GPU GEMM-Kernel Autotuning for scalable machine learners

Johannes Sailer, Christian Frey and Christian Kühnert

Fraunhofer Institute of Optronics, System Technologies and Image Exploitation IOSB, Karlsruhe, Germany

Abstract. Deep learning (DL) is one of the key technologies in the artificial intelligence (AI) domain Deep learning neural networks (DLNN) profit a lot from the overall exponential data growth while on the other hand the computational effort for training and inference strongly increase. Most of the computational time in DLNN is consumed by the convolution step, which is based on a general matrix multiplication (GEMM). In order to accelerate the computational time for DLNN different highly optimized GEMM implementations for Graphic Processing Units (GPUs) have been presented in the last years [1] most of these approaches are GPU hardware specific implementations of the GEMM software kernel and do not incorporate the performance dependency of the training data layout . In order to achieve a maximum performance the parameters of the GEMM algorithm have to be tuned for the different GPU hardware and specific data layout of the training task. In this paper we present a two step autotuning approach for GPU based GEMM algorithms. In the first step the kernel parameter search space is pruned by several performance criteria and afterwards further processed by a modified Simulated Annealing in order to find the best kernel parameter combinations with respect to the GPU hardware and the task specific data layout. Our results were carried out on 160 different input problems with the proposed approach an average speedup against the state of the art implementation from NVIDIA (cuBLAS) from around 12 on a NVIDIA GTX 1080 Ti accelerator card can be achieved.

Keywords: GPU, Matrix Multiplication, Autotuning, automatic gerneration, acceleration, CUDA, BLAS

1 Introduction

1.1 Motivation

Deep learning (DL) is one of the key technologies in the artificial intelligence (AI) domain Deep learning neural networks (DLNN) profit a lot from the overall exponential data growth while on the other hand the computational effort for training and inference strongly increase. Machine learning applications profit a lot from that overall data growth, since the models can be trained more precise.

© The Author(s) 2019
J. Beyerer et al. (Eds.), *Machine Learning for Cyber Physical Systems*, Technologien für die intelligente Automation 9, https://doi.org/10.1007/978-3-662-58485-9_8

However, those algorithms runtime depend heavily on the input data. Most of the computational time in DLNN is consumed by the convolution step, which is based on a general matrix multiplication (GEMM). In order to accelerate the computational time for DLNN different highly optimized GEMM implementations for Graphic Processing Units (GPUs) have been presented in the last years [1]. In order to achieve a high computational throughput, most of these approaches are based on a hardware specific software kernel implementation of the GEMM algorithm. Usually the different hardware dependent kernel parameters are tuned manually, which involves expertise about the specific GPU architecture. Furthermore the performance of the GEMM kernel is strongly affected by the shape of the input data processed different data sizes have a huge impact on the computational runtime of the GEMM kernel due to the different memory layouts of the GPU accelerators.
In order to achieve a maximum performance the parameters of the GEMM algorithm have to be tuned hardware and task specific. In the last years, several autotuning approaches of GEMM kernel parameters have been proposed [2] - the basic idea is to automatically tune a limited number of essential GPU kernel parameters in order to achieve a maximum performance. Usually the approaches do not take into account the size and shape of the given input data, which yields to varying computational runtimes.
The motivation of the presented work is to develop an autotune procedure for GPU based GEMM kernels, which takes into account a comprehensive set of kernel parameters and varying shapes of the data in the input task.
Proposed autotuning solutions such as [2] usually require a lot of computational runtime to find an optimal kernel parameter set. The kernel parameter space e.g. in the MAGMA GEMM kernel [4] is very large and therefore restrictions are made to reduce the search space for the kernel parameters followed by a brute search mechanism. This usually results in high search times for the kernel parameters to be set.

1.2 Related Work

Well known autotuning concepts like the Automated Tuned Linear Algebra Software Project (ATLAS) [5] or the Optimized Sparse Kernel Interface (OSKI) [6] focus on the optimization of CPU calculations. There are only a few approaches, which introduce concepts for autotuning GPU kernel parameters [7] the approaches focus only on a small number of tuning parameters and therefore the achieved performance cannot be compared reasonable to the proposed approach in this work. In order to achieve optimal performance a comprehensive set of GPU kernel parameters have to be taken into account.
In literature there are several more autotuning approaches such as [8, 9] . While the work presented in [8] focuses on 3D TFT, the approach in [9] focuses on sparse matrices and optimizing the GPU kernel based on a statistical model. The concepts presented in [10] and [11] focus on automatic generating GPU kernel code and autotune over different generated kernels. Since the generated code is not optimized with respect to the underlying GPU architecture, usually

