Hardware Acceleration of Transformer Networks using FPGAs

Georgios Tzanos
Dep. of Electr. and Computer Engineering
NTUA
Athens, Greece
grg.tzan@gmail.com

Christoforos Kachris
ICCS-NTUA
& DUTH
Greece
kachris@microlab.ntua.gr

Dimitrios Soudris
Dep. of Electr. and Computer Engineering
NTUA
Athens, Greece
dsoudris@microlab.ntua.gr

***Abstract*—Natural Language Processing (NLP) allows program computers to process and analyze large amounts of natural language data. In the last few years, NLP has shown tremendous growth, with many organizations presenting models such as BERT (Bidirectional Encoder Representations from Transformers), GPT2 (Generative Pre-trained Transformer 2), GPT3 (Generative Pre-trained Transformer 3), etc. The cornerstone of these models and the reason for the growth of the NLP is mainly due to the Transformer networks. However, very few architectures have been presented, for the acceleration of the Transformer networks using FPGAs. In this paper, we propose a novel architecture for Transformer Networks optimized on FPGAs. The performance evaluation on a Xilinx Alveo U200 FPGA achieved up to 80.5x speed-up over a single-core CPU and up to 2.3x speedup over a 40-thread Xeon CPU running BERT model.**

I. Introduction

Transformer Networks [2] are behind every state-of the art Language model, such as BERT [5], GPT3 [6], etc. The reason for this is that Transformers were developed to solve the problem of sequence transduction, or neural machine translation. That means any task that transforms an input sequence to an output sequence. This includes speech recognition, text-to-speech transformation, etc. Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) [3,4] have been used to deal with this problem because of their properties but it turned out to be not enough. This computational complexity motivated efforts to enhance these tasks using hardware-specific optimizations by leveraging different heterogeneous architectures combining CPUs, GPUs, FPGAs. The increasingly growing demands for efficient and fast processing of the new generation algorithms today can be addressed by high performance computing systems such as FPGAs. This architecture shows immense parallelization and reconfigurability and can be mapped well to repetitive tasks, such as Transformer Networks. Specifically, many companies tend to search for efficient ways for high performance and low energy cost solutions, and FPGAs play a major role in this evolution.

FPGAs as parallel platforms can reduce the total computational overhead with an underlying hardware solution which boosts the performance on these kind of tasks. An optimized implementation can take advantage of the FPGA hardware resources for high performance and much lower energy footprint, something very crucial for data centers that operate today on these large scale tasks. In this paper, we present an implementation of a Transformer network on FPGAs and we achieve a speed-up compared with CPU multi-core solutions. More specifically, in this work we make the following contributions:

- We perform a thorough analysis of the most computational intensive tasks that could be offloaded to FPGAs
- We implement a novel Transformer network fully utilizing the advantages of FPGAs
- We evaluate the speed-up of the Transformer network on an FPGA accelerator (Up to 80.5x speedup).

II. Related Work

While a lot of work has been published in software optimization for CPU and hardware optimizations on GPU for Transformer networks, very little work has been published related to custom hardware acceleration of any Transformer-based networks on FPGA. Two ASICs have been recently proposed, OPTIMUS [7] and A^3 [8], that each accelerate different parts of Transformer inference. OPTIMUS optimized matrix multiplication for Transformers by exploiting sparsity. It has dedicated exponential, divider, and square root components for non-linearities, leading to wasted area since each is only used a small fraction of the time. FTRANS [1] has currently published an FPGA accelerator for BERT and related Transformer net-works. FTRANS takes a very specialized approach for implementing Transformers, in which it has dedicated encoder and decoder modules. Each of these modules has many specialized components. Moreover another publication on this direction is NPE [9]; an FPGA-based overlay processor that can efficiently execute a variety of NLP models. NPE offers software-like programmability to the end user and, unlike FPGA designs that implement specialized accelerators for each nonlinear function, can be upgraded for future NLP models without requiring reconfiguration. Compared to our work, both Ftrans and NPE use custom ways to implement the Transformer network, while in our case we directly replace all or some of the parts of the Transformer network with FPGA kernels.

979-8-3503-9958-5/22/$31.00 ©2022 IEEE

III. Transformer Networks

BERT adopts the structure of the encoders from Transformers. While there are many BERT variants, the particular structure can be described by three parameters: $number\ of\ encoders\ L$, $number\ of\ attention\ heads\ A$, and $hidden\ layer\ size\ H$. We focus on $BERT_{BASE}$, which is composed of 12 encoders, each with 12 attention heads and a hidden size of 768 ($L = 12, A = 12, H = 768$). The general structure for BERT can be found in [2]. The model starts with an embedding layer that converts each input language sequence into features. For instance, an input sequence with 512 tokens in $BERT_{BASE}$ would be converted to a 512×768 matrix, where each token is replaced by a 768-length feature vector. Here, a token refers to a few adjacent characters, where a word is made up of one or more tokens. The embedding step has negligible computations but requires lots of memory. Therefore, we assume this initial step is performed off-chip and we focus on accelerating the computationally-intensive encoders.

