DNNs), Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). NNs are created in such a way, containing layers that learn more complex concepts from smaller ones and can be represented as graphs with many layers so they fall in to Deep Learning. There are many different domains and applications for Deep Learning, some of these include image classification, self driving cars, speech recognition and others.

The representation of Deep Learning models falls in the framework used, different frameworks define the models differently. These models have a defined size input which the data need to have. They are first trained on a dataset knowing its output and then they are used to guess the output for some unknown data. This entails the training phase of the model, where the network is trained with labelled data and gets updated many times. Taking a batch of data and using it for training makes an iteration. After a specified amount of iterations the network will calculate its accuracy, trying to predict the output of a batch and then comparing to the labelled one. For both training and testing phases of the network, before feeding the data into the network, this later needs to be transformed to the model’s definition. For cases when the model takes images as a input, the images should be transformed (cropped, resized or have their mean value subtracted) to the dimensions of the model. These transformations usually happen once before the training of the model begins. In some cases there can be other transformations as well to improve the learning of the model, i.e. random mirroring, in the training phase.

The metric for the training phase is calculated by the throughput of the application, how many images it consumes per second or how fast it performs an iteration. On the other hand
the metric for testing or inference is calculated by the latency, how fast the network will produce the result for some input.

The operations required by Deep Learning are independent and can be parallelised. For this reason GPUs are a perfect match. Training can occur on a single or multi GPU system. In case of a multi GPU system, training on Caffe\(^1\) and Caffe\(^2\) for instance is cooperative. With cooperative training, each GPU will get a copy of the networks with its weights, get a batch, compute the gradients, exchange them, compute the updated weights and update all models. TensorFlow\(^3\) on the other hand lets the user define how the training should happen on a multi GPU system.

2.2 Caffe

There are many different machine learning frameworks that are currently available. Some of these frameworks include Caffe, Caffe\(^2\) and TensorFlow and each one of them works differently. Caffe\(^2\) is being developed by Berkeley Artificial Intelligence Research (BAIR)\(^3\) under an open source licence. Caffe is build with modularity in mind, allowing the addition of custom functionality. This has caused many different forks\(^4\) on GitHub, which try to extend its functionality. Some forks worth mentioning are Intel’s fork of Caffe\(^5\) and NVCaffe\(^6\). Intel’s fork of Caffe aims to improve the performance when running on specific CPU architectures\(^7\) and it has been incorporated to the main Caffe repository as a different branch. Another variant of Caffe is NVCaffe which is targeted for the NVIDIA products and adds various features. Some of theses additions are support for the NVIDIA technologies.

Models in Caffe are separated from the implementation of the framework. Each model in Caffe is defined as a text file in the Protocol Buffers\(^5\) (protobuf) format. This text file defines the architecture of the model and with a separate file the training parameters for the model are defined. These models are being composed using a set of layers that are implemented in Caffe, including community implementations. Layers can have both CPU and GPU implementations, where the GPU implementation is optional. Caffe supports both CPUs and GPUs and switching between each other is done by the change of a parameter. When executing with the use of GPUs and a layer does not have a GPU implementation, the CPU implementation will be used as fall back. Caffe also supports the use of multiple GPUs for training\(^9\).

Data in Caffe are stored in 4D arrays called blobs. Blobs are also used for passing data from one layer to another and hide the data migration between the CPU and GPU.

2.3 Dataset

The collection of input data to train a neural network is named a dataset. Datasets usually have a specific topic, i.e. images of flowers. There are a lot of different datasets online with images from different categories. Many different organisations have their own datasets.

In the experiment section of this thesis there was a need for a dataset large enough that will not fit entirely in memory. The computer in use for the experiment has 128 GB of RAM and for this reason the dataset from ImageNet Large Scale Visual Recognition Competition (ILSVRC) 2012\(^10\) was used. ILSVRC 2012 contains over 1.4 million images in 1000 different categories.

---

\(^1\)http://caffe.berkeleyvision.org/

\(^2\)https://caffe2.ai/

\(^3\)https://www.tensorflow.org/

\(^4\)Creating a copy of the source code under a different user giving full access to the copy of the repository [4, p. 136-137].

\(^5\)Protocol Buffers is a platform-neutral, language-neutral mechanism to serialise structured data [8].
classes. More specifically it contains around 1.2 million training, 50000 validation and 100000 test images. The images from this dataset come in various sizes and shapes.

2.4 Benchmarks

Multiple parameters can be adjusted at the set-up of the training phase. The most important ones being: the number of iterations, the batch size, test iterations and test interval.

- The maximum number of iterations is the parameter that defines when to stop the training.
- Batch size defines the amount of images to be run in an iteration. The batch size is usually a power of 2 for better performance, typical values are in the range 32-256 while using small values for deep models [1].
- Test iterations describe the amount of iterations to be run each time the network would require to get its accuracy. The batch size of training and testing does not have to be the same value.
- The test interval depicts how often testing the network should happen.

Apart of these parameters, the model used is important too. A list of such models can be found at Caffe’s website 6.

2.5 Data format: LMDB

In order to access the dataset fast enough Caffe and other Deep Learning frameworks use a specific data format or layout to store their dataset. There are many different ways to store the data which have different performance impact as shown in a previous study [11]. One data format is the Lightning Memory-mapped Database (LMDB) a key-value database. LMDB uses mmap to map the file in the Virtual Memory space. Through this technique files are loaded indirectly into the memory rather than explicitly reading the file.

The use of indirect memory access helps in cases of complex data structures, which LMDB uses. The underlining representation is a B+ tree and for which LMDB uses the page size of the Operating System (OS) page size. This makes LMDB being architecture dependent and does not make the same database usable across different platforms. This data format has been adopted in Caffe for batch processing than handling individual files, as handling individual files requires a lot of Input/Output (I/O) operations and is considered slower. However, LMDB has not been initially designed for the Deep Learning frameworks but for LDAP, an application protocol for accessing and maintaining distributed directory information services over an Internet Protocol network. Due to the easy utilisation it was reused for deep learning applications inheriting thus the complexity of the tree representation.

LMDB provides a solution to keep a common data structure for different applications while trying to remove the bottleneck of accessing the storage system and improving scalability [12], [13]. However, it has been shown that LMDB is not a good solution to be used in High Performance Computing (HPC) systems. Explicitly handling the access to the storage system is expected to improve performance at the expense of the usability.

6http://caffe.berkeleyvision.org/model_zoo.html
2.6 NVIDIA GPU

GPUs are widely used for Deep Learning applications in order to address the need for computational power. The architecture of GPUs intrinsically suits convolution techniques and other low level kernels at the core of Deep Learning. With the help of GPUs, complex Deep Learning tasks can be processed in an acceptable time frame. The computational acceleration provided by GPUs has been a technological enabler and is at the source of the popularity of Deep Learning applications today. The term itself of GPUs nowadays embraced the previous notion of General Purpose Graphics Processing Units (GPGPUs). GPGPUs were introduced to underscore that GPUs are not only used with graphics Application Programming Interfaces (APIs). It is now well accepted that GPU stands for GPGPU. The key characteristic of GPUs is a high level of parallelism. GPUs can execute parallel applications on a massive amount of cores they provide, in the range of 10s of thousands of cores on a single GPU. As a trade-off for this large number of cores, the frequency of GPU cores is lower than CPU’s. Therefore GPUs aim at high throughput not latency driven computations. Thus, GPUs are meant to complement the CPUs and not to replace them. This is a critical point since GPUs perform poorly on sequential tasks where CPUs thrive.

NVIDIA has invested in the deep learning field and has improved their GPUs by adding tensor cores and NVLink to their architecture (Section 2.6.2) [14]. To ease the programmability of their accelerator, NVIDIA provides a programming language named Compute Unified Device Architecture (CUDA). NVIDIA has developed many different software and hardware technologies such as GPUDirect (Section 2.7) to reduce the CPUs involvement, Unified Memory (Section 2.6.3) has been a step forward to improve the easy utilisation of the device with its latest Tesla V100 GPU (Section 2.6.1). The NVLink interconnect has also been introduced in order to allow an efficient use of multiple GPUs in a system (Section 2.6.2).

2.6.1 Tesla P100 and Tesla V100 hardware design

Each GPU generation brings in new capabilities to hardware and software. They get faster, contain more cores, memory and increase the total Floating Point Operations Per Second (FLOPS). The Tesla V100 [14] reaches 125 TeraFLOPS (TFLOPS).

Tesla V100 comes with a different memory model, merging the L1 cache with the shared memory comparing to Tesla P100 [15]. This merger provides a total of 128 KB of combined memory that each application can utilise as it suits it better, for instance programs that do not used shared memory can use all memory area for cache. The combination of the two memory regions provides L1 cache benefits from shared memory, high bandwidth, low latency accesses but requires explicit memory management by the programmer.

Tesla V100 supports the second generation of NVLink that allows up to 6 links and a higher bandwidth of 25 GB/sec of each sub-link between GPU devices. NVLink in more details in Section 2.6.2.

The Streaming Multiprocessor (SM) structure in the Tesla V100 can be seen in Fig. 1. Each SM is partitioned on 4 blocks, each block contains 8 Floating Point (FP)64 cores, 16 Integer (INT)32 cores, 16 FP32 cores and 2 Tensor Cores. Tensor cores are added in Tesla V100, allowing a $4 \times 4$ matrix operation per clock and each SM contains two tensor cores. This operation is $D = A \times B + C$, where all $A$, $B$, $C$ and $D$ are $4 \times 4$ matrices.

A new feature to the Unified Memory (Section 2.6.3) is the Access Counters, able to track the frequency of memory access on other processors. These access counters seek to migrate the memory pages to the processor that most frequently access them.

A major enhancement in the Tesla P100 and Tesla V100 is the new memory technology. There is a shift from traditional Graphics Double Data Rate 5 (GDDR5) to the new High
Figure 1: Tesla V100's SM architecture. Source: https://devblogs.nvidia.com/parallelforall/wp-content/uploads/2017/05/image3.png
Bandwidth Memory 2 (HBM2) memory type enabling higher memory bandwidth. The peak memory bandwidth of the Tesla V100 is 900 GB/sec.

