GPGPU for High-Performance Neural Networks

Sven Lüpke
Department of Informatics
Technical University of Munich
Munich, Germany
luepke@in.tum.de

***Abstract*—General purpose programming capabilities of GPUs provide a significant increase in performance for many applications outside of computer graphics. One of the most notable is the acceleration of neural network training. Without GPUs many breakthroughs in the field of deep learning would not have been possible. We will show why neural networks benefit from the hardware architecture of GPUs and how their training can be scaled in a distributed manner.**

I. Introduction

General purpose programming on graphics processing units (GPGPU) is applied to many fields in high-performance computing such as astrophysics and weather forecasting. In deep neural network learning especially, many advancements would not have been possible without highly parallel processing on GPUs. In this paper we explain how neural network training benefits from GPUs.

First, we provide an insight into modern GPU hardware and how it can be leveraged to improve the performance of general purpose programs. We also outline several GPU interconnect technologies that allow high-bandwidth communication between multiple GPUs. Then we explain how a matrix multiplication can efficiently be implemented on a GPU and provide a performance comparison to a CPU implementation. To show how training neural networks is accelerated on a GPU we describe how the gradient descent algorithm can be almost entirely performed through matrix multiplications. Finally we present general techniques for distributing learning across multiple compute nodes.

Regarding the notation we will use lowercase letters to denote scalars, lowercase bold letters for vectors and uppercase letters for matrices. In some cases Nvidia's and AMD's terminology may differ from one another. We will use Nvidia's terminology in the rest of this paper.

II. GPU Hardware

The primary advantage of GPUs over CPUs is their highly parallel architecture. This section will give an overview of how GPUs developed into general purpose hardware, their architecture today and various GPU interconnect technologies.

A. Evolution of GPGPU

GPUs are primarily designed for efficient 3D graphics rendering. The first consumer GPUs in the nineties could not execute custom programs. The hardware was designed as a **fixed-function graphics pipeline**. In 2001 GPUs started to support programmable vertex and pixel shaders. Although these programs were still a far cry from today's GPU programs in terms of flexibility, they were already used for parallelizing computations not related to graphics, such as fluid dynamics simulations [1]. In 2006 the introduction of geometry shaders allowed an even wider range of general-purpose algorithms to be efficiently implemented on the GPU [2]. At this point each core on a GPU was still designed to run only one specific shader type. The introduction of Nvidia's **Compute Unified Device Architecture** (CUDA) in 2007 unified all cores and allowed general purpose computations to be performed independently from the graphics pipeline [3].

B. GPU Architecture

Even though some of the world's most powerful supercomputers such as *Summit* [4] and *Sierra* [11] are equipped with thousands of GPUs (27.648 and 17.280), efficient program implementations are still important to archive optimal performance. Thus, we give a short overview of today's general GPU hardware architecture and its effect on program performance.

A modern GPU contains thousands of **arithmetic logic units** (ALUs). These can perform both floating point and integer operations. Additionally recent Nvidia GPUs provide Tensor Cores [6]. They are specifically designed for fast matrix multiplications which are fundamental for training neural networks. Ampere Tensor Cores can perform a $8 \times 4 \times 8$ matrix multiplication in a single clock cycle [27].

In a GPU ALUs are grouped together into a **streaming multiprocessor** (SM). Work that is to be executed on the GPU is divided into **warps**. A warp consists of 32 or 64 **threads**, i.e. invocations of the same program. Each warp is processed by one SM, where the program will be executed in lock-step for all threads, each on one ALU. For that reason the GPU ALUs are also called SIMD lanes [5]. A consequence of this design is that divergent branches within one warp should be avoided. If only some threads in warp execute a branch, it is still executed by all ALUs in the SM. This will result in a waste of processing power, as the other threads do not need the results of this branch.

A **warp scheduler** decides which warp to execute and allocates resources for each thread in the current warp [5].

