A Novel Accelerator Design for Natural Language Processing CNNs

Colter, Zachary[a] and Zhang, Wentao[b] and Liu, Bryan[b,c] and Lou, Yuanqing[b,c]

Abstract

In our data-driven world, there is a paramount need for technologies that can make sense of and keep up with the rapid growth of big data. A very large subset of this work involves interpreting complex and highly variable sentences, to answer questions, to paraphrase, to classify and compare them, and to summarize the main takeaways of a much larger piece of text, to name several. Such tasks are used in domains such as targeted user advertising, stock price prediction, and speech recognition. With the vast amounts of new data created every day, NLP technologies must be both accurate, fast, and scalable to keep up with growing demand. Traditional LSTMs and RNNs, which in the past have been used in the NLP field, have faltered in terms of performance and speed when comparing them to CNNs. CNNs have shown great promise to be used for NLP, a domain with a huge range of applications, and thus we have designed an NLP CNN accelerator. Our hardware targets the convolution operation between sentences and a set of filters for acceleration by leveraging 1D convolution for higher data reuse and a fast lookup table to reuse previous computations. In doing this, our accelerator achieved a 23x speedup when compared to the software implementation.

Keywords: Convolutional Neural Network, Natural Language Processing, Accelerator

1 Introduction

The major bottleneck of NLP CNNs is the convolution operation, which we found takes up about 72% of the total run-time in our software implementation[1]. Because of this, we decided to build an accelerator to increase the performance of our convolution operation step in terms of increasing throughput and reducing power consumption, while maintaining or improving latency when compared to our software benchmark on a general purpose processor.

NLP CNNs differ from image processing in two key ways. First, inputs are indexes into an embedded matrix of hyperparameters, where each row of hyperparameters represents a pre-computed feature representation of a word. Second, the filter is asymmetrical with one dimension being the row size of the embedded matrix. Because the filter has the same length of the hyperparameters, this is effectively a 1D convolution operation. The is much less reuse in a naive 1D convolution operation than in a 2D due to how many more weights are required per each fmap.

Our design exploits the fact that there are a finite amount of input indexes. Reducing the input sample space allows us to

1) store newly computed word and filter computations in a lookup table in BRAM for fast lookup when we see that word again for the same filter.
2) greatly increase the number of lookup table hits due to the drastically smaller number of total possible results. Our hardware implements this lookup table idea to accelerate the convolution operation.

Overall, our accelerator's design is split into three different stages: pre-processing, processing, and post-processing.

1. **Pre-processing**
   We train our network in Python using Keras to get our filter weights. We then obtain our embedded matrix from Stanford's NLP glove library and a batch of sentences we want our accelerator to run inference on. Our values in our embedded matrix and filter weights are 32-bit floating point

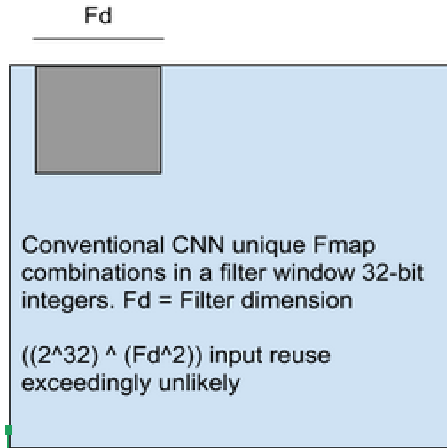

Figure 1: Conventional 2D Convolution

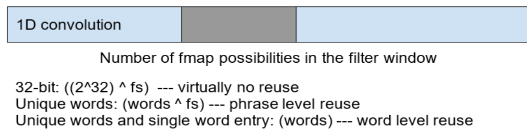

Figure 2: 1D Convolution in NLP CNNs

numbers, while each sentence's word in a sentence batch corresponds to an index into the embedded matrix. We then quantize our 32-bit floating point numbers into 32-bit integers. These values are loaded into our hardware's BRAM upon starting inference.