the performance of these concepts is not optimal. The presented work in this contribution is based on the well-known MAGMA GEMM kernel. The software implementation is characterized by an extensive GPU kernel parameter space. The MAGMA GEMM Kernel, has already been investigates in several autotuning approaches [2,12–16]. The original kernel implementation has been described in [12] and a first autotune concept [13]. With the introduction of the NVIDIA Fermi GPU architecture, the kernel implementation has been revised [14] and an autotuning procedure has been presented in [2]. The approach is characterized by a huge search space for the GPU kernel parameters in conjunction with a brute-force parameter search mechanism, which leads to a high computational effort for finding optimal kernel parameters. With respect to small GEMM operations in [15,16] approaches for batched GEMM operations have been presented and [17] describes the utilization of the Magma GEMM kernel in machine learning procedures. The autotuning approach presented in [18] focuses on energy efficiency of the GPU while processing GEMM operations.
Most of the presented state-of-the-art work is based on a brute-force approach for determining the optimal GEMM kernel parameters. This usually yields to a huge parameter search space and therefore most of the approaches use a parameter combination pre-elimination step in order to reduce the computational effort. The different heuristics for reducing the search space can possibly dismiss optimal kernel parameter combinations. With respect to this suppositions, the presented work focuses on defining optimal heuristics to reduce the search space in combination with a Simulated Annealing(SA) procedure to find efficiently optimal performing GEMM kernel parameters.

2 Solution

Optimal GPU kernel parameters strongly rely on the underlying GPU hardware architecture, the memory layout and the input data size different settings lead to different optimal parameter combinations. Therefore the resulting search space for finding the optimal parameter combination can be enormous. Tuning the parameters by hand is impractical, since it has to be redone for every GPU architecture and every set of input data size again. With respect to these suppositions in the following sections we present a two step autotuning approach for GPU-based GEMM algorithms. In the first step the kernel parameter search space is pruned by several heuristic performance criteria, keeping good performing parameter combinations for a set of different use cases. In the second step based on a modified Simulated Annealing (SA) algorithm the remaining parameter sets are further processed in order to find the best kernel parameter combinations with respect to the GPU architecture and task specific data layout.
In the following sections, the proposed autotuning approach is presented in section 2.1 a short overview of the MAGMA GEMM kernel is given, in section 2.2 we explain the developed heuristics for reducing the search space and in section 2.3 the SA approach is introduced.

2.1 Magma GEMM Stencil structure

The developed autotuning approach is based on the well-known MAGMA GEMM kernel. The original kernel implementation has been described in [12] and is characterized by an extensive GPU kernel parameter space. Algorithm 1 shows the pseudo-code of the kernel. The kernel has 11 parameters - two of the kernel parameters are only relevant for calculations in complex number space. Therefore the kernel parameter space is reduced to nine relevant kernel parameters - the parameters are described in the following:

Blocksizes The Blocksizes BLK_M, BLK_N and BLK_K define how many elements a Threadblock will calculate.

Threadblock dimensions The Threadblock dimensions DIM_X and DIM_Y determine the size of the Threadblock, which calculates a block on the result matrix.

Subdimensions The Subdimensions DIM_XA, DIM_XB, DIM_YA and DIM_YB determine how the Shared Memory(SMEM) is filled.

Algorithm 1: GEMM Kernel Algorithm (simplified)

Data: Matrix A [M x K], Matrix B [K x N], Matrix C [M x N], alpha, beta
Result: C=A x B * alpha C + beta * B
load A_t and B_t to SMEM;
for $i \leftarrow 0$ **to** $KstepBLK_K$ **do**
  A_{t+1} and B_{t+1} to regs;
  for $i \leftarrow 0$ **to** BLK_K **do**
    load A_t and B_t to REG;
    C_{temp}=A_t * B_t
  load A_{t+1} and B_{t+1} to SMEM;
C = C_{temp} * alpha + beta

2.2 Reducing search space

To reduce the search time for finding optimal kernel parameter sets in the first step it is necessary to eliminate parameter sets, which with respect to the underlying GPU hardware layout are not possible and possibly lead to an unstable behaviour of the kernel execution. The following parameters are reduced:

prelimitations
We started with reducing the viable threadcounts respectively the threadblock dimensions. The threadblock dimensions(DIM_X, DIM_Y) can only be 8, 16 or 32 resulting in 64, 256, 512 or 1024 threads. The GPU manufacturer NVIDIA recommends using a minimum of 64 threads [20], which is the lower limit we are applying, the upper limit is given by the hardware specification of the GPU. Other configurations will not map onto the GPU hardware.

