Accelerating Neural Network Inference on Nvidia GPU

Jianchang Su[*]
[*]Friedrich-Alexander-Universität Erlangen-Nürnberg

***Abstract*—GPUs are well suited to perform inference of ANNs but require extensive optimisation to tap their full potential. Consideration of a GPUs memory hierarchy allows for improvements regarding comutational capacity and memory bandwidth when executing matrix multiplication, the most common operation in ANNs, leading to a speedup of up to 25. Even further significant performance improvements are possible with optimisations affecting accuracy like quantisation and pruning, increasing the performance up to 60 times with mediocre accuracy impact. Alternative algorithms for performing convolutions, that have a lower theoretical runtime complexity, did not show observable performance gain. Our experiments regarding an embedded GPU platform show, that these can compete with desktop GPUs in terms of power efficiency while maintaining a low power budget. The available Domain Specific Architecture for ANN inference disappoints because of its tremendously restrictive model requirements.**

I. Introduction

General Purpose Graphics Processing Units (GPGPUs) provide a vast amount of theoretical computational power due to enormous hardware parallelism, that can be utilised for running inference of Artificial Neural Networks (ANNs). Nevertheless, this computational power does not come for free but requires a variety of optimisation techniques to reveal its full potential. In this report we investigate a multitude of optimisation strategies suitable for improving the most important operations performed by ANNs. Optimisation is especially important if the theoretical computational power of the device is rather low, so that every quantum of potential can be used, which is the case with the Jetson Xavier NX, an embedded GPU platform. In this report we investigate how much performance gain we can reach for ANNs on GPU based devices.

II. Background

A. SGEMM Optimisation Strategy

The major operations in Convolutional Neural Networks (CNNs), which are most modern ANNs, consist of matrix multiplications in both the Convolutional and the Fully-Connected Layers (FCLs).

We refer to those muliplications as Single Precision General Matrix Multiplications (SGEMM) [1] and investigate other levels of precision in Subsec. III-E. Here we are only interested in optimisations, that conserve accuracy, and we choose single precision as it is usually the native precision for weights in ANNs.

Before implementing the SGEMM optimizations, we review the GPU memory hierarchy, which is a central condition for effective improvements. The memories exposed by the GPU architecture are registers, L1 cache or Shared memory (SMEM), read-only memory, L2 cache and global memory. Like the CPU, it is important for efficient computing on the GPU to exploit the access speed of the individual pieces of memory and use caching effects. The memory hierarchy is visualised in Fig. 1.

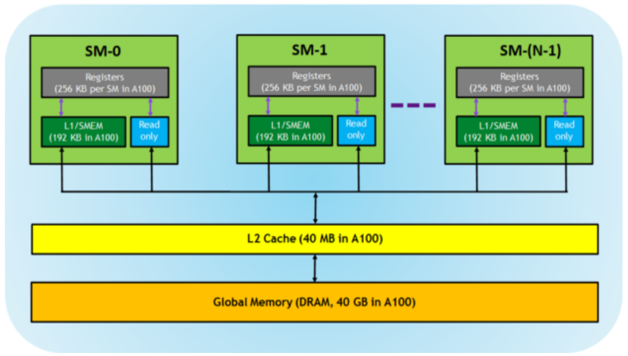

Fig. 1: NVIDIA A100 Memory Hierarchy [2]. Exploiting this architecture provides great optimisation potential.

In general, optimizing SGEMM on GPU and CPU platforms share the same idea: We hide the memory latency with massive parallelism, re-use data on the cache and register level and manually prefetch data [3]. The implementation of these concepts are further discussed in III-A.

B. Convolution Algorithms

As already mentioned are convolutional layers some of the most fundamental components of modern ANNs. Not only do convolutional layers play an important role as trainable feature extractors since the emerge of CNNs [4], but even further as a replacement for FCLs in 1×1 or $n \times n$ kernel configurations. While it is obvious how the outcome of a convolution has to look like, the convolution itself can be implemented using different algorithms producing the same result. Here we mention four of them, the naive procedure, GEMM, the Coppersmith-Winograd-Algorithm and FFT-based convolution, each with individual properties regarding complexity and runtime behaviour.

1) Naive Procedure: With the naive procedure we refer to the intuitive way of computing a convolution as with pen and paper, possibly implemented employing nested for-loops on a single core processor. Although this iterative approach can benefit from optimisation quite decently, it does not utilise the main features of GPUs, i.e. their immense parallelisation, thus we do not discuss this method in more detail here. For reference, the runtime complexity is $O\left(N^3\right)$, where N denotes the side length of a square matrix (W.l.o.g. all matrices are assumed to have square shape).

2) GEMM: General Matrix Multiplication (GEMM) is the most well-established implementation for performing convolutions, although it consumes more memory with redundant information. After rewriting convolutions into blatantly parallelisable matrix multiplications, it inherently takes benefit from a GPUs structure, while still showing high potential for finetuning and optimisation. We elaborate this further in Subsec. III-A. The amount of instructions necessary for computing one matrix multiplication is drastically reduced compared to iterative convolution, specifically using libraries that deploy the operations to a GPU device in a superscalar manner or by execution on dedicated arithmetic units for matrices, as they can be found in Deep Learning Accelerators (DLAs). In Subsec. II-C we discuss these units in more detail and present our investigations and analysis in III-D. While the parallelisation leads to a runtime, that is a small fraction of the naive method, this constant factor is ommitted in runtime complexity analysis, remaining at a scaling of $O\left(N^3\right)$.

3) Coppersmith-Winograd-Algorithm: The Coppersmith-Winograd algorithm, named after its developers, implements a non-standard matrix multiplication and is known as one of the fastest algorithms for computing matrix multiplication. By reordering and reusing precomputed values it can asymptotically reduce the runtime complexity of GEMM in the polynomial exponent from $O\left(N^3\right)$ to $O\left(N^{2.376}\right)$ [5]. It realises a valuable trade-off, lowering the computationally expensive multiplications by increasing the necessary additions [6]. We show a comparison for this in Subsec. III-B.