HBM2 also adds support for native Error Correction Code (ECC) where a separate area of memory is been utilised for ECC bits, comparing to inline ECC where it uses a part of the main memory for this reason. In the Tesla K40, which utilises inline ECC, 6.25% of the GPU’s memory was reserved for ECC bits [14], [15].

Hardware support of page faulting is introduced: if a page is not located in the page tables of the GPU it will be requested. The page does not have to be pinned memory. Address Translation Services (ATS) allow direct access to the CPU’s page tables. With ATS even pages that have not been registered with the Unified Memory driver can be fetched, i.e. addresses allocated with malloc.

In the Tesla V100 architecture, independent thread scheduling is introduced. In NVIDIA’s GPUs 32 threads, called a warp, co-execute under the same instructions in a Single Instruction Multiple Thread (SIMT) way. Prior to Tesla V100, each warp would have a single program counter and call stack for all threads in the warp. In case of divergent code, each divergent code block should execute until it converges back allowing then the other divergent block to be executed. Execution of the threads within a warp is marked with a mask, showing which threads should be executed. Lets take an example into consideration from Tesla V100’s whitepaper in listing 1 where capital letters are statements. In pre Tesla V100 architecture, first the divergent path, A and B, until reconvergence, would be executed. After reconvergence the other divergent path, X and Y would be executed by the other threads and finally all would execute Z. This interleaving is shown in Fig. 2. This thread scheduling requires lock free algorithms, else it can lead to deadlocks.

