ASRPU: A Programmable Accelerator for Low-Power Automatic Speech Recognition

Dennis Pinto, Jose-María Arnau, and Antonio González, Fellow, IEEE

Abstract—The outstanding accuracy achieved by modern Automatic Speech Recognition (ASR) systems is enabling them to quickly become a mainstream technology. ASR is essential for many applications, such as speech-based assistants, dictation systems and real-time language translation. However, highly accurate ASR systems are computationally expensive, requiring on the order of billions of arithmetic operations to decode each second of audio, which conflicts with a growing interest in deploying ASR on edge devices. On these devices, hardware acceleration is key for achieving acceptable performance. However, ASR is a rich and fast-changing field, and thus, any overly specialized hardware accelerator may quickly become obsolete.

In this paper, we tackle those challenges by proposing ASRPU, a programmable accelerator for on-edge ASR. ASRPU contains a pool of general-purpose cores that execute small pieces of parallel code. Each of these programs computes one part of the overall decoder (e.g., a layer in a neural network). The accelerator automates some carefully chosen parts of the decoder to simplify the programming without sacrificing generality. We provide an analysis of a modern ASR system implemented on ASRPU and show that this architecture can achieve real-time decoding with a very low power budget.

Index Terms—Parallel Architectures, Machine Learning, Automatic Speech Recognition, Parallel Architectures, Machine Learning, Automatic Speech Recognition, Real-time On-edge ASR.

1 INTRODUCTION

A UTOMATIC speech recognition (ASR) consists of processing an audio signal (utterance) to obtain a written transcription. Figure 1 shows the overall ASR process. First, the audio signal is broken down into overlapping fragments and transformed into a sequence of feature frames. Then, each of these frames is classified into acoustic tokens by an Acoustic Model (AM). For most systems, the AM is a Deep Neural Network (DNN) whose output is a vector of probabilities over phonetic units. The last stage, decoding, generates a transcription from the acoustic scores. The simplest approach for obtaining a transcription consists of selecting the phonetic unit with the highest probability on each frame. However, using more sophisticated approaches, e.g., integrating a lexicon and a language model, generally result in better accuracy. When the ASR system includes a lexicon or a language model, the best-scoring transcription cannot be obtained by simply taking the single best-scoring phonetic unit on each frame. Instead, a search algorithm, such as Viterbi Beam Search, traverses the graph of possible transcriptions (lattice) to find the sequence of phonetic units with the best overall score.

ASR systems are key components in many game-changing technologies such as automatic language translation [1] and virtual assistants [2, 3, 4] as well as many others like dictation [5], automatic captioning [6], hands-free computing [7, 8], pronunciation evaluation for language learning [9] and many more.

To a higher or lesser degree, all of those applications require that the ASR system provides high-quality transcription while allowing the user to speak naturally, something that past ASR systems could not deliver. Modern ASR systems, however, are starting to excel at that. Figure 2 illustrates how fast ASR accuracy has improved over the last few years. The plot shows the Word Error Rate (WER) of different ASR systems, as reported in various papers published between 2016 and 2021 for librispeech, a large-vocabulary, multi-user continuous speech recognition benchmark. The outstanding improvement is apparent by comparing a 2016 system, DeepSpeech2 [10], with the best 2021 system [11]. In just 5 years, the WER was reduced from 5.33% to 1.4% in test_clean, the basic librispeech task, and from 13.5% to 1.7% in test_other, a librispeech task containing only challenging utterances. For reference, the WER of humans is estimated to be around 5% for test_clean and 13% for test_other [10].

This level of accuracy opens the door for many mainstream uses, as witnessed by the proliferation during the last decade of consumer products based on ASR. It comes, however, at the cost of performing inferences with huge models that require on the order of billions of arithmetic operations per second of speech and expensive searches in large graphs, such as lexicon and language models graphs.
Another challenge for on-edge ASR comes from the wide diversity of ASR systems. ASR systems come in one of two major flavors: Hybrid DNN-HMM systems and End-to-End systems. The former relies on a statistical model called Hidden Markov Model to identify and chain together individual spoken units, and a DNN to generate the inputs required by the HMM from the audio frames, whereas the latter consists of a Deep Neural Network (DNN) that classifies the audio frames into spoken units, generating a valid transcription directly. Additionally, End-to-end systems are often subclassified depending on whether they are Connectionist Temporal Classification (CTC) or seq2seq systems.

Furthermore, ASR systems are evolving rapidly. The impressive drop in transcription errors showcased in figure 2 is possible thanks to the abundant innovations proposed during the last few years. ASR systems are constantly changing to incorporate these innovations. Consequently, any overly specialized chip to accelerate ASR will likely become obsolete at once.

The consequence of these three factors: high computational cost, vast heterogeneity and fast pace of innovation, is that ASR is usually performed on servers rather than on edge devices. Edge devices, such as smartphones and smart appliances are often ill-equipped to perform highly accurate ASR within reasonable latency. In contrast, servers provide more than enough computing power for the task. Furthermore, hardware and software in servers can be easily updated and so the companies can always guarantee state-of-the-art ASR to their clients. This is more challenging if ASR is deployed on edge devices. Edge devices generally require hardware acceleration to provide highly accurate decoding within reasonably latency, but too specialized hardware is likely to become obsolete rather quickly, leaving users stuck with sub-par ASR until they update the device.