Memory can generally be divided into **global memory**, **thread group shared memory** (TGSM) and **registers**. Accessing global memory occurs at a higher latency than on CPUs, because less chip space is dedicated to caches and more

to ALUs [7]. The warp scheduler can hide this latency by quickly switching between different warps. Whenever a warp stalls on a memory read, it is swapped for another warp that is run until it stalls as well. Once the initial warp is scheduled to run again, the data requested from global memory might already be available.

TGSM on the other hand has a relatively limited capacity (128 kB on a Volta GV100), but also a much lower latency, as it is placed directly in the SMs [6], [8]. TGSM is commonly used to reduce the number of accesses to global memory and to share data between threads. If, for example, multiple threads in a thread group need to load data from the same address in global memory, it is sufficient for only one thread to load the data into TGSM where all threads in the group can access it.

Registers are used to store data local to each thread and can be accessed instantly with practically no latency. They are stored in one or multiple **register files** that are placed on the SM. A Volta GV100 SM contains four register files. Each can store up to 16.384 32-Bit values [6].

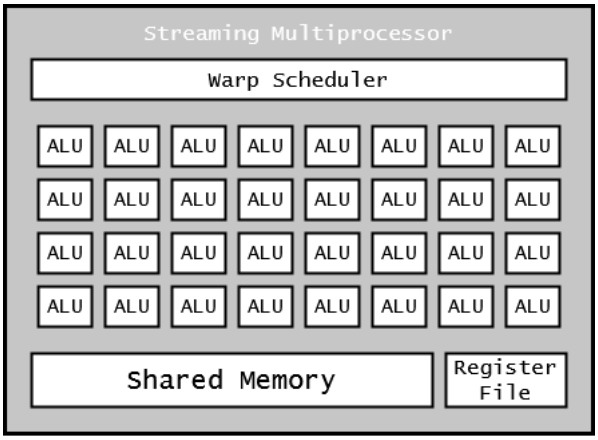

Fig. 1. A streaming multiprocessor containing 32 ALUs

GPU performance can be significantly improved by optimizing memory access patterns. If all threads in a warp read from a continuous region in global DRAM memory, with the k-th thread accessing the k-th element in the region, the read will be **coalesced**. This enables all reads in a half warp (16 threads) to be performed in a single memory transaction [16].

To provide a higher bandwidth for concurrent accesses, TGSM is divided into equally sized **memory banks**. Each bank can serve only one memory access at a time. The bank size can be configured to be either four or eight bytes. If multiple threads access the same bank, this will result in a **bank conflict**. This should be avoided, because in this case the accesses cannot be performed in parallel and must be serialized [18].

Occupancy is a crucial factor to how effective global memory latency can be hidden [5], [7]. Occupancy describes the current number of warps available for execution relative to the maximum number of warps that can be resident on the GPU. To increase occupancy, on the one hand enough work must be submitted to the GPU. On the another hand occupancy can also be affected by register and TGSM usage. The more registers per thread and TGSM per thread group is used the less warps can be held by a SM [16]. While register usage cannot be directly controlled by a programmer a more complex program will generally use more registers than a simpler one.

C. GPU Interconnect

High-performance neural network training on large data sets utilizes multi-GPU setups. Fast communication between the components of such systems is crucial. In the following we examine various GPU interconnect technologies used in modern high-performance systems.

Peripheral Component Interconnect Express Bus (PCIe) is traditionally used to connect multiple GPUs to a single CPU. This results in a tree topology with its root lying at the CPU. Each connection allows bidirectional communication through full-duplex lanes. Each PCIe 3.0 lane provides a bandwidth of 1 GB/s and each GPU bundles 16 lanes [10]. Nonetheless PCIe is still much slower than the connection between a CPU and its DRAM. In GPU-accelerated applications this can quickly become a performance bottleneck [9].