In Tesla V100 each thread has its own program counter and call stack allowing finer grain synchronisation. In the same source code example, Tesla V100’s threads are possible to interleave statements from different divergent paths. It is made possible to execute the statements in the following order, A, X, B, Y, Z. This interleaving shown in Fig. 3. This new thread scheduling allows programmers to use a finer grain synchronisation, allowing the implementation of more familiar synchronisation algorithms.

```plaintext
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

Listing 1: Divergent code snippet. Source: NVIDIA Tesla V100 GPU Architecture [14]

### 2.6.2 NVLink

As the use of multi-GPUs systems has been increasing and the CPU to GPU ratio as well. The main bottleneck [15] in multi-GPU systems has become the Peripheral Component Interconnect Express (PCIe) bandwidth. GPUs in a multi-GPU system have to share this bandwidth. PCIe 3rd generation ×16, provides a peak bidirectional transfer rate of 32 GB/s which is not sufficient for the system.

For this reason NVIDIA created a new high speed interconnect technology named NVLink [14]–[16], allowing a better performance scalability, leaving the PCIe bandwidth available to other
system resources. NVLink allows faster GPU to GPU communication and faster access to system memory.

Currently only Tesla P100 and Tesla V100 support this interconnect. NVLink was introduced with Tesla P100 which supports the first generation of NVLink, where the Tesla V100 supports the second generation. The first generation of NVLink supports 4 links per GPU, this number has been increased to 6 for the second generation. The speed has increased between the two generations, increasing connection speed from 20 GB/s to 25 GB/s in a single direction (sub-link) and 40 GB/s to 50 GB/s in bidirectional connection (Link). This allows a total bidirectional bandwidth of 160 GB/s in Tesla P100 and 300 GB/s in Tesla V100.

With NVLink it is possible to directly load or store system’s memory from the GPU and also perform atomic operations on other GPUs, completing on destination.

The second generation of NVLink allows direct access from the CPU to the GPU’s memory, by permitting items available in the GPU’s memory to be stored in the CPU’s cache hierarchy. Atomic operations can be initiated directly from either GPU or CPU and there is a direct access to the CPU’s page table.

NVLink can be used in different topologies, these can be GPU-to-GPU or GPU-to-CPU. There can be multiple connections between two GPUs increasing their bandwidth and performance. An example of such connections can be seen in Fig. 4

In a system where there is a CPU and a GPU connecting all four (1st gen) NVLink links to the CPU achieves a bidirectional bandwidth of 160 GB/s comparing to 32 GB/s peak PCIe 3rd gen.
2.6.3 Unified memory on Tesla P100 and Tesla V100

GPU application developers need to manually handle memory allocations and data transfers. Therefore, historically programming GPUs have required deep understanding of the memory, with explicit data copy and manipulation. To maintain data consistency, programmers have to keep track of the latest updated copy of a given data in the multiple memories of the system. All of these operations lead to the developers waste of time and effort and lead to error prone implementations. For instance, in case of sparse memory accesses, data prior to their transfer to GPU have to be coalesced, which additionally to the implementation difficulties degrade the performance by increasing transfer latency by moving more data than used.

NVIDIA has created the Unified Memory [14], [15], [17] which provides a unified address space for the memory system of the CPUs and GPUs in the system. This is a pool of managed memory that is shared with CPUs and GPUs allowing data accesses with the same pointer. With Unified Memory it is easier to program and port applications to GPUs, making it easier to use complex data structures on the GPUs. It enables programmers to create their prototypes faster, since they do not have to worry about the memory management and transfers and it simplifies debugging.

On Tesla P100 and Tesla V100 49-bit virtual addressing is being utilised for all system and
GPU addresses, which is enough since modern systems use a 48-bit virtual addressing. This allows access to all of the memory space of the CPUs and GPUs.

The hardware support added on Tesla P100 for memory page faulting enables the GPUs to page-fault if a requested page is not resident in the GPU’s memory. When a page fault occurs the SM’s stops until the page is brought in to the GPU memory. Page migration is also possible, moving pages from another system memory unit (i.e. CPU) to the GPU. Page migration can occur on-demand which can improve performance, having frequently used pages migrated on the GPU to increase access speed. Page-faulting allows memory oversubscription, exposing more virtual memory than the physical available. Oversubscription allows the use of datasets larger than the system’s memory making it possible to use large datasets on the GPUs. Although Unified Memory tries to make it easier for the programmer to use the GPU and hide memory transfers, it also allows the programmer to insert memory usage hints to optimise memory usage.

Tesla V100 adds support for access counters which count the access count of pages and can cause a page migration if a page is used more frequently on another device which tries to improve memory access performance. Another thing which is added with Tesla V100 is Address Translating Services (ATS) over NVLink which can directly access the CPU’s page tables. ATS gives full access to the CPU’s memory, even requesting memory areas that are not registered by the Unified Memory driver. This means that memory areas that have been allocated with malloc or new can be used with Unified Memory with no further actions.

Some benchmarks provided by NVIDIA [18] detail the speed-up brought by Unified Memory, hints and oversubscription. Additional results are also available to analyse page migration cost and throughput.

It is crucial to mention that in Kepler’s and Maxwell’s architecture Unified Memory’s support is limited, all of the pages have to be migrated to the GPUs before kernel launch, they do not support hardware page faulting and the size of the Unified Memory’s size is limited by the physical memory.

This feature is however not available yet in Deep Learning frameworks like Caffe (Section 2.2).

2.7 GPUDirect

When the GPU is used in conjunction with other devices CPU overheads are introduced. For example, when a GPU needs to send or retrieve data from another device, i.e. GPU or 3rd party device, the critical data path contains the CPU, where it copies data from one memory area corresponding to one device, i.e. the GPU, to another device, i.e. the network interface. This copy from the CPU is unnecessary and can be removed with some driver changes. Removing this memory copy would reduce the CPU’s utilisation allowing it to be utilised elsewhere and reducing the data access latency.

NVIDIA created the GPUDirect technology family [19], [20] to eliminate the involvement of the CPU. This contains different products:

- GPUDirect,
- GPUDirect Peer-to-peer (P2P) Communication,
- GPUDirect Remote Direct Memory Access (RDMA) and
- GPUDirect async.
These, require additions to the drivers of the 3rd party device which will communicate directly with the GPU. Most of these aim to reduce CPU utilisation which derive from unnecessary memory copies, consequently reducing latency and allowing more efficient resource management.

The first solution has been GPUDirect [21] (see Fig. 5) which enables sharing the pinned memory, between the network interfaces and GPUs. Pinned memory [22, p. 45-46] is a special type of memory that cannot be swapped to disk in contrast with pageable memory that can get swapped to disk. Use of pinned memory is important because it is required in order to achieve memory transfers between RAM and GPU. In the case of memory transfers that includes pageable memory, the content of the memory area has to be copied to a pinned memory to proceed with the transfers.

Using the same pinned memory reduces the amount of copies required while using data from the GPU’s memory on the network interface. This reduces the CPU’s overhead which arrives from the data copy from one pinned memory to another and also reduces the data’s latency over the network.

The work flow without GPUDirect is:

1. copy memory from GPU to pinned memory 1,
2. copy pinned memory 1 to pinned memory 2 (network),
3. network adapter uses pinned memory 2.

With GPUDirect, both devices share the same pinned memory eliminating the extra memory copy. The new work flow is:

1. copy memory from GPU to pinned memory,
2. network devices uses the pinned memory.

This is illustrated in Fig. 5.

Another GPUDirect product is P2P direct access and direct transfers [19], [20]. These allows GPUs in the same node to access and transfer data directly, reducing the system’s involvement in memory accesses and copies between two GPUs. The scope of the direct communication is limited from the topology of the GPUs, they need to share the same PCIe root complex to be able to bypass the CPU. In case they do not share the same PCIe root complex, the memory transfer will fall back to a memory transfer through the CPU.
Figure 6: GPUDirect vs GPUDirect P2P. Source: https://devblogs.nvidia.com/parallelforall/wp-content/uploads/2013/03/GPUDirectP2P.png

An example demonstrating the use of GPUDirect P2P, when a kernel wants to copy data from GPU1 to GPU2, when the GPU1 and GPU2 do not share the same PCIe root complex or the driver does not support GPUDirect P2P, the driver

1. copies data from GPU1 to pinned memory and
2. copies data from pinned memory to GPU2.

With GPUDirect P2P the data transfer would be copied from GPU1 to GPU2, removing the CPU from the data path. This can be seen in Fig. 6.

Another GPUDirect technology is Remote Direct Memory Access (RDMA) [23], [24]. This allows 3rd party devices to directly read and write memory on the GPU, such devices can be network interfaces or storage adapters. The direct data path between these devices occur with features of PCIe. The devices need to share the same PCIe root complex to be able use this and this is only available on NVIDIA’s Tesla and Quadro GPUs. Such devices need to make some changes to their driver to use GPUDirect RDMA. GPUDirect RDMA reduces latency by entirely bypassing the CPU. It eliminates the extra memory copy between the GPU and the pinned memory, since the 3rd party device directly access the GPU’s memory.

Without GPUDirect RDMA, when data located in the GPU need to be transferred over the network,

1. Data are copied to the pinned memory,
2. Network interface copies the data from the pinned memory.

With the use of GPUDirect RDMA, the network interface directly copies data from the GPU’s memory, eliminating the CPU from the data path. An illustration of this can be seen in Fig. 7 with GPUDirect and GPUDirect RDMA.

The newest addition is GPUDirect async [25], [26]. With GPUDirect async it there can be a direct control path with the GPUs and 3rd party devices. This currently has been tested with network interfaces. GPUDirect async does not require GPUDirect RDMA to operate, one complements each other. With GPUDirect async, a different work flow is been introduced, where the operations to the network adapter are requested from the GPU kernels and then the GPU kernels wait for task completion.

In an example with GPUDirect async, where the GPU will do a matrix multiplication and then transmit the results over the network, the control path is:

1. CPU submits the matrix multiplication work to the GPU and waits for synchronisation with the GPU,
2. CPU prepares the network interface,
3. Network interface copies the data from the GPU’s memory and
4. Network interface sends the data.

The new control flow follows:
1. CPU prepares matrix multiplication work and communication tasks for the GPU,
2. GPU completes its computation tasks and triggers the communication over the network interface,
3. Network interface fetches the data and
4. Network interface sends the data.

This makes the GPU responsible for triggering and polling for task completion (the network interface for communication in this example). It moves the poll for completion from the CPU to the GPU, requiring less CPU utilisation. It is worth mentioning that GPUDirect async will not always improve execution time, this depends on the amount of network operations that need to be done, if there are many waiting times it could be more efficient to have the CPU to trigger the communication and asynchronously submit tasks to the GPU.

Adaptation of GPUDirect RDMA is mainly focused on network interfaces. Usage of GPUDirect RDMA with Message Passing Interface (MPI) shows a x4 increase in bandwidth and a latency reduction of 80% [27]. There are a few attempts to implement GPUDirect RDMA on storage adapters but they are not available on drivers created by hardware vendors. Implementing such drivers by the users is hard because it requires the appropriate hardware, NVIDIA Tesla or Quadro and knowledge about DMA transfers.

2.8 Access to the storage system through the file system