Transformer networks use a technique called attention. The attention, as we would say in the field of neuroscience, is the ability to be able to selectively concentrate on specific data while ignoring other data of our environment. In deep learning we imitate this technique through attention mechanisms and one way to achieve this is to encode a sequence not into a single fixed vector but to create a model that produces a vector for each output step by adding a set of weights which will later be optimized. Consequently it does not simply learn what to produce in the output but how to put weights selectively on specific input data maximizing the probability of a correct output. Bert has multiple Attention blocks. Each Attention block converts the input using **GEMM (General Matrix Multiplications)** operations and then uses **GEMM (General Matrix Multiplications)** operations and **Non-Linear** functions such as $Softmax$, $Layernorm$ and $Gelu$ to produce the output.

GELU Activation. The $Gaussian\ Error\ Linear\ Unit$ activation is defined by the following equation:

$$GELU = xP\left(X \leq x\right) = x \cdot \frac{1}{2}\left[1 + erf\left(x/\sqrt{2}\right)\right] \quad (1)$$

It is commonly approximated using the tanh function as in Equation(2) and can also be approximated directly using a lookup table.

$$GELU \approx 0.5 \times \left(1 + \tanh\left[\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right]\right) \quad (2)$$

Layer Normalization. Layer normalization first requires computing the mean and variance of a matrix across rows. Given a matrix of dimension NxK, we compute the mean and variance for row i

$$\mu_i = \frac{1}{K}\sum_{k=1}^{K} x_i, k \quad , \quad \sigma_i^2 = \frac{1}{K}\sum_{k=1}^{K}\left(x_{i,k} - \mu_i\right)^2 \quad (3)$$

Then, the mean and variance are applied to each element using Equation(4) to get the normalized output $\widehat{x}$. Finally, each $\widehat{x}_i$ is scaled and shifted by trained vectors γ and β to get the layer normalization output y, as shown in Equation(5)

$$\widehat{x}_{i,k} = \frac{x_{1,k} - \mu_k}{\sqrt{\sigma_k^2 + \varepsilon}} \quad (4)$$

$$y_{i,k} = \widehat{x}_{i,k}\gamma_k + \beta_k \quad (5)$$

Softmax. The definition of $Softmax$ is shown in Equation (6). $Softmax$ can be difficult to implement in hardware because of the exponential and division operations. It can be directly realized using dedicated exponential and division units, at the cost of under-utilized resources and extra area overhead.

$$softmax(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}} \quad (6)$$

IV. Accelerator Architecture

A. Transformer Network Architecture

To select which parts of Transformer Network are worth implementing on hardware, we first had to perform a detailed profiling on a 40-thread Intel Xeon processor . Then we identified the algorithms of the hardware function we are going to accelerate and all the optimizations on the host and kernel side so to reduce the overall latency of the design.

TABLE I
Profiling of the Transformer Network.

Task	Percentage
GEMM Ops	70%
GeLU2	10%
SoftMax	2.7%
LayerNorm	1.85%
Misc	15.45%

For the implementation of the Transformer network we used the following architecture. Our goal was to speedup the network compared to a CPU implementation. To achieve this we developed our implementation on two main ideas. The first one was to optimally accelerate the GEMM (GEneral Matrix Multiply) operations of the network $sgemm$ and batch_sgemm, which consume about 70% percent of the runtime (Table 1). The acceleration of these two kernels sets our implementation in position to compete with the CPU. The second one was to integrate $sgemm$ and batch_sgemm, which are the most computational intensive parts of the Transformer network, with functions of very little impact on the runtime, such as $Gelu$, $Softmax$ and $Layernorm$, in order to prevent unnecessary data transfers from the host to the kernel and vice versa something that it would definitely cost to our implementation.

The first part of our implementation (Fig 1.) contains the parallel execution of three $sgemm$ kernels, since the

multiplication of vectors $q(query)$, $k(key)$, $v(value)$ with embedding vectors of the Transformer network are completely independent to each other. In the second part of our implementation (and for all the parts after this, we enter a sequential order of execution), we implemented a $batch\_sgemm$ kernel combined with a $Softmax$ kernel. In the third part of our implementation we chose to implement a standalone $batch\_sgemm$, while in the fourth part we have a combination of $sgemm$ and $Layernorm$. Finally, in the fifth part with another combination of kernels, by combining the $sgemm$ and $Gelu$(activation function) kernel, while for the last part we can reuse the computational kernel of the fourth part.

