A Low-Power, High-Performance Speech Recognition Accelerator

Reza Yazdani, Jose-Maria Arnau, and Antonio González
Department of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona, Spain
Email: {ryazdani, jarnau, antonio}@ac.upc.edu

Abstract—Automatic Speech Recognition (ASR) is becoming increasingly ubiquitous, especially in the mobile segment. Fast and accurate ASR comes at high energy cost, not being affordable for the tiny power-budgeted mobile devices. Hardware acceleration reduces energy-consumption of ASR systems, while delivering high-performance. In this paper, we present an accelerator for large-vocabulary, speaker-independent, continuous speech-recognition. It focuses on the Viterbi search algorithm representing the main bottleneck in an ASR system. The proposed design consists of innovative techniques to improve the memory subsystem, since memory is the main bottleneck for performance and power in these accelerators' design. It includes a prefetching scheme tailored to the needs of ASR systems that hides main memory latency for a large fraction of the memory accesses, negligibly impacting area. Additionally, we introduce a novel bandwidth-saving technique that removes off-chip memory accesses by 20%. Finally, we present a power saving technique that significantly reduces the leakage power of the accelerators scratchpad memories, providing between 8.5% and 29.2% reduction in entire power dissipation. Overall, the proposed design outperforms implementations running on the CPU by orders of magnitude, and achieves speedups between 1.7x and 5.9x for different speech decoders over a highly optimized CUDA implementation running on Geforce-GTX-980 GPU, while reducing the energy by 123-454x.

Index Terms—Automatic Speech Recognition (ASR), Viterbi Search, Hardware Accelerator, WFST, Low-Power Architecture.

1 Introduction

Automatic Speech Recognition (ASR) has attracted the attention of the architectural community [2], [3], [4], [5] and the industry [6], [7], [8], [9] in recent years. ASR is becoming a key feature for smartphones, tablets and other energy-constrained devices like smartwatches. ASR technology is at the heart of popular voice-based user interfaces for mobile devices such as Google Now, Apple Siri or Microsoft Cortana. These systems deliver large-vocabulary, real-time, speaker-independent, continuous speech recognition. Unfortunately, supporting fast and accurate speech recognition comes at a high energy cost, which in turn results in fairly short operating time per battery charge. Performing ASR remotely in the cloud can potentially alleviate this issue, but it comes with its own drawbacks: it requires access to the Internet, it might increase the latency due to the time required to transfer the speech and it increases the energy consumption of the communication subsystem. Given the issues with software-based solutions running locally or in the cloud, we believe that hardware acceleration represents a better approach to achieve high-performance and energy-efficient speech recognition in mobile devices.

The most time consuming parts of an ASR pipeline can be offloaded to a dedicated hardware accelerator in order to bridge the energy gap, while maintaining high-performance or even increasing it. An ASR pipeline consists of two stages: the Deep Neural Network (DNN) and the Viterbi search. The DNN converts the input audio signal into a sequence of phonemes, whereas the Viterbi search converts the phonemes into a sequence of words. Figure 1 shows the percentage of execution time required for both stages in Kaldi [10], a state-of-the-art speech recognition system widely used in academia and industry. As it can be seen, the Viterbi search is the main bottleneck as it requires 73% of the execution time when running Kaldi on a recent CPU and 86% when running on a modern GPU. Moreover, the two stages can be pipelined and, in that case, the latency of the ASR system depends on the execution time of the slowest stage, which is clearly the Viterbi search.

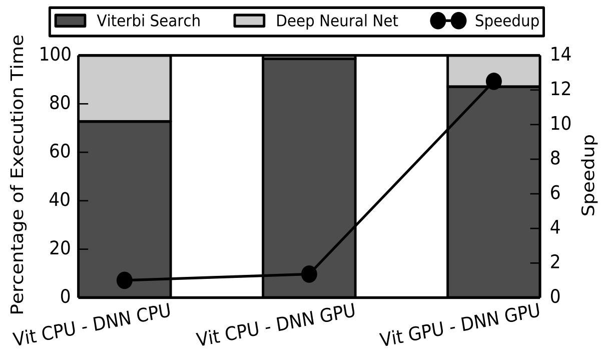


Fig. 1: Percentage of execution time for the two components of a speech recognition system: the Viterbi search and the Deep Neural Net. The Viterbi search is the dominant component, as it requires 73% and 86% of the execution time when running on a CPU and a GPU respectively.

In recent years there has been a lot of work on boosting the performance of DNNs by using GPUs [11] or dedicated accelerators [12], [13], achieving huge speedups and energy

- *This paper is an extension of the conference paper "An Ultra Low-Power Hardware Accelerator for Automatic Speech Recognition", presented in Micro'49, Taipei, Taiwan, 2016 [1].*

savings. However, the DNN is just a small part of an ASR system, where the Viterbi search is the dominant component as shown in Figure 1. Unfortunately, the Viterbi search algorithm is hard to parallelize [14], [15], [16] and, therefore, a software implementation cannot exploit all the parallel computing units of modern multi-core CPUs and many-core GPUs. Not surprisingly, previous work reported a modest speedup of 3.74x for the Viterbi search on a GPU [17]. Our numbers also support this claim as we obtained a speedup of 10x for the Viterbi search on a modern high-end GPU, which is low compared to the 26x speedup that we measured for the DNN. Besides, these speedups come at a very high cost in energy consumption. Therefore, we believe that a hardware accelerator specifically tailored to the characteristics of the Viterbi algorithm is the most promising way of achieving high-performance energy-efficient ASR on mobile devices.

In this paper, we present a hardware accelerator for speech recognition that focuses on the Viterbi search stage. Our experimental results show that the accelerator achieves similar performance to a high-end desktop GPU, while reducing energy by two orders of magnitude. In the second step, by analyzing the memory behavior of our accelerator, we propose two techniques to improve the memory subsystem. The first technique consists in a prefetching scheme inspired by decoupled access-execute architectures to hide the memory latency with a negligible cost in area. The second proposal consists in a novel bandwidth saving technique that avoids 20% of the memory fetches to off-chip system memory. Finally, we introduce a power-control mechanism, based on low-power drowsy caches [18], that significantly reduces on-chip memory leakage, which is the main power bottleneck of the accelerator.

This paper focuses on energy-efficient, high-performance speech recognition. Its main contributions are the following:

- We present an accelerator for the Viterbi search that achieves from 1.7x to 5.9x speedup over a high-end desktop GPU for decoding various speech datasets, while consuming 123x to 405x less energy.
- We introduce a prefetching architecture tailored to the characteristics of the Viterbi search algorithm that provides 1.87x speedup over the base design.
- We propose a memory bandwidth saving technique that removes 20% of the accesses to off-chip system memory.
- We present a low-power mechanism for the accelerator's memory components that mitigates most of the leakage power and reduces total power of the accelerator between 8.5% to 29.2%.

The remainder of this paper is organized as follows. Section 2 provides the background information on speech recognition systems. Section 3 presents our base design for speech recognition. Section 4 introduces two new techniques to hide memory latency and save memory bandwidth. Section 5 elaborates on the power-control approach for the accelerator's scratchpad memories. Section 6 describes our evaluation methodology and Section 7 shows the performance and power results. Section 8 reviews some related work and, finally, Section 9 sums up the main conclusions.

TABLE 1: WER comparison of different ASR decoders for Librispeech non-clean dataset [21].

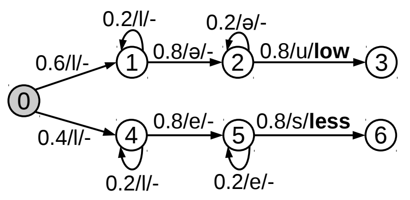
System	Type	WER(%)
Human	-	12.69
Kaldi's ASR (DNN + Viterbi)	**Hybrid**	**10.62**
DeepSpeech2 [22]	End-to-End	13.25
Deep bLSTM with attention [23]	End-to-End	12.76
wav2letter++ [24]	End-to-End	11.24

2 Speech Recognition with WFST

Speech recognition is the process of identifying a sequence of words from speech waveforms. An ASR system must be able to recognize words from a large vocabulary with unknown boundary segmentation between consecutive words. The typical pipeline of an ASR system works as follows. First, the input audio signal is segmented in frames of 10 ms of speech. Second, the audio samples within a frame are converted into a vector of features using signal-processing techniques such as Mel Frequency Cepstral Coefficients (MFCC) [19]. Third, the acoustic model, implemented by a Deep Neural Network (DNN), transforms the MFCC features into phonemes' probabilities. Context-sensitive phonemes are the norm, triphones [20] being the most common approach. A triphone is a particular phoneme when combined with a particular predecessor and a particular successor. Finally, the Viterbi search converts the sequence of phonemes into a sequence of words. The Viterbi search takes up the bulk of execution time, as illustrated in Figure 1, and is the main focus of our hardware accelerator presented in Section 3. The complexity of the search process is due to the huge size of the recognition model employed to represent the characteristics of the speech.

Even though the End-to-End (E2E) ASR models based on standalone RNN or CNN are getting popular since they simplify the overall pipeline, however, the Kaldi's hybrid system combined of DNN and Viterbi beam search still achieves higher accuracy, especially for noisy audio. Furthermore, E2E systems also require a beam search based on a language model to achieve accuracy comparable to hybrid systems [22]. We compared the accuracy of Kaldi's ASR decoder [21] with different E2E systems: Baidu's DeepSpeech2 [22], a state-of-the-art LSTM network with attention mechanism [23] and Facebook's wav2letter++ [24]. Table 1 shows the Word Error Rate (WER) collected for one of our speech corpora, Librispeech, that includes several hours of challenging noisy speech.