To serve large amounts of data, file system are external to the computing unit. Using the network card the computing system can access a storage of much larger capacity than the number of drives it could host physically in the compute server.

Most of the external storage in enterprise is based on NFS (Network File System). NFS is a well known and well accepted technology. NFS is a Distributed File System in the sense that multiple disks can be managed by a single server. By aggregating the capacity of multiple disks (up to racks of disks) a single NFS server can provide access to a very large amount of
files through the connected compute system. However, as a constraint a file is always placed on a single disk. Therefore, even if there are multiple disks in the server, the access speed of a file is determined by the speed of a single disk.

To alleviate this bottleneck and take advantage of the multiple disks, not only for the capacity but also for the bandwidth, the HPC community has developed parallel file systems.

A parallel file system is conceptually similar to a distributed file system, except that a single file is dispatched on multiple disks. Therefore accessing a file is equivalent to reading or writing to several disk in parallel. One of the most successful parallel file system is Lustre. GPFS is an older file system developed by IBM but remains widely used due to its robustness.

To provide an understanding of the order of magnitude of the time required to access a file while using these different file systems, the performance of NFS, two different Lustre systems, and a GPFS system are evaluated. For every system the bandwidth has been evaluated when the access pattern is sequential and random. The measurements have been obtain with fio, used with mmap and 4 KiB block size. These measurements have been conducted on clusters provided by ATOS and can be seen in table 1.

It appears from these measurements that the file systems are very sensitive to the access patterns. While sequential performance is in the range of GB/s, for random access the performance drops to the MB/s range.

### 3 Problem: how to have direct storage - GPUs communication?

Deep Learning applications need a lot of data and are expected to need even more data in the following years. This makes the way storage is used and how data are fetched critical. GPUs are fast data consumers, they are used in deep learning applications because they are excellent for the computations needed in such applications. The current state of the data flow is as following:

1. Data are fetched from the storage system to the RAM and
2. Data are moved from the RAM to the GPU’s memory.

An extra-step is introduced due to the presence of the CPU in the data path. This thesis address this problem of data path length. The core contribution is a proposal for a direct data

---

Table 1: Read measurements on different file systems with mmap as the read engine and block size of 4 KiB measured by fio.

<table>
<thead>
<tr>
<th>System Name</th>
<th>Access Pattern</th>
<th>Measured bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>bw_storage_ram_nfs</td>
<td>Sequential</td>
<td>1445 MB/s</td>
</tr>
<tr>
<td>bw_storage_ram_lustre_ddn</td>
<td>Sequential</td>
<td>1889 MB/s</td>
</tr>
<tr>
<td>bw_storage_ram_lustre</td>
<td>Sequential</td>
<td>1869 MB/s</td>
</tr>
<tr>
<td>bw_storage_ram_gpfs</td>
<td>Sequential</td>
<td>533 MB/s</td>
</tr>
<tr>
<td>bw_storage_ram_nfs</td>
<td>Random</td>
<td>73.2 MB/s</td>
</tr>
<tr>
<td>bw_storageRam_lustre_ddn</td>
<td>Random</td>
<td>31.3 MB/s</td>
</tr>
<tr>
<td>bw_storageRam_lustre</td>
<td>Random</td>
<td>5.341 MB/s</td>
</tr>
<tr>
<td>bw_storageRam_gpfs</td>
<td>Random</td>
<td>0.827 MB/s</td>
</tr>
</tbody>
</table>

---

9https://github.com/axboe/fio
transfer from the storage system to the GPU’s memory, eliminating the CPU from the data path, hence reducing of the data transfer latency. A data tracking tool is also introduced to help the applications that need such data transfers from removing them from the low level implementation details.

**Example:** Lets consider the training of a neural network with the batch size of 256.

The data required to be read from storage to RAM is equal to 39.6 MB and then the data transfer from RAM to GPU is 158.3 MB. Loading this amount of data from storage to RAM takes 39.68 ms and then 249.08 ms to preprocess the data before they are fed into the neural network. The data transfer from RAM to the GPU’s memory takes 13.5 ms and 18.3 ms after that the data are being consumed by the GPU. The GPU is being utilised for approx. 137 ms and then it remains idle for another 140 ms while waiting for more data to be ready.

The overall behaviour can be seen in Fig. 8, where the red part of the graph represents the time the GPU is waiting for data which consists of around 50% of the time. The cyan colour represents the time that the Deep Learning framework is utilising the GPU usefully. More details on this can be found in Section 4.

Reducing this waiting time is crucial to take a bigger advantage of the GPU in use. It is important to locate the cause of this slow down, since it dominates a big part of the training process and making the GPU faster will not make a big difference in the performance. The problem is even more acute for multiple GPUs, since training even with a single Tesla V100, data are not sent fast enough to keep the GPU running without idle time. Hence, removing the CPU from the data path would speed up the whole process. This can be done either by having the computations done on the GPU, which requires a 10% of the time as shown in table 2, or by having all of the preprocessing that is possible to be done offline or both of them.

Improving the GPU utilisation would both make the computations faster and cheaper since less GPU time would be required, since the GPU is considered as the most expensive part of the computation.

![Figure 8: An iteration of batch size of 256 as captured in nvvp. The GPU is waiting for around 50% of the time to get the data and start the iteration.](image)

There are different causes for this problem. The first one, is an implementation issue of the data parser and the underlying data format used. Another cause is that there is some preprocessing happening online and specifically how this preprocessing occurs.

---

10Online in this context means, the preprocessing is happening while the training of the neural network occurs.
The data parser and how data are stored in Caffe have been specifically investigated in the work of this thesis (Section 4). LMDB is being utilised as the underlying data storage format. This is a single file database which creates a smaller I/O overhead but it is based on `mmap()` which has implicit I/O and affects the performance and scaling. Handling many small files produces an I/O overhead which was one of the things that LMDB tried to solve. However, using implicit I/O harms the scaling capabilities for larger datasets due to the way the OS handles page faults. It can increase the waiting time while accessing data from storage when there are many cache misses or random reads. An other drawback, LMDB stores data with the OS’ page size which is much smaller than the page size that the storage medium utilises. Also, because LMDB uses a B+ tree for its underlying representation, it does not fully utilise sequential reads, affecting negatively the I/O throughput. The way the LMDB database is created, the offline preprocessing only contains resize and shuffling and then the database file creation follows.

This leads to the second issue; a part of the preprocessing is happening online. The online preprocessing which is handled by the ”transformer” (the module that does the preprocessing) is sequential. This means that only a single thread is used to perform these operations. This sequential part is slowing down the training process. Making this operation in parallel would significantly reduce the time spent waiting for data. This additional preprocessing stage is slowing down the data traffic towards the GPU. As a result Caffe fails to feed one Tesla V100 fast enough. The operations which are done online include mean value subtraction, random cropping and random mirroring. Mean value subtraction can be done in the offline preprocessing and would severely improve performance. Random cropping and mirroring, if they are required, cannot happen offline but they can be done in the GPU. These two issues are responsible for the red bar in Fig. 8.

There are different ways to address all of these issues. One way is to improve the implementation of Caffe. For instance, LMDBIO [12], [13] tries to improve the speed of the I/O subsystem. NVCaffe improves performance for NVIDIA products and the data parsing. These options remain implementation optimization specific to a given framework and not generic enough.

A more drastic option would be to introduce a new data format in order to replace LMDB. Such a format should use explicit I/O over implicit to allow the implementation to scalable solutions and suffers from the underlying I/O bottleneck.

A third option is to make the data arrive faster to the GPU. This would require to remove the CPU from the data path between storage and GPU, similarly to the GPUDirect RDMA solution for some network adapters which can directly read and write to the GPU’s memory (Section 2.7).

Currently it is not possible to use a storage to GPU data transfer using GPUDirect RDMA. There are no available drivers implementing this functionality at the moment and this requires each storage driver to be implementing this. There is currently some work in progress that tries to do this on Non-Volatile Memory express (NVMe) devices, such as PG-Strom 6.2 and ssd-gpu-dma. Another reason is the shortcoming of GPUDirect RDMA’s availability. This is only available on NVIDIA’s Tesla and Quadro GPUs, which limits the development environments.

Handling transfers from the storage to the GPU would require to restructure and adapt parts of the memory model used in the framework. This also includes data tracking and movement. Moreover, LMDB cannot be moved to the GPU as it would require parsing the underlying data structure.

To our knowledge, the solution proposed in this thesis is new, and has not been addressed by the community. Previously LMDB’s performance were been good enough and datasets were

\[\text{https://github.com/enfiskutensykkkel/ssd-gpu-dma}\]
small enough. With larger datasets and faster GPUs, the performance penalty of using LMDB is increasing and expected to increase even more in the future, which makes it now an important problem to solve. Even stacking more hardware, with increased RAM capacity up to store dataset entirely in the RAM, the performance imbalance between processing capabilities will remains as accesses will still suffer from the drawback of LMDB’s design.

GPUs are constantly improving, they are getting even faster and their memory size and speed improving as well. Let’s take the example of DGX-1 [28], it comes with eight GPUs, Tesla P100 or Tesla V100, and DGX-2 [29] with 16 GPUs, Tesla V100. Having more GPUs in the system requires a fast storage subsystem in order to feed data to the GPUs without leaving the GPUs waiting for the data. In such systems the GPU to GPU communication is limited by the PCIe bandwidth and so NVIDIA created NVLink (Section 2.6.2) to solve this bottleneck and allow fast inter GPU communication.

Until now, the metric for Deep Learning frameworks and models has been the speed of learning, iterations per second which is affected by the throughput of the system, in the training phase. Now the focus has started to include latency, how fast a result will be given for some specific data. This is important in the inference phase (the testing of the trained neural network) as well when the network has been trained and an output of the network is needed, mainly when the network has been deployed. The inference phase is latency dependent it will also gain from a shorter data path.

The solution presented in this thesis includes two stages and the usage of a different data format. The idea is to use GPUDirect RDMA as it has been used with third party network devices that reduces the CPU involvement and the latency of the system. This makes the GPU using data directly from storage without having to wait for data from the CPU, allowing data pre fetching in the GPU memory, increasing the training speed and data latency.

The first step for the suggested solution is to change how the framework handles the data. There is a need for a tool doing this by providing a suitable interface which would be integrated. The tool should allocate the memory on the GPU and orchestrate all memory copies in order to make the data arrive on the GPU.

The second step is to remove the CPU from the data path, moving data directly from the storage to the GPU, requiring driver support for GPUDirect RDMA. Then the tool in the first step should use the driver’s support to move data directly to the GPU.

This requires the use of a different data format than LMDB that can be done with explicit I/O. Another problem is that the data is serialised in the protobuf format that does not have a CUDA implementation. Deserialisation would mean that GPU cycles would be needed in order to be able to use the data. The new data format proposed simply requires the data to be loaded in blocks in the GPU and the GPU is able to use the data directly.