To make the final decision for our architecture we had to take into consideration the need of balancing between resources utilization of a single device and the maintainability of the kernels in the SW/HW co-design. The factors above led us to this hybrid implementation, which neither contains all the functions in a single kernel, nor implements each function as a standalone kernel.

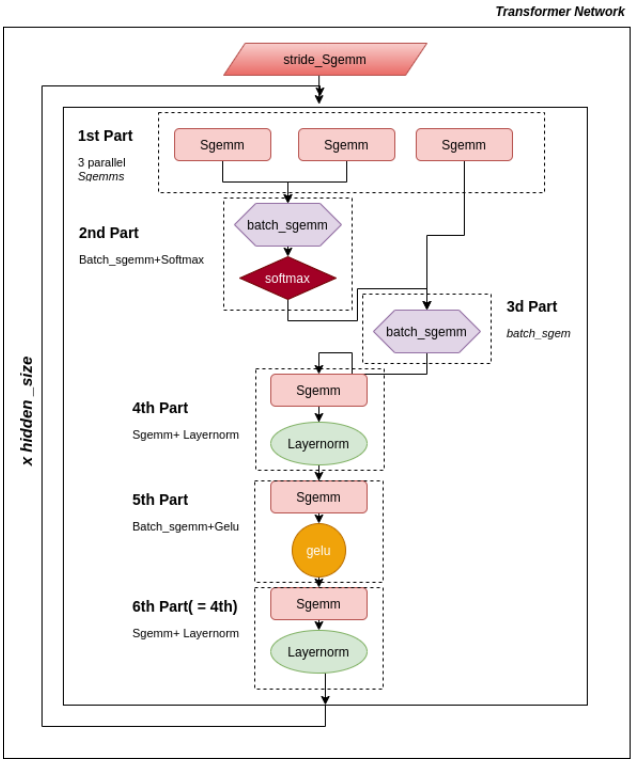

Fig. 1. FPGA Transformer Network Architecture

Moreover the dataflow between so many functions in a single kernel was not supported on the compiler. Furthermore, the re-configuration of the network, with different parameters, would be a lot more difficult in a compact and monolithic kernel. At the same time the combination of some functions into a single kernel helped us to optimize some critical data movements. Finally the need for complete parallelization of the first three $sgemms$ of the first part is another reason that lead us, in some occasions, to create separate but identical kernels that can be called and be executed in parallel.

B. GEMM kernels

In the case of $sgemm$ kernel, we applied block-tiled matrix multiplication, for parallel multiplication of multiple elements, while for the batch_sgemm kernel the small dimension of the sub-matrices allow us to use a systolic array approach for the multiplication.

C. Softmax, Layernorm, Gelu

In the case of $Softmax$, $Layernorm$ and $Gelu$ functions, the implementations does not aim at any speedup compared to the CPU implementations. Any different approach would result in consumption of additional hardware resources without any impact on the overall speedup.

The specific architecture is being repeatedly executed in a loop as much as the number of hidden layers and the number of attention heads. For future work, this gives us the opportunity for an architecture with multiple devices that can run in parallel as each attention head can be executed completely simultaneously and independently of the others.

V. Implementation

The kernels have been developed on Vivado HLS targeting the Alveo U200 FPGA card. We managed to keep the resources in balance allowing kernels to run at a high clock frequency of $411MHz$.

A. Sgemm

1) Host Optimizations: In order to accelerate $sgemm$ we followed the steps below. First, we used multiple DDR BANKS assigning each one of the matrices that participating in the GEMM operation (with A and B as matrix inputs, $alpha$ and $beta$ as scalar inputs, and C as a pre-existing matrix which is overwritten by the output), in a different DDR BANK.

$$C = (alpha)AB + (beta)C \tag{7}$$

The use of multiple DDR BANKS results in faster communication between device and host as it enables the use of multiple communication channels between them, allowing the host to send data in parallel, from the Global Memory to the FPGA. The next step was to achieve the absolute parallelism in the execution of the three $sgemm$ kernels of the first part of the architecture scheme. Thus, in order to do so we used the $clEnqueueTask$ command in a for-loop, enabling the parallel execution between these tasks. Finally we transpose matrix $A \rightarrow A^T$ in order to have the two matrices parallel to their common dimension.

