An Ultra Low-Power Hardware Accelerator for Automatic Speech Recognition

Reza Yazdani, Albert Segura, Jose-Maria Arnau, Antonio Gonzalez
Computer Architecture Department, Universitat Politecnica de Catalunya
{ryazdani, asegura, 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 a high energy cost which is not affordable for the tiny power budget of mobile devices. Hardware acceleration can reduce power 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, that represents the main bottleneck in an ASR system. The proposed design includes innovative techniques to improve the memory subsystem, since memory is identified as the main bottleneck for performance and power in the design of these accelerators. We propose a prefetching scheme tailored to the needs of an ASR system that hides main memory latency for a large fraction of the memory accesses with a negligible impact on area. In addition, we introduce a novel bandwidth saving technique that removes 20% of the off-chip memory accesses issued during the Viterbi search.

The proposed design outperforms software implementations running on the CPU by orders of magnitude and achieves 1.7x speedup over a highly optimized CUDA implementation running on a high-end Geforce GTX 980 GPU, while reducing by two orders of magnitude (287x) the energy required to convert the speech into text.

I. Introduction

Automatic Speech Recognition (ASR) has attracted the attention of the architectural community [1], [2], [3], [4] and the industry [5], [6], [7], [8] 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.

978-1-5090-3508-3/16/$31.00

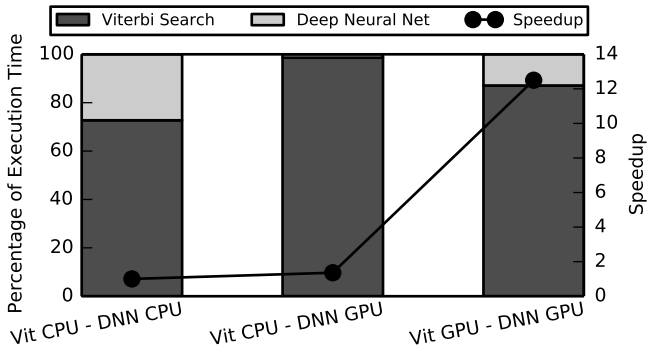


Fig. 1: Percentage of execution time for the two components of a speech recognition system: the Viterbi search and the Deep Neural Net. The Viterbi search is the dominant component, as it requires 73% and 86% of the execution time when running on a CPU and a GPU respectively.

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 [9], 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.

In recent years there has been a lot of work on boosting the performance of DNNs by using GPUs [10], [11], [12] or dedicated accelerators [13], [14], achieving huge speedups and energy 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 [12], [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 [10]. 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. Furthermore, we analyze the memory behavior of our accelerator and 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.

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 1.7x speedup over a high-end desktop GPU, while consuming 287x 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.

The remainder of this paper is organized as follows. Section II provides the background information on speech recognition systems. Section III presents our base design for speech recognition. Section IV introduces two new techniques to hide memory latency and save memory bandwidth. Section V describes our evaluation methodology and Section VI shows the performance and power results. Section VII reviews some related work and, finally, Section VIII sums up the main conclusions.

II. 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) [17]. 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 [18] 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 III. The complexity of the search process is due to the huge size of the recognition model employed to represent the characteristics of the speech.

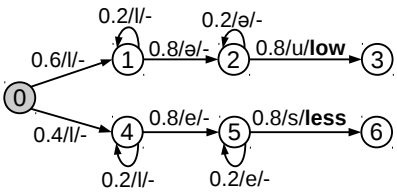
The state-of-the-art in recognition networks for speech is the Weighted Finite State Transducer (WFST) [19] approach. A WFST is a Mealy finite state machine that encodes a mapping between input and output labels associated with weights. In the case of speech recognition, the input labels represent the 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 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:

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

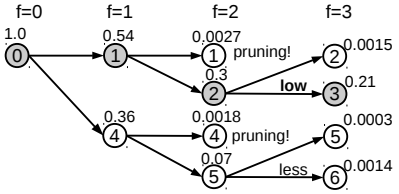
where w_{ij} is the weight of the arc from state i to state j, and $b(O_f; m_k)$ is the probability that the observation vector O_f



(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*.