4) FFT-based Convolution: FFT-based convolution has its foundation in a central theorem hailing from signal processing, the convolution theorem. It claims, that in certain circumstances, convolution in one domain equals multiplication in another domain. The domain in general states the representation of a signal, e.g. a signal in the time domain is represented as its amplitude over the time, whilst the frequency domain takes the amplitude over the frequency as its basis.

This connection can be exploited for calculating the convolutions in an ANN: Take the input signal and the trained filters and transform them to the frequency domain, multiply them pointwise and transform the result back to the time domain. The multiplication itself consumes linear runtime, the standard algorithm for computing the coefficients of a Discrete Fourier Transform (DFT) to the frequency domain as shown in Eq. 1

$$\hat{a}_k = \sum_{j=0}^{N-1} \mathrm{e}^{-2\pi \mathrm{i} \cdot \frac{jk}{N}} \cdot a_j \qquad \text{for } k = 0, \dots, N-1 \tag{1}$$

obviously runs quadratic in the total number of elements, resulting in $O\left(N^4\right)$ as N denotes just one side length of a matrix. Using the Fast-Fourier-Transform (FFT), which is a fast implementation of the DFT for sizes that are powers of two, the runtime complexity of the transform can be reduced to $\mathcal{O}(N_{total} \cdot \log(N_{total}))$. Together with the necessary but asymptotically irrelevant multiplication we receive Eq. 2

$$\begin{aligned} \mathcal{O}(CONV) &= \mathcal{O}(N_{total} \cdot \log(N_{total}) + N_{total}) \\ &= \mathcal{O}(N^2 \cdot \log(N^2) + N^2) \\ &= \mathcal{O}(N^2 \cdot \log(N^2)) \\ &= \mathcal{O}(N^2 \cdot \log(N)) \end{aligned} \tag{2}$$

which is asymptotically faster than any polynomial with exponent truly greater than two - the greatest lower bound for the runtime of GEMM.

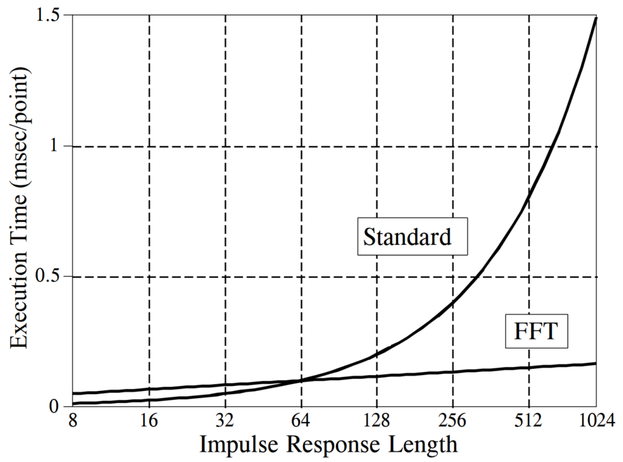

Fig. 2: Performance of FFT-based compared to naive convolution for increasing filter size [7]. Although this experiment is quite old, the message stays the same, FFT-based convolution provides a large asymptotic speedup.

Unfortunately the implementation of a FFT is a difficult matter, as it requires for instance a complex number representation, difficult padding to avoid circular convolution and additional complexity for implementing strides and input paddings. We therefore refer to Fig. 2, where FFT-based convolution is compared to the naive method. The experiments in this plot are run on a 100 MHz processor, which is of course not contemporary anymore, but the gist is the same, FFT-based convolutions provide a significant asymptotic speedup.

C. Jetson Xavier NX

Most applications for ANNs with real-time requirements involve classical pattern recognition tasks in smart home applications and mobility in an human-computer interaction based embedded environment like speech or sign recognition, e.g. in Fig. 3. Ensuring broad availability of these services in consumer products for various circumstances demands low-power low-cost devices with sufficient capabilities.

For evaluating the applicability of embedded GPU hardware for deep learning inference we decided to run our benchmarks on a Nvidia Jetson Xavier NX [10]. The Jetson is a System on a Module (SoM) single board computer comparable to devices like the Raspberry Pi, that has a Thermal Design

Fig. 3: Sign recognition in a car by Tesla. ANNs are running on an embedded platform to provide features for the autopilot [8].

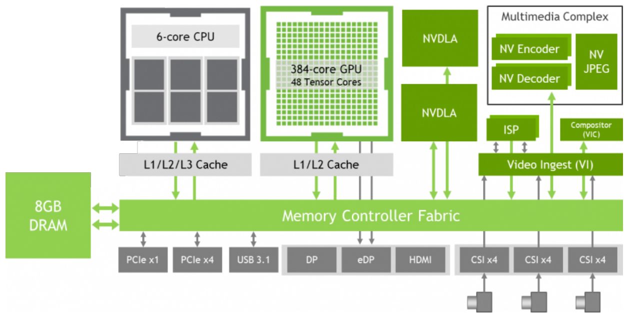

Fig. 4: Block diagram of the Jetson Xavier NX' computer architecture [9]. It computes with an ARMv8 CPU, a Nvidia Volta GPU and two Deep Learning Accelerators (NVDLA).

