Acceleration through Fusion of AvgPool2D and Silu Kernels

Andreas Falkenberg

Dr Falkenberg Technology Consulting Inc
Escondido, California 92026

Abstract. The need to accelerate LLM (large language models) requires the use of always advancing compiler technologies. Operator fusion is one of the promising techniques to considerably improve the throughput of LLMs. This paper discusses the impact of operator fusion on the direct operator performance. The paper compares throughputs between pure CPU implementation, versus two kernel implementations versus a fused single kernel solution for AvgPool2D fused with Silu.

Keywords: AvgPool2D, Silu, Kernel, AI, LLM, GPU, CPU

1 Introduction

In recent years the push for and the growth of Large Language Models (LLM) provided new challenges for processor and compiler developers to improve the throughput and performance of such models. The main challenge is to stay within the constraints of already existing hardware. Although hardware accelerators enter the market in relative rapid succession, companies may not be willing to upgrade their compute ressources with each new generation of processors and accelerators. Usually a software upgrade and tool upgrade is much more common and easier to achieve compared to an actual hardware upgrade. Essentially there are two areas, which are targeted to improve performance of LLMs. As already mentioned, there is the improvement of the hardware by utilizing new hardware architectures and improving existing hardware architectures by adding specific features, which benefit the processing of neural network workloads. The other area is the improvement by software tools. Specifically AI compilers are under development, which boost the performance of exising hardware by utilizing the specific AI accelerating features. [1] [2] [3] They also need to adapt quickly to support new features of new hardware. These optimizations are often in the area of parallel execution of workloads, faster data transfer between nodes and faster loading of kernels into the hardware to name a few. One advanced optimization step in the development process of large language models (LLM) is the use of fused operators [4]. Operator fusion targets the shortcomings of using accelerators, which require to upload and download the data between each operation, by removing some of the intense data transfer operations.

This paper specifically provides some measurements of the direct improvement of operator fusion by comparing operator performance with and without fusion being applied. Once a set of operators is identified as fuseable, it then can be combined into one operator and finally mapped to the appropriate fused kernel. This paper discusses fusion [5] of a AvgPool2D [6] operator with a Silu operator [7]. The following sections provide a brief discussion of the underlying idea, then discuss some of the implementation and test features and environment. Lastly the measured results of a AvgPool2D fused with a Silu operator are presented and a conclusion is given.

2 Motivation

A neural network workload constitutes of many nodes which mostly are executed individually or in eager mode. Eager mode means even with accelerating hardware, each node is executed on an individual basis and therefore the data is loaded in and out of the accelerator after each operation. A huge amount of data may be loaded to the accelerator, which then in a very aggressive highly parallel effort, performs such an operation quickly utilizing the parallel feature of the hardware accelerator and then load the result of said operation back to the host processor. A second node is processed in the exact same way, which means that the same or maybe a reduced or enlarged amount of data is loaded into the accelerator and the appropriate process is performed again. This neverthless means that data, which might as well be processed within the same hardware, is uploaded and downloaded between processing steps despite being already at the correct location or stored in the correct memory. By fusing nodes the additional upload and download step of interim results are removed and the appropriate steps are directly performed. This usually leads to a considerable improvement in throughput and performance of the exact same model. We can see this specifically in the combination of matrix multiplication with immediately following activation functions. The optimization of matrix multiplication and its fusion with activation functions seems to be already widely discussed [8], [9]. In most publications the overall impact of the fusion procecss on existing models is meansured, whereas there is little to no discussion about the impact of each individual contributing node. We think that measuring the acceleration of individual nodes should be of utmost importance for developers of original neural network architectures. In this paper we discuss one specific combination of operations, which are very common in the context of Large Language Models. The first is the AvgPool2D function and the second is the Silu function. Other publications will follow to discuss other combinations of operations.

3 The Approach

Different solutions were executed to compare the runtime of a AvgPool2D operator and a Silu operator between CPU and CUDA solutions. A small model with one

AvgPool2D operator and one Silu operator with flexible shapes is implemented in C++ to run on a CPU. Then a total of three kernels are developed in CUDA. The first kernel represents the AvgPool2d operation, the second kernel represents the Silu operator and the third kernel represents the fused kernel, which constitutes of the said AvgPool2D operator directly followed by tbe Silu operator implemented in one CUDA kernel.