Despite the advantages of servers for ASR, on-edge ASR is the preferred solution for the long term. Service availability and low latency requirements are difficult or plain impossible to guarantee when ASR is provided as a cloud service. Even more important are privacy issues that arise when sending sensitive audio data to company-owned servers.

Acknowledging the clear advantages of ASR on the edge, we propose ASRPU, a processing unit for ASR, to tackle the challenges of executing ASR on edge devices. This accelerator is built around a pool of general-purpose cores, which gives the programmer flexibility to write alternative ASR implementations and perform software updates and optimizations as needed. The pool of cores is supported by an ASR controller and a unit that sorts and prunes hypotheses to automatize as much of the ASR process as possible without removing much flexibility. The accelerator also contains a specialized memory hierarchy adapted to the memory requirements of ASR. As a result, ASRPU provides enough flexibility to implement most of the current (and probably future) ASR systems with a simple and comprehensive API, enabling real-time state-of-the-art ASR on the edge.

Even though there is a plethora of accelerators for DNNs and ASR, including examples and alternative algorithms. Section 2 provides background on Automatic Speech Recognition, introducing most of the important concepts and algorithms related to ASR, including Automatic Speech Recognition, Section 3 is a detailed description of ASRPU. Section 4 describes the detailed implementation of a modern ASR system on ASRPU as an example. Section 5 presents our estimations of the performance and power consumption of a low-power configuration of ASRPU executing the ASR system introduced in section 4. Finally, section 6 contains a summary of the conclusions and a discussion on both the limitations of this work and possible directions for future work.

2 Automatic Speech Recognition

As previously stated, the purpose of ASR is to obtain a written transcription from an utterance. Most current ASR systems do it by following the same overall algorithm consisting of three steps: (1) Feature extraction, (2) Acoustic Scoring and (3) Decoding. Feature extraction receives the raw signal and generates a sequence of feature frames. These feature frames represent the signal in a convenient way, making the identification of phonetic units (e.g. phonemes) easier. The acoustic scoring step receives the feature frames and, employing an acoustic model, identifies the phonetic unit contained within each of the feature frames. The acoustic model generates, for each feature frame, a probability distribution over phonetic units. The decoding step combines the acoustic scores with scores from other sources (such as language models) to generate transcription hypotheses. When all the acoustic score frames are incorporated in the hypotheses by the decoder, the overall best hypothesis is regarded as the final transcription for the input utterance.

Despite the homogeneity implicit in the previous description, there are many alternative approaches to implement those processes. Feature extraction usually consists of a Mel-Frequency Cepstral Coefficients (MFCC) extractor preceded by a signal processing step that enhances certain
2.1 Feature Extraction

Feature Extraction is the first component of most ASR systems. It receives the raw signal (a sequence of amplitude values) and generates a sequence of feature vectors. These are usually MFCC features, which are computed roughly as follows (figure 3): the signal is broken into overlapping frames, usually, 25ms frames shifted by 10ms, then, a Fourier Transform is applied on the signal segments to convert them into the frequency domain. The frames in the frequency domain are mapped to the mel scale using overlapping triangular windows (mel filterbanks). Usually, 80 of the resulting filterbanks compose a feature frame, whereas the rest are discarded. Finally, a cosine transform is applied to the element-wise log of the resulting frames. Dynamic characteristics of the signal and reduces noise. However, MFCCs are not the only features used for ASR, and signal pre-processing can be performed in a variety of ways. The second process, acoustic scoring, usually consists on a DNN inference, but there is a wide variety of DNNs used among ASR systems (e.g. TDNN [34], TDS [35], DeepSpeech [10], LAS [23] and Transformer [20]) and each comes with its own peculiarities. Decoding too can be performed in a variety of ways. Hybrid systems, such as [11], [35] do it by traversing a complex graph, which contains transition probabilities between a heterogeneous set of symbols, such as tri-phones, phonemes and words; end-to-end systems rely on simpler graphs. For example, wav2letter [19] systems employ a tree structure of phonemes, word-pieces or words. This structure serves to limit the symbols to which each symbol can transition. However, this graph does not contain scores. Additionally, a language model graph or DNN can be included in the systems. The language model consists of a graph that contains scores that represent transition probabilities between words.

The following sections provide a more detailed description of the common components of ASR systems and the alternative algorithms.

2.2 Acoustic Scoring

The feature frames produced by the feature extraction process are processed by the acoustic model to generate a sequence of score vectors, usually one vector per feature frame. Depending on the system, these vectors may contain probabilities or log-likelihoods. The set of acoustic tokens is defined by the ASR designer. Many recent systems use word pieces, but characters and phonemes are not rare.

Acoustic scoring is essentially a classification problem, so it is most commonly implemented as a DNN inference. However, Neural Networks is a broad and fast changing domain, with many architectures to chose from and new ones proposed every year. Common architectures include: Time Delay Neural Network (TDNN) [34], ResNet [39] and Time-Depth Separable (TDS) [40] networks (figure 4).

2.2.1 Acoustic Tokens

The acoustic model classifies each feature vector by generating a vector of probabilities over Acoustic tokens. On Hybrid systems [38], [41], the acoustic tokens are HMM states. A WFST graph maps these tokens to phonemes and finally to words. End-to-End systems do not use HMM. Instead, on those systems, feature vectors are directly classified into phonemes, characteristics or word-pieces [11], [42]. The latter, word-pieces, are currently among the most popular type of acoustic tokens for end-to-end systems. They consist of arbitrary pieces of words obtained by an optimization algorithm that takes words from a text corpus and breaks them apart on different parts to minimize a target cost function.

