A Programmable Accelerator for Streaming Automatic Speech Recognition on Edge Devices

Dennis Pinto
Universitat Politècnica de Catalunya
Barcelona, Spain
dpinto@ac.upc.edu

Jose-Maria Arnau
Universitat Politècnica de Catalunya
Barcelona, Spain
jarnau@ac.upc.edu

Antonio González
Universitat Politècnica de Catalunya
Barcelona, Spain
antonio@ac.upc.edu

***Abstract*—Automatic Speech Recognition (ASR) is quickly becoming a mainstream technology, mainly driven by the outstanding accuracy achieved by modern systems based on machine learning. However, these systems often require billions of arithmetic operations to decode a second of audio and relying on cloud services for ASR is usually inconvenient. Even though deployment of ASR systems directly on the edge is highly desirable, the requirements for high performance and low energy consumption, combined with the fast pace of evolution and heterogeneity of existing ASR systems, result in challenges for effective deployment of ASR on edge devices. In this work, we propose a programmable accelerator to efficiently support a variety of ASR implementations. We estimate the performance of our system by implementing a recently proposed streaming ASR system and show that it can perform real-time streaming decoding with a tight power budget and low area footprint while offering great flexibility to implement a variety of different models.**

***Keywords*-Programmable Accelerator, ASR, Speech Recognition**

I. INTRODUCTION

Voiced-based applications are quickly becoming mainstream. This widespread adoption is fueled by the outstanding improvement experienced by the *Automatic Speech Recognition (ASR)* systems that power them [1]–[3], [8].

Despite the impressive progress in automatic speech recognition technology, deployment of ASR systems on edge devices remains challenging and thus ASR systems are commonly deployed on servers. This approach is problematic due to high decoding latency and high energy consumption from the network subsystem of the edge device [5]. However, the biggest concerns come from the security and privacy issues related to sending personal data to external servers.

In order to move ASR to the edge, edge devices must provide enough computing power to perform expensive DNN inferences and graph searches. The compute power requirements can be fulfilled by including hardware accelerators in existing SoCs [6]. However, generic accelerators may not be enough to provide consistent performance and very specific accelerators may quickly become obsolete given the number of alternative implementations available for ASR and the fast pace of innovation in the ASR field.

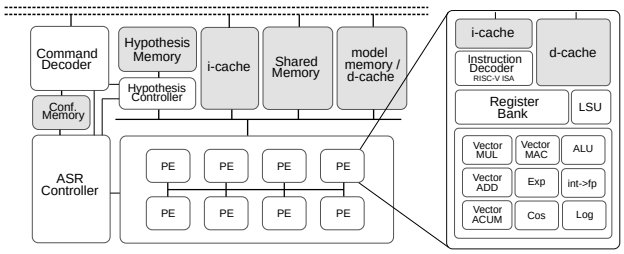

Figure 1. Diagram of the accelerator

In this work, we propose a programmable accelerator for ASR. A programmable accelerator can handle a large variety of different ASR implementations while providing enough compute power in an efficient manner through some degree of specialization. We explore a design that enables enough computing performance in a low-power setup for real-time streaming decoding. Furthermore, we provide a programming model that abstracts away some of the complexities of the system for ease of programming and flexibility.

II. ACCELERATOR

The accelerator, depicted in figure 1, relies on a pool of general-purpose cores (PE) to support parallel execution. Each core has a data cache and an instruction cache and implements a RISC-V ISA with extensions for vector operations and special functions.

A. Command Decoder

The accelerator is accessed through a set of commands. These include commands to set up the beam size of the hypothesis expansion, configure the kernels that implement the ASR system, start a decoding step and finish the current decoding process. The command decoder stores the required parameters in the ConfMemory, which is later accessed by the ASR controller during run time.

