Scalable Low-Latency Persistent Neural Machine Translation on CPU Server with Multiple FPGAs

Eriko Nurvitadhi, Andrew Boutros, Prerna Budhkar, Ali Jafari, Dongup Kwon, David Sheffield,
Abirami Prabhakaran, Karthik Gururaj, Pranavi Appana, Mishali Naik
Intel Corporation. Email: eriko.nurvitadhi@intel.com

***Abstract*—We present a CPU server with multiple FPGAs that is purely software-programmable by a unified framework to enable *flexible* implementation of modern real-life complex AI that scales to large model size (100M+ parameters), while delivering real-time inference latency (~ms). Using multiple FPGAs, we scale by keeping a large model persistent in on-chip memories across FPGAs to avoid costly off-chip accesses. We study systems with 1 to 8 FPGAs for different devices: Intel® Arria® 10, Stratix® 10, and a research Stratix 10 with an AI chiplet. We present the first multi-FPGA evaluation of a complex NMT with bi-directional LSTMs, attention, and beam search. Our system scales well. Going from 1 to 8 FPGAs allows hosting ~8× larger model with only ~2× latency increase. A batch-1 inference for a 100M-parameter NMT on 8 Stratix 10 FPGAs takes only ~10 ms. This system offers 110× better latency than the only prior NMT work on FPGAs, which uses a high-end FPGA and stores the model off-chip.**

Keywords—AI, multi-FPGA server, neural machine translation

I. Introduction

Artificial intelligence (AI) is becoming pervasive. Cloud service providers now offer real-time intelligent services, such as natural language processing (NLP) services (e.g., Amazon Alexa, Apple Siri). Neural Machine Translation (NMT) is a popular deep learning (DL) approach for NLP. The ever-increasing demands of NMT in particular, and DL algorithms in general, pose a number of challenges for our current compute platforms on three main aspects: scalability, complexity and performance, as illustrated in Fig. 1a.

Firstly, as DL accuracy improves, the computational and memory demands increase substantially (e.g., modern NMT can have ~300M parameters). Moreover, many DL algorithms used in real-time AI applications – such as recurrent neural networks (RNNs), long short-term memories (LSTMs), and NMT – are *memory-bound with very limited data reuse and low compute-to-memory ratio* [2]. Therefore, this requires a scalable compute solution to host larger models, ideally in on-chip memories that are close to compute units to avoid expensive off-chip transfers.

Secondly, real-life AI applications are very complex, with a myriad of operations and irregular data movements. They go beyond the simpler RNN kernels, which are used as subroutines. Unfortunately, prior work mainly study only such kernels (e.g., [5]). This paper studies NMT, which uses an encoder and a decoder of stacked multi-layer bi-directional RNN/LSTM kernels along with additional *attention* mechanisms to learn alignments between speech frames [4] and *beam search* to keep track of multiple candidate translation sequences [3].

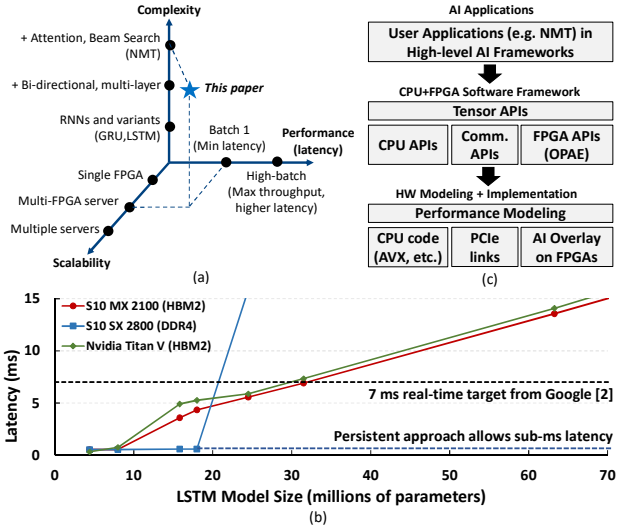

Fig. 1. (a) Challenges in real-time AI inference. This paper focus on highly complex NMT (b) Modern AI models beyond ~32M parameters cannot meet real-time target. (c) This work evaluates NMT on CPU+FPGAs framework.