The selection of acoustic tokens influences the decoding phase. Word pieces and characters can be decoded with very simple decoding graphs or even without any graph. HMMs...
2.3 Decoding

Once the sequence of acoustic scores is generated, they are consumed during the decoding stage to generate transcriptions hypotheses. This can be so simple as taking, for each frame, the phonetic unit with a higher score or as complicated as processing a large decoding graph that combines a Hidden Markov Model with additional graphs that model pronunciation and grammar.

2.3.1 Decoding in Hybrid ASR

Decoding in hybrid systems is performed by traversing a decoding graph while consuming acoustic score frames, looking for the best scoring path. The principal characteristic of hybrid systems is the inclusion of Hidden Markov Models (HMM) [43], a set of weighted directed graphs representing acoustic units (usually tri-phones). HMMs are represented as Weighted Finite-State Transducers (WFST) [44] and then merged with a Lexicon, i.e. a pronunciation model, and a Grammar, i.e. a language model. Additionally, a Context dependency graph is included in the mix to make the HMMs compatible with the Lexicon. These HMMs are trained on labelled utterances to learn how to align the HMM nodes to the acoustic vectors, i.e. which nodes should have a better score for which acoustic vectors. The acoustic model DNN is trained on aligned utterances to learn and generalize the alignments learnt by the HMM. The graph resulting from combining the HMM, context-dependency, lexicon and grammar graphs is called HCLG graph and is the standard decoding graph for Hybrid ASR systems.

To traverse the HCLG graph, the Viterbi Beam Search algorithm is used. This algorithm starts from a special start node. At each step, the algorithm checks all the nodes reachable from the set of active nodes, computing the score of the resulting paths (from the start node). The nodes with scores within the beam, a threshold computed from the score of the best node, will compose the active set for the next traverse step. The process finishes when all the acoustic vectors are consumed. To backtrack the best path once the algorithm ends, after each traverse step, for every active node, a pointer to the parent node is recorded. If a node was reachable from several parent nodes, all but the best scoring are discarded.

2.3.2 Decoding in end-to-end ASR

End-to-End ASR does not include an HMM to model acoustic units. Instead, the acoustic model DNN is trained from scratch to learn how to align the utterances. One of the consequences in this model is that tri-phones are not necessary, and consequently, neither are context-dependency nor lexicon. The acoustic model DNN in these systems processes feature vectors and generates scores over high-level phonetic units, such as characters or word-pieces.

The most straightforward algorithm consists in taking, for each frame, the best scoring phonetic unit. However, this approach often results in poor accuracy. Introducing a lexicon and a language model restricts the possible paths, leading to more compelling results.

2.4 Streaming Decoding

Streaming decoding (sometimes called online decoding) refers to the process of decoding the speech in real-time while it is generated and, hence, not having the entire utterance available. In streaming decoding, frames are decoded one-by-one, or in a batch of enough input frames to generate one output. In contrast, in non-streaming, or offline decoding the entire input utterance is available at the start of the decoding process. This distinction is important because the difference in decoding latency among the two options may be significant. streaming decoding generates partial transcriptions in real-time with very low latency, whereas offline decoding will generate the complete transcription after the speaker has finished an utterance plus some delay.

If the ASR is executed locally (on-edge), streaming decoding is desirable in order to grant the best user experience, e.g. immediate feedback and early detection of transcription mistakes are likely important advantages from the point of view of the user. However, these advantages come at a cost. During streaming decoding, only a small number of input frames are available, which means that there is less potential for data reuse during DNN inference. This is critical during the execution of fully-connected layers. These layers usually contain a huge number of parameters, which are used just once per input frame. Consequently, the parameters from fully connected layers will have to be accessed from external memory many times, incurring higher energy costs.

3 Architecture of ASRPU

Despite the differences among ASR systems, most of them follow a similar overall algorithm. We leverage that to design an accelerator that provides enough flexibility to support the differences between them while automating wherever possible to speed up the ASR process and simplify the software implementation.

This section provides a detailed description of the architecture of ASRPU. The accelerator (Figure 5) is divided into 3 major blocks: Command decoder, Execution unit and Hypotheses unit. The command decoder provides the interface to the accelerator via a set of commands. This includes commands to start an ASR decoding step, to finish decoding an utterance and to configure different parameters of the accelerator. The execution unit executes the program that implements the ASR system. This program is composed of small sub-programs, kernels, written by the ASR designer to implement each part of the ASR system. The execution unit contains a pool of Processing Elements (PE) to execute the code of the kernels. The hypothesis unit sorts and prunes transcription hypotheses. It also keeps them in memory from one decoding step to the following.
In this example, the last kernel implements a DNN layer with 9000 neurons. Each neuron computing the score for one of the 9000 phonetic units modelled by the acoustic model. Because of how the kernel is written, each thread computes a single neuron.