2. **Processing**
   We start by loading a single filter into a small L0 cache. Then we run 1D convolution on our batch of sentences. Each sentence is computed word-by-word. For each word, we first check if the result has already been computed and lies in our lookup table. If it has, we have our result. Otherwise, we must load the word's embedding from the BRAM buffer and take the dot product of the filter column and the embedding vector. Once this is completed for a single filter, the next filter is loaded and the process repeats.

3. **Post-processing**
   We send all our convolution results back to the general purpose processor or another accelerator, remembering to convert our values back into 32-bit floating point. For the particular software we tested, there is a global max pooling layer that is followed by logical regression to obtain the final decision.

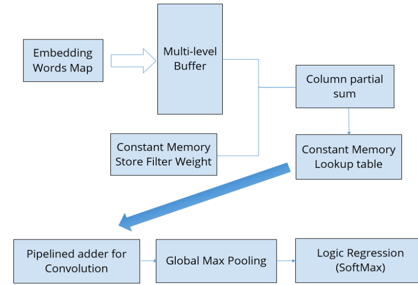

Figure 3: Design Flow

2 Background and Motivation

In this section, we provide some necessary background on how the CNNs are used to deal with the NLP problems which are needed for the rest of this paper.
Embedding Matrix: Embedding Matrix is used to map each word into an array in size $1 \times$ Dimension. These arrays show how words are related with each other. The Embedding Matrix we used for testing is pre-trained Glove[2] data set, which has 4000 unique words represented with 1×300 arrays, stored as 32-bit floating point numbers.
Basic CNNs Algorithm for NLP: The algorithm is proposed in the paper "ABCNN: Attention-Based Convolutional Neural Network."[3] The paper proposed to use CNNs to solve how to model the pair of sentences. Two sentences with the length of 50 unique words in maximum will first fetch the corresponding unique word entries in the Embedding Word Matrix, transforming into an 50×300 array respectively. Then each matrix will be fed into corresponding CNN layer. After that, Pooling layer and Logic Regression layer are used to get the final result. Because it is mentioned in the paper that increasing the number of layer doesn't increase the accuracy and the GlobalMaxPooling gives the better result, we just maintain our CNNs to be one layer with filters size $120 \times 300 \times 4$ and use GlobalMaxPooling for pooling layer.
Motivation: After performing the work analysis for the above algorithm, we found that the Convolution layer takes 72% of the total running time. And due to the relatively high reuse rate in the CNN for NLP, we tried to design the accelerator using the lookup table structure to reduce the number of calculation, thus saving the energy and improving the performance.

3 Design

In order to efficiently calculate the 1D convolution, past results are stored in a lookup table for future use. There are several ways to store these results, one considered method was to store a single value for a set of

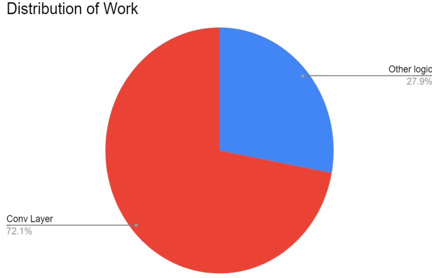

Figure 4: Distribution of Work

inputs. This type of reuse will be called "phrase level reuse" because using the lookup table requires that future inputs have a group of words in the same order as a past group of words. The advantage of this method is that no further computation is required for the convolution layer because final results are stored but not partial sums. The problem with this method is, even with a reduced amount of possible inputs, the amount of output possibilities grows too quickly for reuse to be the common case. For a common filter with a width of 4 and 4000 possible unique words there are 4 × 4000 output possibilities. If we assumed that the order of the words was random, this would be useless. However, because the words are not random and common phrases are often formed, this design might still be more efficient than the original software implementation for some test cases. Our next proposed design, which is the design we implement, gives a speedup for all test cases.