4 Analysis

In order to better understand the deep learning framework and its data flow, it is paramount to detect the bottlenecks and analyse the performance and the data transfer requirements. This section describes an analysis methodology using multiple tracing tools, capable of understanding the data pipeline of the deep learning frameworks and raising points that can be improved.

The main two phases of the analysis include tracing methods to track the data transfers from CPU to the storage system and from CPU to the GPU. Thus, the measurements include the time to read images, parsing the underlying data format (LMDB) and deserialising the content. Reading the images from storage may or not be reflected in the measurements because of the implicit I/O access used (mmap solution). Other measurements include the time to do online
preprocessing of the data (images). Through this analysis it was possible to collect timings from training on the GPU and track the dependencies between the CPU and GPU.

4.1 Target points of analysis and the corresponding tools

This analysis focuses on a few different points. These points include the application’s throughput, the memory footprint GPU, the GPU and CPU utilisation and the I/O and preprocessing time in Caffe.

In order to collect all this analysis information many different tools have been used. For the GPU nvidia-smi, nvprof and nvvp were used, for the CPU ps and time. The lack of automatic tools leaded to integrating multiple instrumentation points in the Caffe framework. Additionally stress-ng was used to clean up the node after each measurement.

- ps\textsuperscript{12} and time\textsuperscript{13} are Linux utilities in charge with displaying information about a selection of the active processes as well information about the resources used. Such information was necessary in order to track the behaviour of Caffe.

- nvidia-smi\textsuperscript{14} (Nvidia System Management Interface) provides information about the state of the GPUs and can also administrate the GPUs in the system. It provides some sub utilities such as pmon and dmon with provide information about processes that run in the GPUs (pmon) and utilisation statistics per device (dmon). Nvidia-smi pmon was used to obtain SM and Memory utilisation measurements. Nvidia-smi was also used for debug reasons, check GPU temperatures and GPU’s memory allocation.

- nvprof\textsuperscript{15} is a command line profiling tool allowing to perform automatic measurements. Nvprof traces a timeline of CUDA-related activities on both CPU and GPU, including kernel execution, memory transfers, memory set and CUDA API calls and events or metrics for CUDA kernels.

- nvvp\textsuperscript{16} (NVidia Visual Profiler) is a tool that visualises information gathered from nvprof. Some of this information include CUDA kernel execution, memory copy and stream information. The kind of information that this includes is the start and end time, duration, launch parameters ordering, bandwidth, etc. Nvvp can be a very useful tool to understand an application’s GPU behaviour.

- Stress-ng\textsuperscript{17} can stress test a computer system in a selectable manner. It was used to stress the node’s memory so its cache buffers could be cleaned up from data previously populated by mmap.

The implicit I/O requests done by LMDB through the mmap made it difficult to trace the actual requests. Multiple tools have been used but they did not lead to obtaining a full I/O trace. They are enumerated: dio-pro, an internal tool of DDN Storage, not publicly available, perf\textsuperscript{18} and SystemTap\textsuperscript{19} were investigated as well but did not provide relevant information for the analysis.

\textsuperscript{12}https://gitlab.com/procps-ng/procps
\textsuperscript{13}https://www.gnu.org/software/time/
\textsuperscript{14}https://developer.nvidia.com/nvidia-system-management-interface
\textsuperscript{15}https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview
\textsuperscript{16}https://developer.nvidia.com/nvidia-visual-profiler
\textsuperscript{17}http://kernel.ubuntu.com/~cking/stress-ng/
\textsuperscript{18}https://github.com/torvalds/linux/tree/master/tools/perf
\textsuperscript{19}https://sourceware.org/systemtap/
In order to overcome the lack of this information the solution was instead to keep track of the memory used by the application (trigger by mmap) to calculate the throughput. This although did not give a correct result especially in the case of lustre and DDN where the memory footprint increased and decreased without any visible cause.

4.2 Caffe Parameters used for the analysis

Caffe requires setting a few software parameters like the training parameters and the model and dataset used. The training parameters for the experiments include the batch size, maximum amount of iterations, test iterations and test interval. The analysis has been done by using different batch sizes (16-1024) and setting the amount of iterations to train the network to 20000. The test iterations were set to 10 and were only run once, by setting the value of test interval to 100000.

The model that these parameters were applied is the blvc_alexnet and the dataset fed into it is the ILSVRC 2012 converted in LMDB format with the images having the dimensions of $227 \times 227$ which is the requirement for Alexnet. The preparation of the dataset happened as the guidelines for Caffe suggest, cropping of images happened offline while the mean subtraction took place online.

4.3 Hardware platform evaluation

The computers used for the experiments have been provided by ATOS ²⁰, and have the following specifications:

- 2× Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz
- 128 GB of RAM
- 2× Tesla V100-PCIE-16GB
- Red Hat Enterprise Linux Server 7.4 (Maipo)
- PCIe 3rd gen ×16

After each time the measurements were collected, the computer’s RAM was cleaned up, evicting cache pages from the system. This was done with the stress test tool stress-ng.

4.4 Evaluation

In this section different phases of Caffe are traced in order to have a better understanding how Caffe works and its performance bottleneck. The figures showing the measurements contain a box plot and a reference line depicting the average labelled with its value. They depict the main analysis points, the data flow (Fig. 9) and control flow (Fig. 10).

The main steps of the data flow, that is how data is currently transferred from storage to the GPU consists in:

1. CPU loads the data from storage into RAM (Storage to RAM),
2. Data gets preprocessed (cropping, random mirroring, mean value subtraction) on the CPU (RAM to RAM),

²⁰https://atos.net/en/
Figure 9: Caffe’s data flow.

Figure 10: Caffe’s control flow.
3. Data is moved to the GPU asynchronously and when the operation is done it is added to the "ready data" vector.

These steps are running on a loop on a separate thread which is responsible of loading and preparing the data. This data preparation thread is always running except if "ready data" vector is full.

The control flow, that is what is invoked from the point of view of the training thread is as:

a. Data layer waits until there is an item in the "ready data" vector and removes the item,

b. orchestrates all of the CUDA kernels and
c. training on the batch takes place on the GPU.

In this analysis section, the element of analysis is the batch size as it is the main parameter used by the Caffe user to tweak the performance. The measurements taken, contain the points from the data flow and control flow, Fig. 9 and 10 respectively, since these are crucial for the analysis. They are going to be explained using the enumeration 1, 2 and 3 for the data flow and a, b and c for the control flow as previously explained. The graphs are plotted with respect to time per image, making it possible to compare them together in one graph.

Fig. 11 shows the overall training performance measured in images per second, treating Caffe as a black box. The measurements contain the training part which consists of 20000 iterations run with different batch sizes. The time spend on initialisation phase has not been included in these measurements. The best performance is taken while using a batch size of 256.

Upon a closer analysis of Caffe, the time spent on each of the 20000 iterations was measured. This contains points a, b and c from the control flow and can be seen in Fig. 12. As expected the batch size of 256 has the smallest average value followed closely by 128.

The following step is splitting the iteration time to finer points, point a which corresponds to the iterations’ time spent waiting for data to be ready and points b and c which correspond...
Figure 12: Graph of the time measurements for 20000 iterations with different batch sizes on NFS. The combined time waiting for data and performing the calculations on the GPU are depicted, per image.

Figure 13: Graph of the time measurements for 20000 iterations with different batch sizes on NFS. Left: The time of an iteration spent on the CPU waiting for data, per image. Right: The time of an iteration spent on the GPU executing computations, per image.
Figure 14: Graph of the time measurements for 20000 iterations with different batch sizes on NFS. The time to load and preprocess an image.

Figure 15: Graph of the time measurements for 20000 iterations with different batch sizes on NFS. Left: Time for an image to be loaded in RAM. Right: Time required to preprocess an image.
to the time the neural network was doing some useful work. Point \(a\) is shown as the "Data layer" graph and points \(b\) and \(c\) as "Actual GPU time" in Fig.13. The measurements show the time spent per image. "Data layer" will be explained when Fig. 14 is introduced.

Considering how the "Actual GPU time" scales, it confirms the hypothesis where small batch sizes would perform poorly since the work they have to do is smaller comparing to the sequential part that are necessary to set things up, memory transfers and invocation overheads. That aside, GPUs come with many cores that can do the same computations across many different SMs which enables to do more work at the same time. The CUDA kernel launch parameters (count, "dimensions", etc.) can effect the performance, depending how well they are aligned with the hardware in use. This can be the reason that the batch size of 128 performs better than the other sizes. Thus the tendency is the bigger the batch size, the faster the training process per image becomes. With bigger batch sizes the values measured are more stable and of a smaller variance which make the execution more predictable and favourable over the bigger variance.

In order to better understand the data path, these next measurements will trace the main phases. Points \(I\), \(2\) and \(3\) are measured as one. The measurements are shown in Fig. 14 ("Batch-time") and it follows a similar trend as in Fig. 12. Points \(I\), \(2\) and \(3\) combined, are responsible for the waiting time in point \(a\) ("Data layer") and the "Data layer" is a combination of "Batch-time" with the time that can be overlapped from "Actual GPU time". "Data layer" has the same trend for values 64-1024 as "Batch-time" but for values 16 and 32 it keeps a straight line which is due to the high "Actual GPU time" corresponding for those values.

Dissecting the values from "Batch-time" to point \(I\), the time to read the data from storage, and point \(2\), the time required to pre-process a batch. These are shown as "Read-time" and "Transformation-time" in Fig. 15 and plotted as the time per image than the time per batch. On "Read-time" it would be expected that all batch sizes would have more or less the same value and have a small variation. On batch sizes of 16 and 32 the average is big but the mean value is significant lower than the other batch sizes. OS scheduling has a role in this but the deeper cause would be the way that the files are loaded in memory. Using LMDB, many small

---

**Figure 16:** GPU SM utilisation (top) and GPU Memory utilisation (bottom) for different batch sizes.
pages (4 KiB) are requested to the OS to be loaded that consists of a random I/O pattern where the storage in use (NFS) facilitates a block size of 1 MiB. The combination of the I/O pattern and page and block sizes lead to this behaviour. Since smaller batch sizes require less data at each iteration, it is more visible with them the difference if data reside in RAM prior to their access. They are faster when data reside in RAM but when the OS needs to access the storage, the cost they pay is bigger relatively. This is the cause for the big variation between the median and average values in batch sizes of 16 and 32. As the batch size gets bigger, the lower end of each measurement gets bigger which corresponds to the fact that more accesses to the storage are required, hence increasing the access cost.