After the acoustic scoring phase concludes, ASRPU switches to the hypothesis expansion phase. During the hypothesis expansion phase, the accelerator executes only one kernel, Hypothesis expansion. Each thread of this kernel is responsible of expanding a single hypothesis. The expansion of an hypothesis generally results in many output hypotheses, which are generated according to the specific decoding algorithm. Depending on the implementation, the acoustic scoring phase can generate one or more acoustic vectors. During hypothesis expansion, the accelerator executes the hypothesis expansion kernel once per acoustic vector. In the example system of the figure, the accelerator launches nHyps threads (determined in run-time) of the hypothesis expansion kernel. The self-referencing arrow indicates that the kernel is executed three times. This number will also depend on the implementation. For example, the feature extraction kernel may extract three frames on each decoding step, resulting in three repetitions of the hypothesis expansion kernel. Some DNNs, particularly convolutional DNNs, apply sub-sampling during acoustic scoring, meaning that they generate less acoustic vectors than feature frames. In this case, six feature frames will result in three acoustic vectors if the DNN AM apply a sub-sampling of two frames.

If ASRPU is integrated in an SoC that also contains a CPU, there may be an external process responsible of capturing the signal as it is produced. This process communicates with the accelerator, starting decoding steps after capturing enough values from the microphone.

### 3.1 Decoding on ASRPU

Figure 6 illustrates the overall process of decoding an utterance in ASRPU with an example ASR system. The decoding process in the accelerator is divided in Decoding Steps. Each step decodes a portion of the signal, extracting feature frames, computing acoustic scores and finally expanding the hypotheses left from the preceding decoding step. We divide each decoding step in two phases: (1) The Acoustic Scoring phase and (2) the Hypothesis Expansion phase.

As previously mentioned, a set of kernels implement every component of the ASR system. The acoustic scoring phase consist of the sequential execution of most of these kernels (except for the last one). These kernels implement the feature extraction algorithm and the acoustic model. The example of the figure shows a sequence of N+1 kernels executed within the acoustic scoring phase. These kernels are executed sequentially on the accelerator. However, they consist of parallel code. The execution of each kernel is carried on by the execution unit, which launches as many threads of the kernel code as required on the PEs. The number inside the parenthesis shows the number of threads required by each kernel. The first kernel implements feature extraction (which may include code for signal preprocessing) and requires 100 threads. Subsequent kernels implement each a layer of a DNN AM, each requiring a different number of threads. The last kernel requires 9000 threads, which is entirely dependant on the implementation.

### 3.2 Setup Thread

The ASR designer can include a special setup program along each of the acoustic scoring and hypothesis expansion kernels. That is, each kernel is complemented with a setup program. This setup program is executed to completion in a single thread before the associated kernel can start executing.

These setup programs provide the accelerator with greater flexibility. For example, the setup program for a specific kernel that implements a convolutional layer of a DNN can determine how many outputs can be computed from the available inputs and notify the hardware to launch the appropriate number of kernel threads so as to maximize data reuse. The setup program associated to the hypothesis expansion kernel can access the number of outputs generated by the acoustic scoring phase and notify it to the hardware so it executes the hypothesis expansion kernels as many times as necessary. In both cases, if the available inputs are not enough to compute even a single output, the setup thread can notify to the accelerator to stop the decoding step. The following section provides more details about this process.

These setup program can also be used to manage the input and output buffers of the kernels in shared memory. Each kernel will generally read inputs from an input buffer and store outputs in an output buffer (in shared memory).
Before executing each kernel, the associated setup thread will first determine the number of outputs that can be generated from the inputs available in the input buffer. Then, it will remove from the input buffer those inputs that can not be further reused and reserve space in the output buffer for the new outputs. Finally, before finishing, it notifies the hardware the required number of kernel threads. After the setup thread finishes, the accelerator launches the required number of kernel threads.

Another advantage of the setup threads is that they allow to reuse code among different kernels. Generally, DNNs contain many convolutional and fully-connected layers. The ASR designer can write a single convolutional and a single fully-connected parametric routines. All the convolutional and fully-connected layers can be configured to execute the same kernels and the associated setup threads will set the appropriate parameters in shared memory before executing the kernel.

### 3.3 Execution Unit

The execution unit consists of a pool of PEs and an ASR controller. The ASR controller handles the overall decoding procedure. It first waits until the command decoder receives a new commit signal. At that moment, it starts a decoding step. First, the controller reads from the Configuration memory the address of the first setup program and configures a PE to execute it by setting its program counter. Once the setup thread finishes executing, it notifies to the ASR controller the required number of kernel threads. The ASR controller then starts dispatching kernel threads to idle PEs. Every time a PE becomes idle, it notifies the ASR controller, which reacts by dispatching a new thread to the PE, until there are no more threads to dispatch. When the last thread finishes, the ASR controller repeats the same procedure for the subsequent kernel.

As mentioned in section 3.2 if a setup thread returns a value of zero, the ASR controller stops the decoding step. This is meant to be used when a program is not ready to be launched, usually when there are not enough inputs to compute even a single output. For example, a convolutional layer with a window of ten frames will check during setup time (during the execution of the setup thread) how many inputs there are available, computing and returning an appropriate number of threads. If there are less than ten inputs, it will return zero notifying the ASR controller to stop the decoding step.

After all the programs in the Acoustic Scoring sequence have been executed, the decoder starts the hypothesis expansion phase. It first accesses the number of active hypotheses, provided by the hypothesis unit, and launches a thread for each active hypothesis. These threads will execute the code from the hypothesis expansion kernel. The setup thread of the hypothesis expansion kernel will determine how many outputs were generated by the acoustic scoring phase and notify the ASR controller to execute the hypothesis expansion kernel that number of times.