Instead of storing final results at a phrase granularity, we store partial results at the word granularity. By doing this we reduce the output sample space from 4 × 4000 to 4000. Unlike the previous method we are no longer storing final results, rather we are storing the partial results of the vector multiplied by a single filter column. Because of this, the amount of values that need to be stored per input is equal to the filter width. To be able to store every possibility for a single filter, ((unqiue words) × (filter width) × (32-bits)) is required. For the particular software algorithms we tested, this comes out to be 64 kB.

By storing results at a word granularity, the chance for reuse is greatly increased over the previous methods, but it is still not the common case. With the max sentence length of an input being 50 for the software we tested it is possible for every word to be unique and therefore the lookup table would have no use. In order to force reuse, the algorithm calculates a batch of 256 simultaneously. Since 256 × 50 is greater than 4000 unique possibilities, the pigeonhole principle tells us that there has to be reuse. For the worst case, 8800 out of the 12800 words, or 68.75%, of the inputs can be found in the lookup table. In reality, the amount of reuse is much higher because every batch only has a small subset of unique words.

Once a word is multiplied with the filter or looked up from the cache, it is to the output buffer which is a group of shift registers and adders. After the filter width-1 cycles a final result will be outputted for every for word input. This operation is fully in order, so that means that if a word is not in the lookup table the entire accelerator would need to stall and wait for the multipliers to calculate the result. To reduce the impact of the multiplier latency, we implement a SIMT-esque architecture.

The current design simultaneously looks at 4 sentences within the batch. Each of these sentences has its own output buffer. If the next word to be processed in any of the sentences is not in the lookup table, that word will be sent to the multiplier array if it is not currently in use. Simultaneously, if any of the following words be processed exists in the lookup table, one of the words will be found and sent to the appropriate output buffer.

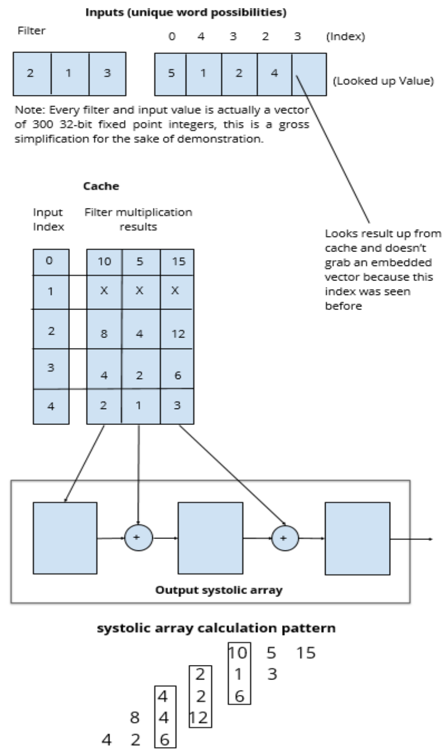

Figure 5: Design Detail

4 Implementation Details

4.1 In Depth Details

To simulate our accelerator, we used the Alveo U200 FPGA. Our design utilized 63% of the BRAM at 1350

blocks, but only 20% of the LUTs. Some of the LUTs are used by the array of 300 mults that calculate product of a filter and a vector in 4 cycles. However, a significant portion of the LUTs were used for creating the cache. The embedded buffer and filter buffer were made out of BRAM, however the cache was made out of LUTs to allow for an extreme multi-ported design in the future because BRAM in Xilinx FPGAs has the limitation of only being dual-ported. Clearly, this shows that our design requires a lot of local memory, but it doesn't need much hardware for computation. This is advantageous because the memory can be easily reused for other accelerators on the chip.
On this board we have a clock speed of 10 ns. This number could easily be improved with proper pipelining as most of our time was focused on the algorithmic and high level parts of our design. However, instead of pipelining to get a 2 or 3× improvement, it would most likely be far more beneficial to instead make the design more parallel. Of course these optimizations are not exclusive, but there is a massive amount of performance that can be unlocked with small tweaks to the cache.
As previously mentioned, the cache was made out of LUTs to allow for parallelism. Our current design can look at 4 different addresses, but only accesses 1 per cycle. Changing this design to a 4, 8, or even 16 ported lookup table would give a massive performance increase with a minimal increase in power consumption.