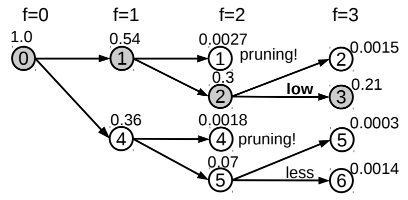
The state-of-the-art in recognition networks for speech is the Weighted Finite State Transducer (WFST) [25] approach. A WFST is a Mealy finite state machine that encodes a mapping between input and output labels associated with weights. In the case of speech recognition, the input labels represent the phonemes and the output labels the words. The WFST is constructed offline during the training process by using different knowledge sources such as context dependency of phonemes, pronunciation and grammar. Each knowledge source is represented by an individual WFST, and then they are combined to obtain a single WFST encompassing the entire speech process. For large vocabulary ASR systems, the resulting WFST contains millions of states and arcs. For example, the standard transducer for English



(a) Example of WFST

frame	l	ə	u	e	s
1	**0.9**	0.025	0.025	0.025	0.025
2	0.025	**0.7**	0.012	0.25	0.012
3	0.025	0.025	**0.9**	0.025	0.025

(b) Acoustic likelihoods



(c) Viterbi search trace

Fig. 2: Figure (a) shows a simple WFST that is able to recognize two words: *low* and *less*. For each arc the figure shows the weight, or transition probability, the phoneme that corresponds to the transition (input label) and the corresponding word (output label). Dash symbol indicates that there is no word associated to the transition. Figure (b) shows the acoustic likelihoods generated by the DNN for an audio with three frames. Figure (c) shows a trace of the Viterbi algorithm when using the WFST in (a) with the acoustic likelihoods shown in (b), the figure also shows the likelihood of reaching each state. The path with maximum likelihood corresponds to the word *low*.

language in Kaldi contains more than 13 million states and more than 34 million arcs.

Figure 2a shows a simple WFST for a very small vocabulary with two words. The WFST consists of a set of states and a set of arcs. Each arc has a source state, a destination state and three attributes: weight (or likelihood), phoneme (or input label) and word (or output label). On the other hand, Figure 2b shows the acoustic likelihoods generated by the DNN for an audio signal with three frames of speech. For instance, frame one has 90% probability of being phoneme *l*. Finally, Figure 2c shows the trace of states expanded by the Viterbi search when using the WFST of Figure 2a and the acoustic likelihoods of Figure 2b. The search starts at state 0, the initial state of the WFST. Next, the Viterbi search traverses all possible arcs departing from state 0, considering the acoustic likelihoods of the first frame of speech to create new active states. This process is repeated iteratively for every frame of speech, expanding new states by using the information of the states created in the previous frame and the acoustic likelihoods of the current frame. Once all the frames are processed, the active state with maximum likelihood in the last frame is selected, and the best path is recovered by using backtracking.

More generally, the Viterbi search employs a WFST to find the sequence of output labels, or words, with maximum likelihood for the sequence of input labels, or phonemes, whose associated probabilities are generated by a DNN. The word sequence with maximum likelihood is computed using a dynamic programming recurrence. The likelihood of the traversal process being in state *j* at frame f, $\psi_f(s_j)$, is computed from the likelihood in the preceding states as follows:

$$\psi_f(s_j) = \max_i\{\psi_{f-1}(s_i) \cdot w_{ij} \cdot b(O_f; m_k)\} \tag{1}$$

where w_{ij} is the weight of the arc from state i to state j, and $b(O_f; m_k)$ is the probability that the observation vector O_f corresponds to the phoneme m_k, i. e. the acoustic likelihood computed by the DNN. In the example shown in Figure 2, the likelihood of being in state 1 at frame 1 is: $\psi_0(s_0) \cdot w_{01} \cdot b(O_1; l)$ = 1.0 · 0.6 · 0.9 = 0.54. Note that state 1 has only one predecessor in the previous frame. In case of multiple input arcs, all possible transitions from active states in the previous frame are considered to compute the likelihood of the new active state according to Equation 1. Therefore, the Viterbi search performs a reduction to compute the path with maximum likelihood to reach the state j at frame f. In addition to this likelihood, the algorithm also saves a pointer to the best predecessor for each active state, that will be used during backtracking to restore the best path.

For real WFSTs, it is unfeasible to expand all possible paths due to the huge search space. In practice, ASR systems employ pruning to discard the paths that are rather unlikely. In standard beam pruning, only active states that fall within a defined range, a.k.a. beam width, of the frame's best likelihood are expanded. In the example of Figure 2c, we set the beam width to 0.25. With this beam, the threshold for frame 2 is 0.05: the result of subtracting the beam from the frame's best score (0.3). Active states 1 and 4 are pruned away as their likelihoods are smaller than the beam. The search algorithm combined with the pruning is commonly referred as Viterbi beam search [26].

On the other hand, representing likelihoods as floating point numbers between 0 and 1 might cause arithmetic underflow. To prevent this issue, ASR systems use log-space probabilities. Another benefit of working in log-space is that floating point multiplications are replaced by additions.

Regarding the arcs of the recognition network, real WFSTs typically include some arcs with no input label, a.k.a. epsilon arcs [17]. Epsilon arcs are not associated with any phoneme and they can be traversed during the search without consuming a new frame of speech. One of the reasons to include epsilon arcs is to model cross-word transitions. Epsilon arcs are less frequent than the arcs with input label, a.k.a. non-epsilon arcs. In Kaldi's WFST only 11.5% of the arcs are epsilon.

Note that there is a potential ambiguity in the use of the term *state*, as it might refer to the static WFST states (see Figure 2a) or the dynamic Viterbi trace (see Figure 2c). To clarify the terminology, in this paper we use *state* to refer to a static state of the WFST, whereas we use *token* to refer to an active state dynamically created during the Viterbi search. A *token* is associated with a static WFST state, but it also includes the likelihood of the best path to reach the state at frame f and the pointer to the best predecessor for backtracking.

The WFST approach has two major advantages over alternative representations of the speech model. First, it provides flexibility in adopting different languages, grammars, phonemes, etc. Since all these knowledge sources are compiled to one WFST, the algorithm only needs to search over the resulting WFST without consideration of the knowledge sources. This characteristic is especially beneficial for a hardware implementation as the same ASIC can be used to recognize words in different languages by using different types of models: language models (e.g., bigrams or trigrams), context dependency (e.g., monophones or triphones), etc. Therefore, supporting speech recognition for a different language or adopting more accurate language models only requires changes to the parameters of the WFST, but not to the software or hardware implementation of the Viterbi search. Second, the search process with the WFST is faster than using alternative representations of the speech, as it explores fewer states [27].

3 Hardware Accelerated Speech Recognition

In this section, we describe a high-performance and low-power accelerator for speech recognition. The accelerator focuses on the Viterbi beam search since it represents the vast majority of the compute time in all the analyzed platforms as shown in Figure 1. On the other hand, the neural-network used to produce acoustic likelihoods runs on GPU and in parallel with the accelerator. Figure 3 illustrates the architecture of the accelerator, which consists of a pipeline with five stages. In addition to a number of functional blocks, it includes several on-chip memories to speedup the access to different types of data required by the search process. More specifically, the accelerator includes three caches (State, Arc and Token), two hash tables to store the active states for the current and next frames of the speech, and a scratchpad memory to store the acoustic likelihoods computed by the DNN.

We cannot store all data on-chip due to its huge size. A typical WFST such as the one used in the experiments of this work [10] has a large vocabulary of 125k words and it contains more than 13M states and more than 34M arcs. The total size of the WFST is 681 MBytes. Regarding the representation of the WFST, we use the memory layout proposed in [3]. In this layout, states and arcs are stored separately in two different arrays. For each state, three attributes are stored in main memory: index of the first arc (32 bits), number of non-epsilon arcs (16 bits) and number of epsilon arcs (16 bits) packed in a 64-bit structure. All the outgoing arcs for a given state are stored in consecutive memory locations namely array of arcs; the non-epsilon arcs are stored first, followed by the epsilon arcs. For each arc, four attributes are stored packed in 128 bits: index of the arc's destination state, transition weight, input label (phoneme id) and output label (word id), each represented as a 32-bit value. The State and Arc caches speed up the access to the array of states and arcs respectively.

On the other hand, the accelerator also keeps track of the tokens generated dynamically throughout the search. Token's data is split into two parts, depending on whether the data has to be kept until the end of the search or it is only required for a given frame of speech: a) the backpointer to the best predecessor is required for the backtracking step to restore the best path when the search finishes, so these data is stored in main memory and cached in the Token cache; b) on the other hand, the state index and the likelihood of reaching the token are stored in the hash table and are discarded when the processing of the current frame finishes.

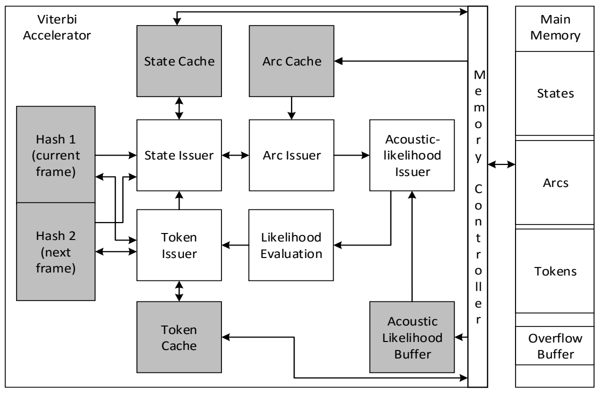


Fig. 3: Architecture of the accelerator for speech recognition.

Due to the pruning described in Section 2, only a very small subset of the states are active at any given frame of speech. The accelerator employs two hash tables to keep track of the active states, or tokens, created dynamically for the current and next frames respectively. By using a hash table, the accelerator is able to quickly find out whether a WFST state has already been visited in a frame of speech. If so, the accelerator obtains the memory location for the token associated with the WFST state from the hash table.

After a brief explanation of the overall ASR system, the following sections describe how the pipeline of the accelerator works and provide more details on the analysis to select appropriate parameters for the different hardware structures.

3.1 Overall ASR System

The system considered here is based on the hybrid approach which combines DNN and Viterbi search. This model represents the state-of-the-art in ASR as it provides better accuracy than other schemes such as GMM +Viterbi. The DNN, which runs on GPU, calculates the acoustic likelihoods of each batch, containing 256 frames of speech, scheduled to be reconized on the search phase. For the Viterbi search, which is the main bottleneck of the whole system, we use the proposed accelerator as it outperforms other implementations on different platforms like GPU.