Finally, inference using such complex and large DL still has to be performed under strict real-time latency constraints. As an example, Microsoft's Brainwave real-time AI inference strive for batch-1 to minimize latency [1], while Google TPU [2] targeted 7 ms maximum latency. Fig. 1b shows the estimated latency of running a 50-step LSTM with varying model size on Stratix 10 FPGAs with DDR4 and HBM memories, as well as a Titan V GPU. As models can no longer fit in on-chip memory, latency increases substantially due to off-chip memory transfers. This highlights that current single-node solutions cannot meet the 7 ms target latency for model sizes beyond 32M parameters.

To address the aforementioned complexity, scalability, and performance challenges of real-time AI, we propose an AI inference system combining a server CPU host with multiple FPGA cards that are software programmable through a unified software framework as shown in Fig. 1c. We evaluate systems with three FPGA devices (Arria 10, Stratix 10, experimental Stratix 10 with AI chiplet [6]), and scaling from 1 to 8 FPGAs. We show the effectiveness of our system on a real-life large and complex NMT applications. Our evaluation shows that this approach enables better scalability and easier software-programmability to handle complex NMT algorithm, while still offering extremely low latency (e.g., 10 ms for 100M parameters NMT with attention on a CPU server with 8 Stratix 10 FPGAs).

978-1-7281-2943-3/19/$31.00 ©2019 IEEE
DOI 10.1109/ICFPT47387.2019.00054

II. Background on Neural Machine Translation (NMT)

Neural Machine Translation (NMT) is a widely used DL model for language translation, which often uses RNN/LSTM as its building blocks. NMT takes an input sentence of a source language and produce a translated sentence sequence in another target language. The architecture of an RNN-based NMT with attention [4] and its pseudo-code is shown in Fig. 2.

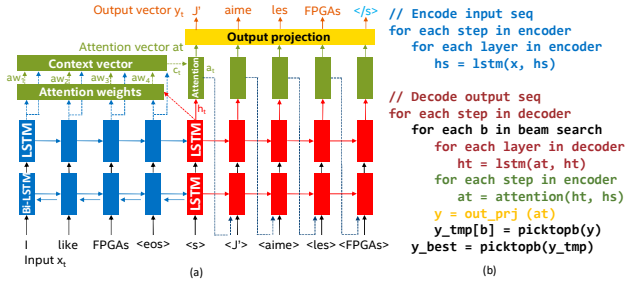

Fig. 2. NMT architecture illustration (a) and pseudo-code (b).

The *encoder* processes the input sequence and the *decoder* generate the output sequence. They are typically based on multi-layer RNN (or LSTM) and can use bi-directional cells to analyze the input sequence and its reverse to capture the relation between a word and its previous and subsequent words. *Attention* correlates historical step-by-step results from encoding the input sequence with each of the results of the output, which incurs a non-trivial amount of computation. *Output projection* takes the result of attention and apply further transformations, such as softmax and embedding, to output the final translation. More details of NMT can be found in [4]. A popular NMT variant uses *beam search* which calls the decoder multiple times to track the top k candidate output and choose the best one, where k is the beam search width. We study both NMT variants in this paper. NMT-Luong is a popular variant from [4], while NMT-Li is used in a prior NMT study on FPGA [3]. We show in our study that our system can flexibly support both.

III. CPU + Multi-FPGA System for AI Inference

Our system is shown in Fig. 1c. Combining CPU with FPGAs allows taming the complexity challenge by offloading only FPGA-friendly functions (e.g., matrix ops) to FPGA, while relying on general-purpose CPU for any algorithmic irregularities. To scale, we distribute a large AI model in on-chip memories across FPGAs (e.g., 224MBs in 8 Stratix 10s). This allows *scaling up* within a server, to complement *scaling out* across servers over Ethernet [1]. Finally, FPGA compute is placed right next to on-chip weight matrices to utilize the high on-chip memory bandwidth and deliver low matrix operation latency. Only vectors are transferred via PCIe between the CPU and FPGAs, which can be done in a few microseconds achieving low overall latency even as the system processes larger models.