Power (TDP) of 10 to 20 Watts. The SoC on the Jetson Xavier NX comes from the Nvidia Tegra family as shown in Fig. 4, which are also used in mobile devices like the Nintendo Switch or Nvidia Shield and cars from Audi or Tesla. It consists of an ARMv8 6-core CPU, a Nvidia Volta GPU with 384 Streaming Processors (SPs) and 48 mixed precision "Tensor" SPs, two Nvidia Deep Learning Accelerators (NVDLAs), 8GB of DRAM, some coders and periphery. The CPU is not meant for significant computational effort, it serves as a control unit for the distribution of data and instructions to the other units located on the SoC instead. The GPU is the main component designated to performing heavily parallelised instructions on a large amount of data. Since the GPU is a general purpose device it is not restricted to just perform inference of ANNs, in contrast to the NVDLAs. These are DSA, that consist of building blocks designed for executing the most common operations required for running neural nets, as depicted in Fig. 5. By design, the NVDLAs should provide a high throughput for inference of ANNs. We could not prove this in our experiments, that can be reviewed in Subsec. III-D.

III. Experiment

A. SGEMM Optimisation

For SGEMM, we use a column-major implementation as in Fortran indexing style.

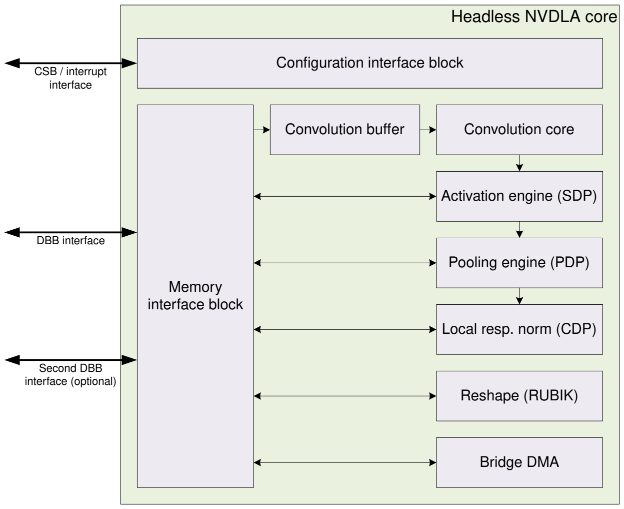

Fig. 5: Block diagram of a NVDLA core [11]. It contains pipelined building blocks for the most popularly combined layers used in modern ANNs like ResNet.

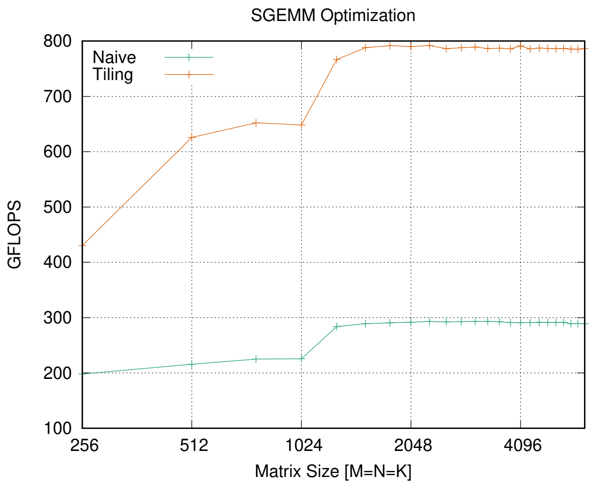

Fig. 6: Performance of the naive GEMM compared to the Tiling optimisation. Tiling offers a speedup of 2 to 3.

1) Tiling: We apply tiling with dimensions `{Ms,Ns,Ks}={32,32,32}`, that partitions the matrices `A` and `B` into blocks of 32×32. Before being loaded for calculation of GEMM, these blocks are first put into the shared memory. Each thread is responsible for loading and storing one matrix element when putting data into the shared memory, which is also known as packing in GEMM for CPUs. We set 1024 threads per Thread Batch (TB) using `__launch_bounds__(1024)`. As soon as all threads have been synchronized, they may begin computing for their respective element. Because each TB has to calculate a 32×32 matrix `C`, every thread must still accept a single `C` element.

The result of applying Tiling is visualised in Fig. 6. The improvement compared to the naive method lies between 2 and 3 times the performance for all probed batch sizes.

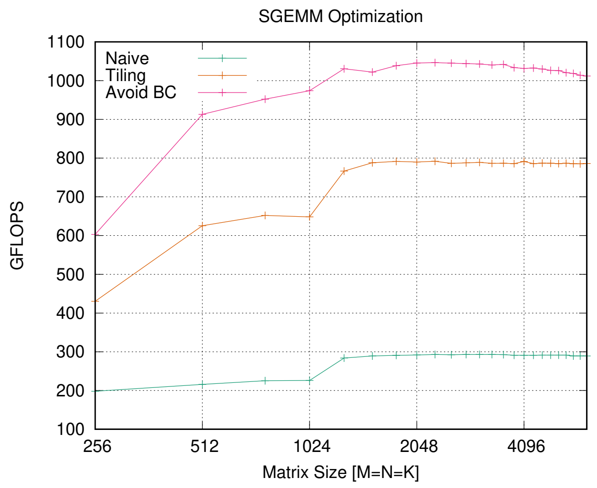

Fig. 7: Performance impact of reducing memory bank conflicts. The optimisation provides an improvement by half additional to Tiling.

2) *Reducing Memory Bank Conflicts:* In the shared memory we reorder the memory access pattern by making all stored data matrices column-major but transposing matrix `B` when packing it in.

Fig. 7 shows the impact of the optimisation, which raises the performance approximately by half.