utilization criteria
The idea behind this approach is to make use of the Latency Hiding Principle of the GPU explained in [21]. Basically when the GPU chip loads data from the off-chip Global Memory (GMEM), it will pause the corresponding warp, which is a bundle of 32 threads. The GPU will schedule another warp, while previous one is waiting. Typically loading data from GMEM takes many hundred GPU cycles so Latency Hiding this is essential for performance. To enable Latency Hiding it is essential GPU kernels keep enough warps available and the GPU can switch between contexts while loading data.
The number of available warps on the GPU is described by the utilization. The utilization is limited by the available SMEM and number of Registers (REG) used by the GPU kernel itself. Based on these resources the upper limit of the achievable utilization can be calculated. The resource consumption and the maximum utilization can be determined by analysing the kernel source code - a similar approach can be found in [2]. Important to note is, that the presented work measures the utilization in Warps per Streaming Multiprocessor (SM). The GPU schedules everything in Warps so this seems to be a reasonable approach. Furthermore we are forcing similar utilization levels of SMEM and REG. This constraint avoids parameter combinations, which heavily utilize one resource while barely utilizing the other one. Parameter combinations, which are heavily limited in utilization due to REG suffer from poor performance as well as those, which are heavily limited through SMEM. Those parameter combinations, which are heavily limited in utilization due to SMEM, are keeping to few entries from the result matrix, for the utilization they achieve. Therefore, data has to be loaded more frequently from GMEM than necessary. Parameter combinations, which are highly restricted with REG, are keeping to less data to read for achieving faster times. Therefore, they have to load and wait more frequently.

efficiency criteria
The presented work introduces a further criteria for finding optimal kernel parameters: The efficiency criteria describes how long a parameter combination can work, until data has to be reloaded from GMEM. The efficiency criteria is calculated based on the kernel source code by the equations given in 1 to 3.

- Equation 1 describes how often data is loaded from SMEM, minus how often data is loaded from GMEM.

- Equation 2 describes how often data is read from SMEM compared to loading data from GMEM.

- Equation 3 describes the size of workload per thread.

Equation 1 and 2 prefer combination with high SMEM consumption. Equation 3 prefers squared fields, which are not proven to be better.

$$\begin{aligned}\text{SMEM Accessdiferenz (SMRW)} = \text{BLK_K} *\\ ((\text{BLK_M} / \text{DIM_X}) + (\text{BLK_N} / \text{DIM_Y})) -\\ \text{BLK_K} / \text{DIM_YA} * \text{BLK_M} / \text{DIM_XA} -\\ \text{BLK_N} / \text{DIM_YB} * \text{BLK_K} / \text{DIM_XB}\end{aligned} \tag{1}$$

$$\begin{aligned}\text{SMEM Reuse (SMR)} = \text{BLK_K} *\\ ((\text{BLK_M} / \text{DIM_X}) + (\text{BLK_N} / \text{DIM_Y})) /\\ (\text{BLK_K} / \text{DIM_YA} * \text{BLK_M} / \text{DIM_XA} +\\ \text{BLK_N} / \text{DIM_YB} * \text{BLK_K} / \text{DIM_XB})\end{aligned} \tag{2}$$

$$\text{Work per Thread (WpT)} = (\text{BLK_M} / \text{DIM_X}) * (\text{BLK_N} / \text{DIM_Y}) \tag{3}$$

Because of the contradictory definition of the efficiency criteria and the utilization criteria, it is not possible to optimize both at once. The efficiency criteria will force contexts, which will reduce the reload operations from GMEM and therefore enforce higher resource consumption. On the other hand, the utilization criteria will favour shorter working times for the contexts by consuming less SMEM and REG resources. The approach of this work is to use those parametrizations for the subsequent SA autotuning step, which forces to achieve the highest efficiency criteria on a specific utilization level. This ensures long living contexts on a specific utilization level with respect to the latency hiding principle from Paragraph 2.2. With respect to these suppositions, the resulting search space reduces to 84 meaningful parameter combinations.

2.3 Simulated Annealing

Simulated annealing (SA) is a probabilistic technique for finding optimal parameter combinations in a given search space - a detailed overview of the concept is given in [22]. For our approach SA is fitting, because of its ability to ignore local minima and converge to the global one. Sorting the search space after different criteria enforces grouping of parameter combinations with similar runtime on similar problems in the search space, resulting in faster convergence of SA. The parameter combinations found in Paragraph 2.1 are sorted according to their achieved utilization on the GPU and processed in the SA step. It should be noted, that other possible criteria for SA could be the blocksizes (BLK_M * BLK_N) or the leading dimension (DIM_X) from Paragraph 2.1.