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: $\psi_0(s_0) \cdot w_{01} \cdot b(O_1; l) = 1.0 \cdot 0.6 \cdot 0.9 = 0.54$. 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 [20].

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 [10]. 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 [21].

III. 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 speed-up 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 tokens for the current and next frames

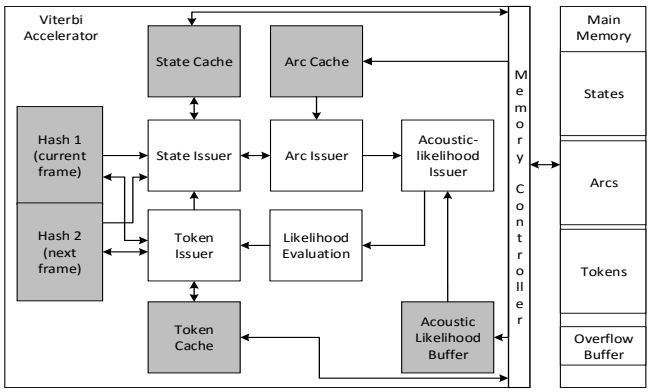


Fig. 3: Architecture of the accelerator for speech recognition.

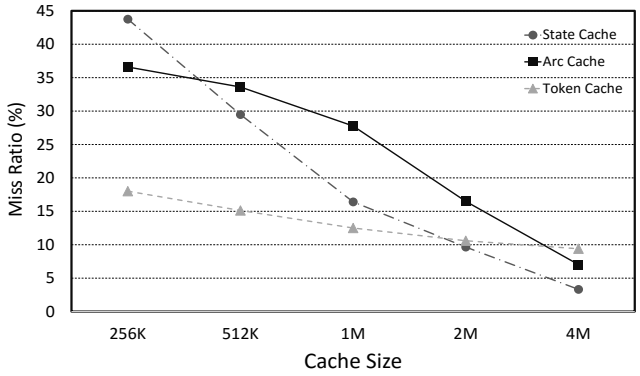


Fig. 4: Miss ratio vs capacity for the different caches in the accelerator.

of the speech, and a scratchpad memory to store the acoustic likelihoods computed by the DNN.

The accelerator cannot store the WFST on-chip due to its huge size. A typical WFST such as the one used in the experiments of this work [9] 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 618 MBytes. Regarding the representation of the WFST, we use the memory layout proposed in [2]. 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 are 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.

Due to the pruning described in Section II, 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.

A. Overall ASR System

Our ASR system is based on the hybrid approach that combines a DNN for acoustic scoring with the Viterbi beam search, which is the state-of-the-art scheme in speech recognition. DNN evaluation is easy to parallelize and, hence, it achieves high performance and energy-efficiency on a GPU. On the other hand, the Viterbi search is the main bottleneck and it is hard to parallelize, so we propose a dedicated hardware accelerator specifically tailored to the needs of Viterbi beam search algorithm. Therefore, our overall ASR system combines the GPU, for DNN evaluation, with an accelerator for Viterbi search.

In our ASR system, the input frames of speech are grouped in batches. The accelerator and the GPU work in parallel processing different batches in a pipelined manner: the GPU computes the DNN for the current batch while the accelerator performs Viterbi search for the previous batch. Note that the results computed by the GPU, i. e. the acoustic scores for one batch, have to be transferred from GPU memory to accelerator's memory. In order to hide the latency required for transferring acoustic scores, a double-buffered memory is included in the accelerator, named *Acoustic Likelihood Buffer* in Figure 3. This memory stores the acoustic scores for the current and the next frame of speech. The accelerator decodes the current frame of speech while the acoustic scores for the next frame are fetched from memory, overlapping computations with memory accesses.