We utilize PCIe slots in a CPU server within a rack to deploy multiple Intel FPGA Programmable acceleration cards (PACs) implementing a software-programmable overlay. We develop a unified software framework to program the CPU and FPGAs in C/C++, containing CPU and FPGA APIs for compute as well as communication and control routines. Any AI application can be implemented using a composition of these API calls, offering an agile AI inference framework and fast experimentation for application development and partitioning across the CPU and FPGAs. We also build a modeling tool to estimate the performance of an AI application written using these API calls.

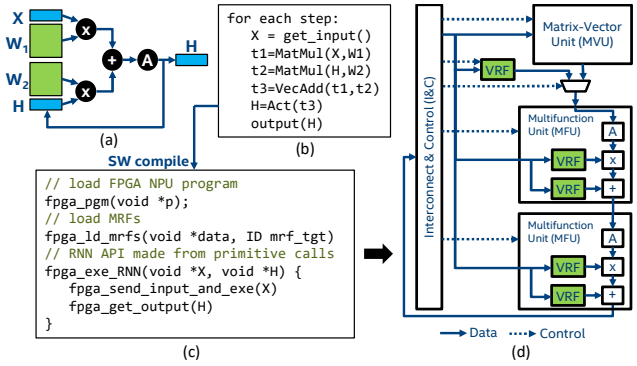

Fig. 3. The programming flow for our system: (a) RNN dataflow graph. (b) RNN overlay program. (c) FPGA software APIs. Each FPGA implemnets an overlay architecture (d). This study used NPU AI overlay in [1][6].

FPGA Overlay. The FPGAs in our system implement a software-programmable AI overlay. We used a faithfully replicated state-of-the-art overlay: Microsoft's Brainwave NPU [1] (Fig. 3d). The reader is referred to [1] and [6] for details of the NPU implementation. We used the NPU from [6], which has been shown to be comparable (and better in some workload scenarios) to the Brainwave NPU in [1]. Although this paper uses NPU, our approach is general to any FPGA overlay that offers software APIs for the target application domain.

TABLE I. Software APIs for CPU, FPGA, and Communication

CPU APIs	
`cpu_add(v1,v2)`	elementwise addition of two vectors
`cpu_mul(v1,v2)`	elementwise multiplication of two vectors
`cpu_reduce(v)`	reduces the elements of a vector into a scalar
`cpu_dot(v1,v2)`	dot product of two vectors
`cpu_axpy(a,v1,v2)`	vector scale and add $av_1 + v_2$
`cpu_tanh(v)`	vector elementwise hyperbolic tangent
`cpu_exp(v)`	vector elementwise natural exponential function
`cpu_softmax(v)`	vector softmax function $y = exp(v)/\sum_{i=0}^{S} v_i$
`cpu_matvec(v,m)`	vector-matrix multiplication
`cpu_pickTop(v,n)`	finds the largest n elements in a vector
FPGA APIs	
`fpga_matvec(v,m)`	vector-matrix multiplication
`fpga_add(v1,v2)`	elementwise addition of two vectors
`fpga_mul(v1,v2)`	elementwise multiplication of two vectors
`fpga_tanh(v)`	vector elementwise hyperbolic tangent
`fpga_sigmoid(v)`	vector elementwise sigmoid
`fpga_relu(v)`	vector elementwise rectified linear unit
Communication and Control APIs	
`fpga_pgm()`	loads an overlay program to a specified FPGA
`fpga_ld_input()`	transfers data from CPU to overlay input buffer
`fpga_ld_mrfs()`	initializes weights in the overlay MRFs
`fpga_execute()`	triggers overlay execution
`fpga_get_out()`	transfers data from overlay output buffer to CPU

Software APIs. We implement software APIs listed in Table I, for CPU and FPGA compute as well as communications. For CPU APIs, we utilize the best known methods (e.g., using AVX-512) to implement CPU compute kernels. We use Open Programmable Acceleration Engine (OPAE) (opae.github.io) framework to access the Intel FPGAs.

To program the NPU overlay, we first compile any given NPU program into a binary. Then, we use our software API `fpga_pgm()` to load the compiled binary to the NPU. We also implement other APIs to load states into the overlay, such as `fpga_ld_mrf()` and `fpga_ld_input()` to load weight matrices and input vectors to the NPU register files (MRF, VRF). These API calls are used to implement AI applications in our system. Fig. 3 illustrates the programming flow. An example RNN graph (Fig. 3a) is written as an overlay program (Fig. 3b), that is compiled to the overlay. We use other API calls to load vector X, trigger FPGA execution, and get output vector H (Fig. 3c).