The GPU and the accelerator work in parallel processing the DNN and the Viterbi search respectively in a pipelined manner. While the accelerator decodes the current batch, the GPU computes the acoustic scores for the next batch of frames. In order to hide the latency required for transferring the acoustic likelihoods from the GPU memory to the accelerator, a double-buffered memory is equipped in the accelerator's architecture which both stores and receives the acoustic scores of the current and the next frame respectively. These two buffers are swapped at the end of each frame's evaluation in order to have the two phases of ASR pipeline running simultaneously.

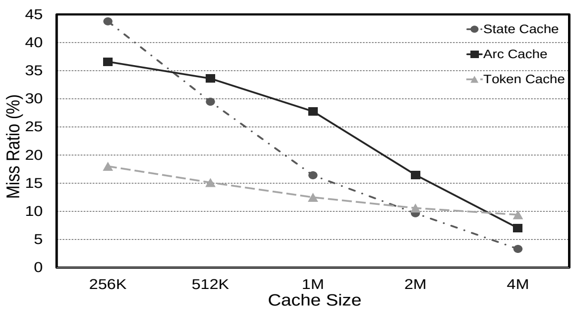


Fig. 4: Miss ratio vs capacity for the different caches in the accelerator.

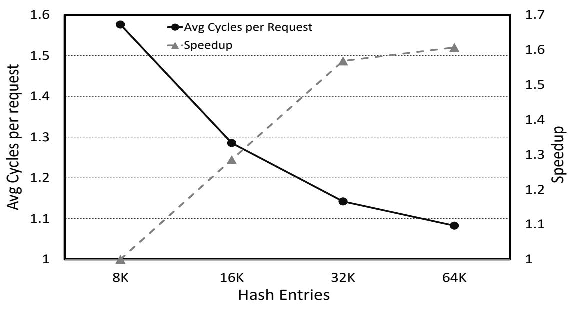


Fig. 5: Average cycles per request to the hash table and speedup vs number of entries.

3.2 Accelerator Pipeline

The pipeline of the accelerator illustrated in Figure 3 works as follows. First, the State Issuer fetches a token from the hash table that contains the list of tokens that have to be expanded for the current frame. Each token's data read from the hash contains two values: the likelihood of reaching the token and the state index. The State Issuer then performs the pruning and, hence, the token is only expanded if its likelihood is bigger than the threshold, otherwise it is discarded. If the token passes the pruning, the state's data is fetched from memory by using the State cache. Once the cache provides the data, it is forwarded to the Arc Issuer together with the token's likelihood read from the hash.

The Arc Issuer uses the state's data that contains the index of the first arc and the number arcs to generate memory requests to the Arc cache in order to fetch all the outgoing arcs for that state. When the Arc Issuer receives the information for a new arc from the cache, it forwards this information together with the token's likelihood to the next pipeline stage. Next, the Acoustic Likelihood Issuer employs the input label of the arc, i. e. the index of the phoneme, to fetch the corresponding acoustic likelihood from the scratchpad memory, labeled as Acoustic Likelihood Buffer in Figure 3.

In the next pipeline stage, the Likelihood Evaluation unit receives all the data required to compute the likelihood of the new token according to Equation 1. This unit computes the summation of the source token's likelihood read from the hash, the weight of the arc read from Arc cache and the acoustic likelihood. Note that additions are used instead of multiplications as the accelerator works in log-space. This unit also keeps track of the maximum likelihood for all the tokens created in a given frame, as this value is required to compute the threshold for pruning.

Finally, the Token Issuer employs the index of the arc's destination state to access the second hash table containing the tokens for the next frame. If a token for that state has already been created, the likelihood for the new token is compared with the likelihood stored in the hash. If the new token has a smaller likelihood it is discarded, whereas if it is bigger both the cost in the hash and the backpointer in main memory are updated, since a better path to reach this destination state has been found. If the destination state does not exist in the hash table, a new hash entry is allocated to store the likelihood, whereas the backpointer is saved at the end of the array of tokens in main memory using the Token cache.

Once all the tokens for the current frame have been expanded, the two hash tables are swapped, so the hash table for the next frame with the new tokens becomes the hash table for the current frame containing the source tokens to be expanded and vice versa.

The hash table is accessed using the state index. Each entry in the hash table stores the likelihood of the token and the address where the backpointer for the token is stored in main memory. In addition, each entry contains a pointer to the next active entry, so all the tokens are in a single linked list that can be traversed by the State Issuer in the next frame. The hash table includes a backup buffer to handle collisions (states whose hash function maps them to the same entry). In case of a collision, the new state index is stored in a backup buffer, and a pointer to that location is saved in the original entry of the hash. Each new collision is stored in the backup buffer and linked to the last collision of the same entry, all collisions forming a single linked list.

On the other hand, in case of an overflow in a hash table, the accelerator employs a buffer in main memory, labeled as Overflow Buffer in Figure 3, as an extension of the backup buffer. Overflows significantly increase the latency to access the hash table, but we found that they are extremely rare for common hash table sizes.

The Acoustic Likelihood Buffer contains the likelihoods computed by the DNN. In our system, the DNN is evaluated in the CPU or the GPU and the result is transferred to the aforementioned buffer in the accelerator. The buffer contains storage for two frames of speech, so the accelerator can be working in the current frame while the DNN is evaluated for the next frame and the result is copied in the buffer.

The result generated by the accelerator is the dynamic trace of tokens in main memory, together with the address of the best token in the last frame. The backtracking is done on the CPU, following backpointers to get the sequence of words in the best path. The execution time of the backtracking is negligible compared to the Viterbi search so it does not require hardware acceleration.

3.3 Analysis of Caches and Hash Tables

Misses in the caches and collisions in the hash tables are the only sources of pipeline stalls and, therefore, the parameters for those components are critical for the performance of the overall accelerator. In this section, we evaluate different configurations of the accelerator to find appropriate values for the capacity of the State, Arc and Token caches and the hash tables.

Figure 4 shows the miss ratios of the caches for different capacities. As it can be seen, even large capacities of 1-2 MBytes exhibit significant miss ratios. These large miss ratios are due to the huge size of the datasets, mainly the arcs and states in the WFST, and the poor spatial and temporal locality that the memory accesses to those datasets exhibit. Only a very small subset of the total arcs and states are accessed on a frame of speech, and this subset is sparsely distributed in the memory. The access to the array of tokens exhibits better spatial locality, as most of the tokens are added at the end of the array at consecutive memory locations. For this reason, the Token cache exhibits lower miss ratios than the other two caches for small capacities of 256KB-512KB.

Large cache sizes can significantly reduce miss ratio, but they come at a significant cost in area and power. Furthermore, they increase the latency for accessing the cache, which in turn increases total execution time. For our baseline configuration, we have selected 512 KB, 1 MB and 512 KB as the sizes for the State, Arc and Token caches respectively. Although larger values provide smaller miss ratios, we propose to use instead other more cost-effective techniques for improving the memory subsystem, described in Section 4.

Regarding the hash tables, Figure 5 shows the average number of cycles per request versus the number of entries in the table. If there is no collision, requests take just one cycle, but in case of a collision, the hardware has to locate the state index by traversing a single linked list of entries, which may take multiple cycles (many more if it has to access the Overflow Buffer in main memory). For 32K-64K entries the number of cycles per request is close to one. Furthermore, the additional increase in performance from 32K to 64K is very small as it can be seen in the speedup reported in Figure 5. Therefore, we use 32K entries for our baseline configuration which requires a total storage of 768 KBytes for each hash table, similar to the capacities employed for the caches of the accelerator.

4 IMPROVED MEMORY HIERARCHY

In this section, we perform an analysis of the bottlenecks in the hardware accelerator for speech recognition presented in Section 3, and propose architectural extensions to alleviate those bottlenecks. There are only two sources of pipeline stalls in the accelerator: misses in the caches and collisions in the hash tables. In case of a miss in the State, Arc or Token cache, the ASIC has to wait for main memory to serve the data, potentially introducing multiple stalls in the pipeline. On the other hand, resolving a collision in the hash table requires multiple cycles and introduces pipeline stalls, as subsequent arcs cannot access the hash until the collision is solved.

The results obtained by using our cycle-accurate simulator show that main memory latency has a much bigger impact on performance than the collisions in the hash tables. The performance of the accelerator improves by 2.11x when using perfect caches in our simulator, whereas an ideal hash with no collisions only improves performance by 2.8% over the baseline accelerator with the parameters shown in Table 2. Therefore, we focus our efforts on hiding the memory latency in an energy-efficient manner. In Section 4.1 we introduce an area-effective latency-tolerance technique that is inspired by decoupled access-execute architectures.

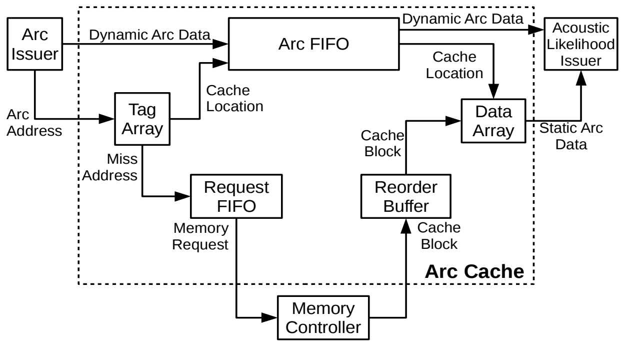


Fig. 6: Prefetching architecture for the Arc cache.

On the other hand, off-chip DRAM accesses are known to be particularly costly in terms of energy [12]. To further improve the energy-efficiency of our accelerator, we present a novel technique for saving memory bandwidth in Section 4.2. This technique is able to remove a significant percentage of the memory requests for fetching states from the WFST.

4.1 Hiding Memory Latency

Cache misses are the main source of stalls in the pipeline of the design and in consequence, main memory latency has a significant impact on the performance of the accelerator for speech recognition presented in Section 3. Regarding the importance of each individual cache, the results obtained in our simulator show that using a perfect Token cache provides a minor speedup of 1.02x. A perfect State cache improves performance by 1.09x. On the other hand, the Arc cache exhibits the biggest impact on performance, as removing all the misses in this cache would provide 1.95x speedup.