To further improve performance, Nvidia DXG-1 and DXG-2 deep learning systems and the supercomputers Summit and Sierra utilize **NVLink**. In addition to connecting GPUs to CPUs it can also be used to connect GPUs with each other. Like PCIe it is a bidirectional interface. NVLink-V2 can transfer data at 50 GB/s per connection. Volta V100 GPUs support up to six NVLink-V2 connections. This results in a total bandwidth of 300 GB/s [6]. In Summit three V100 GPUs and one Power-9 CPU use NVLink-V2 to form a fully connected mesh. All pairs of nodes in this network are connected by two NVLink links [9].

Fast neural network training requires all GPUs in a system to communicate with each other. However, with NVLink only up to 7 GPUs can be fully interconnected. This approach does not scale well, as in a setup with more than 7 some GPUs would need to communicate through the much slower PCIe interface. This is where **NVSwitch** [15] comes into play. It is a NVLink-V2 based switch that contains 18 NVLink ports per switch. In DXG-2 six switches are used to fully interconnect 16 V100 GPUs. Each GPU is connected to all switches. This provides the full NVLink-V2 bandwidth of 50 GB/s between each pair of GPUs.

GPUDirect-RDMA [19] allows third-party devices such as network adapters and SSDs to directly access GPU memory. This reduces CPU overhead as data does not need to be temporally loaded to CPU memory.

III. GPU Accelerated Matrix Multiplication

Before diving deeper into neural networks on GPUs, we compare how much faster a GPU is than a CPU at matrix multiplication, an important component to training neural networks using the gradient descent algorithm.

For a $n \times m$ matrix A and a $n \times p$ matrix B, the matrix product $AB = C$ is a $n \times p$ matrix. The elements c_{ij} for $i \in [1, m], j \in [1, p]$ are defined as:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj} \tag{1}$$

On a CPU C can be calculated by looping over all i and j and computing the sum from (1) for each c_{ij}. Since all elements can be computed independently, all c_{ij} can be processed in parallel. While this parallelization can also be done on a multi-core CPU, it will not scale well for large matrices.

The simplest way to implement this algorithm on a GPU is to have each thread calculate the sum from (1). For large matrices this already results in a speed-up compared to our optimized CPU version. However, in this simple implementation many of the same elements from the matrices A and B will be loaded by multiple threads. This results in more global memory reads than necessary.

This issue can be mitigated by using TGSM. A common way to utilize TGSM in a matrix multiplication is to divide C into *submatrices* and have each thread group compute all elements one submatrix. The algorithm for computing the complete matrix C from submatrices is almost identical to computing it from individual elements. Instead of multiplying and summing elements, submatrices are multiplied using the standard matrix multiplication and summed. The matrix multiplication algorithm that utilizes TGSM consists of the following steps:

1) Initialize all elements in C to zero.
2) Load the first pair of submatrices from A and B required to compute the C-submatrix of the current thread group into TGSM.
3) Wait until all threads in the group finished loading into TGSM.
4) Compute the product of the two submatrices in TGSM.
5) Add the resulting matrix to the C-submatrix of the current thread group.
6) Repeat from step 2 with the next pair of submatrices from A and B until the C-submatrix of the current thread group has been calculated completely.

In our benchmark this optimized version executed more than 20 times faster than the first one. All results are summarized in table I. The times were measured on an i7 2600k running at 4.4 GHz and a GTX 1070. The CPU version is optimized with cache blocking and OpenMP's loop parallelization.

TABLE I
TIME IN MILLISECONDS TO MULTIPLY TWO $n \times n$ MATRICES.

n	CPU[1]	GPU[2]	GPU optimized[2]
512	23	16.22	0.84
1024	227	117.4	4.77
2048	1941	824.12	32.41

[1]Compiled with msvc /O2 optimization.
[2]Includes transfer of data to and from GPU memory via PCIe 2.0.

IV. NEURAL NETWORKS ON GPUS

Neural Network training and inference can be greatly accelerated using GPUs. While there exist a variety of neural networks types, such as recurrent neural networks and spiking neural networks, we will focus on accelerating **feedforward neural networks** with GPUs.