Figure 7 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). 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. The execution of the following kernels \( (a_{s_{i}} \text{ in the figure}) \) starts by dispatching the setup thread for the next kernel \( (a_{s_{i+1}} \text{ alongside the kernel threads of } a_{s_{i}} \text{).} \) The ASR controller keeps dispatching \( a_{s_{i}} \text{ threads until the kernel is completely executed. If a setup thread determines that the corresponding thread cannot be launched it will notify the controller. Additionally, it can pre-fetch the model data for kernel 0 to skip step during the next decoding step. After the current kernel finishes, the controller will interrupt the decoding step and wait for the next decoding command, which will start a new decoding step from or depending on whether the model data for kernel 0 is pre-loaded or not. 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, the accelerator ends the acoustic scoring phase.

### 3.4 Processing Elements

The Processing Element (PE) pool contains a number of programmable and independent PEs (i.e. cores). Each PE, shown in figure 8, implements a general-purpose RISC-V ISA.

The ISA includes extensions for additional operations, such as a vector Multiply and Accumulate (MAC). This operation receives three operands, the first operand is a 32-bit value that carries the accumulation between MAC operations. The other two operands are vectors of 8-bit values. These operands are multiplied element-wise and accumulated. The result is added to the first operand. It also includes vector multiplication and additions, along with special function units to compute logarithms, exponential and cosine functions, usually required during feature extraction, the activation function in neural network layers and for the computation of the hypothesis score during hypothesis expansion. Each PE contains data and instruction caches. These are regular caches managed by the hardware. Each PE also contains a register bank with 2 sets of registers: 32-bit floating-point registers, which are used as operands for
the FP ALU and the special function units, and vectors of 8-bit values used as operands for the vector operations. PEs are connected to the Hypothesis unit, the shared memory and the shared caches through a bus. Another bus connects all the PEs to the ASR controller. This bus is used by the ASR controller to configure the PEs and by the PEs to notify values to the ASR controller.

3.5 Hypothesis unit

The hypothesis unit contains a hypothesis memory and a controller. During any decoding step, the active hypothesis and the newly generated hypotheses reside inside the hypothesis memory. This unit is connected to the internal bus and accessed via a special memory address from the PEs. Hypothesis Expansion threads send hypotheses to the hypothesis controller. Each hypothesis is a data structure that contains some fields. These fields include a hash to identify the hypothesis, the hypothesis score, and others defined by the programmer. The hypothesis unit sorts and prunes them according to their score field and the beam score. The score beam is configured beforehand via configuration commands.

3.6 Memory Hierarchy

Each PE contains data and an instruction cache. Outside the PE pool, there are shared instruction and data caches too. The global data performs two different functions. During acoustic scoring, this memory stores model weights that were pre-fetched beforehand. This maximizes data reuse and hides the latency to access external memory. During hypothesis expansion this scheme would not be of much use. The graph structures used by the hypothesis expansion algorithms are generally in the order of hundreds of MB or even GB, much larger than what is reasonable to store in a low power accelerator. Additionally, the threads access the graph structures following a random pattern. Consequently, during the hypothesis expansion phase, the data cache acts as a regular LRU cache to leverage locality in the access to the graph structures.

ASRPU also includes a scratchpad memory (the Shared Memory) that is be accessed from the threads executing in the PEs. This is were the kernel buffers and the kernel configuration parameters are stored, along with any other variables defined by the programmer.

3.7 Command Decoder

The command decoder is the interface between ASRPU and the rest of the units in the SoC. It provides a set of commands (Table 1). These commands include some to configure the kernels and setup programs for the ASR phases: ConfigureASR_AcousticScoring, ConfigureASR_HypExpansion and commands to configure other parameters (ConfigureBeamWidth). These configuration commands must be used to configure the decoder before any decoding begins. In addition to those, the API contains commands for run-time operations. DecodingStep is to indicate the accelerator to decode a given signal. This signal is not decoded in isolation. Instead, it is appended to previously decoded signals, extending the current transcription hypotheses. Once the utterance is finished, CleanDecoding can be called. This command notifies the accelerator that the utterance is finished. In response, the accelerator prepares itself to decode a new utterance, cleaning the hypotheses memory and resetting the internal state.

4 Case Study

To illustrate the versatility and simplicity of our programming model for ASR, we present the implementation of one of the end-to-end systems from wav2letter. Features are 80-dim MFCCs computed from the pre-processed audio signal. The acoustic model is a TDS network, built from TDS blocks (Figure 4b). It is mostly composed of fully-connected and convolutional layers. The activation function for most layers is a ReLU, followed by a layer normalization. Hypotheses are extracted by traversing a lexicon tree that includes all the words in the vocabulary and a mechanism to handle out of vocabulary words. Additionally, an n-gram language model provides language model scores for the hypothesis.

In our implementation, the kernels that implement the acoustic scoring phase will first pre-process the signal and generate the MFCCs frames. Then, they perform inference with the TDS network to obtain the acoustic scores from each of the computed frames. On each hypothesis expansion execution, all the hypotheses are expanded one node forward in the lexicon tree, covering each reachable node. Every reached node in the tree is a new hypothesis for the following hypothesis expansion execution. Every time a hypothesis reaches a node in the lexicon tree that represents a word, a link in the n-gram language model graph is traversed. The n-gram graph contains language model scores that are included, along with the acoustic scores, the word penalty and others, in the computation of the hypothesis score. Hypotheses are compared based on this score and those with a lower score are pruned away by the hypothesis unit. In addition to the reachable nodes in the lexicon tree, hypothesis expansion generates two more
hypothese as part of the CTC algorithm: the blank symbol and the repetition.