2) Kernel Optimizations: On the kernel side, after we had transposed matrix A, we were able to apply multiple burst reads accessing the data of the matrices in a sequential way like a FIFO, preventing further delays while we copy the data to local BRAMs of two dimensions. Matrix multiplication [10] requires to access matrix B multiple times, so in order to prevent multiple reads from the global memory to local memory we copy all the matrix to local BRAMs. Therefore, we avoided memory requests of the same data and also

achieved faster burst data transfers to global memory as bigger chunks of memory are transferred at higher rates. while the matrix A is being copied in tiles of 8 rows $Asub[8][n]$, keeping the resources utilization low. Finally we, created 1 more extra table in the local memory, the $Bsub[8][m]$ in which we load the rows of table B in blocks of 8, in order to be able to enforce HLS directives such as $\#pragma\ HLS\ ARRAY\_PARTITION$, for maximum parallelism.

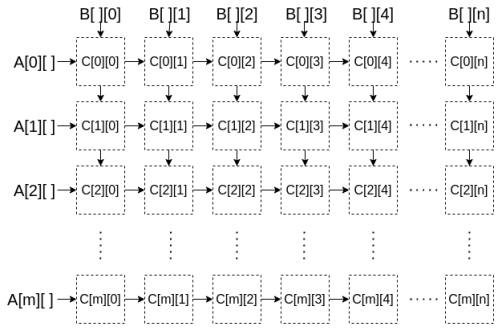

Fig. 2. Systolic array for Batch_sgemm

B. Batch_sgemm

In the implementation of batch_sgemm now, in which we batch multiple independent small GEMM operations into a group, we used the technique of systolic array multiplication (Fig. 2) and the reason we used this particular technique is that the small dimensions of the submatrices, makes them suitable for loop unrolling, leading to better latency.

From the host side, to retain the correct sequential order of execution of the kernels we used the $clEven$ objects from the OpenCL API to enforce the kernels to be executed in the correct order. Moreover, from the kernel side, to further improve the systolic array technique we used four rows of each matrix to be multiplied concurrently decreasing the iteration over the common dimension by a factor of four, increasing the parallelism. Finally we applied the HLS directive DATAFLOW (enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput of the design), to minimize the iteration interval over the batches iteration.

C. Rest of the kernels

The rest of the kernels we implemented, were not used alone but as parts of the combination with GEMM kernels.

- $Batch\_sgemm + Softmax$
- $Sgemm + Layernorm$
- $Sgemm + Gelu$

The functions $Layernorm$, $Softmax$ and $Gelu$ due to the limited impact they have on the runtime, are not key kernels for the Transformer network acceleration, so their implementations were based on their low resource consumption in order for the integration to the device to be feasible. During the implementation on the part of the host we send all the data that is necessary for the calculation of the respective combination of kernels. At this point, inside the kernel we use again the HLS pragma DATAFLOW ensuring that the output of the first function will be used as input to the followed function as soon as it is available,minimizing by this way the time for the data transferring between the two functions.

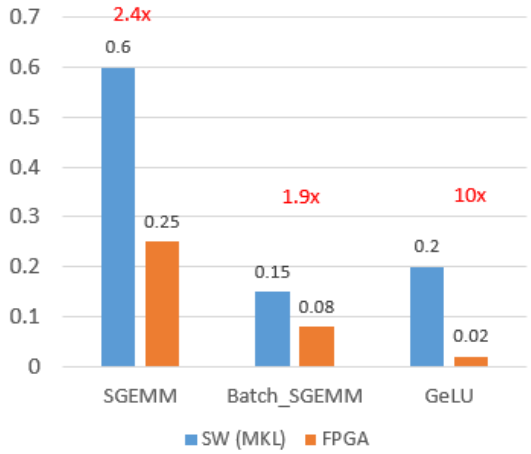

Fig. 3. Kernels evaluation

D. Common Optimizations

To achieve large throughput we had to enable a high degree of parallelism in application execution by avoiding data dependencies. We constructed a highly parallel and pipelined architecture with minimum latency. We achieved initiation interval II=1 in every loop and by using the dataflow directive the kernel can process all the data as soon as they arrived at memory interfaces and write them back to DDRs as soon as they are ready. Last but not least, AXI4-master interface for burst read and write was used, inferring multiple bursts of 512-bit width in each kernel. This specific option gives us the ability to send a large volume of data to the kernels, reducing the time needed to copy the data in the local memory of the FPGA. Specifically what we did was to create vectors from 32 float16 numbers each. This has the effect of taking advantage of the full bandwidth offered by the AXI-4 protocol by reading and writing 32 numbers simultaneously per cycle.