The basic building block of a neural network is an **artificial neuron**. Given input values x_i, weights w_i for $i \in [1, m]$, a **bias** w_0 and an **activation function** g, its **activation** a is computed as:

$$a = g(w_0 + \sum_{k=1}^{m} x_k * w_k). \tag{2}$$

The activation function is usually non-linear to enable non-linear decision boundaries. Common choices for the activation function include the **logistic function**

$$g(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

and the **rectified linear unit** (ReLU)

$$g(x) = max(0, x). \tag{4}$$

A common property of most activation functions is their easy to compute derivative.

A **multi-layer perceptron** (MLP) consists of multiple layers of neurons. The layers between the input and output layers are called **hidden layers**. The activations $\boldsymbol{a}^{l-1}$ of the layer $l-1$ are fed to the inputs of neurons of the next layer l. The weights of layer l can be packed into a matrix W^l. The element w_{ij} of this matrix equals the weight of the activation ${a_j}^{l-1}$ to the i-th neuron of layer l and ${w_{i0}}^l$ denotes its bias. This allows the neuron activations of a layer l with n neurons, given the activations of the previous layer $l-1$ with m neurons, to be computed by the matrix-vector product

$$\boldsymbol{a}^l = g(\begin{pmatrix} {w_{10}}^l & {w_{11}}^l & \dots & {w_{1m}}^l \\ \vdots & \vdots & \ddots & \vdots \\ {w_{n0}}^l & {w_{n1}}^l & \dots & {a_{nm}}^l \end{pmatrix} \cdot \begin{pmatrix} 1 \\ {a_1}^{l-1} \\ \vdots \\ {a_m}^{l-1} \end{pmatrix}) \tag{5}$$

which can be efficiently parallelized on a GPU, as seen in section III.

Training neural networks on massive datasets is considerably more time-intensive than inference. Thus, computational speed becomes even more important. In the following we will see how GPUs can be used to accelerate training with the **gradient descent** algorithm. We will also show how training can be scaled on distributed systems.

A. Gradient Descent on a GPU

The goal of training a neural network is to minimize a given **loss function** by adjusting the network's weights. The loss function $Loss(\boldsymbol{W})$ is a measure of the difference between the output $f(\boldsymbol{x}) = \hat{\boldsymbol{y}}$ of the neural network and the target

output $\boldsymbol{y}$, given an input $\boldsymbol{x}$. A commonly used loss function for regression problems is the **mean squared error** (MSE)

$$Loss_{MSE} = \frac{1}{n}\|\boldsymbol{y} - \hat{\boldsymbol{y}}\|^2, \tag{6}$$

where n is the number of neurons in the output layer. Computing the output of a neural network during training is called **forward propagation**. For each layer $W^l \cdot \boldsymbol{a}^{l-1} = \boldsymbol{z}^l$ needs to be stored as it will be needed later.

The gradient descent algorithm uses the gradient of the loss function w.r.t. all weights in the network to progressively step towards its minimum. The update rule of a single weight w with a **learning rate** α is given by

$$w \leftarrow w - \alpha \cdot \frac{\partial Loss}{\partial w}. \tag{7}$$

Batch gradient descent computes the gradient from the average loss of all training samples. This method is guaranteed to converge to a global minimum if the learning rate sufficiently small. The downside is that it can be very slow to calculate for large data sets.

Stochastic gradient descent (SGD) mitigates this problem by using only a *random subset* of the training set to compute the loss at each step. This subset is often referred to as a **mini-batch**. While this is not guaranteed to converge, it is generally faster than batch gradient descent. Forward propagation can be performed in parallel for all mini-batch samples by turning (5) into a matrix-matrix product

$$A^l = g(W^l \cdot A^{l-1}). \tag{8}$$

The input of the network is now a matrix X where each column is one sample from the mini-batch. Similarly, the columns of A^l contain the neuron activations at layer l for each sample in X. The matrix Z^l is defined correspondingly.

In neural networks the gradient of the loss is calculated through **backpropagation** [20], [21]. The algorithm can be derived through repeated application of the **chain rule**.