4.1 The Main Process
The main process residing in the CPU orchestrates the overall decoding of utterances. It does so by calling commands from the API of the accelerator. Before the decoding starts, the main process configures the accelerator, setting all the necessary parameters, including the addresses in external memory of the kernels that implement the ASR system.

During decoding, the main process collects reading from the microphone. This example ASR system performs streaming decoding, meaning that every few milliseconds, the main process calls the submittSignal command to perform a decoding step on a partial signal. If this was not the case, the main process would capture the signal until the end of the utterance is reached and then call a submittSignal on the entire signal.

4.2 Acoustic Scoring
The acoustic scoring phase executes the code that implements the feature extraction and the acoustic model. The acoustic scoring phase consists of a set of programs executed in sequence. In this case study, the first kernel performs signal pre-processing and extracts MFCC features frames from the input signal, whereas the rest implement each a layer of the TDS DNN.

Before executing the feature extraction kernel, its setup thread is launched to check the size of the input signal and determine how many output frames can be computed from the available input. Then, it reserves memory for the output, marks the inputs as consumed and notifies the controller about how many main threads must be launched. The kernel threads then process the inputs to generate feature frames. Each thread computes a single feature frame, which means that for each output frame to compute, a feature extraction thread will be launched.

The subsequent kernels in the acoustic scoring phase implement the TDS DNN. It is implemented in a sequence of 79 kernels: 18 CONV, 29 FC and 32 LayerNorms, each preceded by its corresponding setup thread. To avoid repeating very similar code, the programs for CONV, FC and LayerNorm are parameterized. The setup thread sets the parameters in shared memory to the values corresponding to the current layer, which are accessed by the layer threads.

Each setup thread checks the number of inputs available (those generated by the previous layer), reserves memory for the outputs and notifies the ASR controller to launch the required number of threads for the layer program. Each CONV and FC thread compute a single neuron of the layer.

4.3 Hypothesis Expansion
The hypothesis expansion kernel implements the CTC decoding algorithm with lexicon and language model. Each thread processes a single hypothesis. The algorithm first accesses the node in the lexicon graph associated with the hypothesis, then, it traverses all the output links to access reachable nodes, generating a new hypothesis for each of them. Each hypothesis also contains a link to the language model graph, pointing to the last n-gram in the hypothesis. If a newly reached node in the lexicon graph represents a word, the hypothesis expansion thread will access the node in the language model graph associated with the hypothesis and expand it one node further following the link that represents the newly added word. The node contains a language model score that is added to the score of the hypothesis.

In addition to the hypotheses generated by traversing the lexicon graph, the CTC algorithm implemented in the hypothesis expansion thread requires the generation of two more hypotheses: the first one obtained by appending to the hypothesis the last phonetic unit in the hypothesis to account for repetitions, which produce valid CTC paths. The other hypothesis is obtained by appending the blank unit, which represents a frame that does not contain a phonetic unit.

5 Evaluation
This section provides estimations on the performance of ASRPU when running the ASR system described throughout the previous sections. The goal of this section is to provide proof of the capacity of the proposed design to enable real-time ASR on very low-power devices. To that purpose,
we studied a possible implementation for the TDS-based system described in the previous section and estimated its performance on the accelerator, configured to enable real-time ASR with that system. Furthermore, we estimate the power consumption and area footprint of that specific configuration.

5.1 Methodology and Scope
To estimate performance, we count the number of instructions for each kernel. For example, a loop will usually consist of two instructions for the comparison and conditional jump, one instruction for the variable update and the instructions for the loop body, all multiplied by the average number of iterations. Additionally, one instruction is added for the variable initialization. We assume that each PE executes one instruction per cycle, so we divide the number of instructions by the clock frequency of the PEs to obtain execution time.

To estimate chip area, we rely on several tools. Cacti for the memories, McPat for the PEs and the PE bus and Design compiler (using the Saed32hvt cell library), which provides cell models at 32nm technology node) for the special function units.

Peak power is estimated by adding together the leakage power and peak dynamic power for the logic units as obtained from the Power Compiler. The case of memories is slightly different. Cacti reports leakage power and access energy. In this case, we assume as peak power the scenario where all the ports are accessed once per cycle. Adding the energy consumed for those accesses, divided by the clock period gives the dynamic power, which we add, along with the leakage power given by Cacti, to the power consumed by the logic.

This estimation, albeit not exhaustive, should provide a good approximation of the potential of the accelerator proposed in this work.

5.2 Accelerator Configuration
Table 2 contains the details of the accelerator. This configuration was chosen to allow real-time ASR with the ASR system described in previous sections. Particularly, the number of PEs and the size of the memories was chosen to match the performance requirements. We include 8 PEs, each loaded with an 8-dim MAC unit, which allows us to exploit plenty of parallelism. The implemented algorithm stores about 275KB of intermediate data in between decoding steps. It stores inputs for the convolutional layers. Due to the shifting input window used in convolutions, inputs are reused in several consecutive executions. We include 512KB of shared memory to store these inputs and other temporal outputs that may be necessary to store if the decoding step is interrupted due to insufficient inputs for one of the kernels.