The impact of the Arc cache on the performance of the overall accelerator is due to two main reasons. First, the memory footprint for the arcs dataset is quite large: the WFST has more than 34M arcs, whereas the number of states is around 13M. So multiple arcs are fetched for every state when traversing the WFST during the Viterbi search. Second, arc fetching exhibits poor spatial and temporal locality. Due to the pruning, only a small and unpredictable subset of the arcs are active on a given frame of speech. We observed that only around 25k of the arcs are accessed on average per frame, which represents 0.07% of the total arcs in the WFST. Hence, the accelerator has to fetch a different and sparsely distributed subset of the arcs on a frame basis, which results in large miss ratio for the Arc cache (see Figure 4). Therefore, efficiently fetching arcs from memory is a major concern for the accelerator.

The simplest approach to tolerate memory latency is to increase the size of the Arc cache, or to include a bigger second level cache. However, this approach causes a significant increase in area, power and latency to access the cache. An-

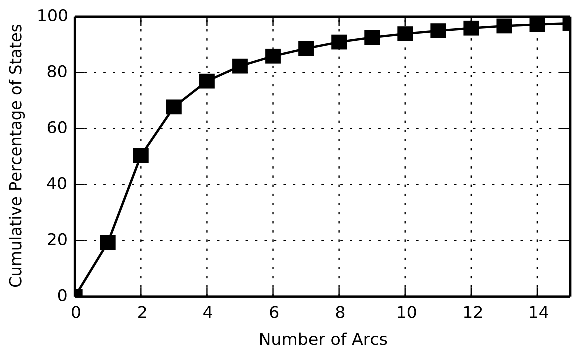


Fig. 7: Cumulative percentage of states accesses dynamically vs the number of arcs. Although the maximum number of arcs per state is 770, 97% of the states fetched from memory have 15 or less arcs.

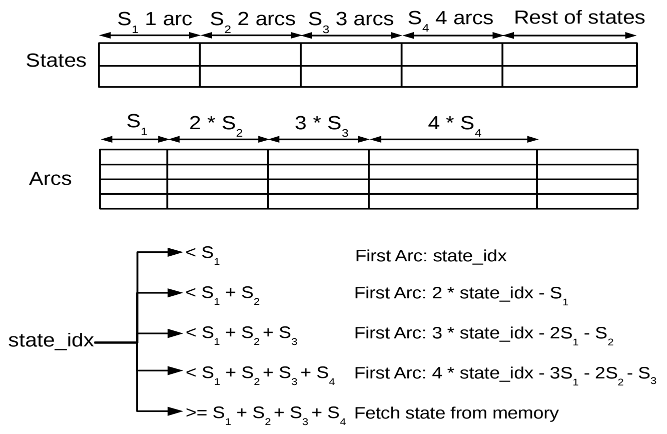


Fig. 8: Changes to the WFST layout. In this example, we can directly compute arc index from state index for states with 4 or less arcs.

other solution to hide memory latency is hardware prefetching [28]. Nonetheless, we found that the miss address stream during the Viterbi search is highly unpredictable due to the pruning and, hence, conventional hardware prefetchers are ineffective. We implemented and evaluated different state-of-the-art hardware prefetchers [28], [29], and our results show that these schemes produce slowdowns and increase energy due to the useless prefetches that they generate.

Our proposed scheme to hide memory latency is based on the observation that arcs prefetching can be based on computed rather than predicted addresses, following a scheme similar to the decoupled access-execute architecures [30]. After the pruning step, the addresses of all the arcs are deterministic. Once a state passes the pruning, the system can compute the addresses for all its outgoing arcs and prefetch their data from memory long before they are required, thus allowing cache miss latency to overlap with useful computations without causing stalls. Note that the addresses of the arcs that are required in a given frame only depend on the outcome of the pruning. Once the pruning is done, subsequent arcs can be prefetched while previously fetched arcs are being processed in the next pipeline stages.

Figure 6 shows our prefetching architecture for the Arc cache, which is inspired by the design of texture caches for GPUs [31]. Texture fetching exhibits similar characteristics to the arcs fetching, as all the texture addresses can also be computed much in advance from the time the data is required.

The prefetching architecture processes arcs as follows. First, the *Arc Issuer* computes the address of the arc and sends a request to the Arc cache. The arc's address is looked up in the cache tags, and in case of a miss the tags are updated immediately and the arc's address is forwarded to the *Request FIFO*. The cache location associated with the arc is forwarded to the *Arc FIFO*, where it is stored with all the remaining data required to process the arc, such as the source token likelihood. On every cycle, a new request for a missing cache block is sent from the *Request FIFO* to the *Memory Controller*, and a new entry is reserved in the *Reorder Buffer* to store the returning memory block. The *Reorder Buffer* prevents younger cache blocks from evicting older yet-to-be-used cache blocks, which could happen in the presence of an out-of-order memory system if the cache blocks are written immediately in the Data Array.

When an arc reaches the top of the *Arc FIFO*, it can access the Data array only if its corresponding cache block is available. Arcs that hit in the cache proceed immediately, but arcs that generated a miss must wait for its cache block to return from the memory into the *Reorder Buffer*. New cache blocks are committed to the cache only when its corresponding arc reaches the top of the *Arc FIFO*. At this point, the arcs that are removed from the head of the FIFO read their associated data from the cache and proceed to the next pipeline stage in the accelerator.

The proposed prefetching architecture solves the two main issues with the hardware prefetchers: accuracy and timeliness. The architecture achieves high accuracy because the prefetching is based on the computed rather than predicted addresses. Timeliness is achieved as cache blocks are not prefetched too early or too late. The *Reorder Buffer* guarantees that data is not written in the cache too early. Furthermore, if the number of entries in the *Arc FIFO* is big enough the data is prefetched with enough anticipation to hide memory latency.

Our experimental results provided in Section 7 show that this prefetching architecture achieves performance very close to a perfect cache, with a negligible increase of 0.34% in the area of the Arc cache and 0.05% in the area of the overall accelerator.

4.2 Reducing Memory Bandwidth

The accelerator presented in Section 3 consumes memory bandwidth to access states, arcs and tokens stored in off-chip system memory. The only purpose of the state fetches is to locate the outgoing arcs for a given state, since the information required for the decoding process is the arc's data. The WFST includes an array of states that stores the index of the first arc and the number of arcs for each state. Accessing the arcs of a given state requires a previous memory read to fetch the state's data.

Note that this extra level of indirection is required since states in the WFST have different number of arcs ranging from 1 to 770. If all the states would have the same number of arcs, arcs indices could be directly computed from state index. Despite the wide range in the number of arcs, we have observed that most of the states accessed dynamically have a small number of outgoing arcs as illustrated in Figure 7. Based on this observation, we propose a new scheme that is based on sorting the states in the WFST by their number of arcs, which allows to directly compute arc addresses from the state index for most of the states.

Figure 8 shows the new memory layout for the WFST. We move the states with a number of arcs smaller than or equal to *N* to the beginning of the array, and we sort those states by their number of arcs. In this way, we can directly compute the arc addresses for states with *N* or less arcs. In the example of Figure 8, we use 4 as the value of *N*.

To implement this optimization, in addition to changing the WFST offline, modifications to the hardware of the *State Issuer* are required to exploit the new memory layout at runtime. First, *N* parallel comparators are included as shown in Figure 8, together with *N* 32-bit registers to store the values of S_1, $S_1 + S_2$... that are used as input for the comparisons with the state index. Second, a table with *N* entries is added to store the offset applied to convert the state index into its corresponding arc index in case the state has *N* or less arcs. In our example, a state with 2 arcs has an offset of $-S_1$, which is the value stored in the second entry of the table.

The outcome of the comparators indicates whether the arc index can be directly computed or, on the contrary, a memory fetch is required. In the first case, the result of the comparators also indicates the number of arcs for the state and, hence, it can be used to select the corresponding entry in the table to fetch the offset that will be used to compute the arc index. In addition to this offset, the translation from state index to arc index also requires a multiplication. The factor for this multiplication is equal to the number of arcs for the state, which is obtained from the outcome of the comparators. The multiplication and addition to convert state index into arc index can be performed in the Address Generation Unit already included in the *State Issuer*, so no additional hardware is required for those computations.

For our experiments we use 16 as the value of N. With this value we can directly access the arcs for more than 95% of the static states in the WFST and more than 97% of the dynamic states visited at runtime. This requires 16 parallel comparators, 16 32-bit registers to store the values of S_1, $S_1 + S_2$... and a table with 16 32-bit entries to store the offsets. Our experimental results show that this extra hardware only increases the area of the State cache by 0.36% and the area of the overall accelerator by 0.02%, while it reduces memory bandwidth usage by 20%. Furthermore, we have applied our technique for three other ASR decoders using either Librispeech [32], Tedlium [33], or Voxforge [34] corpora, which yields almost similar decrease in the memory requests, 21.7%, 21.8%, and 22%, respectively.

5 Power-Controlled Hash Architecture

In this section, we describe a novel power-control scheme applied on the accelerator's Hash structure. First, we analyze the potential of such technique for reducing the accelerator's power dissipation. As our evaluation shows, the Hashes dissipate nearly 53% of the total power, including both the leakage and dynamic activities. Thus, we focus on controlling the power of these modules as they represent the main bottleneck. On the other hand, the static power accounts for 72% of the accelerator's total power and almost 50% of this amount is dissipated by the Hashing mechanism. Therefore, we target decreasing the Hashes' leakage power, in order to efficiently lower both the accelerator's power and energy consumption.