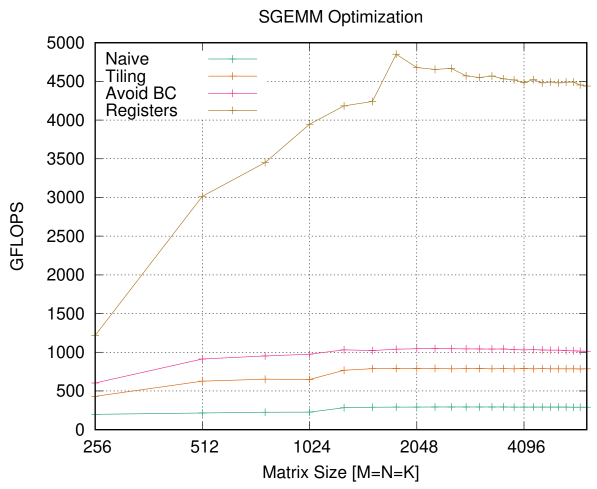

Fig. 8: Effect of Register Buffering to the Performance of executing GEMM. The speedup increases with higher batch sizes from 2 to 4 additional to all previous optimisations.

3) *Register Buffering:* Given that each TB has enough registers (64K), and we only give 256 threads to each TB, it should be safe to assign more tasks to each thread in terms of speed. In comparison to the previous phase, we now ask each thread to calculate a 4×4 sub-matrix of `C`, resulting in tremendous data re-use at the register level.

We can also raise `Ms` and `Ns`, while maintaining enough TBs for mapping to the streaming multiprocessors, when the input matrices are large. To retain the same shared memory usage, we raise `{Ms,Ns}` from the previous `{32,32}` to `{64,64}`, but lower `Ks` from `32` to `16`. Because everything says identical except the two parameters `{Ms,Ns}`, we conclude that requesting TBs to complete more jobs improves performance when the input matrices are large enough.

This optimisation has the greatest influence on the performance gain for GEMM. It provides an speedup from 2 for low batch sizes up to more than 4 for large batches, observable in Fig. 8.

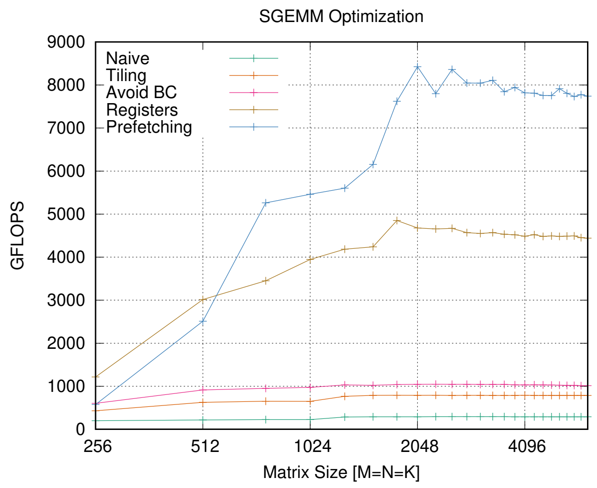

Fig. 9: Performance gain by prefetching data. For lower batch sizes prefetching even worsens the computational power, whereas larger batches profit with a speedup of about 2.

4) *Prefetching:* When performing memory prefetching we load not only the currently necessary piece of data from the memory, but also the next three entries. This access can be realised with a burst memory access, that does not demand significantly more time than access to a single peace of information. The big advantage is, that in the next cycles the demanded data is already loaded, so that the previously necessary overhead for memory access is ommitted.

The effect of prefetching is visible in Fig. 9. For smaller batches this adjustment even reduces the performance slightly, probably because the overhead for afterwards not needed prefeched memory is to costly. With higher batch sizes the advantage of about double the performance is clearly visible.

5) *CuBLAS:* For reference, we use the the Nvidia cuBlas library for GEMM, which provides optimised algorithms for basic linear algebra.

The result of our optimisations are visualised in Fig. 10. Each measure improves the performance, most significantly when applying the register level optimisation. The best performance we achieve is 8.1 TFLOPS, while Nvidia cuBlas' best performance lies slightly above at 8.73 TFLOPS, as the library has surely been finetuned a lot during development. The possible peak performance of a RTX 2080 amounts to 10.07 TFLOPS, which we come close to but do not reach. GEMM

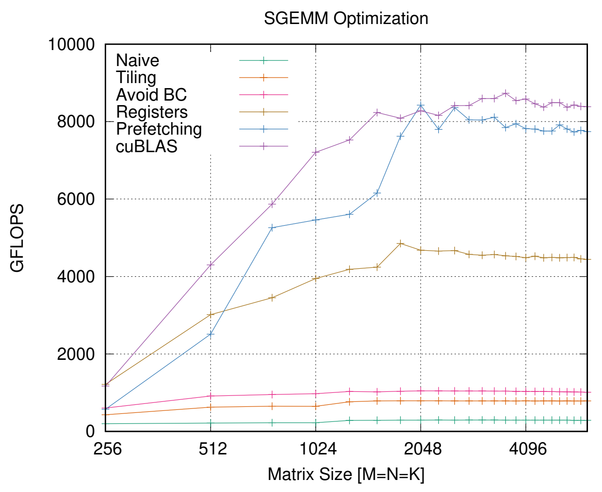

Fig. 10: Performance of SGEMM using various optimisations.

can probably not exploit all resources of the RTX' GPU, other tasks might have a genuinely higher hardware utilisation.

B. Convolution Optimisation

We discuss the optimisation of GEMM in Subsec. III-A. Here we compare the performance of the Coppersmith-Winograd-Algorithm to GEMM in terms of runtime.

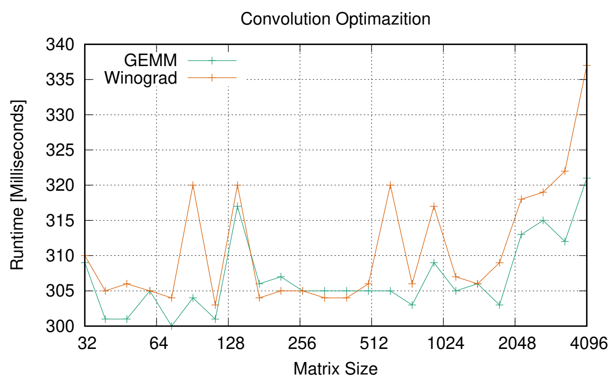

Fig. 11: Inference runtime of the Coppersmith-Winograd-Algorithm compared to GEMM. Although the Coppersmith-Winograd-Algorithm has a higher asymptotic performance in theory, GEMM achieves slightly better results due to high optimisation potential.