In the description of the algorithm we will use the **Hadamard product**

$$A \odot B, \tag{9}$$

that is the element-wise product of two matrices A and B. It is trivial to parallelize the Hadamard product on the GPU.

Assuming we performed forward propagation for all samples in a mini-batch in parallel (8), the gradients can be obtained through the following matrix-based algorithm:

1) Compute the error matrix E^r at the output layer r:

$$E^r = L \odot g'(Z^r)$$

where L is a matrix whose q-th column contains

$$\begin{pmatrix} \frac{\partial Loss}{\partial {a_1}^r} \\ \vdots \\ \frac{\partial Loss}{\partial {a_k}^r} \end{pmatrix}$$

for the q-th sample in the mini-batch. k is the number of output neurons.

2) For $l = r - 1$ to 2, backpropagate the error:

$$E^l = ((E^{l+1})^T \cdot W^{l+1}) \odot g'(Z^l)$$

3) Obtain the gradient for the q-th sample through:

$$(\frac{\partial Loss}{\partial w_{ij}^l})_q = a_{jq}^{l-1} \cdot {e_{iq}}^l$$

4) Compute the average of the gradients of the individual samples.

In step 4 it might be efficient to use a **parallel reduction** [22], [23], if the number of samples in the mini-batch is sufficient.

For further reference Nielsen [21] provides a detailed explanation of the backpropagation algorithm.

Wang et al. [28] explain how the tiled matrix multiplication algorithm can be performed on multiple GPUs with a linear speedup by distributing the computation of the output tiles among the GPUs. In their experiments, this accelerated neural network training on four GPUs by a factor of 2.5 compared to a single GPU.

B. Distributed Learning

The next step to further accelerate training is to distribute the calculations across multiple compute nodes.

To perform distributed learning, various types of parallelism can be exploited. In **data parallelism** [25] the same model is replicated on multiple nodes. The current step's mini-batch is then split into multiple sub-batches and each node will use one sub-batch to compute a gradient. At the end of the step all gradients are loaded to a single node and averaged. The model then needs to be updated on all nodes before the next gradients can be computed. This approach is known as **synchronous SGD** (S-SGD). However, the model synchronization quickly becomes a bottleneck. Koliousis et al. [24] propose to use **model averaging** to reduce synchronization.

On the contrary **model parallelism** can be used to split the model and train each part on a separate node. It is also used when the model is too large to fit onto a single device. A simple fully interconnected MLP for example can be split to two nodes by training the first half of the layers on one node and training the second half of the layers on the other node (Fig. 2). In general, finding good model splits for parallelization is a complex task. One approach is to use reinforcement learning to find a split [26]. A disadvantage of this method is the high communication cost between the nodes.

Pipeline parallelism is a combination of data and model parallelism [25]. The model is split, and each partition is trained on a separate node. All nodes together form a pipeline that first forward propagates micro-batches from one node to the next. After the forward propagation is completed, backward propagation can be performed in the same manner.

Laanait et al. [29] showed that efficient scaling of distributed deep learning can be achieved by exploiting only data parallelism. Their key optimization is a gradient reduction strategy that allows for an optimal overlap of computation and communication. By that they were able to realize a scaling

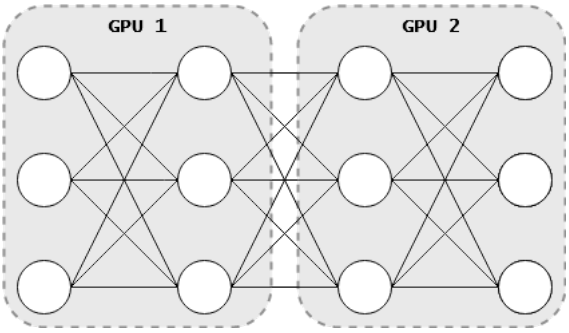

Fig. 2. Model parallelism on a MLP

efficiency of 0.93 at distributed learning on 4600 Summit nodes (27600 GPUs). The model used in the tests was a fully-convolutional dense neural network with 200 million parameters.