3.1 AvgPool2D kernel implementation

The implementation of the AvgPool2D kernel is based on the mathematical representation shown in equation 1:

$$avgPool2D(N_i, C_j, h, w) = \frac{1}{kH * kW} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n) \tag{1}$$

The kernel is implemented in CUDA. The inner two loops are implemented on each of the nodes, which therefore would perform $kH * kW$ operations each. The outer loops are mapped to the nodes of the CUDA hardware.

3.2 Silu kernel implementation

The Silu kernel is based on the mathematical definition in equation 2:

$$Silu(x) = \frac{x}{1 + e^{-x}} \tag{2}$$

Each kernel performs the above Silu operation, therefore mulitple of these operations can be mapped onto the CUDA platform to be executed in parallel. For this specific exercise there is no further optimization performed and the original Silu function is used as in Figure 1. Nevertheless it is recognized that there are proposals to improve the Silu operation considerably by using partial functions with linear approximations.

In the following sections the results of timing measurements are presented.

4 Results

In this section we show the results when running the individual kernels on CPU, then we show the combined solution running on CPU. Further we show the performance of the two CUDA kernels running back to back on a CUDA platform, and finally compare that with the fused kernel solution, which shows the real benefit of performing fusion in the first place.

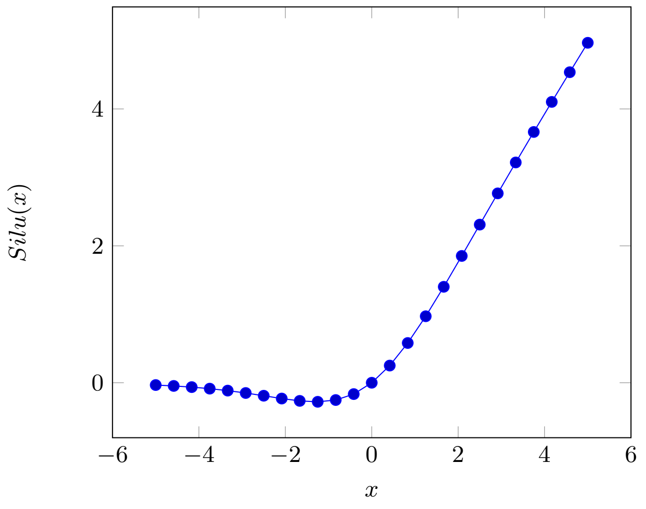

Fig. 1. Silu function

4.1 CPU implementation of AvgPool2D

The first result is based on a CPU implementation of AvgPool2D in C++ on a PC. In Figure 2 we can clearly identify a linear behaviour based on the number of input elements. The kernel size as well as the stride size are set to eight for each dimension.

4.2 AvgPool2D in sequence with Silu on CPU

The next implementation combines the AvgPool2D algorithm with Silu. Since it is performed on the CPU we do not need to make a difference between the sequential and the parallel execution as well as the combined execution of the two operators.

The Figure 3 shows the result for AvgPool2D with Silu in sequence. The inclusion of Silu hardly contributes to the overall result on CPU.

In Figure 4 a direct comparisong of AvgPool2D without and with Silu is shown. It is recognized that exercising Silu on a CPU does not significantly change the overall runtime. In this case, unless we would zoom further in, we do not even see a difference on this plot. The main explanation is that there is no additional allocation of memory involved since Silu directly writes back to the same element. Further the number of elements is reduced now by 64, because the avgPool2D operation uses a 8x8 inner dimensions for our example.

4.3 Performance of AvgPool2D kernel on CUDA

In the following subsections the CUDA results are presented. The first results as shown in Figure 5 are based on the AvgPool2D cuda kernel. It shows that the par-