3 Performance Evaluation

The performance evaluation of the proposed work is based on a NVIDIA Pascal GPU (MSI Geforce GTX 1080 Ti Aero 11G OC) in combination with a Intel

Xeon E5-1620 with 96 GB Memory host system. The operating system is Windows 64 Bit with NVIDIA Driver Version is 390.65 and CUDA 8. To evaluate the performance of the proposed approach different data sets are used - Table 1 gives on overview of the different matrix shapes for evaluation. These matrix shapes have been chosen, because cuBLAS proven to perform very well. An evaluation test consists of three Matrices A,B and C with format M x K, K x N and M x N $\in$ N. Additionally in order to illustrate the flexibility of the proposed approach, several other matrix shapes have been evaluated. The results of the performance evaluation are shown in Figure 1 and Table 2. Figure 1 shows the achieved speedups with respect to the matrix shapes compared to cuBLAS. It can be seen, that the larger N the lower the performance speedup. In the worst case the achieved result of the proposed approach is 1.3 times faster than the highly optimized cuBLAS routine, in the best case the speedup is 187 times faster than cuBLAS.
Table 1 shows a comparison between the best-found solutions with a standard the brute-force approach to the proposed approach based on SA proposed in this work. The speedup for finding optimal kernel parameters with the proposed SA approach is nearly five to six times faster than the standard brute force approach, while the performance loss for GEMM kernel execution is maximum 10%.

Algorithm 2: Procedure for proving performance capability of this work. The algorythm generates examples in the form of three matrices A, B and C with the formats M x K, K x N and M x N $\in$ N. After 152 generated examples the process terminates.

```
for M = 25; M < 1000000; M = M + 25 do
    for K = 25; K < 1000000; K = K + 25 do
        if M * K = 6250000 or 25000000 or between 2000 and 1000
        then
            N = 25;
             Brute-force search space (M,N,K);
             Simulated Annealing (M,N,K);
             N = 0.5 * M;
             Brute-force search space (M,N,K);
             Simulated Annealing (M,N,K);
             N = M;
             Brute-force search space (M,N,K);
             Simulated Annealing (M,N,K);
             N = 5 * M;
             Brute-force search space (M,N,K);
             Simulated Annealing (M,N,K);
```

Matrix Entries			Matrix Number	Matrix Entries			Matrix Number
M	K	N		M	K	N	
10000	25	10000	1	1250	5000	1250	5
10000	200	10000	2	25	1000000	25	6
5000	500	5000	3	4000	50000	25	7
2500	2500	2500	4	25	10000	2000	8

Table 1. Data Matrix sizes for performance evaluation.

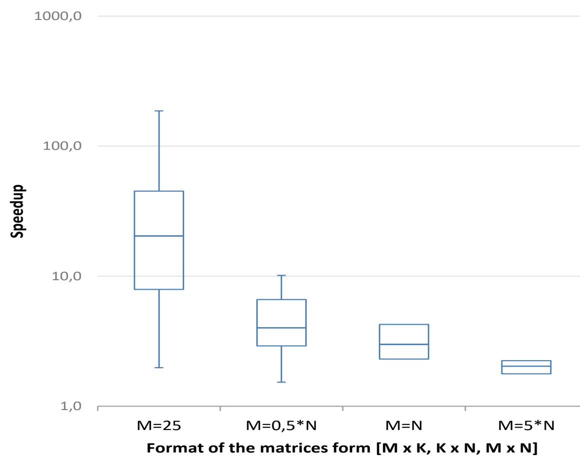

Fig. 1. Comparison of the speedup times against cuBLAS with the brute-force approach on the examples from Algorythm 2. The minimum Speedup was 1,3, the maximum was 187 times as fast as cuBLAS. The average was 12.3 compared to 11.9 in the Simulated Annealing approach. The figure shows, that with an increasing size of N compared to M the speedup reduces. But there was no negative speedup in this test so the results are always faster than the calculation with cuBLAS.