Fig. 11 presents the runtime of convolution for the Coppersmith-Winograd-Algorithm and the implicit GEMM performed by CuBLAS. We discuss the background for both algorithms in Subsec. II-B. Although the Coppersmith-Winograd-Algorithm provides a lower asymptotical runtime complexity in theory, it is slightly outperformed by GEMM. The used GEMM implementation from the cuBlas library is called `IMPLICIT_PRECOMP_GEMM`, requires a small amount of workspace for optimisation and achieves a significantly higher performance than the standard approach [12]. Our suggestion, why GEMM has a narrow advantage over the Coppersmith-Winograd-Algorithm, is the high potential for optimisation possible for GEMM as we show in Subsec. III-A. The reordering of operations in the Coppersmith-Winograd-Algorithm, that heavily reduces the amount of necessary multiplications and makes it faster in general, probably leads to a bad data organisation. When performing GEMM, we profit from the shape of the matrices and their organisation in the memory in terms of caching and prefetching, and can easily distribute the operations to the SPs on the GPGPU. With the precomputed and reused values in the Coppersmith-Winograd-Algorithm, bad cache effects might be a consequence and burst memory accesses might be less effectively used. This could explain the lower performance during our benchmarks.

C. CNN Benchmarks

We implemented the most important layers occuring in modern ANNs using CUDA: FCLs, Convolutional layers, Rectified Linear Unit (ReLU) Activation Functions, Batch Normalisation and Pooling layers. Those layers were tested and benchmarked on various available devices designated to multiple purposes, including desktop GPGPUs (RTX 2080 and GTX 1050 Ti) and an embedded platform (Jetson Xavier NX). We compare the throughput and power consumption running a classification task with a ResNet-18 architecture on the CIFAR-10 dataset.

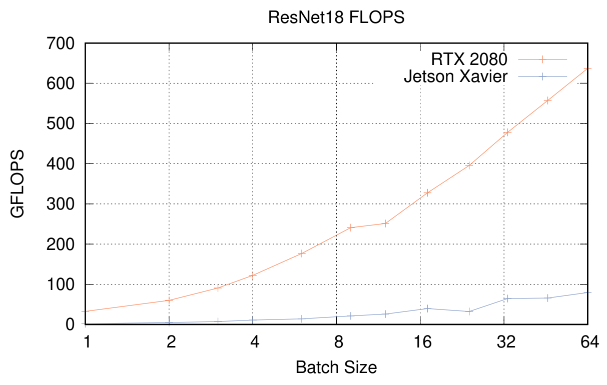

Fig. 12: Performance of RTX 2080 and Jetson Xavier NX.

1) Performance Measurement: Fig. 12 shows the performance of the RTX 2080 and the Jetson Xavier NX in GFLOPS depending on the provided batchsize. While the RTX 2080 has a higher overall performance, it also shows larger and more constant scaling with higher batchsizes, when the Jetson Xavier NX' performance already runs into saturation. This effect probably hails from the RTX' higher amount of execution units. While the RTX can still provide shader cores for running larger batchsizes in parallel, the Jetson runs out of hardware ressources and has to serialise the execution, thus the performance gain stagnates.

2) Power Measurement: The costs for high computational power become visible in Fig. 13. The average power budget of the RTX reaches from 80 to 90 Watts, while the Jetsons

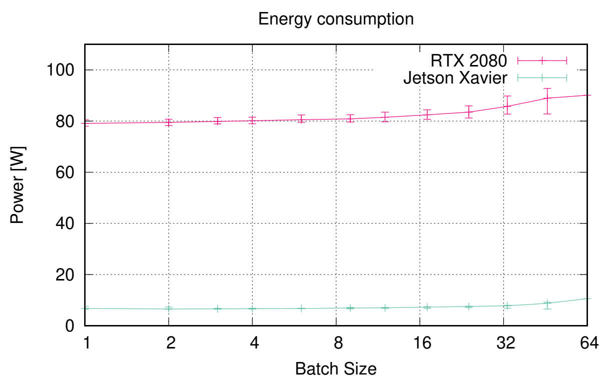

Fig. 13: Energy consumption of RTX 2080 and Jetson Xavier NX. Both devices demand (slightly) more additional power with increasing batch size, while the Jetson as an embedded system draws significantly less power overall.

mean energy consumption lies underneath the 15 Watts mark. The Thermal Design Power (TDP) of the RTX is 225 Watts, what indicates that its computational power is not exhausted with the provided maximum batch size. The Jetson on the other hand has a TDP of 10 to 20 Watts, which is almost reached with a peak power consumption of about 18 Watts, much likely constituting another reason for the stagnation in performance at higher batch sizes.

The provided values have to be handled with care, as they originate from software based power measurement due to the lack of a possibility of hardware measurement using a circuit analyser. The RTX can be profiled using tools by Nvidia, the Jetson however does not provide utility for this purpose, therefore the power values have to be read out directly from the memory mapped sensors. The gained values tend to be lower than the actual power consumption, but they still serve as a rule for an appropriate comparison.