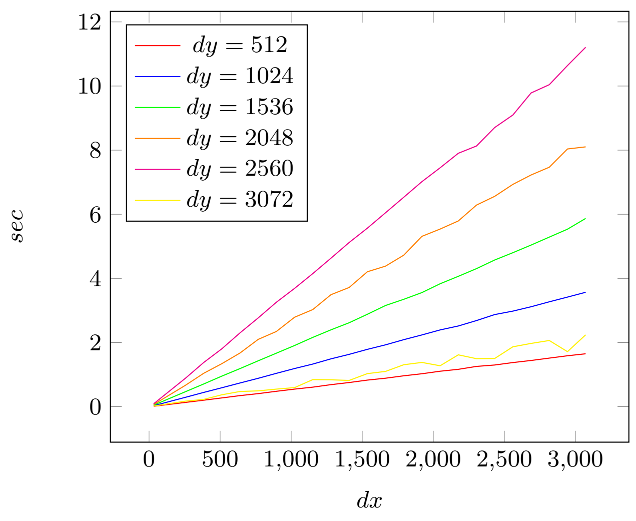

Fig. 2. AvgPool2D on CPU

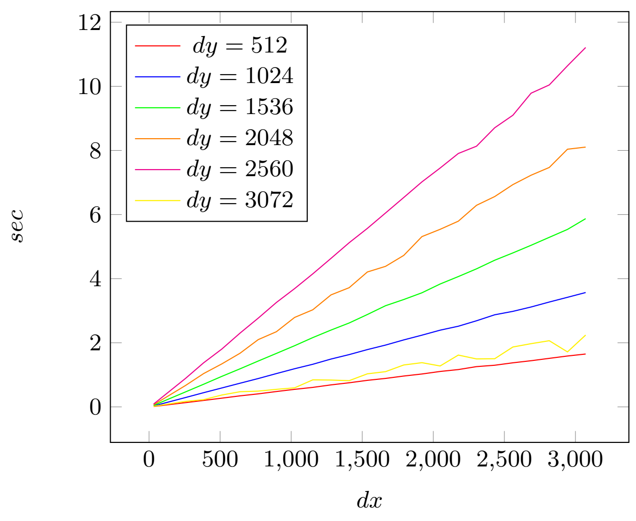

Fig. 3. AvgPool2D with Silu on CPU

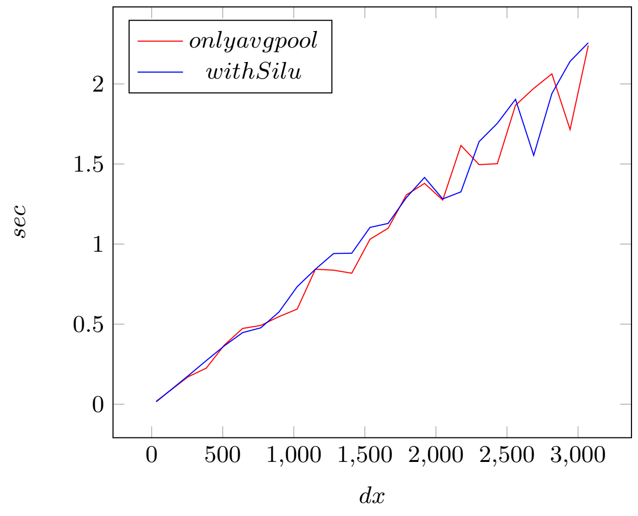

Fig. 4. AvgPool2D with/without Silu on CPU

allel execution leads to almost no difference between smaller and larger workloads. The presented workloads are still small and therefore the major impact comes from the overhead of engaging with the GPU in the first place. Nevertheless the results clearly show that the CUDA implementation has a huge impact on the performance.

4.4 Performance of Silu kernel on CUDA

If we only run the Silu kernel Figure 9 shows that for most cases the overhead to engage with the GPU has a higher impact than performing Silu directly on the CPU. Nevertheless part of it is misleading since on the CPU we do not count the impact of the memory allocation, which is already done as part of the AvgPool2D output matrix memory allocation.

4.5 AvgPool2D and Silu in sequence

To provide some better datapoints Figure 7 shows running the avgPool2D and the Silu back to back, yet using separate kernels. We can clearly also see that there is some overhead compared to the prior solution.

The best result is achieved by using the fused kernel. The fused kernel incorporates avgPool2D and Silu into one kernel and therefore the output of the AvgPool2D is directly processed. The first result does not need to be copied to the main memory and back to the kernel memory of the Silu kernel any more. This allows to shave off a significant portion of the runtime and improve the performance significantly. Figure 8 shows the result of the fused kernel running on a CUDA capable device.