B. ASR Controller

Figure 2 shows the overall process of decoding an utterance with our accelerator. This process is controlled by the

ASR Controller. In streaming decoding, the signal is decoded in *decoding steps*. Each step decodes a few milliseconds of the audio signal. An external process captures an audio segment and commands the accelerator to start a decoding step. Each decoding step is divided into two stages: (1) the first stage is the *acoustic scoring*, which consists of producing acoustic score vectors by processing the raw audio segment; (2) after the acoustic scores are generated, the hypothesis expansion stage starts. That stage generates transcriptions hypotheses from the acoustic scores and additional models, such as lexicon and language models.

During acoustic scoring, the ASR controller launches a sequence of parallel kernels which collectively implement the feature extraction and acoustic model processes in the ASR system. These kernels access data from the shared memory, which can be used to store and retrieve intermediate results, and the model memory, which is used to pre-load model parameters, such as DNN weights. The kernels are launched sequentially, meaning that the next kernel will not start executing until all the threads from the current kernel have finished.

Every kernel consists of a kernel program and a setup program. The setup program reserves and frees space in the shared memory, configures the model memory DMA to pre-fetch model data and check whether there are enough inputs for the kernel program to execute or not. Figure 3 shows how the different threads are scheduled in the PE pool during acoustic scoring. Each square represents a PE executing a setup thread (yellow) or a kernel thread (blue). (1) First, the setup thread of kernel 0 is dispatched. It configures the DMA to load the model data for kernel 0 in model memory and waits for it to finish. (2) The execution of the following kernels (as_i in the figure) starts by dispatching the setup thread for the next kernel (as_{i+1}) alongside the kernel threads of as_i. (3) The ASR controller keeps dispatching as_i threads until the kernel is completely executed. If a setup thread determines that the corresponding thread cannot be launched (4), it will notify the controller. Additionally, it can pre-fetch the model data for kernel 0 to skip step 1 during the next decoding step. After the current kernel finishes (5), the controller will interrupt the decoding step and wait for the next decoding command, which will start a new decoding step from (1) or (2), depending on whether the model data for kernel 0 is pre-loaded or not. (6) The setup for the hypothesis expansion phase is launched alongside the threads for the last acoustic scoring kernel. Finally, when all the threads for the last acoustic scoring kernel finish (7), the accelerator ends the acoustic scoring phase.

After the acoustic scoring phase, the ASR controller enters the hypothesis expansion phase. For this phase, the programmer provides only one kernel. The controller dispatches as many threads of this kernel as the number of hypotheses that resulted from the previous hypothesis expansion phase. Each thread reads a hypothesis and appends to it every possible

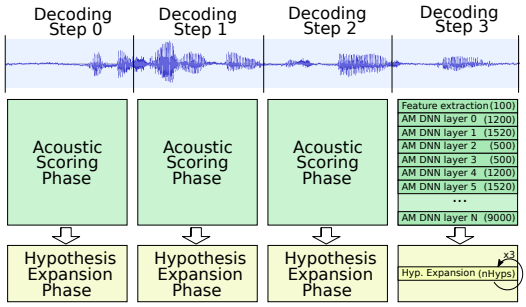

Figure 2. ASR decoding in the proposed accelerator

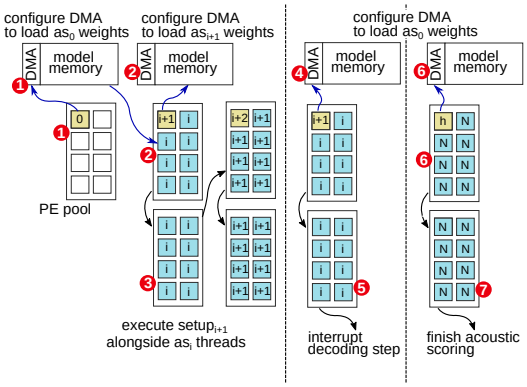

Figure 3. Scheduling of kernel and setup threads in the PE pool

acoustic token to generate new hypotheses. During this stage, the model memory acts as a data cache to leverage the existing locality in the access to the graph structures [7]. The setup program of the hypothesis expansion kernel determines how many hypothesis expansions must be performed. This is useful in the case that the acoustic scoring phase produced more than one scoring vector.