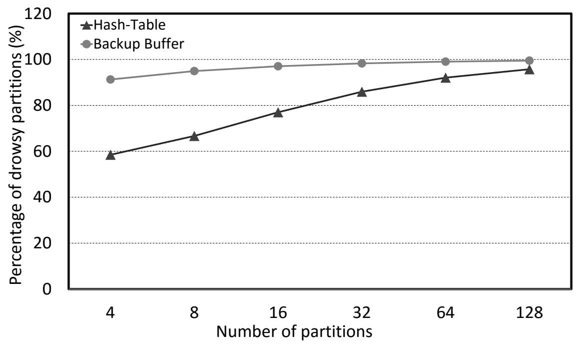


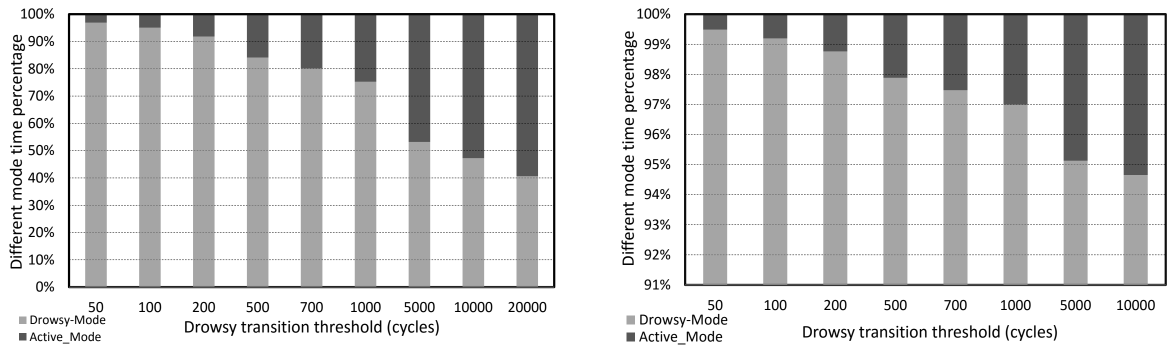
Fig. 9: Percentage of hash in drowsy mode for different number of partitions.

Regarding static power optimization, power gating [35] is a very effective technique used in many designs. However, we cannot apply this approach for the Hash's table since it would lose the data, which is needed for the next frame's evaluation. The reason is that this table uses a well-dispersed hashing function, which therefore distributes all the tokens almost uniformly across the table's entries. Nevertheless, we can power-gate a moderate portion of Hash's backup buffer on average, due to its entries consecutive allocation.

Another less aggressive method of reducing leakage energy consumption is to put the buffers into a low-power drowsy mode in which the state of memory is preserved. To do so, two schemes have been introduced at the circuit level: one using adaptive body-biasing with multi-threshold CMOS [36]; and the other one chooses between two different voltages for each operation mode [18]. Because of the short-channel effects in deep-submicron process and the low operating voltage (for the second approach), a significant reduction of the leakage power is yielded, achieving most of the advantage of a power-gating strategy.

As aforementioned, we can use power-gating for the backup buffer, partially, by conservatively selecting the partitions to turn off. A more aggressive approach would incur in some performance degradation in case the active partitions do not suffice to store all the active states, and some partitions need to be reactivated. Consequently, we exploit the drowsy technique as the base of our power-control scheme for both Hash tables and backup buffer. We based our implementation on the approach which uses dual voltages as it is simpler and more effective.

Regarding the switching between drowsy and active modes for the entire Hash structure, we use a simple yet effective policy which is based on the access pattern of the memory partitions. Unlike the static scheme proposed in [18], we use a dynamic threshold for transitioning a partition from active to drowsy mode. Moreover, we explore



(a) Hash-table

(b) Backup buffer

Fig. 10: Percentage of drowsy vs active time for several transition thresholds considering 128 partitions in each buffer.

the optimal number of partitions both for the Hash table and backup buffer. The power of each partition is controlled by its own drowsy control logic.

Our power-control system works as follows. Whenever there is an access to a partition of the Hash table, it is switched on by setting the voltage of that partition to the active value (1 V for 28 nm technology). Then, a counter, called change-mode counter, is started in order to measure a statically-adjusted period. The counter starts over at each access to the same partition. As the resetting of the counter happens repetitively because of consecutive accesses to the same partition, the active period varies dynamically throughout the execution time and for the different partitions. When the change-mode counter exceeds a given threshold, the partition's voltage is set to the drowsy mode, lowering the voltage to 0.3V, for which the state of all the entries is preserved. To manage the power-control mechanism of each partition, a very simple logic is required in addition to the change-mode counter: two pass transistors for connecting different voltages to the partition, and a 1-bit SRAM cell to keep track of the drowsy control signal.

Figure 9 shows the average percentage of partitions that are in drowsy mode when decoding speech signal for different numbers of partitions in the hash table and backup buffer. As it can be seen, the more partitions we use for the Hash table, the larger the percentage that stays in the drowsy mode. The main reason is that the hash function maps tokens to entries in a nearly uniformly-distributed manner. Consequently, as we decrease the size of partitions, they are less frequently accessed, which results in a higher drowsy percentage. On the other hand, accesses to the backup buffer are more predictive since partitions are written in order starting from the first entry. Thus, although the buffer can be in the drowsy mode more time by increasing the number of partitions, the improvement is not as huge as for the hash table.

In order to further evaluate our technique, we have explored various drowsy mode transitioning thresholds to decide when an active partition is switched to drowsy mode. Figure 10 shows the breakdown of the percentage of time in which a partition is in drowsy or active mode, on average, for different transition thresholds. As it can be seen, by increasing the threshold, which determines the minimum cycles that a partition is active, the percentage of drowsy mode decreases in either Hash table or backup buffer. This reduction is substantially larger for the Hash table because of the more frequent accesses to each partition when using a longer period, which triggers more resets of the change-mode counter, resulting in a higher percentage for the active mode operation. On the other hand, by setting an aggressive period such as 50 cycles for both the hash table and backup buffer, it imposes negligible overhead, as the delay and energy cost of switching a partition to different operation modes are relatively small. Based on our H-SPICE simulation results, each transition takes 0.12 ns (much less than a cycle) and consumes 2.83×10^{-15} J energy at 28 nm technology. Furthermore, we can hide almost all the delay of turning on a partition by notifying the hash table in advance when a request for either updating or adding a token is received in the Token Issuer.

By using the aforementioned exploration and statistics, we decided to use 128 and 32 partitions in the Hash table and backup buffer, which results in 95.8% and 98.3% of drowsy mode operation on average, respectively. Moreover, we set the transition threshold as 50 cycles (83 ns) due to its good trade-off between saving leakage power and energy overhead of switching between different operation modes. Furthermore, we extended the Hash's microarchitecture in a way to add a snooping port that the Token Issuer uses to report a future access to a drowsy partition. By notifying the access to the partition in advance in the pipeline, our accelerator is able to hide the transition delay. Regarding the implementation of this technique, the area of the Hash table is increased by 0.16% while only incurring in a 0.05% overhead in the total accelerator's area.

6 Evaluation Methodology

We have developed a cycle-accurate simulator that models the architecture of the accelerator presented in Section 3. Table 2 shows the parameters employed for the experiments in the accelerator. We modified the simulator to implement the prefetching scheme described in Section 4.1. After exploring different sizes of 32, 64, and 128 entries for the Arc FIFO, we select 64 for both the Request FIFO and Reorder Buffer, achieving the best trade-off between the area-cost, energy-consumption and accelerator's performance, in order to hide most of the memory latency. Furthermore, we have implemented the bandwidth saving technique proposed in

TABLE 2: Hardware parameters for the accelerator.

Technology	28 nm
Frequency	600 MHz
State Cache	512 KB, 4-way, 64 bytes/line
Arc Cache	1 MB, 4-way, 64 bytes/line
Token Cache	512 kB, 2-way, 64 bytes/line
Acoustic Likelihood Buffer	64 KB
Hash Table	768 KB, 32K entries
Memory Controller	32 in-flight requests
State Issuer	8 in-flight states
Arc Issuer	8 in-flight arcs
Token Issuer	32 in-flight tokens
Acoustic Likelihood Issuer	1 in-flight arc
Likelihood Evaluation Unit	4 fp adders, 2 fp comparators

TABLE 3: CPU parameters.

CPU	Intel Core i7 6700K
Number of cores	4
Technology	14 nm
Frequency	4.2 GHz
L1, L2, L3	64 KB, 256 KB per core, 8 MB

TABLE 4: GPU parameters.

GPU	NVIDIA GeForce GTX 980
Streaming multiprocessors	16 SMs (each with 2048 threads)
Technology	28 nm
Frequency	1.28 GHz
L1, L2 caches	48 KB, 2 MB

Section 4.2. We use 16 parallel comparators and a table of offsets with 16 entries in order to directly compute the arc indices for the states with 16 or less arcs.

We configure the Viterbi search parameters in the same way as configured for the Kaldi toolkit. In particular, we use the beam width as 15, which is representative for the very low probability of 3.1e-7. Thus, the Viterbi's hypotheses whose score (likelihood) gets higher than beam 15 (lower than 3.1e-7) are pruned away, to keep the search space tractable.

In order to estimate area and energy consumption, we have implemented the different pipeline components of the accelerator in Verilog and synthesized them using the Synopsys Design Compiler with a commercial 28 nm cell library. On the other hand, we use CACTI to estimate the power and area of the three caches included in the accelerator. We employ the version of CACTI provided in McPAT [37], a.k.a. enhanced CACTI [38] which includes models for 28 nm.

We use the delay estimated by CACTI and the delay of the critical path reported by Design Compiler to set the target frequency so that the various hardware structures can operate in one cycle (600 MHz). In addition, we model an off-chip 4GB DRAM using CACTI's DRAM model to estimate the access time to main memory. We obtained a memory latency of 50 cycles (83 ns) for our accelerator.

Regarding the WFST datasets, in addition to the WFST for English language provided in the Kaldi toolset [10], we use several other ASR decoders, such as Librispeech [32], Tedlium [33] and Voxforge [34], in order to extensively evaluate our accelerator. Each of these systems' WFSTs is generated based on various language models using the vocabularies from 13.9K to 200K words. Moreover, they use different approaches for the acoustic-scoring, either Gaussian Mixture Model (GMM) (Tedlium and Voxforge) and DNN (Librispeech and Kaldi's English dataset). Overall, our system is evaluated for thousands of hours of speech and hundreds of different speakers, with different phonemes evaluation approaches.

6.1 CPU Implementation

We use the software implementation of the Viterbi beam search included in Kaldi, a state-of-the-art ASR system. We measure the performance of the software implementation on a real CPU with the parameters shown in Table 3. We employ Intel RAPL library [39] to measure energy consumption. We use GCC 4.8.4 to compile the software using -O3 optimization flag.