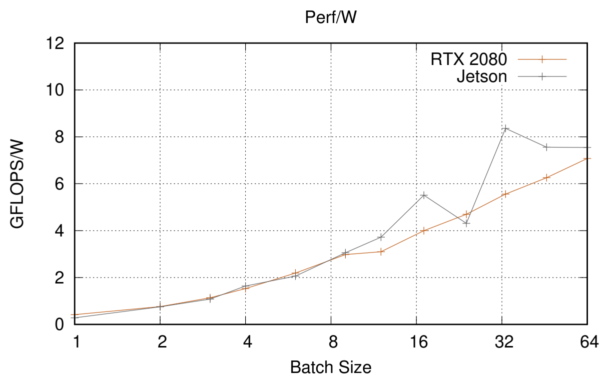

Fig. 14: Power efficiency of RTX 2080 and Jetson Xavier NX.

3) Efficiency: The efficiency of the RTX 2080 and the Jetson Xavier NX in GFLOPS of performance per Watt of power consumption is shown in Fig. 14. Although the line of the Jetson fluctuates rather strong, it can be assumed to have a comparable or even better power efficiency than the RTX 2080. This is somehow remarkable; The device with the overall lower power consumption still provides a competitive efficiency. This is interesting especially for mobile applications that possibly use the Jetson Xavier NX for inference of ANNs. The point is, that usually no vast computational power is required, but the potentially provided energy is bound to strict and rather low limits. For instance in automotive applications that rely on combustion engines, all electrical devices have to be supplied by the lighting dynamo or recuperation of brake energy. In this context it is quite astonishing, that a small device like the Jetson Xavier NX, that is able to meet strict demands concerning power limits, can still compete with larger devices in terms of power efficiency.

D. Inference on NVDLAs

Our experiments concearning the Jetson Xavier NX also included the NVDLAs on the SoC. The access to the NVDLA cores is capped, making it necessary to use Nvidias TensorRT library to run inference on the NVDLAs. TensorRT is a runtime environment for loading, building, optimising and running ANNs on Nvidia devices. For measuring the performance of the NVDLAs, we deploy a ResNet-18 model to TensorRT running on the Jetson Xavier NX and compare the performance between GPU (CUDA) and NVDLA (DLA).

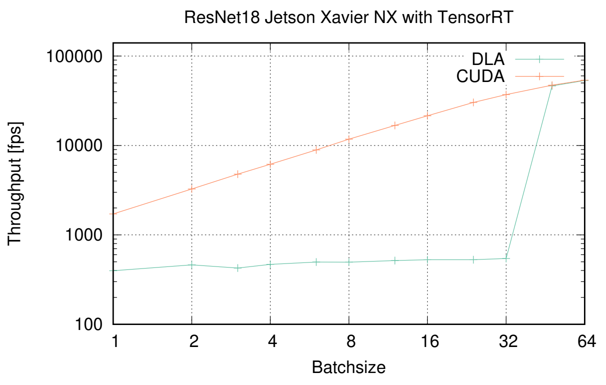

Fig. 15: Performance of DLA compared to GPU running ResNet-18 with TensorRT. The DLA cannot execute all necessary layers, causing the poor performance. Batchsizes larger than 32 are not supported by the DLA, the ANN runs on the GPU instead.

Fig. 15 depicts the throughput in Hz of the ResNet-18 over the batch size of the provided input on a logarithmic scale. The performance of running inference on the GPU develops as expected, the throughput increases sublinearly but continuously with larger batch sizes. The NVDLA on the other hand delivers rather disaponting results, since the throughput is significantly smaller than for the GPU and its scaling lies far underneath the improvements of the GPU with larger batch sizes until batch size 32.

An Explanation for this behaviour is the DLAs lack of support for certain layers of ANNs, in this case the Global

Average Pooling (GAP) layer, that is part of the ResNet-18 architecture. Unfortunately, although the DLA contains a building block entitled as pooling engine, it cannot execute GAP, requiring a GPU fallback. This means, that the respective unsupported instruction needs to be executed on the GPU instead, or the inference fails. Execution on the GPU demands for two memory transfers forth and back, leading to a huge overhead and eventually decreasing the measured performance significantly.

At batch size 48 the performance then jumps up to the values for the GPU. The reason for that is not a drastic performance gain of the DLA but its hardware-limited incapability of performing operations with batch size larger than 32. At this point all operations have to fall back on the GPU, making the memory transfer between DLA and GPU gratuitous and reaching the GPUs performance, as this is the unit that runs the inference now.

It is unfortunate to not be able to observe the real performance of the DLAs. We have tested inference on the DLA with other model architectures like AlexNet-3, LeNet-5, ResNet-50 and some architectures for MNIST, but none of them only uses layers supported by the DLA. But this circumstance just underlines the central statement of our investigations: Vast theoretical computational power of hardware is pointless, if the power cannot be utilised, which requires optimisation, or as in the case with the DLAs, model adjustment to supported layers.

E. Quantisation

We evaluate how quantisation effects the performance and accuracy of running inference on GPUs.

1) Concept: With quantisation we refer to the amount of bits we use to represent one piece of information in an ANN, for instance a trained weight. The less bits are spent on the representation of numbers, the higher is the expected performance, since operations on smaller data representations can usually be computed in less clock cycles. Furthermore, in a superscalar fashion, the Streaming Multiprocessors (SMs) of modern GPUs can contain more Arithmetic Units (AUs) the simpler its operations are, resulting in a higher Instruction Level Parallelism (ILP) for lower quantised data and therefore additionally increased performance. Nevertheless, this comes with a tradeoff as it impacts the accuracy of an ANNs predictions, which has to be kept in mind.

2) Performance Measurement: We test the performance gain of quantisation with single precision FP32, half precision FP16 and INT8 on a Nvidia GTX 1050Ti and the Jetson Xavier NX.