B. Accelerator Pipeline

The Viterbi accelerator, illustrated in Figure 3, implements the Viterbi beam search algorithm described in Section II. We use the example shown in Figure 2 to illustrate the behavior of the accelerator. More specifically, we describe how the hardware expands the arcs of frame 2 to generate the new tokens in frame 3. Initially, the hash table for the current frame (Hash 1) stores the information for tokens 1, 2, 4 and 5. The State Issuer fetches these tokens from the hash table, including the index of the associated WFST state and the likelihood of reaching the token. The State Issuer then performs the pruning

using a threshold of 0.05: the likelihood of the best token (0.3) minus the beam (0.25). Tokens 1 and 4 are pruned away as their likelihoods are smaller than the threshold, whereas tokens 2 and 5 are considered for arc expansion. Next, the State Issuer generates memory requests to fetch the information of WFST states 2 and 5 from main memory, using the State Cache to speed up this process. The state information includes the index of the first arc and the number of arcs. Once the State Cache serves these data, they are forwarded to the Arc Issuer.

The Arc Issuer generates memory requests to fetch the outgoing arcs for states 2 and 5, using the Arc Cache to speed up the arc fetching. The arc information includes the index of the destination state, weight, phoneme index and word index. For example, the second arc of state 2 has destination state 3, weight equal to 0.8 and it is associated with phoneme *u* and word *low*. Once the Arc Cache provides the data of an arc, it is forwarded to the next pipeline stage. The Acoustic Likelihood Issuer employs the phoneme's index to fetch the corresponding acoustic score computed by the DNN. A local on-chip memory, the Acoustic Likelihood Buffer, is used to store all the acoustic scores for the current frame of speech.

In the next pipeline stage, the Likelihood Evaluation, all the data required to compute the likelihood of the new token according to Equation 1 is already available: the likelihood of reaching the source token, the weight of the arc and the acoustic score from the DNN. The Likelihood Evaluation performs the summation of these three values for every arc to compute the likelihoods of reaching the new tokens in the next frame. Note that additions are used instead of multiplications as the accelerator works in log-space. In the example of Figure 2c, the second arc of token 2 generates a new token in the next frame, token 3, with likelihood 0.21: the likelihood of the source token is 0.3, the weight of the arc is 0.8 (see Figure 2a) and the acoustic score for phoneme *u* in frame 3 is 0.9 (see Figure 2b).

Finally, the Token Issuer stores the information for the new tokens in the hash table for the next frame (Hash 2). In this example, it saves the new tokens for states 2, 3, 5 and 6 together with their associated likelihoods in the hash table. In addition, the index of the source token and the word index are stored in main memory, using the Token Cache, as this information is required for backtracking in order to recover the best path. Note that different arcs might have the same destination state. In that case, the Token Issuer only saves the best path for reaching the destination state, i. e. the path with maximum likelihood among all the different ingoing arcs.

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 GPU and the result is transferred to the aforementioned buffer in the accelerator. The buffer contains storage for two frames of speech. The accelerator expands the tokens for the current frame while it fetches the acoustic scores for the next frame, overlapping computations with memory accesses.

The result generated by the accelerator is the dynamic trace of tokens in main memory (see Figure 2c), 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.

C. 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 IV.

Regarding the hash tables, Figure 5 shows the average number of cycles per request versus the number of entries

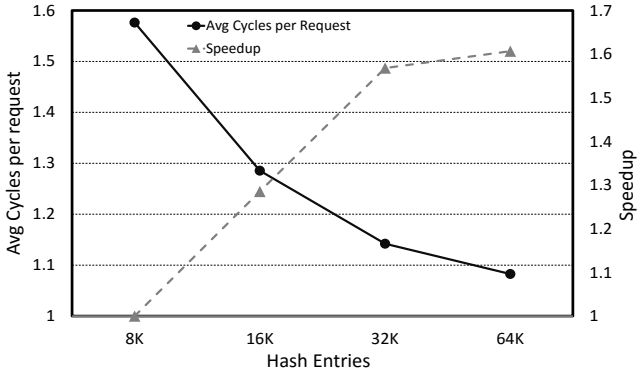


Fig. 5: Average cycles per request to the hash table and speedup vs number of entries.