We include 1MB of model memory. During acoustic scoring, this memory is used for caching the DNN parameters and other model data. The size of the TDS network layers vary significantly (figure 2). Convolutional layers fit in a few KB whereas most fully-connected layers range in the MB. We solve this by trivially partitioning FC layers into several kernels, each less than 1MB. Given that each thread in our implementation of the FC kernels computes a neuron, we partition the layers in various kernels, each computing some of the neurons. For example, each of the first FC layers consists of 1200 neurons with 1200 inputs each, which results in 1.4MB of model data. We divide each of these layers into 2 kernels, each computing 600 neurons (700MB).

5.3 Area and Power
Figure 10 shows an estimation of the area and peak power of ASRPu, broken down by component. At a 32nm technology node, the total area is 11.68mm², 65% of which is dedicated to the execution unit (PEs, PE d-cache, PE i-cache and PE bus), 32% is dedicated to the shared and model memories. The hypothesis unit accounts for less than 1%. Regarding power, the accelerator consumes slightly more than 1.8 W assuming peak power. That is, if every PE is in execution and every memory is accessed. Around 800 mW come from static power, mostly from the PE cores and the shared and model memories, whereas the rest comes from dynamic power, mainly from the PE cores.

5.4 Performance
Each decoding step in our implementation decodes 80ms of audio. According to our estimations, ASRPu takes about 40ms to perform a decoding step. In other words, the accelerator executes the ASR system in 2x real-time. Figure 11 shows the execution time of the ASR system kernels, including the feature extraction and the hypothesis expansion kernels. The left plot shows the execution time for the kernels that implement the convolutional layers and the hypothesis expansion, whereas the right plot shows the execution time taken by fully-connected layers and the feature expansion.
6 CONCLUSIONS

Automatic Speech Recognition is becoming a mainstream technology. Higher recognition accuracy opens the door to more use cases, while increased public acceptance towards natural human-machine interaction increases demand for ASR products. Furthermore, data privacy and Quality of service motivate an interest in performing ASR locally, on-device. There are some challenges not tackled by this work. Furthermore, given the simplicity and the few constraints supported to implement most of the current ASR algorithms. Furthermore, given the simplicity and the few constraints imposed by the programming model on the ASR algorithms, it is likely that future algorithms can be supported, as well. The results included in this paper show how a modern ASR system can be implemented to be executed in the proposed architecture to perform real-time stream decoding with very low-power and area requirements.

There are some challenges not tackled by this work. ASR systems based on a encoder-decoder architecture (such as LAS and RNN-T) are challenging to implement in our platform. Encoder-decoder systems work as follows: The encoder part of the DNN first processes the entire sequence of features. Then, the decoder component generates score vectors until a special output is generated. This scheme does not fit well with the programming model of the proposed architecture. Supporting these systems is an interesting future work given the increasing interest in them in the ASR space.

ACKNOWLEDGMENTS

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] P. KR, R. Mukhopadhyay, J. Philip, A. Jha, Y. Namboodiri, and C. Jawahar, “Towards automatic face-to-face translation,” in Proceedings of the 27th ACM International Conference on Multimedia, 2019, pp. 1428–1436.

[2] “Cortana,” https://www.microsoft.com/en-us/cortana [Online; accessed 29-Oct-2021].

[3] “Alexa,” https://en.wikipedia.org/wiki/Amazon_Alexa [Online; accessed 29-Oct-2021].

[4] “siri,” https://en.wikipedia.org/wiki/Siri [Online; accessed 29-Oct-2021].

[5] T. G. Poder, J.-F. Fisette, and V. Déry, “Speech recognition for medical dictation: overview in quebec and systematic review,” Journal of medical systems, vol. 42, no. 5, pp. 1–8, 2018.

[6] S. Evain, B. Lecouteux, F. Portet, I. Estève, and M. Fabre, “Towards automatic captioning of university lectures for french students who are deaf,” in The 22nd International ACM SIGACCESS Conference on Computers and Accessibility, 2020, pp. 1–4.

[7] M. N. Sahadat, A. Alreja, N. Mikail, and M. Ghovanloo, “Comparing the use of single versus multiple combined abilities in conducting complex computer tasks hands-free,” IEEE Transactions on Neural Systems and Rehabilitation Engineering, vol. 26, no. 9, pp. 1868–1877, 2018.

[8] A. Škraba, A. Koložvari, D. Kofjac, R. Stojanovic, E. Semenkin, and V. Stanovov, “Development of cyber-physical speech-controlled wheelchair for disabled persons,” in 2019 22nd Euromicro Conference on Digital System Design (DSD). IEEE, 2019, pp. 456–463.

[9] V. Arora, A. Lahiri, and H. Reetz, “Phonological feature-based speech recognition system for pronunciation training in non-native language learning,” The Journal of the Acoustical Society of America, vol. 143, no. 1, pp. 98–108, 2018.

[10] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen et al., “Deep speech 2: End-to-end speech recognition in english and mandarin,” in International conference on machine learning. PMLR, 2016, pp. 173–182.

[11] 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.

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

[13] S. Bhatt, A. Jain, and A. Dev, “Acoustic modeling in speech recognition: A systematic review.”

[14] Y. Wang, A. Mohamed, D. Le, C. Liu, A. Xiao, J. Mahadeokar, H. Huang, A. Tjandra, X. Zhang, F. Zhang et al., “Transformer-based acoustic modeling for hybrid speech recognition,” in ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2020, pp. 6874–6878.