4.2 Power Consumption

To calculate software power consumption, we multiply the run time of convolution layer by the power of our CPU for implementation, which is 1.78s × 45W = 80.1J. Note that this number is just a rough estimation. The actual power consumption is lower than this.
Since we haven't implement on a real FPGA board, we are estimating hardware power consumption with Xilinx Vivado, which turns out to be an incredibly low 3.72W. Out the total power, 1.225W is dynamic power while 2.495W is static. Our low power consumption can be attributed to our large clock period as well as low data movement. The leakage doubling our dynamic consumption show just how dominant the memory is in our design. By multiplying the simulation time with this power, we have the energy consumption to be 0.288J.
By dividing these two energy consumption, we can see that the optimal energy saving is 99.64%, which means we can accomplish the same amount of work with 0.36% of the original energy.

5 Evaluation

We ran our accelerator using DeepLearn's [3]dataset "glove.6B.300d.txt". We verified our accelerator's values are consistent with the software implementation results. We run both software and hardware versions 3 times and average their respective runtimes. Overall, we observe 23 × improvement in performance, as shown in the following figures.(CPU Version: Intel 9th i7, 2.60GHz)

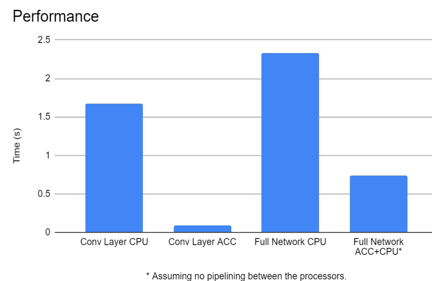

Figure 6: Performance Analysis

6 Future Work

For now, we've achieved 23× speed up in CNN layer and this definitely can be further improved. What we want to do next is getting more speed up by improving pipelining, increasing number of lookup tables and adjusting hyperparameters such as batch size, number of sentences executed in parallel, etc.
So far, we are simulating our design with the help of Xilinx Vivado. Implementing it on a real FPGA board will be our focus in the future. We intend to use Xilinx VU9P FPGA provided by Amazon web Services.

7 Conclusion

We designed an accelerator specific for solving the heavy workload in the convolution layer in NLP CNNs models. Using the lookup table and systolic array computing unit, our accelerator achieves 23× performance improvement and using 0.36% percent of original energy with the 0.1% loss in the convolution layer result due to the float to int transformation.

8 Team and Team Member Contributions

- Zachary Colter:
  Hardware verilog Design, Hardware Implementation debug, Paper Writing, Poster

- Wentao Zhang:
  Software Implementation, Perf, Hardware Implementation debug, Paper Writing, Poster
- Bryan Liu:
  Software Simulation Implementation, Hardware Implementation debug, Paper Writing, Poster
- Yuanqing Lou:
  Hardware Implementation debug, Paper Writing, Poster

References

[1] G. Bhatt, ABCNN: Attention-Based Convolutional Neural Network for Modeling Sentence Pairs (2017).
URL https://github.com/GauravBh1010tt/DeepLearn

[2] J. Pennington, R. Socher, C. D. Manning, Glove: Global vectors for word representation, in: Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1532–1543.
URL http://www.aclweb.org/anthology/D14-1162

[3] W. Yin, H. Schütze, B. Xiang, B. Zhou, Abcnn: Attention-based convolutional neural network for modeling sentence pairs, Transactions of the Association for Computational Linguistics 4 (2016) 259–272.