Matrix Format	brute-force Speedup against Simulated Annealing (%)
N=25	10,0
N=M/2	8,4
N=M	3,7
N=5M	4,3
average	5,8

Table 2. Comparison between the best achieved brute-force solution in comparison to the found solution with the Simulated Annealer on examples in the form of three matrices A, B and C with the formats M x K, K x N and M x N $\in$ N

4 Conclusion

The computational throughput of Machine Learning algorithms is limited by the available computational power of the underlying hardware. Most of the computation power in DLNN is consumed by the convolution step, which is based on a general matrix multiplication (GEMM). To accelerate the computational time in Machine Learning applications different highly optimized GEMM implementations for GPUs have been presented in the last years - usually these software libraries have been optimized for a specific GPU version and a specific layout of the data to be processed.
In order to achieve a maximum performance the kernel parameters of the GEMM algorithm have to be tuned hardware and learning task specific. With respect to these suppositions, we have presented a two-step autotuning approach for GPU-based GEMM algorithms: In the first step, the kernel parameter search space is pruned by analysing the kernel source code with several developed performance metrics. In the second step a modified Simulated Annealing algorithm is utilized, which enables a fast searching process for performance optimal kernel parameters, while maintaining search runtimes lower than state of the art brute-force implementations. We have shown that the proposed approach for autotuning MAGMA-GEMM kernels yields high performance and adapts to the GPU hardware and the data layout. Our results have been carried out base on 160 different input problems - we get an average speed up against the state of the art GEMM implementation from NVIDIA (cuBLAS) from around 12 on Pascal based NVIDIA accelerator cards. The key concepts of this contribution can be generalized, to autotune the kernel parameters of other performance sensitive GPU kernels.

5 Acknowledgements

This work was developed in the Fraunhofer Cluster of Excellence "Cognitive Internet Technologies".

References

1. Theano: Deep learning on gpus with python Bergstra, James and Bastien, Frédéric and Breuleux, Olivier and Lamblin, Pascal and Pascanu, Razvan and Delalleau, Olivier and Desjardins, Guillaume and Warde-Farley, David and Goodfellow, Ian and Bergeron, Arnaud and others, NIPS 2011, BigLearning Workshop, Granada, Spain
2. Autotuning GEMMs for Fermi Jakub Kurzak, Stanimire Tomov, Jack Dongarra 2011
3. Fast k nearest neighbour search using GPU Vincent Garcia, Eric Debreuve, Michel Barlaud available at: http://vincentfpgarcia.github.io/kNN-CUDA/ access 13.06.2018
4. Magma project page http://icl.cs.utk.edu/magma/ access 13.06.2018

5. Automated empirical optimization of software and the ATLAS project R. Clint Whaley and Antoine Petitet and Jack J. Dongarra 2001
6. OSKI: A library of automatically tuned sparse matrix kernels Richard Vuduc and James W. Demmel and and Katherine A. Yelick 2005
7. Application-independent Autotuning for GPUs Martin TILLMANN and Thomas KARCHER and Carsten DACHSBACHER and Walter F. TICHY 2013
8. Auto-Tuning 3-D FFT Library for CUDA GPUs Akira Nukada and Satoshi Matsuoka 2009
9. Performance Prediction Based on Statistics of Sparse Matrix-Vector Multiplication on GPUs Ruixing Wang and Tongxiang Gu and Ming Li 2017
10. Automatic C-to-CUDA Code Generation for Affine Programs Muthu Manikandan Baskaran and J. Ramanujam and P. Sadayappan 2010
11. A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code MALIK KHAN and NUST PROTONU BASU and GABE RUDY and MARY HALL and CHUN CHEN and JACQUELINE CHAME 2013
12. Benchmarking GPUs to Tune Dense Linear Algebra Vasily Volkov, James W. Demmel 2008
13. A Note on Auto-tuning GEMM for GPUs Yinan Li1, Jack Dongarra, Stanimire Tomov 2009
14. An Improved Magma Gemm For Fermi Graphics Processing Units Rajib Nath, Stanimire Tomov, Jack Dongarra 2010
15. Performance, Design, and Autotuning of Batched GEMM for GPUs Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov and Jack Dongarra 2016
16. Novel HPC Techniques to Batch Execution of Many Variable Size BLAS Computations on GPUs Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov and Jack Dongarra 2017
17. Brute-Force k-Nearest Neighbors Search on the GPU Shengren Li and Nina Amenta 2015
18. Experiences in autotuning matrix multiplication for energy minimization on GPUs Hartwig Anzt, Blake Haugen1, Jakub Kurzak1 and Piotr Luszczek and Jack Dongarra 2015
19. Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs Jee W. Choi, Amik Singh and Richard W. Vuduc 2010
20. https://www12.informatik.unierlangen.de/edu/map/ss08/talks/Best Practices for GPU Programming.ppt, Access 26.2.2018 16:21. Best Practise for GPU Programming.
21. Understanding Latency Hiding on GPUs, Vasily Volkov, 2016
22. AARTS, Emile; KORST, Jan. Simulated annealing and Boltzmann machines. 1988

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.