UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONA**TECH**

Facultat d'Informàtica de Barcelona

Master in Innovation and Research in Informatics

High Performance Computing

Master Thesis

Characterization of Speech Recognition Systems on GPU Architectures

Author:	Albert Segura Salvador
Advisor:	Antonio Gonzalez (DAC)
Co-Advisor:	Jose-Maria Arnau (DAC)
Date:	July 4, 2016

Abstract

Automatic speech recognition is one of the most important applications in the area of cognitive computing. Mobile devices, such as smartphones, have incorporated speech recognition as one of the main interfaces for user interaction. This trend towards voice-based user interfaces is likely to continue in the next years. Effective speech recognition systems require real-time recognition, which involves a huge effort for CPU architectures to reach it. GPU architectures offer parallelization capabilities which can be exploited to increase the performance of speech recognition systems. However, efficiently utilizing the GPU resources for speech recognition is challenging, as the software implementations exhibit irregular and unpredictable memory accesses and poor temporal locality.

Our key ambition is to characterize the performance and energy bottlenecks of speech recognition systems when running on a modern GPU, with the aim of providing useful information for designing future GPU architectures. First, we develop a GPU version of the Viterbi search algorithm, which is known to be the main bottleneck by far in speech recognition systems. Second, we analyse the GPU architecture to find the main sources of stalls in the pipeline and the energy bottlenecks. We show that memory stalls are the main reason for the low utilization of GPU resources.

We then focus on the exploration of a number of architectural modifications to state-of-the-art GPU architectures in order to deal with the performance limiting factors, i.e. the memory bottlenecks, and propose a GPU configuration highly tuned for speech recognition. The exploration evaluates different parameters for the memory hierarchy, including the L1 data cache, the L2 cache and the memory controller. We also consider modifications to the core resources and frequency scaling, in order to significantly reduce the number of idle cycles waiting for the memory and the underutilization of functional units. Our proposed GPU configuration is able to achieve real-time performance for large-vocabulary speech recognition, while increasing the issue rate from 5.1% to 18.1%, and achieving a power reduction of 31.6%, an energy reduction of 24% and area shrinkage of 17.96%.

Keywords

Speech recognition, Viterbi, Kaldi, GPU, GPGPU-Sim.

Contents

List of Figures

1 Introduction

1.1 Motivation

Automatic Speech Recognition (ASR) is rising as a core technology for next generation smart devices. ASR systems allow multimedia content to be transcribed from acoustic waveforms to word sequences. This technology is emerging as a critical component in data analytics for a wealth of media data that is being generated everyday. Commercial usage scenarios are already appearing in industry such as broadcast news transcription [14, 23], voice search [28], automatic meeting diarization [1] or real-time speech translation in video calls [29] Moreover, voice-based user interfaces are becoming increasingly popular, significantly changing the way people interact with computers. Assistant tools like Google Now [12] from Google, Siri [2] from Apple or Cortana [21] from Microsoft rely on ASR as the main interaction system with the user on mobile devices. We can only expect a broader adoption of this interaction system with the anticipated popularization of IoT devices and wearables.

To ensure a compelling user experience it is critical that the ASR systems are accurate, have low latency and provide sufficient coverage over the extremely large vocabularies that the system may encounter. In order to obtain high recognition accuracy, state-of-the-art ASR systems may perform recognition with large vocabularies of hundreds of thousands or even millions of words, massive acoustic models with millions of parameters and extremely large language models. Such enormous models can be applied in offline speech recognition tasks, but they are impractical for real-time software-based ASR systems running on the CPU due to the large computational cost required to convert the speech into words.

Graphics Processing Units (GPUs) enable tremendous compute capabilities in personal desktop and laptop systems. The CUDA [10] programming model from NVIDIA provides an implementation path for a broad spectrum of applications beyond graphics processing, such as speech recognition. GPU-accelerated ASR systems take advantage of the high throughput capabilities of graphics processors to deliver large-vocabulary speech recognition in real-time [9, 8]. By using the GPU, thousands of alternative interpretations of the input speech utterance can be evaluated in parallel to find the most likely sequence of words.