The main slow down of the training process is caused by the "Transformation-time" as it requires double the time of the "Actual GPU time" so it can not be fully overlapped with it and its impact can be seen by the "Data layer". The process behind "Transformation-time" is fully sequential and does not take advantage of all of the cores in the system. Making this process parallel would improve the performance significantly, in this study it could improve it by a factor of 10.

Taking the view of a black box for the GPU (Fig. 16), it can be seen that the overhead caused by the small batch sizes is big. Utilising many small memory copies than fewer larger ones, effects the overhead of the memory copy (from the API invocation to the actual copy) which is expensive and the memory copy’s transfer bandwidth. Up to the batch size of 32 the overhead for the GPU is big, fulfilling the expectations. For the GPU, batch sizes from 128 and bigger start to work better, since they reach zero utilisation, both for SM and memory. The bigger the batch size the more extreme the values get, more time is required to prepare the data (more zero utilisation) and more time to train on the data (more high utilisation).

4.5 Conclusions of the analysis

During the analysis of Caffe, one thing was constant: apart of having a good neural network and a fast Machine Learning framework, it is very important how the neural network’s input data is preprocessed, stored and accessed. Preprocessing data online on the CPU is slow and slows down the training part of network. Moreover, there are some unnecessary memory copies from the device (GPU) to host (CPU). These memory copies insert additional dependencies that slow down the training and fall in an implementation issue of Caffe. Reducing thus the implication of the CPU would reduce the overhead. Storing data in the LMDB format on Lustre leads to slow read performance. LMDB requires many small file reads where Lustre does not perform well. This also makes tracing I/O activity generated by \texttt{mmap} hard and does not allow to be investigated in detail. Future work will rely on creating a tool making it possible to trace I/O calls that are invoked by \texttt{mmap}.
5 Solution to improve the GPU access to the storage system

Fig. 13 clearly shows the GPU is waiting for the CPU to provide it with data and consumes a considerable large amount of time waiting. This makes it very important to speed up this process, reduce the involvement of the CPU on the data path and move directly data from storage to the GPU. This thesis advocates for a two steps solution and the use of data format different than LMDB.

The first step is to prepare the application to handle such data path. Previous solutions required copying data from storage to RAM and then using CUDA functions to copy data from host to device. The suggested solution removes the burden to copy the data to the CPU allowing direct communication from storage to GPU through GPUDirect RDMA. This requires to adjust the memory model of the application, adjust how data are fed into the neural network and create an API to handle all necessary information.

In order to do this, it is necessary to keep track of what data is being currently used, know what data to use next and provide data pointers for the GPU, allowing the CUDA kernels to know on which data to operate on. For this reason this section presents a Data tracking tool (DTT) implemented as part of this thesis. This tool is capable of keeping track of data and has been integrated with Caffe to provide the data pointers that each iteration on the GPU should work on. This tool also handles data movements to the GPU, currently the data is moved from the RAM to the GPU since there is no support for GPUDirect RDMA. The lack of GPUDirect RDMA does not effect how Caffe or any other tool uses DTT. It is agnostic for Caffe how data are transferred to the GPU.

The second step is to implement the direct data movement from the storage to the GPU. The storage adapter’s driver should be adjusted to support GPUDirect RDMA. Last action requires the use of the driver within the restructured application (Step one) and substitute the RAM to GPU with appropriate storage to GPU calls. In case of absence of the driver support, DTT could fallback to transferring the data through the CPU for completeness.

In addition to both steps mentioned above, there is a need to use a different data format than LMDB. This is critical for the proposed solution to work. The proposed format requires storing data simply in a C-contiguous way, making it possible to load data without underlying on a complicated data format that needs parsing.

Step one has been implemented and evaluated in Section 5.1 and the potential of step two is shown by modelling the behaviour and performance in Section 5.2, leaving the implementation as future work.

5.1 Step one: Delegating data loading to DTT (Data Tracking Tool)

Implementation of DTT In order to implement DTT, there was a need to select a framework and base the tool on it. Caffe has been chosen because it is widely used in the community and provides an easy installation setup. The C++ implementation has been an advantage as it provides more flexibility for optimizations. Other alternatives that were considered where Caffe 2 and TensorFlow.

DTT has been implemented as an API. It is not specific to Caffe and can be used and adapted to any other framework and application. In the case of Caffe, DTT has been used in a modified version of ”data layer”, adapted to call the API which requests a data pointer to use in the specific iteration. The only required change of the data layer is to pass through the results of the API’s call. The part that Caffe interacts with DTT is simple and DTT hides the complexity of tracking the data movements, providing a higher level interface. For simplicity DTT has been implemented in one of Caffe’s header files. This includes the public Handling class and all of the internally used classes. Because the target of this thesis was not to fix
specifically Caffe, Caffe has been modified so it passes preprocessed data to the tool on the CPU, imitating the data source that DTT uses instead of a file on disk.

The implementation of DTT is based on providing the following set of functions:

1. Set source data from CPU,
2. Get data pointers for the GPU,
3. Get additional information for a batch that is the same for all iterations and
4. Move to the next batch.

DTT orchestrates all of the memory copies, placing asynchronous memory copies for data that will be used in the future and includes thread safety to be able to concurrently add and remove data. If data are requested from DTT that have not yet arrived to the GPU, the API call will pause until the data are available.

Memory is allocated on the GPU for DTT’s purposes. The reserved memory on the GPU consist of the numbers of batch shown in equation 1.

\[
\text{super\_batch\_size} \cdot \text{gpu\_factor}
\]  

(1)

Super_batch_size defines the amount of batches that should be transferred at once and gpu_factor shows how many super_batch_size should fit in the GPU. The memory hierarchy with super_batch_size and gpu_factor can be seen in Fig. 17.

Let us take an example with super_batch_size = 3 and gpu_factor = 2. When DTT is requested for data the first time, it will move three batches to the GPU synchronously and three more asynchronously. When DTT is asked for data, it computes and provides a pointer with an offset with the data requested, for the GPU. In case the batch’s index is 3 (or 0), the tool will perform an asynchronous data transfer of the super_batch into the memory area that was just stopped being used, that is batches 0 - 2 (or 3 - 5). In the example, there are only
two memory copies per 6 iterations, transferring three batches of data at once leading to higher throughput.

**Evaluation of DTT** The metrics used for the evaluation of DTT include the application’s throughput (img/s), the effective time required to process an image on the GPU and SM utilisation.

Training Alexnet with DTT version of Caffe with super_batch_size = 2 and gpu_factor = 2 can be seen in Fig. 18. Iterations can happen one after another without idle time due to the data feeding. CUDA kernels are getting launched from one iteration to the next without any problem.

DTT version of Caffe fully utilises the hardware which can be seen in Fig. 19. The main benefit of DTT is increasing the GPU utilization, reaching full utilisation on both GPU’s SM and memory. However, the target of DTT is not necessarily to improve performance.

Caffe’s throughput with both original and proposed versions is shown in Fig. 20. The proposed version gives more than a 2× speed up over the original version. It has to be made clear, with the proposed version no preprocessing happens online. DTT serves data fast enough so that the neural network does not stop working.

Evaluating the effective time per image required by the original and DTT version of Caffe, DTT provides the edge while training in all batch sizes tested, produces a lower average and mean value and also provides a smaller variation, as shown in Fig. 21. What is of interest is the fact that as the batch size becomes bigger the training performance is degrading. There is not a definite answer to why this is happening, one thing that was observed was the GPU overheating during the measurement process, that is the GPU’s memory surpassing the maximum operating temperature. It is not clear how the overheating GPU memory module behaves. It could have leave the GPU to either lower its working frequency or halt the execution of the CUDA kernels until working temperature was reached again, which is the SMs’ overheating behaviour [30]. It is necessary to know how the overheating module works, the GPU’s memory is a critical component in order for the SMs to operate and do their computations.

### 5.1.1 Limitations

A few limitations are present in the implementation. These limitations vary from GPU hardware architecture, to GPU temperature issues and availability of GPUDirect RDMA.

In the current GPU architecture there can only be one memory copy per direction at a given
Figure 19: Comparison of SM and memory utilisation between the original and DTT version of Caffe.

Figure 20: Comparison of Caffe’s performance with the original and proposed solution.
moment. The available directions are host to device, device to host and device to device. This means that when two memory copies, submitted in different streams, are serialised. To get the second memory copy started, the first one has to be completed, even if memory copies are independent from each other. For this reason, hidden dependencies are introduced.

In this thesis the Alexnet model is used, which in the duration of the GPU computations, it requires memory transfers from host to device during an iteration. These can interfere with asynchronous memory copies orchestrated by the tool that are done to transfer data in time to the GPU. Such interference increases the time to execute an iteration since the imposed memory copy dependency. This has been illustrated in Fig. 22 where it can be seen that the data copy stalls another memory copy, leaving the GPU underutilised. For this, it was observed that a granularity of $super\_batch\_size \leq 2$ did not impose an extra waiting time for the batch size of 256.

Another limitation found is related to the specific hardware setup; the GPU was reaching maximum SM and memory temperatures, decreasing performance with larger batch sizes. Last but not least, the implementation is tied to pass the data through the CPU since there is no support for GPUDirect RDMA. With support for GPUDirect RDMA the new work flow would be like:

1. Specify file from which to load the batches,
2. Prefetch data to the GPU (Storage to GPU),
3. Caffe requests a data pointer and moves to the next batch and
4. DTT prefetch’s the next batch if necessary.
5. Repeat steps 2 - 4.

In the current implementation, step one differs from the one just described. The current step one provides preprocessed batches on the CPU. With the support of GPUDirect RDMA the work flow of the tool would look like in Fig. 23.
Figure 22: CUDA limitations on memory copies. The training process is paused until the data transfers are finished.

Figure 23: Proposed workflow, in this workflow data have the opportunity to be directly moved to the GPU without passing through the CPU.
5.2 Step two: GPUDirect RDMA model

There is no support yet for GPUDirect RDMA for storage devices. Providing support for GPUDirect RDMA is the responsibility of storage device driver producers. Such support is critical for DTT, since it will be built on top of it. Since it is not currently available, a methodology is proposed to model such behaviour and describe the performance benefit. The methodology relies on two steps, (1) tracing and measuring several points of Caffe’s original implementation and extract the values corresponding to the parts that will not require change by using the proposed solution and (2) replace some steps in the implementation in order to use the proposed solution and estimate the potential of the approach.

This section takes as part of the hypothesis the following parameters, the model is agnostic to their values and they are taken as example only. The values have been obtained through calibration methodology and are presented in the Analysis section (Section 4). The parameters will first be explained and their measured values, for batch_size of 256, 512 and 1024, presented in Table 2. The values for the parameters on the following sections are being calculated with batch_size of 256.