C. Hypothesis Controller

The hypothesis expansion threads send the generated hypotheses to the hypothesis controller, which sorts them according to their score and prunes them according to the beam width and the hardware maximum number of threads. This controller stores the received hypotheses in the Hypothesis memory. The non-pruned hypotheses are kept there in-between decoding steps. Hypothesis expansion threads access the hypothesis memory through the hypothesis controller.

III. Results

We configure the architecture with 8 cores, each containing $4KB$ of i-cache and $24KB$ of d-cache. The vector units are of width 4. Outside of the cores, we include $1MB$ of prefetch buffer/d-cache, $512KB$ of shared memory, $64KB$ of i-cache and $24KB$ of hypothesis memory. This configuration results in around 12mm² at 22nm (64% of which is dedicated for the core pool) and provides a peak

performance of 32GMAC/s at 500MHz. We estimate the peak power at about 1.8W

To estimate performance, we implement a recent end-to-end CTC-based ASR system [4]. It consists of a TDS network that extracts acoustic scores from MFCC features. The hypotheses are expanded by traversing a lexicon tree and a graph-based language model.

The acoustic scoring phase consists of 1 kernel to compute MFCC features and 79 kernels (18 convolutions, 29 fully-connected layers and 32 LayerNorms) to implement the TDS network, each preceded by its corresponding setup thread. We only implement each type once and reuse the kernel code as explained in section II-B.

The ASR system generates 100 MFCC frames per second of audio and the TDS network applies a sub-sampling factor of 8 to the input. This means that decoding a second of audio requires 13 decoding steps. We estimate that it takes about 520ms to decode a second of audio in the proposed accelerator, which exceeds real-time performance. We also estimate that the average power dissipation is slightly over 1W.

ACKNOWLEDGMENT

This work has been supported by the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency (MCIN/AEI) under grant PID2020-113172RB-I00, the ICREA Academia program and the Spanish MICINN Ministry under grant BES-2017-080605.

REFERENCES

[1] "Speech recognition on librispeech test-clean," https://paperswithcode.com/sota/speech-recognition-on-librispeech-test-clean, [Online; accessed 29-Oct-2021].

[2] "Speech and voice recognition market size, share & industry analysis, by component (solution, services), by technology (voice recognition, speech recognition), by deployment (on-premises, cloud), by end-user (healthcare, it and telecommunications, automotive, bfsi, government, legal, retail, travel and hospitality and others) and regional forecast, 2019 – 2026," 2019.

[3] Y.-A. Chung, Y. Zhang, W. Han, C.-C. Chiu, J. Qin, R. Pang, and Y. Wu, "W2v-bert: Combining contrastive learning and masked language modeling for self-supervised speech pre-training," *arXiv preprint arXiv:2108.06209*, 2021.

[4] A. Hannun, A. Lee, Q. Xu, and R. Collobert, "Sequence-to-sequence speech recognition with time-depth separable convolutions," *arXiv preprint arXiv:1904.02619*, 2019.

[5] G. P. Perrucci, F. H. Fitzek, and J. Widmer, "Survey on energy consumption entities on the smartphone platform," in *2011 IEEE 73rd vehicular technology conference (VTC Spring)*. IEEE, 2011, pp. 1–6.

[6] D. Pinto, J.-M. Arnau, and A. González, "Design and evaluation of an ultra low-power human-quality speech recognition system," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–19, 2020.

[7] R. Yazdani, A. Segura, J.-M. Arnau, and A. Gonzalez, "An ultra low-power hardware accelerator for automatic speech recognition," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[8] Y. Zhang, J. Qin, D. S. Park, W. Han, C.-C. Chiu, R. Pang, Q. V. Le, and Y. Wu, "Pushing the limits of semi-supervised learning for automatic speech recognition," *arXiv preprint arXiv:2010.10504*, 2020.