In Fig. 16 we run inference of a ResNet-18 model over the CIFAR-10 dataset on a GTX 1050Ti using three levels of quantisation. ANNs are usually trained using 32 bit single precision floating point numbers, thus this quantisation level represents the original precision. Reducing the amount of bits to 16 per floating point value increases the throughput with a batch size of 128 to approximately 3.5 times its original value. Further decreasing the number of bits to an 8 bit integer

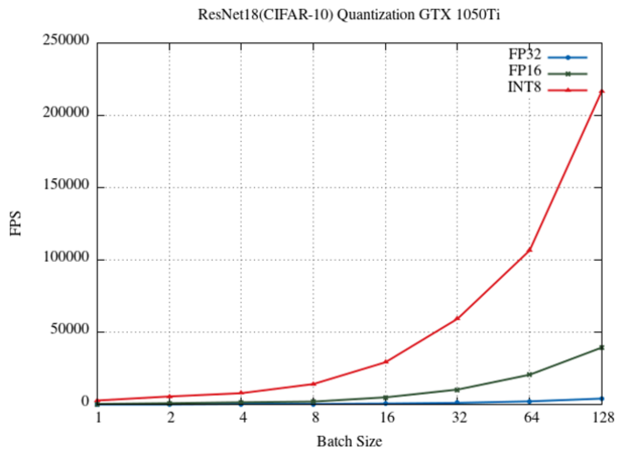

Fig. 16: Performance of GTX 1050Ti with three different quantisation levels. Using less bits for a number representation significantly increases the throughput.

representation elevates the performance to more than 31 times of its single precision value.

Experiments with the Jetson Xavier NX verify this picture, although the performance fluctuates more intensely, visible in Fig. 17. The CPU of the Jetson has rather poor performance, making it more sensible to external system conditions. A resulting irregular deploy of inference tasks could cause troublesome latency.

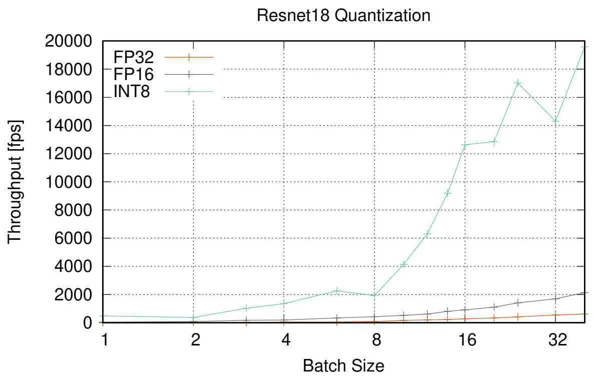

Fig. 17: Performance of Jetson Xavier NX with three different quantisation levels. Using less bits for a number representation increases the throughput.

3) Accuracy Impact: Tab. I shows, that the effect of quantisation on the accuracy of the ANNs prediction is quite limited. Especially the performance gain for reduction of precision to FP16 comes with almost no additional incertitude, so that we recommend this optimisation in most circumstances. If accuracy is not crucial, decreasing to INT8 is an even more valuable improvement.

F. Pruning

Getting faster and smaller networks is important for running these networks on mobile devices, thus we apply pruning to

Precision	Accuracy
FP32	92.1%
FP16	91.8%
INT8	90.2%

TABLE I: Accuracy of ResNet-18 using three levels of quantisation. The deterioration for FP16 is barely measurable and for INT8 still very low, making both a good tradeoff.

the network in order to set a reasonable amount of weights to zero.

Algorithm 1 Pruning weights of ANNs

```
$n \leftarrow norm_{L1/L2}(weights)$
$threshold \leftarrow kthvalue(n)$
**if** $n < threshold$ **then**
    Remove the neuron
**end if**
```

The ranking can be done according to the L1 or L2 norm of neuron weights, their mean activations, the number of times a neuron was not zero on some validation set, and other specialised methods. We perform pruning according to Alg. 1, the according code snippet can be found in List. 1.

```python
def apply(self.weights, amount=0.0, round_to=1):
  if amount <= 0:
    return []
  n = len(weights)

  l1_norm = torch.norm(weights.view(n, -1), p=self.↙
     ↳ p, dim=1)

  n_to_prune = int(amount*n) if amount < 1.0 else ↙
     ↳ amount
  n_to_prune = round_pruning_amount(n, n_to_prune, ↙
     ↳ round_to)
  if n_to_prune == 0:
    return []
  threshold = torch.kthvalue(l1_norm,k=n_to_prune).↙
     ↳ values
  indices = torch.nonzero(l1_norm <= threshold).↙
     ↳ view(-1).tolist()
   #return index
  return indices
```

Listing 1: Python code for the pruning step

If too much pruning happens at once, that means to many weights are set to zero, the network might be damaged severely, so that it is not able to recover.

In practice pruning is an sequential process often referred to Iterative Pruning: A network is pruned, then trained with the pruned weights and this is repeated until a desired speedup is reached.

An Overview on our results can be found in Tab. II.

Fig. 18 visualises the performance gain per iteration of pruning. The performance increases continuously with more rounds of pruning. Both L1 or L2 pruning have no advantage over each other.

The effect on the accuracy of the ANN can be seen in Fig. 19. Until iteration 4 to 5, the loss in accuracy is still reasonable. Afterwards the accuracy decreases rapidly.

Round	Params[K]	Reduced[%]	L1 Acc.[%]	L2 Acc.[%]
0	11181.642	0	92.2	92.2
1	4499.885	59.75	91.94	92.36
2	1904.148	82.97	91.96	92.11
3	850.709	92.39	91.20	91.48
4	408.764	96.34	91.03	90.87
5	210.707	98.11	90.35	90.02
6	118.955	98.93	88.68	88.59
7	70.37	99.37	85.82	86.20
8	46.003	99.58	81.70	82.19
9	32.347	99.71	78.83	79.16
10	23.606	99.78	77.14	77.02
11	17.706	99.84	74.46	74.71

TABLE II: Results of iterative pruning. Until round 4 the speedup and reduction of network size has almost no negative impact on the accuracy.