Despite the performance improvements over CPU-based solutions achieved by state-of-the-art CUDA implementations, ASR is still a challenging problem for GPUs. Large-vocabulary ASR systems use a graph-based recognition network, which is a language database created offline from different knowledge sources using powerful statistical learning techniques. The recognition process requires performing a search on this irregular recognition network to find the most likely word sequence for the input speech signal. Parallel graph traversal on large unstructured graphs is a well-known challenge for parallel architectures [16], especially in the context of ASR [15]. ASR systems exhibit unpredictable memory accesses and poor data locality, these characteristics put significant pressure on the GPU memory subsystem and lead to significant underutilization of GPU functional units.

Properly understanding and characterizing the issues of CUDA-based ASR systems is necessary for improving the performance and energy-efficiency of GPUs when running speech recognition software. Identifying the sources of stalls in the GPU pipeline as well as the main sources of energy drain will provide essential information for designing future GPU architectures, increasing GPU efficiency for ASR applications. Note that ASR is an open research problem and accuracy of state-of-the-art real-time systems is still far from being completely satisfactory [31], especially in noisy environments. Improving the performance of ASR on GPUs will allow the use of more complex speech models with larger vocabularies in real-time, which in turn will provide improvements in accuracy, delivering a more satisfactory experience for end users.

In this Master thesis we analyse the performance and energy consumption of a modern GPU when running a state-of-the-art CUDA-based ASR system. We collect numbers on an NVIDIA GTX 980 GPU to identify the main bottlenecks in the software. Furthermore, we use a cycle-accurate GPU simulator to get more insights on the performance and energy bottlenecks, identifying the most critical components of the microarchitecture. Finally, we perform a design space exploration on the most critical GPU parameters and conclude the thesis with a proposal of a GPU configuration that is better suited for ASR, as it improves GPU utilization by 18.1%, providing 31.6% power savings over the baseline GTX 980 GPU.

1.2 Objectives and contributions

The first objective is to implement in CUDA the main component of modern ASR systems, the Viterbi search algorithm [27]. Open-source ASR toolkits only provide CPU implementations, as the Viterbi search is a challenging algorithm to parallelize [8]. Although several GPU versions have been proposed in the literature [8, 31], the source code for those proposals is not available. We will port the Viterbi search implementation of Kaldi [26], a widely used open-source ASR system, to a GPU including the most important optimizations described in previous work [8, 31].

The second objective is to analyse the performance of the GPU-accelerated speech recognition system, identifying the main bottlenecks in the CUDA implementation. The different CUDA kernels will be analysed to evaluate their execution time and power dissipation on a modern GPU.

The third objective is to characterize the performance and energy consumption of a modern GPU microarchitecture when running ASR systems, by using a cycle-accurate GPU simulator. The target is to characterize the sources of stalls in the GPU pipeline and the main sources of energy drain, in order to identify most critical components of the GPU microarchitecture for ASR applications.

Finally, the last objective is to find optimal parameters for the GPU components that have been identified as critical for ASR applications in the performance and energy characterization. We will perform a design space exploration for those components, analysing the impact of different GPU configurations in performance, power and area. The target is to find a GPU configuration that is highly tuned for ASR applications.

In this Master thesis we claim the following contributions:

- We implement in CUDA the Viterbi search algorithm for ASR systems. Our implementation is based on the state-of-the-art GPU version described in [9], including the optimizations for improving GPU performance presented in [8].
- We characterize the performance and energy bottlenecks, profiling the CUDA implementation on a modern NVIDIA GTX 980.
- We analyse the main sources of GPU stalls in the microarchitecture. We also characterize the energy consumption of the different components in the GPU pipeline when running ASR software.
- We perform a design space exploration to evaluate GPU performance, power and area for different configurations, changing the most critical parameters for ASR applications.
- We propose a GPU configuration that improves GPU functional units utilization by 18.1%, reduces power by 31.6%, energy by 24% and area by 17.96%, while achieving similar performance to the baseline GPU.