Our APIs are extensible. We can create new APIs as long as they are expressible in the overlay ISA. Furthermore, as these APIs are written in C/C++, we can easily construct higher-level applications that incorporate other CPU API calls for operations that are not supported on or optimized for FPGAs.

System-level Performance Modeling Tool. Given an application described using a composition of the supported APIs, the modeling tool associates each API call with its argument and a previously measured or simulated proxy of a similar API call (e.g., previously measured data transfer time between CPU and FPGA or cycle-accurate RTL simulation of NPU compute time). From such piecewise information, the tool produces an aggregate performance estimate and highlights potential bottlenecks. The tool is useful to rapidly estimate the performance of a given target system, and for system-level exploration to find the optimal server configuration (i.e. number and type of FPGAs) to meet given performance constraints.

Problem partitioning across CPU and FPGAs. Problem partitioning is very important and affects operations distribution between the CPU and FPGAs, and thus achieved performance. Our software-programmable approach makes it easier to experiment with various partitioning schemes. In this paper, we manually experiment with different schemes to choose the best mapping for the NMT application server configurations studied.

IV. Evaluation

We study several generations of FPGAs: Arria 10 (A10), the latest Stratix 10 (S10), and an experimental Stratix 10 FPGA with a small AI ASIC chiplet (S10+TR) from Intel [6]. For the A10 and S10 NPU overlays, they run at 200 MHz with around 80% of the available DSP blocks and M20Ks utilized. When not explicitly stated, results are from the performance modeling tool. We validated multiple data points from our model against real hardware measurements on a prototype instance of our system, based on a modern Intel Xeon server with 4×A10 Intel FPGA PACs. Table II summarizes our experimental setup.

TABLE II. Summary of Experimental Setup

Server	Dell PowerEdge R740 Server, 2 sockets
CPU	Intel Xeon Gold 6132 2.6GHz CPU, TDP: 140W/socket
PACs	Arria 10 1150 (6MB on-chip, 20nm, TDP: 60W) Stratix 10 2800 (28MB on-chip, 14nm, TDP: 225W) Stratix 10 2100 + AI chiplet (74MB on-chip, 14/10nm)
Tools	Intel ICC (icpc) compiler version 19.0.2.187 Intel Quartus Prime Pro 18.0 OPAE software for FPGA on Xeon
Precisions	FP32 (CPU) INT8 (FPGA MatMul), INT27 (FPGA vector ops)
Benchmarks	Matrix-vector multiply, LSTM, NMT with Attention

API primitives. We characterized the CPU and communication APIs on our server. Fig. 4a shows the runtimes for the CPU vector operations. Even for 8K-element vectors, all operations can be done within microseconds. Fig. 4b shows the latency of data transfers between the CPU and N FPGAs for varying payload size. Transfers involving 1, 2, and 4 FPGAs take roughly the same latency, since each FPGA card has its own physical PCIe link and the transfers to multiple FPGAs can happen concurrently. For smaller payload sizes less than 32 KB, which are common for our vector transfers, sending/receiving data to/from four FPGAs takes only 1.2−1.4× higher latency than to/from a single FPGA, and can all be done in less than 8 µs. The biggest overlay program within our benchmarks is 2.9 KB and thus, our overlay can be programmed in less than 8 µs.

Scaling FPGA compute to multiple FPGAs. *1) Matrix-Vector Multiplication:* We studied largest square matrices that can be persistent across 1-8 FPGAs. Input vectors are sent to the FPGAs from CPU through PCIe. We studied these partitioning schemes: (a) split matrix by row blocks and concatenate the partial results, (b) split matrix by column blocks and reduce partial results, or (c) a combination of both. For R row blocks and C column blocks, an R × 1 configuration does not require any CPU reduction of partial vectors, while a 1×C configuration employs column-block scheme and hence requires (C − 1) CPU reductions . As shown in Fig. 4c, FPGA matrix operation is very efficient and accounts for only a small fraction of the runtime. Data transfers and CPU compute times dominate. Row-blocking performs better since it does not require CPU reductions. However, column-blocking may be needed to exploit compute parallelism for skewed matrices (i.e. short and wide matrices).