6.2 GPU Implementation

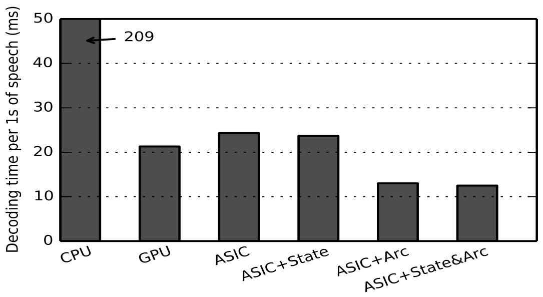
We have implemented the state-of-the-art GPU version of the Viterbi search presented in [17]. Furthermore, we have incorporated all the optimizations described in [40] to improve memory coalescing, reduce the cost of the synchronizations and reduce the contention in main memory by exploiting GPU's shared memory. We use the nvcc compiler included in CUDA toolkit version 7.5 with -O3 optimization flag. We run our CUDA implementation on a high-end GPU with the parameters shown in Table 4 and use the NVIDIA Visual Profiler [41] to measure performance and power.

7 Experimental Results

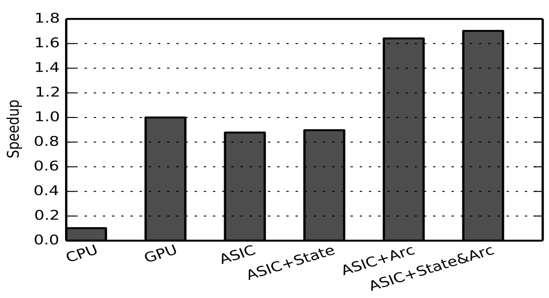
In this section, we provide details on the performance and energy consumption of the CPU, the GPU and the different versions of the accelerator. At first, we consider the Kaldi's English dataset to illustrate our evaluation for the different versions of accelerator. Then, we compare our most optimized version to show the numbers for the rest ASR decoders. Figure 11a shows the decoding time per one second of speech, a common metric used in speech recognition that indicates how much time it takes for the system to convert the speech waveform into words per each second of speech. We report this metric for six different configurations. *CPU* corresponds to the software implementation running on the CPU described in Table 3. *GPU* refers to the CUDA version running on the GPU presented in Table 4. *ASIC* is our accelerator described in Section 3 with parameters shown in Table 2. *ASIC+State* corresponds to the bandwidth saving technique for the State Issuer presented in Section 4.2. *ASIC+Arc* includes the prefetching architecture for the Arc cache presented in Section 4.1. Finally, the configuration labeled as *ASIC+State&Arc* includes our techniques for improving both the State Issuer and the Arc cache.

As it can be seen in Figure 11a, all the systems achieve real-time speech recognition, as the processing time per one second of speech is significantly smaller than one second. Both the *GPU* and the *ASIC* provide important reductions in execution time with respect to the *CPU*. The *GPU* improves performance by processing multiple arcs in parallel. The *ASIC* processes arcs sequentially, but it includes hardware specifically designed to accelerate the search process, avoiding the overheads of software implementations.

Figure 11b shows the speedups with respect to the *GPU* for the same configurations. The initial design of the

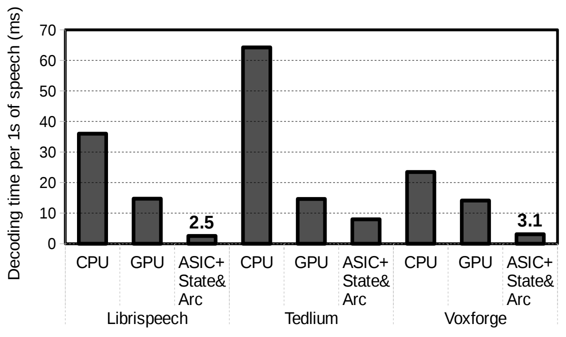


(a) Decoding-time

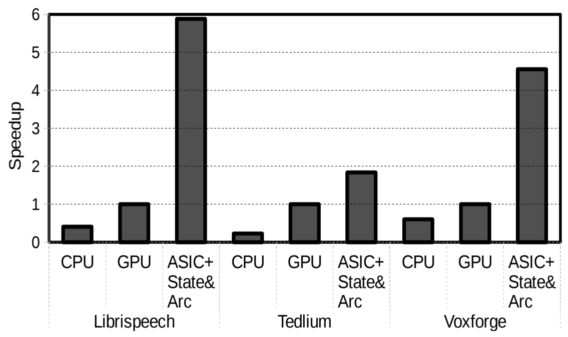


(b) Speedup

Fig. 11: **a**) Decoding time, i. e. the time to execute the Viterbi search, per second of speech. Decoding time is smaller than one second for all the configurations, so all the systems achieve real-time speech recognition. The numbers are measured for Kaldi's English dataset; **b**) Speedups achieved by the different versions of the accelerator. The baseline is *GPU*.



(a) Decoding-time



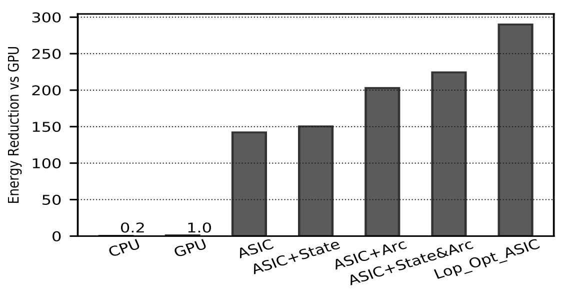
(b) Speedup

Fig. 12: **a**) Comparison of decoding time for the most optimized version of ASIC accelerator versus *CPU* and *GPU*. The numbers are shown for 3 different decoders: Librispeech, Tedlium and Voxforge.; **b**) Speedups achieved for the best ASIC configuration versus *CPU* and *GPU*, for different ASR datasets.

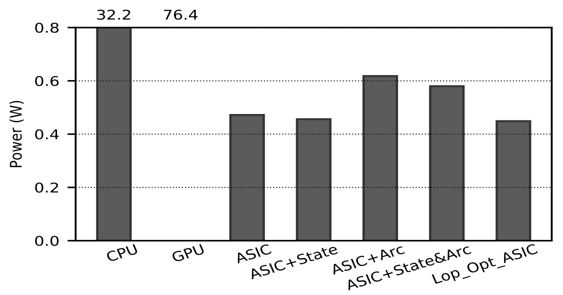
ASIC achieves 88% of the performance of the *GPU*. The *ASIC+State* achieves 90% of the *GPU* performance. This configuration includes a bandwidth saving technique that is very effective for removing off-chip memory accesses as reported later in this section, but it has a minor impact on performance (its main benefit is power reduction as we will see later). Since our accelerator processes arcs sequentially, performance is mainly affected by memory latency and not memory bandwidth. On the other hand, the configurations using the prefetching architecture for the Arc cache achieve significant speedups, outperforming the *GPU*. We obtain 1.64x and 1.7x speedup for the *ASIC+Arc* and *ASIC+State*&*Arc* configurations respectively with respect to the *GPU* (about 2x with respect to the ASIC without these optimizations). The performance benefits come from removing the pipeline stalls due to misses in the Arc cache, as the data for the arcs is prefetched from memory long before they are required to hide memory latency. The prefetching architecture is a highly effective mechanism to tolerate memory latency, since it achieves 97% of the performance of a perfect cache according to our simulations.

In order to measure the applicability of our accelerator, we also compare the decoding time when running several other decoders. Figures 12a and 12b show the execution time per one second of speech and the speedup achieved versus *GPU*, respectively, for Librispeech, Tedlium and Voxforge speech datasets. As illustrated, the accelerator obtains significant performance improvement, speeding up the Viterbi from 1.8x for Tedlium to 5.9x for Librispeech, compared to a high-end GPU. The reason for achieving higher speedup for Librispeech and Voxforge, is that these datasets are significantly smaller in size (37 and 496 MB) comparing to Tedlium (1.1 GB) and Fisher English(681 MB). Moreover, the complexity of the Viterbi search is lower for these ASR decoders as the number of hypotheses dynamically generated is considerably smaller. Thus, we have seen that both the arc and state caches and the token cache exhibit good temporal and spatial locality, which therefore results in higher performance speedup.

Our accelerator for speech recognition provides a huge reduction in energy consumption as illustrated in Figure 13a. The numbers include both static and dynamic energy. The base *ASIC* configuration reduces energy consumption by 142x with respect to the *GPU*, whereas the optimized version using the proposed improvements for the memory subsystem (*ASIC+Arc*&*State*) reduces energy by 224x. The reduction in energy comes from two sources. First, the accelerator includes dedicated hardware specifically designed for speech recognition and, hence, it achieves higher energy-efficiency for that task than general purpose processors and GPUs. Second, the speedups achieved by using the prefetching architecture provide a reduction in static energy. The last configuration, named as *Lop_Opt_ASIC*, includes the Hash's power-control mechanism in addition to all the previous

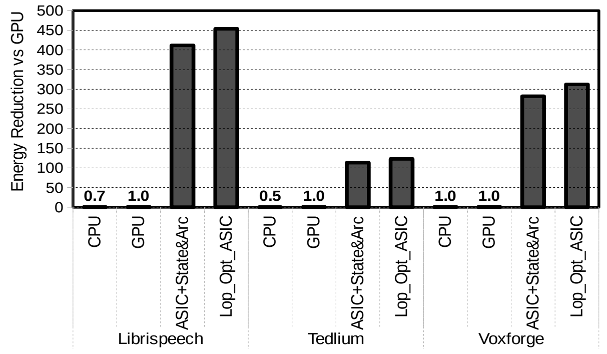


(a) Energy reduction

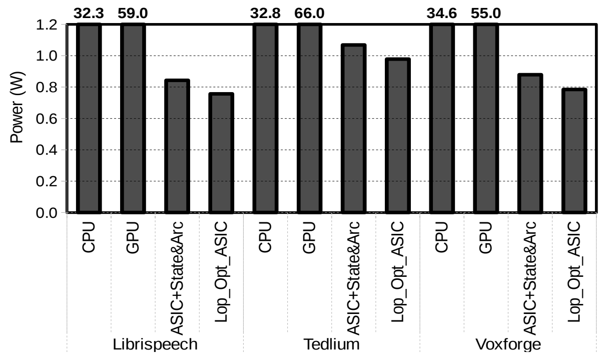


(b) Power dissipation

Fig. 13: **a**) Energy reduction vs *GPU*, for different versions of the accelerator when decoding Kaldi's English dataset; **b**) Power dissipation for the *CPU*, *GPU* and different versions of the accelerator for decoding Kaldi's English dataset.