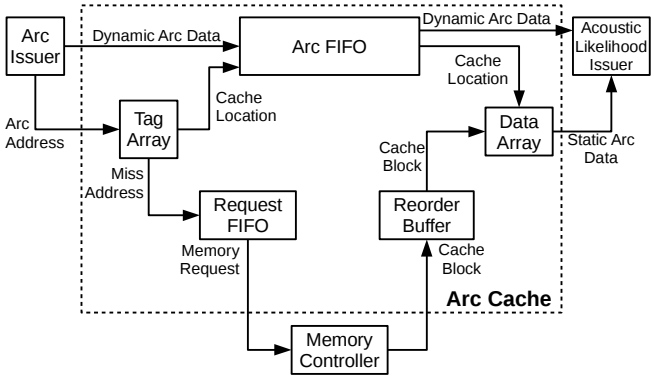


Fig. 6: Prefetching architecture for the Arc cache.

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.

IV. Improved Memory Hierarchy

In this section, we perform an analysis of the bottlenecks in the hardware accelerator for speech recognition presented in Section III, 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 I. Therefore, we focus our efforts on hiding the memory latency in an energy-efficient manner. In Section IV-A we introduce an area-effective latency-tolerance technique that is inspired by decoupled access-execute architectures.

On the other hand, off-chip DRAM accesses are known to be particularly costly in terms of energy [13]. To further improve the energy-efficiency of our accelerator, we present a novel technique for saving memory bandwidth in Section IV-B. This technique is able to remove a significant percentage of the memory requests for fetching states from the WFST.

A. 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 III. 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. Another solution to hide memory latency is hardware prefetching [22]. 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

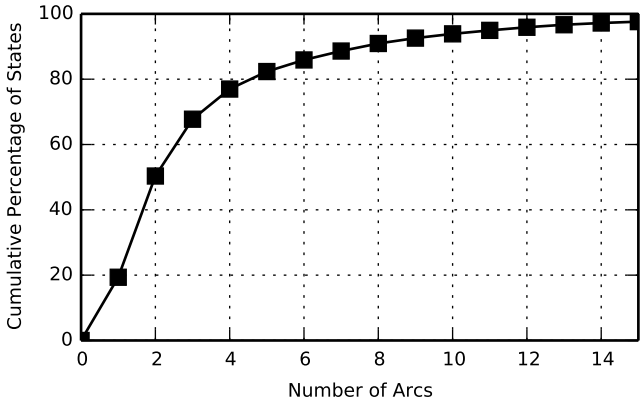


Fig. 7: Cumulative percentage of states accesses dynamically vs the number of arcs. Although the maximum number of arcs per state is 770, 97% of the states fetched from memory have 15 or less arcs.

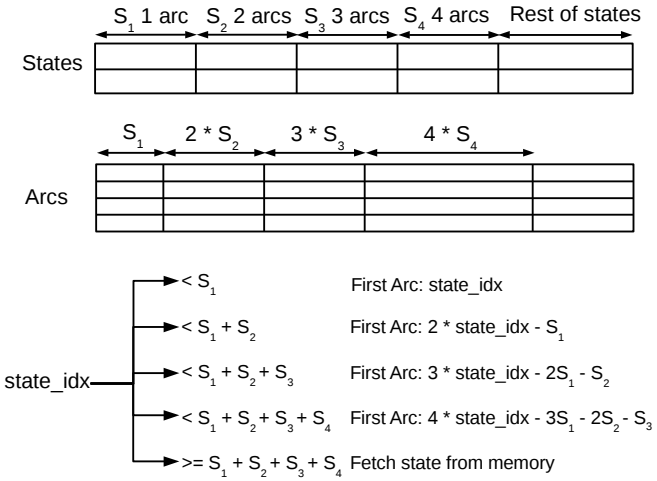


Fig. 8: Changes to the WFST layout. In this example, we can directly compute arc index from state index for states with 4 or less arcs.

ineffective. We implemented and evaluated different state-of-the-art hardware prefetchers [22], [23], 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 [24]. 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 [25]. 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 VI 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.