VI. Evaluation

A. System Setup

To evaluate our implementation we mapped the architecture on an Alveo U200 FPGA card while as a host an Intel Xeon processor with 40 cores has been used.

Moreover in order to integrate our work in a Transformer network we used an existing implementation [11] of a Transformer network written in $C/C++$ which was also part of a

BERT model. To be able to implement our proposed architecture in our device, we had to change some parameters of the network. Specifically, we configured some optional parameters of the Transformer network such as the $batch\_size$, making our network a smaller one. In the specific experiments we used $batch\_size = 1$.

B. Kernels Evaluation

Figure 3 shows the performance of each kernel independently. Intel MKL has been optimized for using the maximum threads of the CPU. The $sgemm$ kernel has been accelerated and it has a speedup of $2.4x$ compared to $cblas\_sgemm$ and $batch\_sgemm$ has speedup $1.9x$ compared to $cblas\_sgemm\_batch$. Moreover $Gelu$ functions although it takes only a 10% of the runtime we have managed to accelerate it with a speedup of about $10x$. The rest of the functions do not show any speedup compared to the CPU implementation and have been omitted in the figure. Integrating all the kernels in the Transformer network we achieve a speedup of $2.3x$ inside the BERT model compared to a 40-thread processor and 80.5x speed-up over a single-core CPU. In terms of resource allocation, Table II shows the utilization of the hardware resources for the Alveo U200 Data Center card.

TABLE II
HARDWARE FUNCTIONS RESOURCES.

Resources	***Used***	***Available***	***Utilization***
DSP	5861	6840	85.7%
BRAM	2592	4320	60%
LUT	910324	1182240	77.4%
FF	1938873	2364480	81.2%

VII. CONCLUSION

In this paper, we described an optimized FPGA implementation of the Transformer Networks. Our work shows that the overall speedup can reach up to 2.3x compared to a 40-thread CPU and up to 80.5x over a single-core CPU. Our architecture can be expanded by using multiple FPGA cards allowing parallel computations to independent multiple Transformer networks inside the BERT model.

REFERENCES

[1] Bingbing Li, Santosh Pandey, Haowen Fang, Yanjun Lyv, Ji Li, Jieyang Chen,Mimi Xie, Lipeng Wan, Hang Liu, and Caiwen Ding. 2020. FTRANS: energy-efficient acceleration of Transformers using FPGA. In Proceedings of the ACM/IEEEInternational Symposium on Low Power Electronics and Design. 175–180.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, JakobUszkoreit, Llion Jones, Aidan N Gomez, LukaszKaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in Neural Information Processing Systems, pages 6000–6010.

[3] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann NDauphin. 2017. Convolutional sequence to sequence learning.arXiv preprintarXiv:1705.03122(2017).

[4] Alex Graves. 2012. Sequence transduction with recurrent neural networks.arXivpreprint arXiv:1211.3711(2012).

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert:Pre-training of deep bidirectional Transformers for language understanding.arXivpreprint arXiv:1810.04805(2018).

[6] Tom B. Brown, Benjamin Mann, Nick Ryder,Melanie Subbiah, Jared Kaplan, PrafullaDhariwal, Arvind Neelakantan, Pranav Shyam,Girish Sastry, Amanda Askell, SandhiniAgarwal, Ariel Herbert-Voss, GretchenKrueger, Tom Henighan, Rewon Child, AdityaRamesh, Daniel M. Ziegler, Jeffrey Wu,Clemens Winter, Christopher Hesse, MarkChen, Eric Sigler, Mateusz Litwin, ScottGray, Benjamin Chess, Jack Clark, ChristopherBerner, Sam McCandlish, Alec Radford,Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners.arXiv:2005.14165 [cs].

[7] Junki Park, Hyunsung Yoon, Daehyun Ahn, Jungwook Choi, and Jae-Joon Kim.2020. OPTIMUS: OPTImized matrix MUltiplication Structure for Transformer neu-ral network accelerator. In Proceedings of Machine Learning and Systems, I. Dhillon,D. Papailiopoulos, and V. Sze (Eds.). Vol. 2. 363–378.

[8] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, YoonhoSong, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al.2020. Aˆ 3:Accelerating Attention Mechanisms in Neural Networks with Approximation. In2020 IEEE International Symposium on High Performance Computer Architecture(HPCA). IEEE, 328–341.

[9] Khan, H., Khan, A., Khan, Z., Huang, L. B., Wang, K., and He, L., "NPE: An FPGA-based Overlay Processor for Natural Language Processing

[10] Introduction to Linear Algebra, Fifth Edition (2016) by Gilbert Strang ISBN : 978-09802327-7-6

[11] https://github.com/zhihu/cuBERT