(a) Energy reduction



(b) Power dissipation

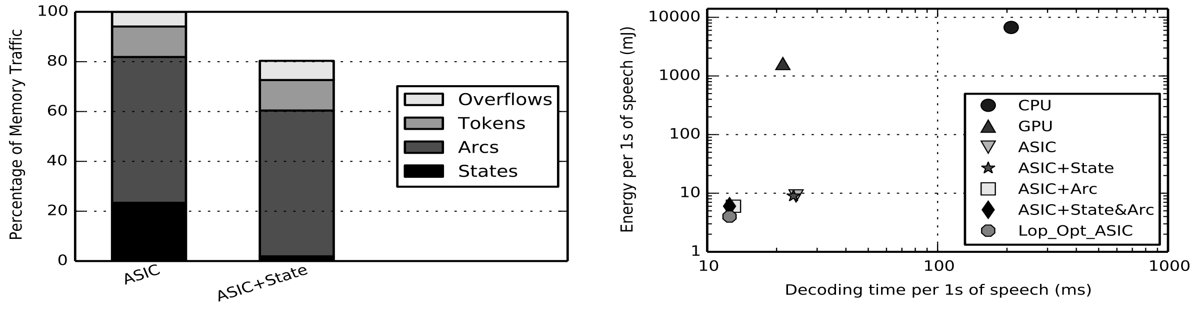
Fig. 14: **a**) Energy reduction of the best versions of Accelerator vs the GPU, for the different ASR decoders; **b**) Power dissipation for the *CPU*, *GPU* and the two optimized and low-power versions of accelerator, regarding different ASR decoders.

optimizations. The *Lop_Opt_ASIC* reduces the energy of the accelerator by 290x with respect the *GPU* configuration. This additional decrease in energy is due to controlling the leakage power of the Hash tables by switching almost 97% of its partitions to the drowsy mode (see Figure 10).

On the other hand, Figure 13b shows the average power dissipation for the different systems, including both static and dynamic power. The *CPU* and the *GPU* dissipate 32.2 W and 76.4 W respectively when running the speech recognition software. Our accelerator provides a huge reduction in power with respect to the general purpose processors and GPUs, as its power dissipation is between 472 mW and 618 mW depending on the configuration. The prefetching architecture for the Arc cache increases power dissipation due to the significant reduction in execution time that it provides, as shown in Figure 11b. With respect to the initial *ASIC*, the configurations *ASIC+Arc* and *ASIC+State&Arc* achieve 1.87x and 1.94x speedup respectively. The hardware required to implement these improvements to the memory subsystem dissipates a very small percentage of total power. The Arc FIFO, the Request FIFO and the Reorder Buffer required for the prefetching architecture dissipate 4.83 mW, only 0.83% of the power of the overall accelerator. On the other hand, the extra hardware required for the State Issuer (comparators and table of offsets) dissipates 0.15 mW, 0.03% of the total power. In the best case, *Lop_Opt_ASIC* configuration achieves 22.64% power reduction versus the optimized version of accelerator, by only dissipating 5.3 mW (1.18% of the total power) for the change-mode counters and the drowsy control logic.

Furthermore, we evaluate the power and energy consumption of the accelerator for the other three ASR decoders, in order to show the general applicability of our proposal in energy-efficient Viterbi acceleration. Moreover, we also include the evaluation of the power-control mechanism for the different benchmarks. Figures 14a and 14b show the energy-savings and power dissipation of the accelerator, respectively, for the various speech datasets. As depicted in Figure 14a, we can achieve between 113x to 412x energy reduction for the optimized version of accelerator (*ASIC+State&Arc*) with respect to the *GPU*. In addition, we can further decrease energy by significantly shrinking the leakage power using the power-control mechanism, resulting in 123-454x energy-saving. The reason for a more modest benefit for the low-power drowsy technique, compared to Figure 13a, is that the power breakdown of the accelerator shows a lower percentage for the hash tables (26.2% on average) for the three decoders shown in Figure 14a. Moreover, the leakage power accounts for 44.4% of the total accelerator's power for these decoders, compared to the 59% in the case of Kaldi's English WFST. Regarding the power dissipation, the best ASIC configuration requires an average of 710 mW power. On average, we can reduce both energy and power and the accelerator by 15% using the power-control scheme, considering all the speech decoders.

As mentioned in Section 3, the main focus is to accel-



(a) Memory bandwidth

(b) Time vs Energy-consumption

Fig. 15: **a**) Memory traffic for the baseline ASIC and the version using the optimization for the state fetching presented in Section 4.2; **b**) Energy vs decoding time per one second of speech. The measurements are collected based on Kaldi's English dataset.

erate the Viterbi search which is the main bottleneck in ASR systems. However, we have also measured the delay and power of the GPU part in order to have a thorough evaluation of our method. Considering an audio file of about 2 minutes of speech and 56 batches of frames, the additional execution time regarding the rest parts running on GPU is 10.23% of total, partitioning into 6.12% and 4.11% for the DNN and backtracking phases respectively. Regarding the DNN implementation on GPU, we have used the latest version of the NVIDIA CUBLAS library [42] that reaches the best performance for the Kaldi's fully-connected DNN computation. The final results show that we have a speedup of 1.87x compared to the GPU running all the ASR application together. The 17% increase in the speedup is because that most of the DNN evaluation is overlapped with the decoding time since both GPU and the accelerator work in parallel, whereas the GPU runs the DNN and Viterbi search in a consecutive manner.

The memory bandwidth saving technique presented in Section 4.2 avoids 20% of the accesses to off-chip system memory as illustrated in Figure 15b. The baseline configuration for this graph is the initial *ASIC* design. The figure also shows the traffic breakdown for the different types of data stored in main memory: states, arcs, tokens and the overflow buffer. Our technique targets the memory accesses for fetching states, which represent 23% of the total traffic to off-chip memory. As it can be seen in the figure, our technique removes most of the off-chip memory fetches for accessing the states. The additional hardware included in the State Issuer is extremely effective to directly compute arc indices from state indices for the states with 16 or less arcs, without issuing memory requests to read the arc indices from main memory for those states. Note that in Figure 15b we do not include the configurations that employ the prefetching architecture, as this technique does not affect the memory traffic. Our prefetcher does not generate useless prefetch requests as it is based on computed addresses.

To sum up the energy-performance analysis, Figure 15a plots energy vs execution time per one second of speech. As it can be seen, the *CPU* exhibits the highest execution time and energy consumption. The *GPU* improves performance in one order of magnitude (9.8x speedup) with respect to the *CPU*, while reducing energy by 4.2x. The different versions of the accelerator achieve performance comparable or higher to the *GPU*, while providing an energy reduction of two orders of magnitude. Regarding the effect of the techniques to improve the memory subsystem, the prefetching architecture for the Arc cache provides significant benefits in performance and energy. On the other hand, the aim of the memory bandwidth saving technique for the State Issuer is to reduce the number of accesses to off-chip system memory. This technique achieves a reduction of 20% in the total number of accesses to off-chip DRAM as reported in Figure 15b. When using both techniques (configuration labeled as *ASIC+State*&*Arc*) the accelerator achieves 16.7x speedup and 1185x energy reduction with respect to the *CPU*. Compared to the *GPU*, this configuration provides 1.7x speedup and 287x energy reduction. Finally, the configuration with the low-power technique keeps the same performance as the ASIC with all the other optimizations, while deceasing the energy by two additional orders of magnitude, 1674x and 290x with respect to *CPU* and *GPU* respectively.

Finally, we evaluate the area of our accelerator for speech recognition. The total area for the initial design is 24.07 mm^2, a reduction of 16.53x with respect to the area of the NVIDIA GeForce GTX 980 (the die size for the GTX 980 is 398 mm^2 [43]). The hardware for the prefetching architecture in the Arc cache, i. e. the two FIFOs and the Reorder Buffer, produce a negligible increase of 0.05% in the area of the overall accelerator. The extra hardware for the State Issuer required for our bandwidth saving technique increase overall area by 0.02%. The power-control approach of the hash table represents only 0.04% of area overhead. In total, the area of the accelerator including all optimizations is 24.11 mm^2.

8 Related Work

Prior research into hardware acceleration for WFST-based speech recognition has used either GPUs, FPGAs or ASICs. Regarding the GPU-accelerated Viterbi search, Chong et al. [17], [40] proposed an implementation in CUDA that achieves 3.74x speedup with respect to a software decoder running on the CPU. We use this CUDA implementation as our baseline and show that an accelerator specifically designed for speech recognition achieves an energy reduction of two orders of magnitude with respect to the

GPU. The GPU is designed to perform many floating point operations in parallel and, hence, it is not well-suited for performing a memory intensive yet computationally limited WFST search.

Regarding the FPGA approach, Choi et al. [3], [44] present an FPGA implementation that can search a 5K-word WFST 5.3 times faster than real-time, whereas Lin et al. [45] propose a multi-FPGA architecture capable of decoding a WFST of 5K words 10 times faster than real-time. Their use of a small vocabulary of just 5K words allows them to avoid the memory bottlenecks that we have observed when searching large WFSTs. Our accelerator is designed for large-vocabulary speech recognition, which is 56 times faster than real-time when searching a 125K-word WFST.

Regarding the ASIC approach, Price et al. [5] developed a 6 mW accelerator for a 5K-word speech recognition system. Our work is different as we focus on large-vocabulary systems. On the other hand, Johnston et al. [4] proposed a low-power accelerator for speech recognition designed for a vocabulary of 60K words. Compared to the aforementioned accelerators, our proposal introduces two innovative techniques to improve the memory subsystem, which is the most important bottleneck in searching larger WFSTs: a prefetching architecture for the Arc cache and a novel bandwidth saving technique to reduce the number of off-chip memory accesses for fetching states from the WFST.