B. Reducing Memory Bandwidth

The accelerator presented in Section III 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

TABLE I: 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 II: 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 III: GPU parameters.

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

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%.

V. Evaluation Methodology

We have developed a cycle-accurate simulator that models the architecture of the accelerator presented in Section III. Table I shows the parameters employed for the experiments in the accelerator. We modified the simulator to implement the prefetching scheme described in Section IV-A. We use 64 entries for Arc FIFO, Request FIFO and Reorder Buffer in order to hide most of the memory latency. Furthermore, we have implemented the bandwidth saving technique proposed in Section IV-B. 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.

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 [26], a.k.a. enhanced CACTI [27] 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 our datasets, we use the WFST for English language provided in the Kaldi toolset [9], which is created from a vocabulary of 125k words, and the audio files from Librispeech corpus [28].

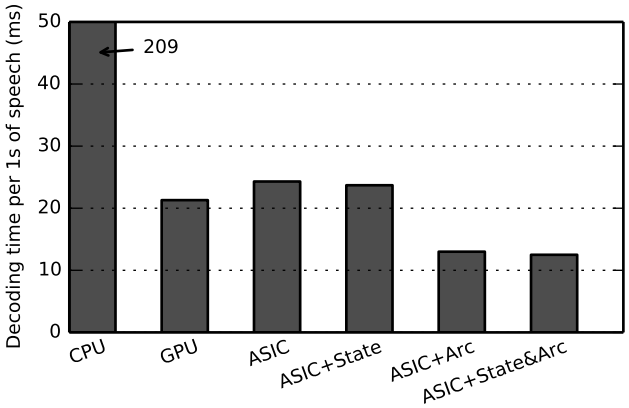


Fig. 9: Decoding time, i. e. time to execute the Viterbi search, per second of speech. Decoding time is smaller than one second for all the configurations, so all the systems achieve real-time speech recognition.

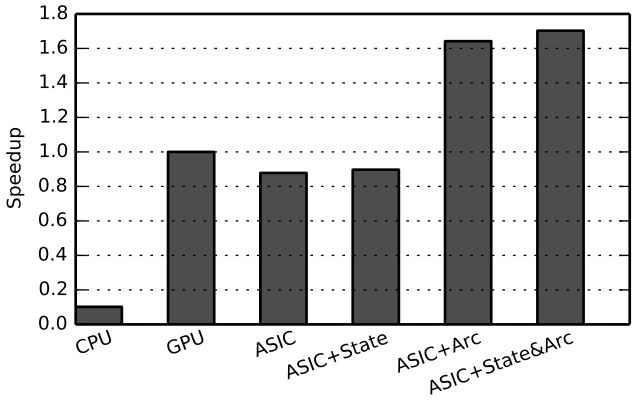


Fig. 10: Speedups achieved by the different versions of the accelerator. The baseline is the GPU.

A. 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 II. We employ Intel RAPL library [29] to measure energy consumption. We use GCC 4.8.4 to compile the software using -O3 optimization flag.

B. GPU Implementation

We have implemented the state-of-the-art GPU version of the Viterbi search presented in [10]. Furthermore, we have incorporated all the optimizations described in [30] 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 III and use the NVIDIA Visual Profiler [31] to measure performance and power.