[15] C. Lüscher, E. Beck, K. Irie, M. Kitza, W. Michel, A. Zeyer, R. Schütter, and H. Ney, “Rwth asr systems for librispeech: Hybrid vs attention-w/o data augmentation,” arXiv preprint arXiv:1905.03072, 2019.

[16] H. Xu, K. Li, Y. Wang, J. Wang, S. Kang, X. Chen, D. Povey, and S. Khudanpur, “Neural network language modeling with letter-based features and importance sampling,” in 2018 IEEE international conference on acoustics, speech and signal processing (ICASSP). IEEE, 2018, pp. 6109–6113.

[17] X. Xu, A. Baevski, T. Likhomanenko, P. Tomasello, A. Conneau, R. Collobert, G. Synnaeve, and M. Auli, “Self-training and pre-training are complementary for speech recognition,” in ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2021, pp. 3030–3034.
M. Rouvier and B. Favre, “Speaker adaptation of dnn-based asr”

N. Dehak, P. J. Kenny, R. Dehak, P. Dumouchel, and P. Ouellet,

A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, “Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks,” in Proceedings of the 23rd international conference on Machine learning, 2006, pp. 369–376.

W. Chan, N. Jaitly, Q. Le, and O. Vinyals, “Listen, attend and spell: A neural network for large vocabulary conversational speech recognition,” in 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2016, pp. 4960–4964.

S. Cheng, Z. Xu, X. Li, X. Wu, Q. Fan, X. Wang, and V. C. Leung, “Task offloading for automatic speech recognition in edge-cloud computing based mobile networks,” in 2020 IEEE Symposium on Computers and Communications (ISCC). IEEE, 2020, pp. 1–6.

D. Pinto, J.-M. Arnau, and A. Gonzalez, “Design and evaluation of a ultra low-power human-quality speech recognition system,” ACM Trans. Archit. Code Optim., vol. 17, no. 4, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3425604

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.

H. Tabani, J.-M. Arnau, J. Tubella, and A. Gonzalez, “An ultra low-power hardware accelerator for acoustic scoring in speech recognition,” in 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, 2017, pp. 41–52.

Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, “Diannao family: energy-efficient hardware accelerators for machine learning,” Communications of the ACM, vol. 59, no. 11, pp. 105–112, 2016.

Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” IEEE journal of solid-state circuits, vol. 52, no. 1, pp. 127–138, 2016.

Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, “Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices,” IEEE journal on Emerging and Selected Topics in Circuits and Systems, vol. 9, no. 2, pp. 292–308, 2019.

M. Putic, S. Venkataramani, S. Eldridge, A. Buyuktosunoglu, P. Bose, and M. Stan, “Dyhard-dnn: Even more dnn acceleration with dynamic hardware reconfiguration,” in Proceedings of the 55th Annual Design Automation Conference, 2018, pp. 1–6.

M. Price et al., “Energy-scalable speech recognition circuits,” Ph.D. dissertation, Massachusetts Institute of Technology, 2016.

B. Liu, H. Qin, Y. Gong, W. Ge, M. Xia, and L. Shi, “Eera-asr: An energy-efficient reconfigurable architecture for automatic speech recognition with hybrid dnn and approximate computing,” IEEE Access, vol. 6, pp. 52227–52237, 2018.

V. Ppeedinti, D. Povey, and S. Khudanpur, “A time delay neural network architecture for efficient modeling of long temporal contexts,” in Sixteenth annual conference of the international speech communication association, 2015.

A. Hannun, A. Lee, Q. Xu, and R. Collobert, “Sequence-to-sequence speech recognition with time-depth separable convolutions,” CoRR, vol. abs/1904.02619, 2019. [Online]. Available: http://arxiv.org/abs/1904.02619

D. Povey, V. Ppeedinti, D. Galvez, P. Galaharman, V. Manohar, X. Na, Y. Wang, and S. Khudanpur, “Purely sequence-trained neural networks for asr based on lattice-free mmi.” in Interspeech, 2016, pp. 2751–2755.

N. Dehak, P. J. Kenny, R. Dehak, P. Dumouchel, and P. Ouellet, “Front-end factor analysis for speaker verification,” IEEE Transactions on Audio, Speech, and Language Processing, vol. 19, no. 4, pp. 798–798, 2011.

M. Rouver and B. Favre, “Speaker adaptation of dnn-based asr with i-vectors: Does it actually adapt models to speakers?” in Fifteenth Annual Conference of the International Speech Communication Association, 2014.

K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

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.

J. Guglani and A. N. Mishra, “Dnn based continuous speech recognition system of punjabi language on kaldi toolkit,” International Journal of Speech Technology, vol. 24, pp. 41–45, 2021.

F. Zhang, Y. Wang, X. Zhang, C. Liu, Y. Saraf, and G. Zweig, “Faster, simpler and more accurate hybrid asr systems using wordpieces,” arXiv preprint arXiv:2005.00150, 2020.

L. R. Rabiner, “A tutorial on hidden markov models and selected applications in speech recognition,” Proceedings of the IEEE, vol. 77, no. 2, pp. 257–286, 1989.

M. Mohri, “Weighted finite-state transducer algorithms: an overview,” Formal Languages and Applications, pp. 551–563, 2004.