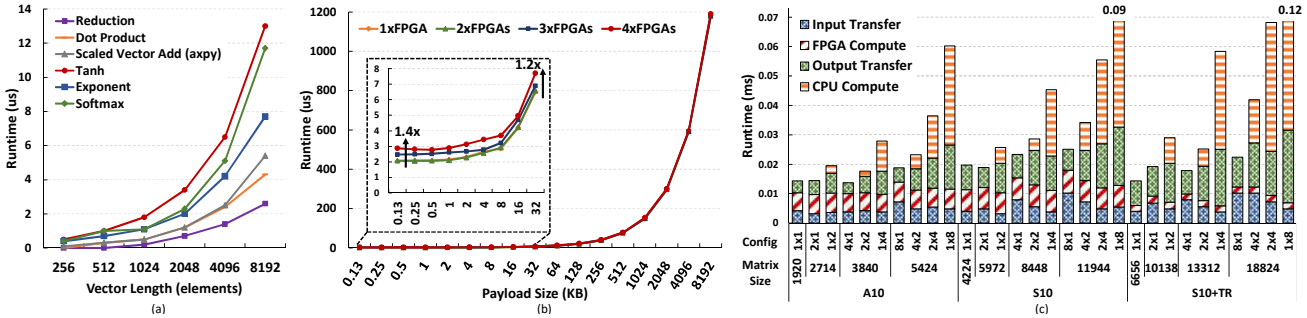

Fig. 4. Runtime results for: (a) CPU vector operations, (b) Data transfers between the CPU and 1 to 4 FPGAs, and (c) Matrix-vector kernels scaled across 1, 2, 4 and 8 FPGAs using different problem partitioning schemes.

Even with dominating transfer time, our system performance scales well, since transfer time scales sub-linearly with quadratically growing problem sizes and increasing number of FPGAs. Even at 11.9K matrix size with 136M elements, latency is less than 0.02 ms using the 8×1 S10 configuration.

2) LSTM: LSTM consists of 8 matrix-vector multiplications followed by several dependent vector operations. We study a partitioning scheme where each FPGA works on a complete LSTM subset with the CPU combining the results at the end of each iteration. (We also experimented with other partitioning schemes, but we will not discuss here for brevity). Fig. 5a shows the results scaling LSTM across multiple FPGAs in our system. We experiment with the largest LSTM we can keep persistent on 1, 2, 4 and 8 FPGAs. The figure shows that our system still scales well with multiple FPGAs even for LSTM. All studied LSTMs achieve below 2 ms latency, even when scaling to a 128M-parameter model on eight S10 FPGAs.

NMT Evaluation. We evaluate the popular NMT-Luong [4] and NMT-Li used in the only prior NMT on FPGA work [3].

1) Scaling Study: The NMT-Luong uses a 2-layer LSTM encoder with a bi-directional cell in its first layer, attention mechanism with general formulation for the score function, and a 2-layer LSTM decoder. We map both LSTM and matrix-vector kernels to FPGA while the other operations, including softmax, vector scaling, vector add, high-precision tanh activation and input/output embedding, are performed on CPU. We evaluate scaling of NMT-Luong on 1, 2, 4 and 8 FPGAs using the biggest model size we can keep persistent. Fig. 5b shows the results. Although this workload contains complicated communication patterns between CPU and FPGAs as well as a wide variety of CPU compute kernels, our system still achieves below 10 ms latency for NMT models with ~100M parameters processing 50-step input sequences on eight S10 cards.

Overall, latency increases slower than model size. We get only 2× higher latency when processing 8× bigger models scaling from one to eight FPGAs. This is because FPGA execution time scales constantly and communication scales sub-linearly, while CPU operations runtime scales linearly. Fig. 5c shows runtime breakdown for CPU, FPGA, and communication for the data points in Fig. 5b to further illustrate such behavior. As we move to a newer FPGA from A10 to S10 and S10+TR, NMT portions running on the FPGA become more efficient, while the CPU and communication become more dominant.