VI. 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. Figure 9 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 II. *GPU* refers to the CUDA version running on the GPU presented in Table III. *ASIC* is our accelerator described in Section III with parameters shown in Table I. *ASIC+State* corresponds to the bandwidth saving technique for the State Issuer presented in Section IV-B. *ASIC+Arc* includes the prefetching architecture for the Arc cache presented in Section IV-A. 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 9, 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 10 shows the speedups with respect to the *GPU* for the same configurations. The initial design of the *ASIC* achieves 88% of the performance of the *GPU*. The *ASIC+State* achieves 90% of the *GPU* performance. This configuration includes a bandwidth saving techinque 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.

Our accelerator for speech recognition provides a huge reduction in energy consumption as illustrated in Figure 11. The numbers include both static and dynamic energy. The base *ASIC* configuration reduces energy consumption by 171x with respect to the *GPU*, whereas the optimized version

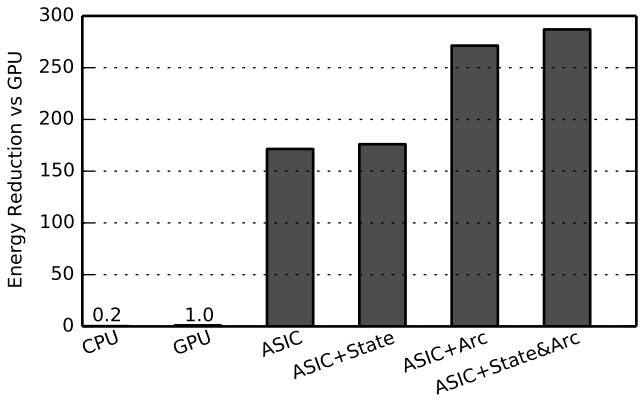


Fig. 11: Energy reduction vs the GPU, for different versions of the accelerator.

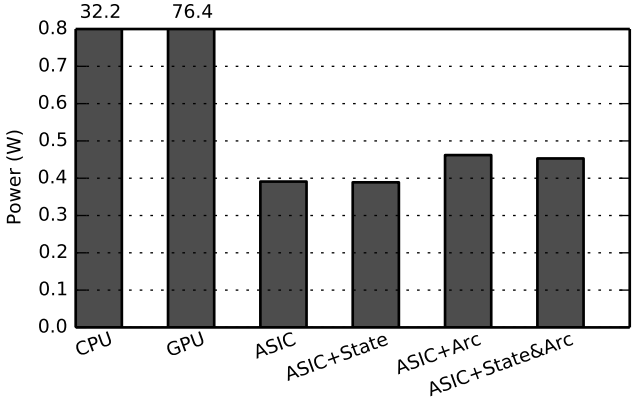


Fig. 12: Power dissipation for the CPU, GPU and different versions of the accelerator.

using the proposed improvements for the memory subsystem (*ASIC+Arc&State*) reduces energy by 287x. 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.

On the other hand, Figure 12 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 389 mW and 462 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 10. 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 1.07% 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.

The Viterbi search represents the main bottleneck by far in state-of-the-art ASR systems and, therefore, our research focuses on this stage. Nevertheless, we have also evaluated the performance of the entire ASR pipeline, including the DNN evaluation and the Viterbi search. More specifically, we compare the performance of a system that runs both stages on the GPU with our ASR system that combines the GPU, for DNN evaluation, with our accelerator for Viterbi beam search. Our results indicate that our system combining GPU with Viterbi accelerator achieves 1.87x speedup over a GPU-only system. The improvement in performance comes from two sources. First, the accelerator achieves 1.7x speedup over the GPU for the Viterbi search as shown in Figure 10. Second, our ASR system is able to run the DNN and the Viterbi search in parallel by using the GPU and the accelerator, whereas the GPU-only system has to execute these two stages sequentially.

The memory bandwidth saving technique presented in Section IV-B avoids 20% of the accesses to off-chip system memory as illustrated in Figure 13. 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 13 we do not include the configurations that employ the prefetching architecture, as this technique does not affect the memory traffic. Our prefetcher does not genereate useless prefetch requests as it is based on computed addresses.