1.3 Organization

The remainder of this report is structured as follows. In Chapter 2 we provide basic background information on the Speech recognition algorithms and GPU programming model and hardware organization. In Chapter 3 we introduce the evaluation methodology, and describe the tools used for the evaluation of the following Chapters. In Chapter 4 we describe the GPU Viterbi algorithm implementation. In Chapter 5 we evaluate the GPU Viterbi algorithm on a baseline architecture. In Chapter 6 we present a design space exploration of the GPU architecture for the Viterbi algorithm and propose an improved hardware configuration. Finally in Chapter 7 we present the main conclusions of the report.

2 Related work

2.1 Automatic Speech Recognition

Automatic Speech Recognition consists on processing an input speech waveform and identifying a sequence of words from it. The process is based on two main stages: a feature extraction and scoring, implemented by a Deep Neural Network (DNN) and a search phase, implemented with the Viterbi search algorithm.

The Viterbi search algorithm is the dominant component of a speech recognition system like Kaldi, where it requires from 73% to 86% of the execution time when running on a CPU and a GPU respectively, while DNN only the remaining percentage.

Speech properties

Speech has a complex structure in which the audio stream is made up of stable and dynamic states. In this change of states one can define classes of sounds or phones which constitutes the basis of a word, but a given phone can be significantly different when factors like the speaker or the phone context changes. Changes between states often contain more information than stable states, for this reason we use figures like: diphones, which define the part of the speech between two consecutive phones, and triphones [3], which contain a phone and its surrounding part of speech. To detect diphones and triphones correctly, we detect senones which represent smaller partitions of distinguishable sounds. Finally, fillers like words or respirations are classified as utterances.

2.1.1 Speech recognition process

A typical ASR procedure works as follows. First the input audio is typically split in segments of 10 ms of speech. Second each frame is processed to extract a vector of features out of it. Third, using the acoustic model, implemented by a DNN, the vector features are translated to phonemes probabilities. Finally the Viterbi search algorithm is used to translate the sequence of phonemes to a sequence of words using a weighted finite state transducer (WFST) state machine [22]. This procedure is represented in Figure 2.1.

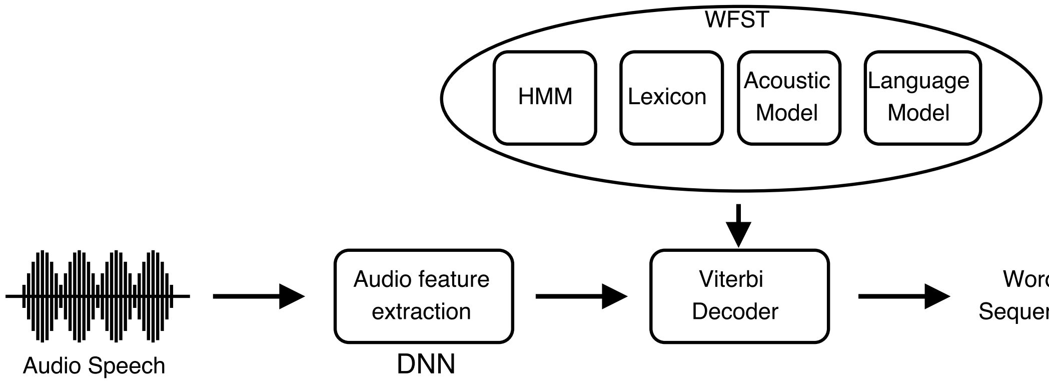


Figure 2.1: Speech recognition procedure

Deep Neural Network (DNN)

The DNN implements the acoustic model, it is the phase of the speech recognition which is in charge to translate the vector features of the audio signal to phonemes probabilities. In recent years there has been a lot of work on boosting the performance of DNNs by using GPUs [31, 9, 8] or dedicated accelerators [7, 11], achieving huge speedups and energy savings. However, it is just a small part of an ASR system where Viterbi is the dominant component.

Weighted Finite State Transducers (WFST)

Speech recognition initially used Hidden Markov models (HMM), which express the relationship between hidden and observable variables: hidden variables being sequences of words, and the observable ones being the acoustic features. Accounting for an increasing number of acoustic features increased the complexity of the models. A more recent approach to reduce the complexity of the HMM are the WFST state machines.