2) Comparison to Prior FPGA NMT Implementation [3]: Unlike NMT-Luong, NMT-Li [3] uses beam search with width 5, a single layer encoder and an attention with concatenation score function [4]. In [3], they use high-level synthesis (HLS) on a single Xilinx Ultrascale+ FPGA targeting FP32 precision. Despite that we use a mix of INT8 and FP32 precisions (shown to work well for NMT), the algorithm and the composition of operations are the same and comparable (we have verified it with the authors of [3]). We implement two variations of the NMT-Li model on a single S10 FPGA using the non-persistent approach similar to [3] in which we load the weight matrices through the host over PCIe or through the DDR4 external memory interface. Then, we scale to keep the whole model persistent with eight S10 PACs. Fig. 5d shows 1.2× better latency even with PCIe, which goes up to 7.4× when using DDR4 for the non-persistent single-FPGA implementation. For the persistent implementation on our system with eight S10 PACs, we achieve two orders of magnitude improvement compared to the 24 seconds latency in [3]. This experiment highlights the power of our agile software framework that allows us to rapidly experiment with different algorithmic variations by changing the sequence and composition of our API calls.

V. Conclusion

This paper proposes a CPU server with multiple FPGAs to provide low-latency AI inference scalable to larger models. The system is software programmable via a unified framework. We evaluate our system on modern complex NMT workload. The system scales well with increasing model sizes, where scaling from 1 to 8 FPGAs allows hosting 8× larger model, while only increasing latency by 2×. Further, software programmability allows rapidly implementing multiple NMT variants.

References

[1] J. Fowers *et al.*, "A Configurable Cloud-Scale DNN Processor for RealTime AI," in Int. Symposium on Computer Architeture (*ISCA*), 2018.

[2] N. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in Int. Symposium on Computer Arch. (*ISCA*), 2017.

[3] Q. Li *et al.*, "Implementing Neural Machine Translation with BiDirectional GRU and Attention Mechanism on FPGAs using HLS," in Asia & South Pacific Design Automation Conf. *(ASP-DAC)*, 2019.

[4] M.-T. Luong *et al.*, "Effective Approaches to Attention-Based Neural Machine Translation," Conference on Empirical Methods in Natural Language Processing (*EMNLP), arXiv:1508.04025*, 2015.

[5] E. Nurvitadhi *et al.*, "Accelerating Recurrent Neural Networks in Analytics Servers: Comparison of FPGA, CPU, GPU, and ASIC," in Field-Programmable Logic and Applications (*FPL)*, 2016.

[6] E. Nurvitadhi *et al.*, "Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs," in (*FCCM)*, 2019.

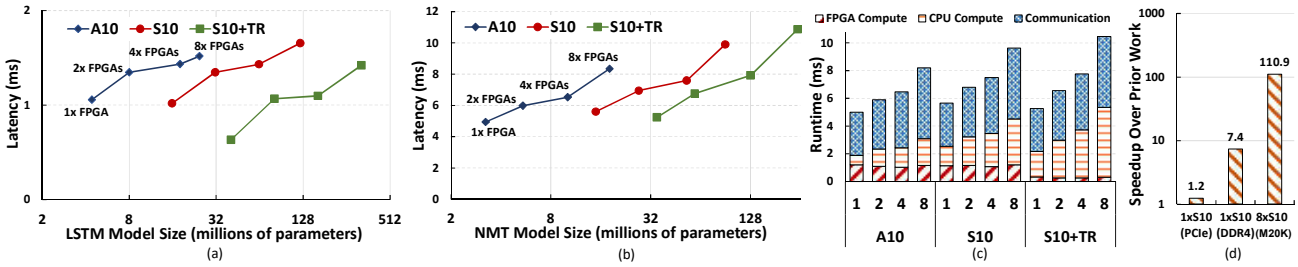

Fig. 5. Results of LSTM and NMT benchmarks on our proposed system: (a) LSTM and (b) NMT scaling on 1, 2, 4 and 8 FPGAs. (c) NMT runtime breakdown. (d) Comparison to prior NMT work [3]. A10 is Arria 10 1150, S10 is Stratix 10 2800, and S10+TR is Stratix 10 with TensorRAM chiplet [6].