Prior work on hardware-accelerated speech recongnition also includes proposals that are not based on WFSTs [46], [47]. These systems use HMMs (Hidden Markov Models) to model the speech. In recent years, the WFST approach has been proven to provide significant benefits over HMMs [17], [27], especially for hardware implementations [44]. Hence, our accelerator focuses and is optimized for a WFST based approach.

9 Conclusions

In this paper we design a custom hardware accelerator for large-vocabulary, speaker-independent, continuous speech recognition, motivated by the increasingly important role of automatic speech recognition systems in mobile devices. We show that a highly-optimized CUDA implementation of the Viterbi algorithm achieves real-time performance on a GPU, but at a high energy cost. Our design includes innovative techniques to deal with memory accesses, which is the main bottleneck for performance and power in theses systems. In particular, we propose a prefetching architecture that hides main memory latency for a large fraction of the memory accesses with a negligible impact on area, providing 1.87x speedup with respect to the initial design. On the other hand, we propose a novel memory bandwidth saving technique that removes 20% of the accesses to off-chip system memory. To further improve the efficiency of the accelerator, a power control mechanism is introduced to remove most of the leakage power of the hash tables, which provides between 8.5% to 29.2% reduction of the total power dissipation. The final design including all the improvements achieves a range of speedups from 1.7x to 5.9x with respect to a modern high-end GPU, for a variety of speech decoders, while providing reductions in energy consumption between 123x and 454x.

Acknowledgment

This work has been supported by the the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU), and the ICREA Academia program.

References

[1] R. Yazdani, A. Segura, J. 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)*, Oct 2016, pp. 1–12.

[2] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 27–40. [Online]. Available: http://doi.acm.org/10.1145/2749469.2749472

[3] J. Choi, K. You, and W. Sung, "An fpga implementation of speech recognition with weighted finite state transducers," in *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, March 2010, pp. 1602–1605.

[4] J. R. Johnston and R. A. Rutenbar, "A high-rate, low-power, asic speech decoder using finite state transducers," in *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, July 2012, pp. 77–85.

[5] M. Price, J. Glass, and A. P. Chandrakasan, "A 6 mw, 5,000-word real-time speech recognizer using wfst models," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 102–112, Jan 2015.

[6] Google Now, https://en.wikipedia.org/wiki/Google_Now.

[7] Apple Siri, https://en.wikipedia.org/wiki/Siri.

[8] Microsoft Cortana, https://en.wikipedia.org/wiki/Cortana_%28software%29.

[9] "IBM Watson Speech to Text," http://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/speech-to-text.html.

[10] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely, "The kaldi speech recognition toolkit," in *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, Dec. 2011, iEEE Catalog No.: CFP11SRW-USB.

[11] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[12] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer."

[13] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 92–104. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750389

[14] J. Chong, E. Gonina, and K. Keutzer, "Efficient automatic speech recognition on the gpu," *Chapter in GPU Computing Gems Emerald Edition, Morgan Kaufmann*, vol. 1, 2011.

[15] A. L. Janin, "Speech recognition on vector architectures," Ph.D. dissertation, University of California, Berkeley, 2004.

[16] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[17] K. You, J. Chong, Y. Yi, E. Gonina, C. J. Hughes, Y. K. Chen, W. Sung, and K. Keutzer, "Parallel scalability in speech recognition," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 124–135, November 2009.

[18] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, 2002, pp. 148–157.

[19] R. Vergin, D. O'Shaughnessy, and A. Farhat, "Generalized mel frequency cepstral coefficients for large-vocabulary speaker-independent continuous-speech recognition," *Speech and Audio Processing, IEEE Transactions on*, vol. 7, no. 5, pp. 525–532, Sep 1999.

[20] L. Bahl, R. Bakis, P. Cohen, A. Cole, F. Jelinek, B. Lewis, and R. Mercer, "Further results on the recognition of a continuously read natural corpus," in *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '80.*, vol. 5, Apr 1980, pp. 872–875.

[21] Y. Wang, V. Panayotov, I. Edrenkin, D. Povey, and G. Chen, "Kaldi speech recognition results," 2019. [Online]. Available: https://github.com/kaldi-asr/kaldi/blob/master/egs/librispeech/s5/RESULTS

[22] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. Engel, L. Fan, C. Fougner, T. Han, A. Y. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Y. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu, "Deep speech 2: End-to-end speech recognition in english and mandarin," *CoRR*, vol. abs/1512.02595, 2015.

[23] A. Zeyer, K. Irie, R. Schlüter, and H. Ney, "Improved training of end-to-end attention models for speech recognition," *CoRR*, vol. abs/1805.03294, 2018. [Online]. Available: http://arxiv.org/abs/1805.03294

[24] N. Zeghidour, Q. Xu, V. Liptchinsky, N. Usunier, G. Synnaeve, and R. Collobert, "Fully convolutional speech recognition," *CoRR*, vol. abs/1812.06864, 2018. [Online]. Available: http://arxiv.org/abs/1812.06864

[25] M. Mohri, F. Pereira, and M. Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech and Language*, vol. 16, no. 1, pp. 69 – 88, 2002.

[26] X. Lingyun and D. Limin, "Efficient viterbi beam search algorithm using dynamic pruning," in *Signal Processing, 2004. Proceedings. ICSP '04. 2004 7th International Conference on*, vol. 1, Aug 2004, pp. 699–702 vol.1.

[27] S. Kanthak, H. Ney, M. Riley, and M. Mohri, "A comparison of two lvr search optimization techniques," in *IN PROC. INT. CONF. SPOKEN LANGUAGE PROCESSING*, 2002, pp. 1309–1312.

[28] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Software, IEE Proceedings-*, Feb 2004, pp. 96–96.

[29] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, ser. MICRO 25. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 102–110. [Online]. Available: http://dl.acm.org/citation.cfm?id=144953.145006

[30] J. E. Smith, "Decoupled access/execute computer architectures," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 289–308, Nov. 1984. [Online]. Available: http://doi.acm.org/10.1145/357401.357403

[31] H. Igehy, M. Eldridge, and K. Proudfoot, "Prefetching in a texture cache architecture," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, ser. HWWS '98. New York, NY, USA: ACM, 1998, pp. 133–ff. [Online]. Available: http://doi.acm.org/10.1145/285305.285321

[32] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: An asr corpus based on public domain audio books," in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, April 2015, pp. 5206–5210.

[33] A. Rousseau, P. Delglise, and Y. Estve, "Enhancing the ted-lium corpus with selected data for language modeling and more ted talks," in *In Proc. LREC*, 2014, pp. 26–31.

[34] V. S. Corpus, "http://www.voxforge.org," 2009.

[35] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-vdd: A circuit technique to reduce leakage in deep-submicron cache memories," in *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, ser. ISLPED '00. New York, NY, USA: ACM, 2000, pp. 90–95. [Online]. Available: http://doi.acm.org/10.1145/344166.344526

[36] K. Nii, H. Makino, Y. Tujihashi, C. Morishima, Y. Hayakawa, H. Nunogami, T. Arakawa, and H. Hamano, "A low power sram using auto-backgate-controlled mt-cmos," in *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, Aug 1998, pp. 293–298.

[37] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures."

[38] S. Li, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "McPAT 1.0: An Integrated Power, Area, and Timing Modeling Framework for Multicore Architectures," Tech. Rep.

[39] V. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with papi," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, Sept 2012, pp. 262–268.

[40] J. Chong, E. Gonina, and K. Keutzer, "Efficient automatic speech recognition on the gpu," Chapter in GPU Computing Gems Emerald Edition, Morgan Kaufmann, 2011.

[41] NVIDIA Visual Profiler, https://developer.nvidia.com/nvidia-visual-profiler.

[42] NVIDIA, "Cuda toolkit 4.2 cublas library," 2012. [Online]. Available: https://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf

[43] GeForce 900 series, https://en.wikipedia.org/wiki/GeForce_900_series.

[44] K. You, J. Choi, and W. Sung, "Flexible and expandable speech recognition hardware with weighted finite state transducers," *Journal of Signal Processing Systems*, vol. 66, no. 3, pp. 235–244, 2011.

[45] E. C. Lin and R. A. Rutenbar, "A multi-fpga 10x-real-time high-speed search engine for a 5000-word vocabulary speech recognizer," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '09. New York, NY, USA: ACM, 2009, pp. 83–92.

[46] P. J. Bourke, "A low-power hardware architecture for speech recognition search," Ph.D. dissertation, Carnegie Mellon University, 2011.

[47] B. Mathew, A. Davis, and Z. Fang, "A low-power accelerator for the sphinx 3 speech recognition system," in *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '03. New York, NY, USA: ACM, 2003, pp. 210–219.

Reza Yazdani is a PhD researcher at the Universitat Politècnica de Catalunya. He is currently pursuing the fourth year of his study on designing high-performance hardware accelerators for the low-power mobile systems. His research interests include high-performance and low-power accelerators for cognitive computing architectures, embedded systems, IoT and edge devices, and VLSI design. He received MS in computer architecture from University of Tehran.

Jose-Maria Arnau received Ph.D. on Computer Architecture from the Universitat Politecnica de Catalunya (UPC) in 2015. He is a postdoctoral researcher at UPC BarcelonaTech and a member of the ARCO (ARchitecture and COmpilers) research group at UPC. His research interests include low-power architectures for cognitive computing, especially in the area of automatic speech recognition and object recognition.

Antonio González (Ph.D. 1989) is a Full Professor at the Computer Architecture Department of the Universitat Politecnica de Catalunya, Barcelona (Spain), and the director of the Microarchitecture and Compiler research group. He was the founding director of the Intel Barcelona Research Center from 2002 to 2014. His research has focused on computer architecture, compilers and parallel processing, with a special emphasis on microarchitecture and code generation. He has published over 370 papers and has served as associate editor of five IEEE and ACM journals, program chair for ISCA, MICRO, HPCA, ICS and ISPASS, general chair for MICRO and HPCA, and PC member for more than 130 symposia. He is an IEEE Fellow.