WFST is a Melay state machine widely used in speech recognition. This state machine allows to separately define different information sources represented as independent WFST and at the same time provides a single and combined WFST for the model. The final WFST, contains the entire speech process and is highly optimized, as the redundancy of the overall network is eliminated.

The representation of the acoustic model with a WFST state machine is made up of different information sources which at the same time are represented with a WFST. Usually, the model is made up of four WFST: an HMM (H) which encodes the pronunciation of each phone, a context dependency (C) which establishes the context rules between each phone, a lexicon (L) defines the relations between words and phones, and finally a grammar (G) for the language model. The final model is the composition of all of them (H o C o L o G). The modularity of the WFST permits higher adaptability to a different language or vocabulary complexity by modifying the required parameter models without requiring changes to the underlying decoder.

A WFST state machine has arcs with five attributes: weight, source and destination states, and input and output labels. In the context of speech recognition, the input labels represent the phonemes and the output labels the words. The WFST of an ASR is constructed offline and is used by the Viterbi search algorithm for the translations of phonemes to words.

Regarding the arcs of the WFST we can recognize two different types: epsilon arcs, which do not have an association with any phone (do not have any input label) and the non-epsilon which do. The reasoning behind this differentiation is to, model the connection between words thus allowing for the transversal of states without consuming a frame of speech. In Kaldi's WFST only 11.5% of the arcs are epsilon.

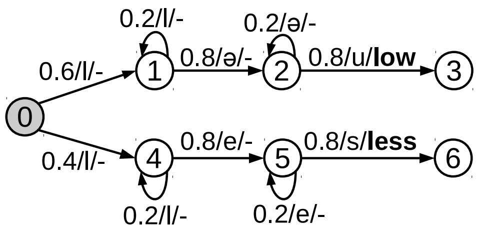
As the weights of the WFST represent probabilities or likelihoods between 0 and 1, using floating points is not reliable due to floating point underflow problem. To prevent this problem, probabilities are represented in log-space, a side effect of this is that probabilities multiplications become additions.

Even though the WFST model reduces the complexity and lowers the redundancy of previous approaches using HMM, the resulting size is significantly big. For example, the WFST of Kaldi contains more than 13 million states and more than 34 million arcs.

Viterbi search algorithm

The Viterbi search algorithm finds the most likely sequence of hidden variables resulting from a sequence of observed variables on an HMM. It is used in applications like speech recognition algorithms, language parsing and bio-informatics [6]. In the context of speech recognition the Viterbi search algorithm is responsible for the translation of a sequence of phonemes, obtained from the processing of the audio in the DNN phase, to a sequence of words. The Viterbi search algorithm is hard to parallelize [8, 15, 16] and, therefore, a software implementation cannot exploit all the parallel computing units of modern multi-core CPUs and many-core GPU

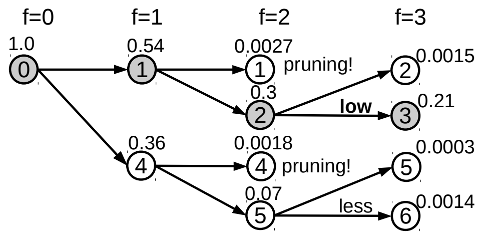
The Viterbi search algorithm uses the information provided by the WFST state machine to determine the most likely sequence of words according to it. For every frame of audio a given phoneme is detected. The Viterbi search algorithm uses the phoneme as an input label on the WFST state machine to expand its corresponding output labels and detect the most likely sequence of words.



(a) Example of a 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 scores



(c) Viterbi search path

Figure 2.2: Viterbi search example

Figure 2.2a contains an example of a small WFST for a vocabulary with two words, low and less. On Figure 2.2b we have the acoustic likelihoods of three frames of a given audio sample generated by the DNN. The first frame has 90% probability of being phoneme 'l'. Finally, the Figure 2.2c shows the Viterbi search exploration of the WFST of Figure 2.2a using the acoustic scores of Figure 2.2b. The search starts at state 0, the new states are created from the previous ones using the acoustic likelihoods corresponding to the states for the given frames. Once all the frames are processed, the active state with maximum likelihood in the last frames is selected and the best path is recovered using backtracking, which finally produces the sequence of words. For this example, the recovered sequence is the word 'low'.