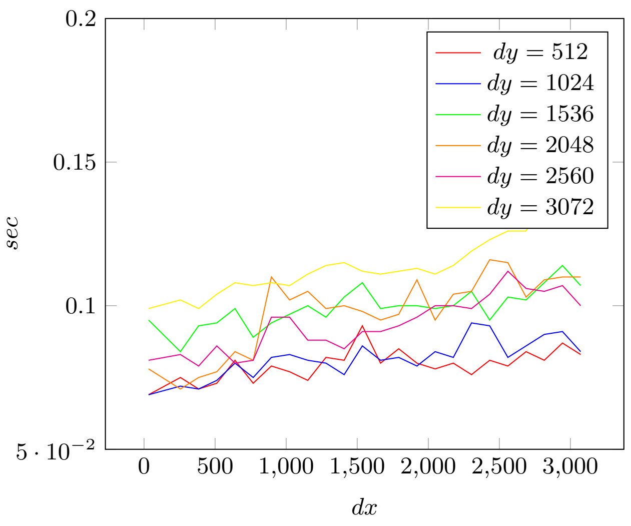

Fig. 5. AvgPool2D on CUDA

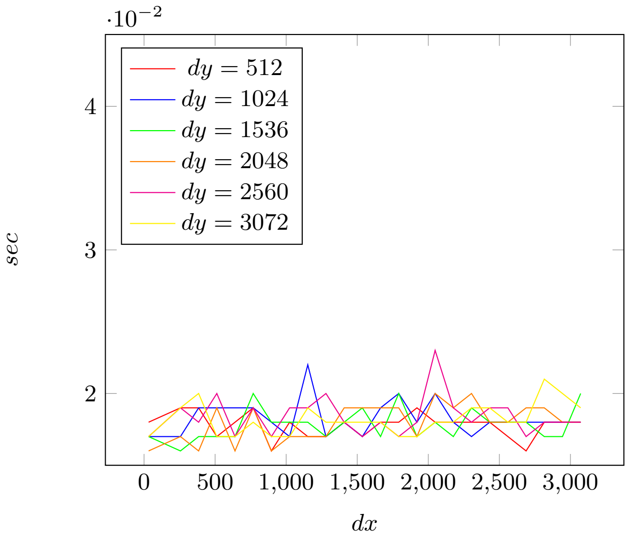

Fig. 6. Silu on CUDA

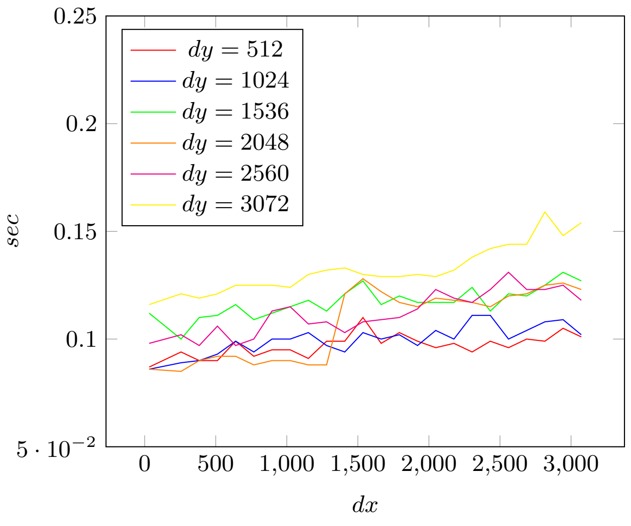

Fig. 7. AvgPool2D and Silu sequential on CUDA

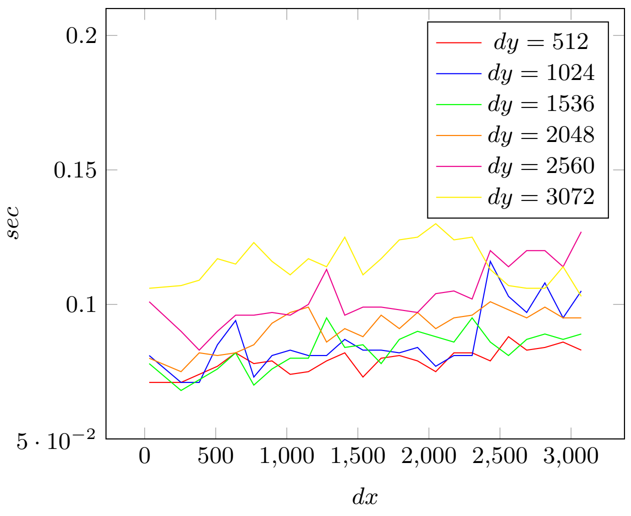

Fig. 8. AvgPool2D and Silu fused on CUDA

5 Conclusion

We conclude that kernel fusion has a significant impact on the performance and the throughput of a model. Although the fused kernel was not integrated into a realistic LLM workload this initial research already shows significant progress. The Figure 9 combines the most significant results. We can see that only when a small matrix is used, i.e. at or below 128x3096, the CPU solution may be faster.

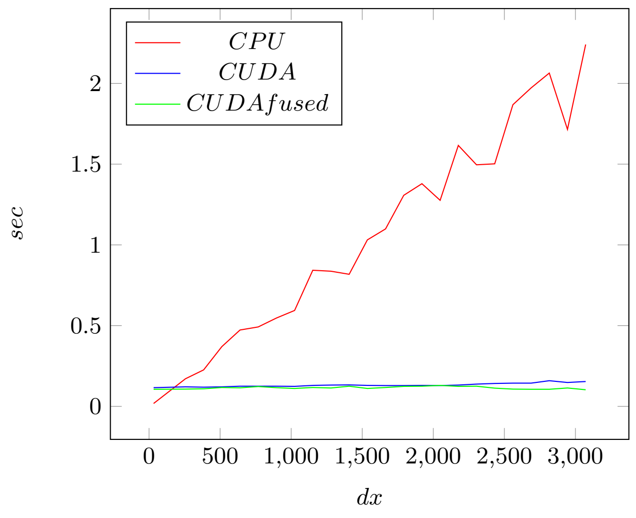

Fig. 9. CPU vs CUDA vs fused CUDA

The herein presented results are based on the implementation of a new CUDA based library, which concentrates solely on fused kernels for AI workloads. The progress of this work can be followed in the github repository [10].

References

1. Alwani, Manoj and Chen, Han and Ferdman, Michael and Milder, Peter, Fused-layer CNN accelerators, 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016.
2. Cai, Xuyi and Wang, Ying and Zhang, Lei, Optimus: An Operator Fusion Framework for Deep Neural Networks, ACM Trans. Embed. Comput. Syst. , Association for Computing Machinery, Jan. 2023.
3. Liu, Zihan and Leng, Jingwen and Chen, Quan and Li, Chao and Zheng, Wenli and Li, Li and Guo, Minyi, DLFusion: An Auto-Tuning Compiler for Layer Fusion on Deep Neural Network Accelerator, 2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), 2020.