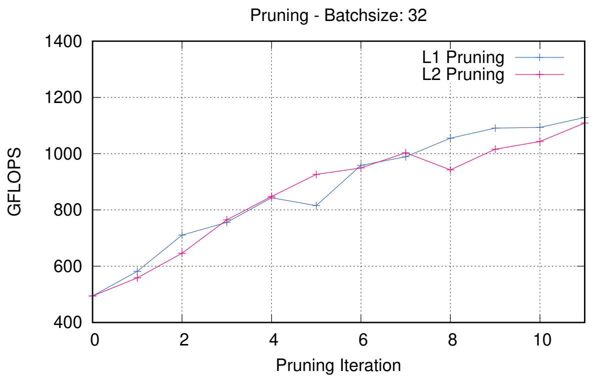

Fig. 18: Performance of a pruned network. L1 and L2 norm both provide a similar improvement.

As a trade-off pruning until iteration 4 to 5 is supposed to be the best compromise. Accuracy does not decreased significantly but the performance doubles.

IV. Conclusion

GPGPUs provide a high level of computational performance, whilst requiring lots of optimisation in order to efficiently use their hardware properties. This can be achieved by trying to use the full comutational capacity, i.e. hardware parallelism, and memory bandwith, e.g. by supporting cache effects. Optimisations affecting the accuracy like quantisation and pruning are excellent choices for further improvement if accuracy is not crucial.

Frameworks, libraries and runtime environments provided by manufacturers like Nvidia exploit the hardware architecture to a remarkable degree, accelerating the inference of ANNs significantly.

Providing a previously optimised environment for running ANNs is crucial for the further rapid development of machine learning on GPUs. This can potentially happen by employing optimised libraries or DSA like DLAs. Unfortunately we could not observe Nvidias NVDLA cores to improve the chips performance at all, due to their lack in flexibility concerning their limited variety of implementable layers.

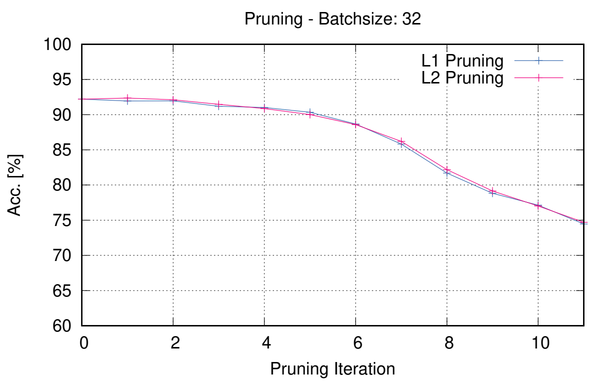

Fig. 19: Accuracy of a pruned network. Until iteration 4 to 5 the loss is reasonable.

For this reason the results from our experiments with the DLAs send an imporant message: It is not enough to simply provide DSA for potentially fast inference of ANNs without further support, especially when working with DLAs it is even more crucial to know the limitations of the specific hardware.

As the more robust solution, we suggest optimised implementations of ANNs on general-purpose shader cores as the more suitible and future-proof solution.

V. Ethical AI

It is common practice in publications concerning AI to subsume with a brief discussion of ethical issues resulting from AI based systems. Although we do not provide any new software architecture for DL, the topic of this report consists of evaluation and improvement of DL execution hardware. Thus, in order to show we are aware of social risks and challenges, we would like to remind of the hazards for society that result out of global availability of AI.

Especially with low power hardware like the Jetson Xavier NX implementing DL based pattern recognition is possible everywhere. This allows risky new possibilities, e.g. for surveillance. Conceivable would be public cameras, that automatically evaluate the faces of people. Being used positively, this could of course allow for better law enforcement, but might also have side effects, e.g. classification of people evaluating how likely it is to commit crime, leading to racial profiling and discrimination. In countries with a decent democratic structure this might not be a big problem, but in other nations under totalitarian regimes, where people already suffer under political or religious persecution, this could lead to a further deterioration of circumstances.

Broad availability of AI does not come without risk.

References

[1] OpenGenus, “Sgemm,” 2022. [Online]. Available: https://iq.opengenus.org/sgemm/

[2] P. Gupta, “Cuda refresher: The cuda programming model,” June 2020. [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

[3] Z. Ye, “Matrix multiplication in cuda,” 2014/2015. [Online]. Available: https://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda

[4] Y. Bengio and Y. Lecun, “Convolutional networks for images, speech, and time-series,” November 1997.

[5] D. Coppersmith and S. Winograd, “Matrix multiplication via arithmetic progressions,” *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251–280, 1990.

[6] D. Mangla, “Understanding ‘winograd fast convolution’,” July 2019. [Online]. Available: https://medium.com/@dmangla3/understanding-winograd-fast-convolution-a75458744ff

[7] S. W. Smith, *The Scientist and Engineer’s Guide to Digital Signal Processing*. California Technical Publishing, 1997-2011.

[8] A. Karpathy, “Ai for full-self driving.” ScaledML 2020, February 2020. [Online]. Available: https://www.youtube.com/watch?v=hx7BXih7zx8

[9] “The World’s Smallest AI Supercomputer,” NVIDIA, November. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/

[10] Nvidia, “Jetson xavier nx,” 2022. [Online]. Available: https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-xavier-nx/

[11] “Hardware Architectural Specification — NVDLA Documentation,” 2018. [Online]. Available: http://nvdla.org/hw/v1/hwarch.html

[12] M. Gupta, “Cutlass convolution,” 2017 - 2021. [Online]. Available: https://github.com/NVIDIA/cutlass/blob/master/media/docs/-implicit_gemm_convolution.md