There is potential ambiguity in the use of the term state, as it might refer to the static WFST states (see Figure 2.2a) or the dynamic Viterbi ones (see 2.2c). To clarify the terminology we use state to refer to the static state of the WFST, whereas we use token to refer to an active state created during the Viterbi search. A token is associated with a static WFST state, but it also includes the likelihood scores of the best path to reach the state and a pointer to the predecessor.

Due to size of real WFST, for instance the one we find on Kaldi, it is infeasible and impractical for the Viterbi search algorithm to expand all possible paths. For this reason, threshold values are taken into account during the arc expansion with the idea to prune the expansion of the less likely branches of Viterbi search, this can be seen in Figure 2.2c, in the expansion of frame 2. This improvement receives the name of Viterbi beam search [19].

2.2 GPU Architectures

For many years, the performance improvements for new processing units was mainly obtained by means of increasing the frequency speed at which they operated. Each new generation of processors provided better performance through increasing the frequency. However, frequency increase directly affects the power dissipation which, once reached a certain limit, forced the industry to look for other ways to improve performance.

The alternative has been the design of processors with multiple cores; to shift the focus from increasing the performance of a single core processor to increase the overall performance by adding multiple cores to the processor, even at the expense of them being individually less capable than their single-core counterparts. Multi-core processors came also with a shift from sequential programs to parallel computing programs.

GPU architectures started featuring programmable pipeline units, or shader units, around 2001. Albeit with much programming effort involved in using the APIs that enabled this programmability (OpenGL and DirectX), this allowed the use of GPU hardware resources dedicated to process graphic elements to be used for general purpose computations, which originated the term General-purpose GPU (GPGPU). Due to the nature of the processing of graphic elements, GPU allowed high arithmetic throughput, useful for applications like matrix multiplications.

From this starting point followed subsequent microarchitectural changes to the GPU architecture focused on general purpose computation: ALU floating point support, read and write memory access for the execution units and special shared memory. Aside from architectural improvements, the APIs used to program GPU were cumbersome requiring knowledge about graphic concepts, this lead to the development of new programming models like CUDA [10] or OpenCL [25] which streamlined the use of GPUs form general purpose computation.

GPU microachitectures feature many, simple cores which consists of no more than execution units rather than a single or few complex cores with many microarchitectural improvements (i.e. branch prediction, out-of-order execution) seen on CPU architectures. GPU architectures are throughput oriented rather than latency oriented: many slow operations computing simultaneously, as well as high memory throughput but high latencies. Whereas CPU architecture focus on low latency rather than high throughput: a few fast operations at a time, as well as low memory latencies due to highly tuned cache hierarchy. The GPU approach to computation is to exploit massive level parallelism to hide the slow execution units and high memory latency.

2.2.1 State-of-the-art GPU architecture

This section is going to cover the main components of modern GPU architectures using as a reference the NVIDIA GTX 980 GPU, which implements the NVIDIA Maxwell Architecture [24].

Figure 2.3 shows the hardware architecture overview of the NVIDIA GTX 980 GPU. Its main components are: 16 Streaming Multiprocessors (SM or SMM), running at 1.28 GHz and organized in 4 graphic processing clusters (GPC); 4 memory controllers and a shared L2 cache, size 2 MB, as a LLC.