- **batch_size** - the number of images to train on in a single iteration.
- **nb_iters** - the number of iterations for the model’s prediction.
- **size_storage_ram** - data size for a batch_size of images from storage to RAM
- **time_*1*_2** - the time for each connection is calculated by the formula size_*1_/ bw_*2_.
- **time_lmdb** - median of ”read time” for all measurements in NFS setting × batch_size. This measurement contains potentially waiting for files to be read, parsing LMDB and deserialising protobuf.
- **time_preprocess_cpu** - time to do the preprocessing on the CPU containing mean subtraction, cropping and random mirroring. This is taken from the average of all measurements, computing the time for an image $\rightarrow 0.973 \cdot batch_size$.
- **time_preprocess_gpu** - time to do the preprocessing on the GPU. This has been measured using ported CPU code and can be optimised and made faster, which is not the goal of the port. Measured values after running the benchmark 50 times for every batch size.
- **size_ram_gpu** - data size for a batch_size of images from RAM to GPU.
- **bw_ram_gpu** - bandwidth for data transfers between RAM and GPU through PCIe (peak theoretical bandwidth 16 GB/s).
- **time_compute_gpu** - time to do the computations on the GPU for an iteration, also known as time to train the model for an iteration.
- **bw_storage_gpu_PCIe** - bandwidth for data transfers between storage and GPU through PCIe (peak theoretical directional bandwidth 16 GB/s).
- **bw_storage_gpu_NVLink** - bandwidth for data transfers between storage and GPU through NVLink (peak theoretical directional bandwidth 150 GB/s).
- **T_average** - the average time for an iteration for a model’s prediction.
- **average_time** - average time measured for an iteration.
<table>
<thead>
<tr>
<th>Variable name</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>batch_size</td>
<td>256</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>nb_iters</td>
<td>20000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size_ram_storage</td>
<td>39.578 MB</td>
<td>79.157 MB</td>
<td>158.314 MB</td>
</tr>
<tr>
<td>time_lmdb</td>
<td>38.65 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time_preprocess_cpu</td>
<td>249.08 ms</td>
<td>498.17 ms</td>
<td>996.35 ms</td>
</tr>
<tr>
<td>time_preprocess_gpu</td>
<td>11.14 ms</td>
<td>12.28 ms</td>
<td>12.08 ms</td>
</tr>
<tr>
<td>size_ram_gpu</td>
<td>158.298 MB</td>
<td>316.596 MB</td>
<td>633.192 MB</td>
</tr>
<tr>
<td>bw_ram_gpu</td>
<td>11.65 GB/s (measured)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time_compute_gpu</td>
<td>137.49 ms</td>
<td>269.64 ms</td>
<td>537.75 ms</td>
</tr>
<tr>
<td>average_time</td>
<td>306.07 ms</td>
<td>645.98 ms</td>
<td>1336.81 ms</td>
</tr>
<tr>
<td>bw_storage_gpu_PCIe</td>
<td>7 GB/s (measured)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bw_storage_gpu_NVLink</td>
<td>150 GB/s (peak directional bandwidth)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c_caffe</td>
<td>4.49 ms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Values for the parameters explained for batch sizes of 256, 512 and 1024.

- **c_caffe** - constant random time for Caffe. This value is calculated through $T_{\text{average}_0}$ which has $c_{\text{caffe}} = 0$ and then it is calculated as
  
  \[ c_{\text{caffe}} = T_{\text{average}} - T_{\text{average}_0} \]

- **overlap** - the time that can not be overlapped and needs to be executed before the GPU computation can run.

- **t_{iteration}** - end to end latency, reading the images, moving them to the GPU and doing the GPU computations.

This section describes the models obtained in two steps. First the one corresponding to the existing implementation and then iteratively adding new improvements to the existing solutions.

The first one evaluates the Current state of Caffe, that is using LMDB and the preprocessing on the CPU. With the current state, the **best case** will try to be modelled (Section 5.2.4), assuming that all pages reside in RAM prior to their requests. The **worst case** will be modelled where the assumption is that none of the pages reside in to RAM. Two cases are taken, one where data reside into an NFS storage adapter (Section 5.2.2) and one where data resides under the lustre file system (Section 5.2.3).

Next, a model that assumes that no longer LMDB (Section 5.2.5) will be used is considered. In this model, there will be no restrictions on the block size used required by LMDB (4 KiB) that might reduce read performance. Then a model moving the preprocessing to the GPU is shown (Section 5.2.6). Finally two models that utilise GPUDirect RDMA for a direct storage to GPU data transfer, including PCIe (Section 5.2.7) and NVLink (Section 5.2.8), are shown.

### 5.2.1 Model current state of Caffe

This model tries to describe the current state of Caffe, in the general case. With this model, the best and worst cases will be described in the following sections. Data transfers from storage to RAM, time for LMDB, preprocessing and data transfers from RAM to GPU can overlap.

\[
\text{overlap} = \max(0, t_{\text{storage}} + t_{\text{lmdb}} + t_{\text{preprocess}} + t_{\text{ram}} - t_{\text{compute}})
\]

\[
\text{t_{iteration}} = t_{\text{storage}} + t_{\text{lmdb}} + t_{\text{preprocess}} + t_{\text{ram}} + t_{\text{compute}} + c_{\text{caffe}}
\]
\[ T_{\text{average}} = \frac{t_{\text{iteration}} + (nb_{\text{iters}} - 1)(overlap + t_{\text{compute, gpu}} + c_{\text{caffe}})}{nb_{\text{iters}}} \]

5.2.2 Model worst case nfs (wc - nfs)

In this section, a worst case scenario will be examined, that is when all of the mmap pages do not reside in RAM. For this, all data will be loaded synchronously from an NFS storage adapter. The effective bandwidth is affected by LMDB’s limitation that stores data with a block size of 4 KiB. This bandwidth value is taken from a benchmark measurement shown in Section 2.8. The target is to know how Caffe can perform in the worst case with NFS.

\[ \text{overlap} = \max(0, t_{\text{storage, ram, nfs}} + t_{\text{lmdb}} + t_{\text{preprocess, cpu}} + t_{\text{ram, gpu}} - t_{\text{compute, gpu}}) = 704.51ms \]

\[ t_{\text{iteration}} = t_{\text{storage, ram}} + t_{\text{lmdb}} + t_{\text{preprocess, cpu}} + t_{\text{ram, gpu}} + t_{\text{compute, gpu}} + c_{\text{caffe}} = 984.23ms \]

\[ T_{\text{average}} = \frac{t_{\text{iteration}} + (nb_{\text{iters}} - 1)(overlap + t_{\text{compute, gpu}} + c_{\text{caffe}})}{nb_{\text{iters}}} = 846.75ms \]

\[ \text{Speed} \rightarrow \frac{\text{batch size} \cdot 1000}{T_{\text{average}}} \rightarrow 302.33img/s \]

5.2.3 Model worst case - lustre (wc - lustre)

Another worst case scenario will be modelled now using lustre instead of NFS as the storage adapter.

\[ \text{overlap} = \max(0, t_{\text{storage, ram, lustre}} + t_{\text{lmdb}} + t_{\text{preprocess, cpu}} + t_{\text{ram, gpu}} - t_{\text{compute, gpu}}) = 7574.05ms \]

\[ t_{\text{iteration}} = t_{\text{storage, ram}} + t_{\text{lmdb}} + t_{\text{preprocess, cpu}} + t_{\text{ram, gpu}} + t_{\text{compute, gpu}} + c_{\text{caffe}} = 7853.77ms \]

\[ T_{\text{average}} = \frac{t_{\text{iteration}} + (nb_{\text{iters}} - 1)(overlap + t_{\text{compute, gpu}} + c_{\text{caffe}})}{nb_{\text{iters}}} = 7716.17ms \]

\[ \text{Speed} \rightarrow \frac{\text{batch size} \cdot 1000}{T_{\text{average}}} \rightarrow 33.17img/s \]

Lustre performs bad with many small sized reads, this is why the model predicts such a bad performance.
5.2.4 Model Best Case (BC)

In this section the best case scenario of Caffe is modelled. This means that all mmap pages reside into RAM prior of their request, hence the \( t_{\text{storage}, \text{ram}, \text{nfs}} \) is being 0.

\[
\text{overlap} = \max(0, t_{\text{storage}, \text{ram}, \text{nfs}} + t_{\text{lmdb}} + t_{\text{preprocess}, \text{cpu}} + t_{\text{ram, gpu}} - t_{\text{compute, gpu}}) = 163.82 \text{ms}
\]

\[
t_{\text{iteration}} = t_{\text{storage, ram}} + t_{\text{lmdb}} + t_{\text{preprocess, CPU}} + t_{\text{ram, gpu}} + t_{\text{compute, gpu}} + c_{\text{caffe}} = 443.55 \text{ms}
\]

\[
T_{\text{average}} = \frac{t_{\text{iteration}} + (n_{\text{iters}} - 1)(\text{overlap} + t_{\text{compute, gpu}} + c_{\text{caffe}})}{n_{\text{iters}}} = 306.07 \text{ms}
\]

\[
\text{Speed} \rightarrow \frac{\text{batch\_size} \cdot 1000}{T_{\text{average}}} \rightarrow 836.4 \text{img/s}
\]

The actual measurements taken by NFS are very close to this. This can relate to some of the OS’ settings such as the read_ahead value for the disk adapter.

5.2.5 Model 1: Removing LMDB (No LMDB)

In this section a model predicting the performance of Caffe without the use of LMDB is made. It is assumed that all data within LMDB’s substitute are placed sequentially and do not require random I/O. Removing LMDB allows the use of bigger block sizes than 4 KiB which can produce a better I/O performance. 4 KiB as the block size reduces the I/O throughput.

The value that has been used is 1889 MB/s and has been taken from Section 2.8 that consist of benchmark measurements.

\[
\text{overlap} = \max(0, t_{\text{storage, ram, best}} + t_{\text{preprocess, cpu}} + t_{\text{ram, gpu}} - t_{\text{compute, gpu}}) = 146.35 \text{ms}
\]

\[
t_{\text{iteration}} = t_{\text{storage, ram, best}} + t_{\text{preprocess, cpu}} + t_{\text{ram, gpu}} + t_{\text{compute, gpu}} + c_{\text{caffe}} = 426.07 \text{ms}
\]

\[
T_{\text{average}} = \frac{t_{\text{iteration}} + (n_{\text{iters}} - 1)(\text{overlap} + t_{\text{compute, gpu}} + c_{\text{caffe}})}{n_{\text{iters}}} = 228.05 \text{ms}
\]

\[
\text{Speed} \rightarrow \frac{\text{batch\_size} \cdot 1000}{T_{\text{average}}} \rightarrow 887.05 \text{img/s}
\]
5.2.6 Model 2: Moving the preprocessing to the GPU without LMDB (preproc GPU)

A model moving the preprocessing computations to the GPU is shown. The model is based on Model 1, which does not use LMDB but another file format. In this model, data transfers from storage to RAM and from RAM to GPU can be overlapped with the preprocessing on the GPU and the computation time on the GPU, changing the overlap formula. The order of the computations change comparing with the previous.

\[
\text{overlap} = \max(0, t_{\text{storage-ram best}} + t_{\text{ram-gpu}} - t_{\text{preprocess-gpu}} - t_{\text{compute-gpu}}) = 0 \text{ms}
\]

\[
\text{t}_{\text{iteration}} = t_{\text{storage-ram best}} + t_{\text{ram-gpu}} + t_{\text{preprocess-gpu}} + t_{\text{compute-gpu}} + c_{\text{caffe}} = 188.4 \text{ms}
\]

\[
T_{\text{average}} = \frac{t_{\text{iteration}} + (n_{\text{iters}} - 1)(\text{overlap} + t_{\text{preprocess-gpu}} + t_{\text{compute-gpu}} + c_{\text{caffe}})}{n_{\text{iters}}} = 153.64 \text{ms}
\]

\[
\text{Speed} \rightarrow \frac{\text{batch size} \cdot 1000}{T_{\text{average}}} \rightarrow 1666.15 \text{ img/s}
\]

This generates a huge throughput boost, having more than \(2\times\) speedup and reducing latency from 300ms to 200ms.

5.2.7 Model 3a PCIe: Moving the preprocessing to the GPU with direct data path (DD PCI)

In this model the potential of a direct data path from storage to GPU is shown. The data is transferred directly from the storage adapter to the GPU over PCIe and the preprocessing takes place on the GPU with the iteration of the neural network. Data transfers can overlap with the preprocessing and the network’s iteration.

The value used here for \(t_{\text{compute-gpu}}\) is 126.69 ms which was the value measured with the use of DTT. The value is used in this model because the direct data path from storage to GPU requires the deep learning framework to be changed from the current implementation.

\[
\text{overlap} = \max(0, t_{\text{storage-gddr_PCI}} - t_{\text{preprocess-gpu}} - t_{\text{compute-gpu}}) = 0 \text{ms}
\]

\[
\text{t}_{\text{iteration}} = t_{\text{storage-gddr_PCI}} + t_{\text{preprocess-gpu}} + t_{\text{compute-gpu}} + c_{\text{caffe}} = 148.49 \text{ms}
\]

\[
T_{\text{average}} = \frac{t_{\text{iteration}} + (n_{\text{iters}} - 1)(\text{overlap} + t_{\text{preprocess-gpu}} + t_{\text{compute-gpu}} + c_{\text{caffe}})}{n_{\text{iters}}} = 142.84 \text{ms}
\]

\[
\text{Speed} \rightarrow \frac{\text{batch size} \cdot 1000}{T_{\text{average}}} \rightarrow 1792.14 \text{ img/s}
\]

The throughput is slightly better than when data goes through the CPU but the latency reduces by \(\frac{1}{3}\) comparing to Model 2.
5.2.8 Model 3b NVLink: Moving the preprocessing to the GPU with direct data path (DD NVLink)

The model in the section is the same as Model 3a (Section 5.2.7) with the only difference that the data is transferred over NVLink.

\[
\text{overlap} = \max(0, t_{\text{storage,gddr,NVLink}} - t_{\text{preprocess,gpu}} - t_{\text{compute,gpu}}) = 0 \text{ms}
\]

\[
\text{t}_{\text{iteration}} = t_{\text{storage,gddr,NVLink}} + t_{\text{preprocess,gpu}} + t_{\text{compute,gpu}} + c_{\text{caffe}} = 143.1 \text{ms}
\]

\[
T_{\text{average}} = \frac{\text{t}_{\text{iteration}} + (\text{nb}_\text{iters} - 1)(\text{overlap} + t_{\text{preprocess,gpu}} + t_{\text{compute,gpu}} + c_{\text{caffe}})}{\text{nb}_\text{iters}} = 142.84 \text{ms}
\]

Speed \rightarrow \frac{\text{batch}_\text{size} \cdot 1000}{T_{\text{average}}} \rightarrow 1792.14 \text{img/s}