To sum up the energy-performance analysis, Figure 14 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

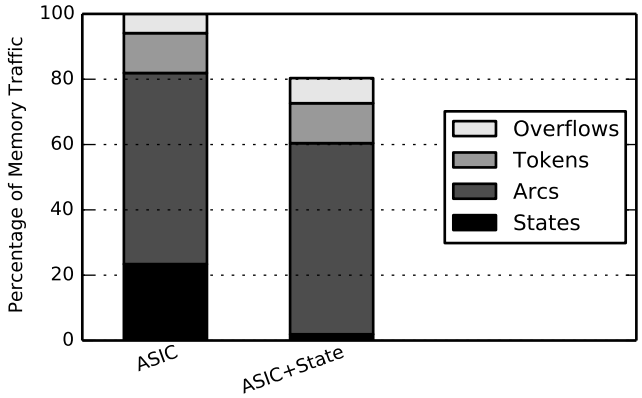


Fig. 13: Memory traffic for the baseline ASIC and the version using the optimization for the state fetching presented in Section IV-B.

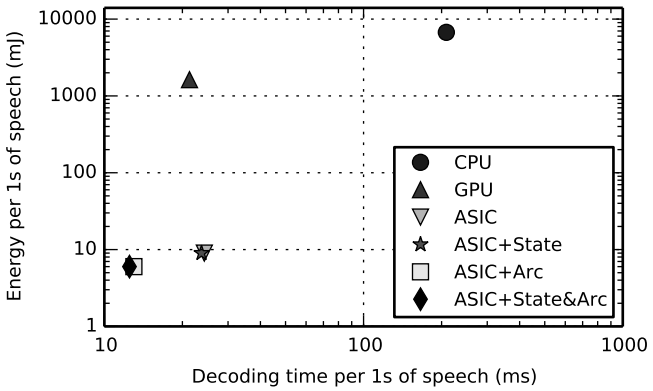


Fig. 14: Energy vs decoding time per one second of speech.

DRAM as reported in Figure 13. 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, we evaluate the area of our accelerator for speech recognition. The total area for the initial design is 24.06 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 [32]). 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 total area for the accelerator including both techniques is 24.09 mm^2.

VII. 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. [10], [30] 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. [2], [33] present an FPGA implementation that can search a 5K-word WFST 5.3 times faster than real-time, whereas Lin et al. [34] 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, it is 56 times faster than real-time when searching a 125K-word WFST.

Regarding the ASIC approach, Price et al. [4] 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. [3] 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 [35], [36]. 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 [10], [21], especially for hardware implementations [33]. Hence, our accelerator focuses and is optimized for a WFST based approach.

VIII. 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. The final design including both improvements achieves a

1.7x speedup with respect to a modern high-end GPU, while reducing energy consumption by 287x.

ACKNOWLEDGMENT

This work is supported by the Spanish Ministry of Economy and Competitiveness and FEDER funds of the EU under contract TIN2013-44375-R.

REFERENCES

[1] 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

[2] 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.

[3] 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.

[4] 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.

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

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

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

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

[9] 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.

[10] 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.

[11] J. Chong, E. Gonina, Y. Yi, and K. Keutzer, "A fully data parallel wfstbased large vocabulary continuous speech recognition on a graphics processing unit," in *in 10th Annual Conference of the International Speech Communication Association (InterSpeech*, 2009.

[12] 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.

[13] 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," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 609–622. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.58

[14] 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

[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] 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.

[18] 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.

[19] 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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0885230801901846

[20] 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.

[21] 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.

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

[23] 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

[24] 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

[25] 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

[26] 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," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 469–480. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669172

[27] 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.

[28] 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.

[29] 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.

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

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

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

[33] 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. [Online]. Available: http://dx.doi.org/10.1007/s11265-011-0587-9

[34] 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. [Online]. Available: http://doi.acm.org/10.1145/1508128.1508141

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

[36] 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. [Online]. Available: http://doi.acm.org/10.1145/951710.951739