V. Conclusion

GPUs can process massive amounts of data in parallel more efficiently than CPUs. It is fundamental to understand the principles of GPUs hardware, especially the memory architecture, to be able to write fast programs. We have seen how a GPU can perform the multiplication of large matrices significantly faster than a CPU. Following that we showed how neural network training can be sped up by expressing the training algorithm through a series of matrix multiplications. Distributed training requires a high-bandwidth interconnect to be efficient. As the processing power of GPUs continues to increase the development of better interconnect technologies will be critical to the performance of future high-performance GPU systems.

References

[1] Mark J. Harris "Fast fluid dynamics simulation on the GPU" in Randima Fernando, GPU Gems, Addison-Wesley, 2004
[2] Franck Diard, "Using the geometry shader for compact and variable-length GPU feedback" in Hubert Nguyen, GPU Gems 3, Addison-Wesley, 2007
[3] Nvidia, "Nvidia CUDA compute unified device architecture: programming guide", 2007.
[4] Veronica G. Vergara Larrea et al., "Scaling the summit: deploying the world's fastest supercomputer", 2019.
[5] Tomas Akenine-Möller et al., "Real-time rendering, fourth edition", CRC Press, 2018, pp. 29-34, 1002-1006
[6] Nvidia, "Nvidia Tesla V100 GPU architecture", 2017
[7] Ashu Rege, "An introduction to modern GPU architecture", 2008
[8] Nvidia, "Whitepaper Nvidia Tesla P100", 2016
[9] Ang Li et al., "Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect", IEEE transactions on parallel and distributed systems, 2019
[10] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl and Volker Markl, "Pump up the volume: processing large data on GPUs with fast interconnects", 2020
[11] LLNL, "Sierra", https://hpc.llnl.gov/hardware/platforms/sierra
[12] Nvidia, "White paper Nvidia DGX-1 system architecture", 2017
[13] Nvidia, "Nvidia DGX-2H the world's most powerful system for the most complex AI challenges", 2018
[14] Denis Foley and John Danskin, "Ultra-performance Pascal GPU and NVLink interconnect", IEEE Micro, 2017
[15] Nvidia, "Technical overview Nvidia NVSwitch", 2018
[16] Mark Harris, "How to Access Global Memory Efficiently in CUDA C/C++ Kernels", https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/, 2013
[17] Andrew Lauritzen, "Future directions for compute-for-graphics", Siggraph, 2017
[18] Mark Harris, "Using shared memory in CUDA C/C++", https://devblogs.nvidia.com/using-shared-memory-cuda-cc/, 2013
[19] Nvidia, "Developing a Linux Kernel Module using GPUDirect RDMA", https://docs.nvidia.com/cuda/gpudirect-rdma/index.html
[20] Stuart Russel, Peter Norvig, "Artificial intelligence: A modern approach, third edition", Pearson, 2014, pp.704-748
[21] Michael Nielsen, "Chapter 2: How the backpropagation algorithm works", http://neuralnetworksanddeeplearning.com/chap2.html, 2015
[22] Mark Harris, "Optimizing parallel reduction in CUDA", 2007
[23] Justin Luitjens, "Faster parallel reductions on kepler", https://devblogs.nvidia.com/faster-parallel-reductions-kepler/, 2014
[24] Alexandros Koliousis et al.,"Crossbow: scaling deep learning with small batch sizes on multi-GPU servers", 2019
[25] Ruben Mayer and Hans-Arno Jacobsen, "Scalable deep learning on distributed infrastructures: challenges, techniques and tools", 2019
[26] Azalia Mirhoseini et al., "A hierarchical model for device placement", 2018
[27] Nvidia, "NVIDIA A100 Tensor Core GPU architecture", 2020
[28] Linnan Wang, Wei Wu,Yi Yang and Jianxiong Xiao, "Large Scale Artificial Neural Network Training Using Multi-GPUs", 2015
[29] Nouamane Laanait et al., "Exascale deep learning for scientific inverse problems", 2019