The result is interesting as it shows that having an NVLink instead of PCI does not improve the overall performance, showing that the main bottleneck is placed at the data management level.

5.2.9 Discussion

DTT provides several benefits for Caffe and can be improved to integrate different optimisations regarding the data placement and tracking. LMDB has been estimated as not being able to scale to large data sets. Thus, Caffe can outsource its file handling to DTT, without caring about the actual data placement. This allows Caffe to make improvements at different places. Just by the use of DTT there is not directly a big performance improvement, it makes Caffe easier to handle data. Moving the online preprocessing to the GPU gives directly a performance
boost to Caffe, which is unrelated to DTT. It should be considered that points of the online preprocessing could happen at the offline preprocessing phase. DTT has the potential to allow a direct data path from storage to GPU that is unrelated and transparent to Caffe. Changing DTT’s implementation to handle storage to RAM to GPU and storage to RAM would not have any effect on the way Caffe is now using DTT.

Moving the preprocessing to the GPU speeds up the training process, even though some of the GPU’s time has to be spend to preprocess the data. According to Fig. 24 the overall performance can be increased with more than 2× by using the proposed model. NVLink or PCI do not change the performance, however a better management of the data is the reason for the presented speedup.

Introducing a different data format than LMDB improves the latency and can make the process of data reading predictable, allowing optimisations at different levels, such as the storage driver or prefetch level allowing to fetch data accurately. Improving the data format can only reduce the latency up to a specific point. Removing the CPU from the critical path can reduce latency even more, eliminating the unnecessary data movements.

It has been shown that the CPU is a bottleneck when using a single GPU. Systems like DGX using multiple GPUs are expecting to face even more performance drawbacks.

6 Related work

Significant amount of work has been done recently in order to improve Deep Learning frameworks as well as the underlying hardware. The usability of these frameworks has been an important requirement as often users are not computer scientists, instead they might have background in biology, image processing, etc. Therefore, several related works target improving specific frameworks for a specific use without changing the habits of the users (e.g. use LMDB). Nevertheless, other fields have encountered the same issues when accessing the storage and using the GPU for computations, like database applications. Other generic solutions like GPUfs target moving the file system entirely to the GPU which can lead to a very complex system. These solutions are presented, that have the same direction as this thesis but that are applied either at the application level or for a different purpose.

6.1 Tensor Comprehensions

Deep Learning is been widely used and developed. New algorithms are created and researchers want to test these out. These researchers might lack the background to make their algorithms perform fast, or the operations required by the algorithms might not perform as fast as optimised libraries provided by vendors, e.g. cuDNN or the optimised libraries might not support them.

Tensor Comprehensions (TC) [32], [33] tries to help, providing highly optimised, native code using Just-In-Time (JIT) compilation. TC tries to find the best implementation for a given set of operations. This product has been recently announced and it is currently at an early stage of development.

TC could be used to improve some layers of Caffe or might help create GPU layers easier. This could be used to create a more optimised way to do the preprocessing on the GPU comparing to the naively ported CPU code used to obtain the measurements in Section 5.2. This is complementary to the study of this thesis.

6.2 PG-Strom

SQL queries on a large amount of data require a lot of processing power. Most data usually get filtered out by ”where” clauses, which is a set of comparison operations. These operations are
highly parallel, which is the kind of operations where the GPUs strive.

PG-Strom \(^{21}\) is an extension for PostgreSQL developed by HeteroDB which tries to take advantage of the massive computational power of the GPUs \(^{34}\)–\(^{36}\). PG-Strom off-loads the heavy computations of a query to the GPU asynchronously. It does not try to use the GPU for all of the operations, it schedules tasks both at the CPU and GPU depending which device is more suitable for each operation. Some of the computations that occur on the GPU are Table scan (where), Table join (join), group-by and projection (select). PG-Strom has been designed for On-line Analytical Processing (OLAP) which is read and computation hungry and it does not try to solve On-Line Transaction Processing (OLTP) whose performance is counted by the the latency. It tries to move the computations within the DBMS system and exporting the result, than exporting the whole database, which in some cases it is very costly due to the database size, and then compute the results with a 3rd party. For this PG-Strom allows the user to define CUDA code to be run within the database, to utilise the locality of the data and avoid the exportation of the data \(^{37}\).

An example of the data path with PG-Strom follows, the GPU acquires data from the storage, processes it, then it sends the results to the CPU and the CPU does any computations that are left. In this way the data sent to the RAM is only those that need more complex computations that the GPU cannot perform efficiently, reducing the memory transfers to RAM and CPU utilisation.

In order to be able to handle the vast amount of data used in the GPU an efficient way to load data from storage is needed. For this reason HeteroDB created a Linux kernel module to directly copy data from the storage Solid State Drive (SSD) to the GPU using Direct Memory Access (DMA). This kernel module has been implemented using GPUDirect RDMA. An older implementation of the driver called nvme-kmod is open source and can be found at GitHub \(^{22}\). The current version of the driver called NVMe-Strom is not open source but binaries can be

\(^{21}\)http://heterodb.com/

\(^{22}\)https://github.com/kaigai/nvme-kmod
found at PG-Strom’s binaries website. The pipeline from the SSD to the GPU can be viewed as an intelligent SSD-GPU storage device that can interpret SQL queries and fetch only relevant data to the RAM.

Performance measurements using PG-Strom are presented in [35], this shows 2 - 3× speedup in the throughput using PG-Strom comparing to PostgreSQL, reaching the SSDs’ theoretical bandwidth. A case study regarding drug discovery [37], PG-Strom was able to reduce the execution time from 50 minutes to 19 seconds, comparing the GPU and CPU implementation.

This related work provides an inspiration for the potential of the proposed solution using GPUDirect RDMA, where the driver support has been implemented and used in a different domain.

6.3 GPUfs

There has been another attempt to change the programming model where the GPU can request that data that it needs, directly from storage. GPUfs [38] exposes the file system directly to the GPU with a Portable Operating System Interface (POSIX)-like API, allowing GPU code to do all the I/O calls for the data that it requires.

GPUfs reduces the complexity of programs that use it since it hides the data transfers from the CPU to the GPU, it shares some similarities with Unified Memory (Section 2.6.3) and allows to create GPU applications that have no CPU code.

GPUfs might be exposing the I/O operations directly to the GPU but the data path contains the CPU. GPUfs has not been widely used and has no active development. Even though the solution can provide multiple benefits it is not adapted to Deep Learning frameworks as it brings a significant change in the data flow of the application as well as potential bottlenecks.

7 Conclusions and Future work

Machine Learning applications are highly demanded today in different fields like biology, image processing, self-driving cars, etc. Significant effort has been done recently to numerically improve Deep Learning frameworks and allow them to provide more accurate results. Having more input data has been a requirement to tackle this goal. The highly parallel structure of these applications makes the GPU eligible as a main processor. However, handling more and more data puts a significant pressure on the storage system and on the way the data is transferred to the GPU, even more when the corresponding CPU host is in charge with handling this data transfer.

This thesis addresses this problem first, by analysing the data flow of Caffe, one of the most commonly used frameworks in the community, and second by proposing a solution to accelerate this data flow.

The first contribution is an analysis methodology, that provides insight into how different parameters of Caffe can impact the performance (e.g. the batch size). The results show that parameters are chosen in order to allow the CPU to be efficient in loading the data in memory and preprocessing it for the GPU, even if this later would be more efficient with larger sets of data. Moreover, the analysis shows that the implementation has a significant bottleneck at fetching the data from the storage system and prepare it to be sent to the GPU. As a result the GPU is utilized at only 50 % Therefore, the CPU’s involvement in the data flow clearly alters the performance. Lowering its involvement and insuring a direct transfer from the storage system to the GPU can thus overcome this issue.

23https://heterodb.github.io/swdc/
The second contribution of this thesis is a proposal to reduce the CPU’s involvement and handle the data placement directly on the GPU. The tool is called DTT (Data Tracking Tool) and is able to track where the data is and place it efficiently to the GPU. DTT is implemented as an API agnostic to Caffe, but integrated with Caffe for the evaluation of this thesis. Even if the purpose of this tool is not to improve the performance directly, up to 10% improvement for a batch size of 256 can be seen. However, the main target of this tool is to improve the GPU utilization, which it does up to 100%.

The last contribution consists in providing a model that estimates the performance benefits of transferring the data directly from storage to the GPU through a GPUDirect RDMA solution. Network adapter companies have already provided a driver implementation to handle this but this is not available for storage systems. The model shows that there is a significant benefit of implementing such a solution increasing the performance to $2 \times$ faster. Nevertheless, using NVIDIA’s NVLink solution instead of PCIe to connect the storage system with the GPU does not bring a performance improvement on a single GPU system, showing thus the bottleneck is not at the hardware level but at the software data management.

One of the main difficulties of this thesis has been LMDB data layout, widely used by most of the Deep Learning applications. It is a solution inherited from LDAP implementations, chosen mainly for the easy usability and not for performance. It’s performance has significant limitations: firstly because it uses the mmap solution which can lead to scalability issues for future larger data sets and secondly because its random I/O pattern has significant performance impact on the file systems and the storage system. The result of this thesis shows that replacing such a data layout with a simple “array-type” format doing explicit I/Os can improve the performance and leave the place for future optimisations even if it can require more effort from the user. Multiple preprocessing phases are already applied offline in order to have the dataset used by LMDB. This could still be the case with a more simpler format. Moreover, Caffe should move to the offline preprocessing phase, other preprocessing operations currently applied online in the current state. Allowing to do as much of the preprocessing as possible while the neural network is not being trained.

Future work consists in proposing such a data layout model that could be run on the GPU directly and implementing the proposed model at kernel level by providing support for GPUDirect RDMA in the storage driver. Independently of this model, another future work consists in implementing a tool able to trace I/O operations executed as a result of the mmap operation and better understand the I/O patterns of applications using such a method to do implicit I/O operations.

A very interesting thing to be analysed in the future is how Deep Learning frameworks work in multi-GPU environments. It is interesting to investigate how they scale and what the bottlenecks are in such environments, as well how the resources are shared between the multiple GPUs. The future seems to going towards multi-GPU systems, making it important to understand how they work and the challenges they face.

---

24https://github.com/flanaras/caffe
References