4. Wei Niu, Jiexiong Guan, Yanzhi Wang , Gagan Agrawal and Bin Ren, DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion, https://arxiv.org/abs/2108.13342, 2021.
5. Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Alexandre V. Evfimievski and Prithviraj Sen,On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML, https://arxiv.org/abs/1801.00829, 2018.
6. Fanyi Wang and Haotian Hu and Cheng Shen, BAM: A Balanced Attention Mechanism for Single Image Super Resolution, https://arxiv.org/abs/2104.07566, 2021.
7. Abien Fred Agarap, Deep Learning using Rectified Linear Units (ReLU), https://arxiv.org/abs/1803.08375, 2019.
8. Acharya, Aravind and Bondhugula, Uday and Cohen, Albert, Effective Loop Fusion in Polyhedral Compilation Using Fusion Conflict Graphs. ACM Transactions on Architecture and Code Optimization, Association for Computing Machinery (ACM), Sep. 2020.
9. Weishi Li, Yong Peng, Miao Zhang, Liang Ding, Han Hu and Li Shen, Deep Model Fusion: A Survey, https://arxiv.org/abs/2309.15698, 2023.
10. Andreas Falkenberg, Fused Operators in CUDA, https://github.com/afalkenberg/AIFusionCuda, 2024.

Authors

Andreas Falkenberg is a independent consultant. He received his PhD in Computer Science from TU Dortmund in 1998 and a second PhD in Telecommunication Systems from University of Glamorgan in 2007. His research interests are in kernel development for large language models and Compiler design.