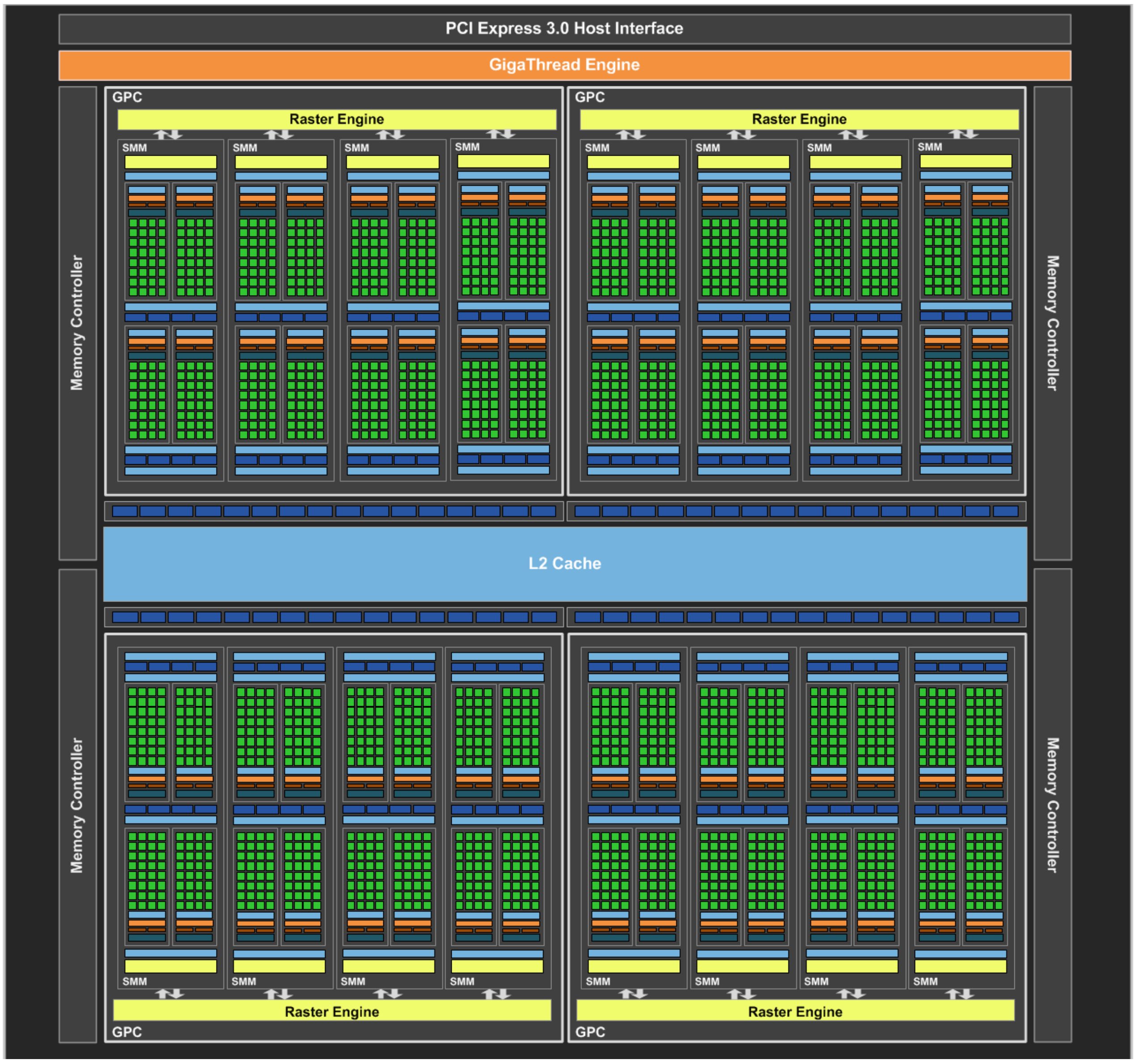


Figure 2.3: NVIDIA GTX 980 GPU, Maxwell Architecture [24].

Streaming multiprocessors

A Streaming Multiprocessor, depicted in the Figure 2.4, is composed of: 4 warp schedulers; a L1 instruction cache, of undocumented size; a L1 data cache, size 48 KB; and a shared memory of 96 KB.

Warp scheduler

Each warp scheduler unit has 2 warp dispatch units, which enables to dispatch 2 warps per cycle to the execution units, depicted as the Core, LS/ST and SFU boxes in Figure 2.4, each Core executes an instruction per cycle. Associated with the warp scheduler we have a register file which is used by the dispatched warps of that warp scheduler, 16K registers per warp scheduler, total of 65K per SM.

A warp is what NVIDIA calls a group of threads, 32 threads in total, executing an instruction on multiple data: single instruction multiple threads (SIMT). The execution of branch instructions in a warp is predicated, meaning all the instructions of the branch will be executed but the result discarded if a given thread did not chose a path of the branch. Due to this predication, high control flow divergence in a warp incurs in an important degradation of performance.

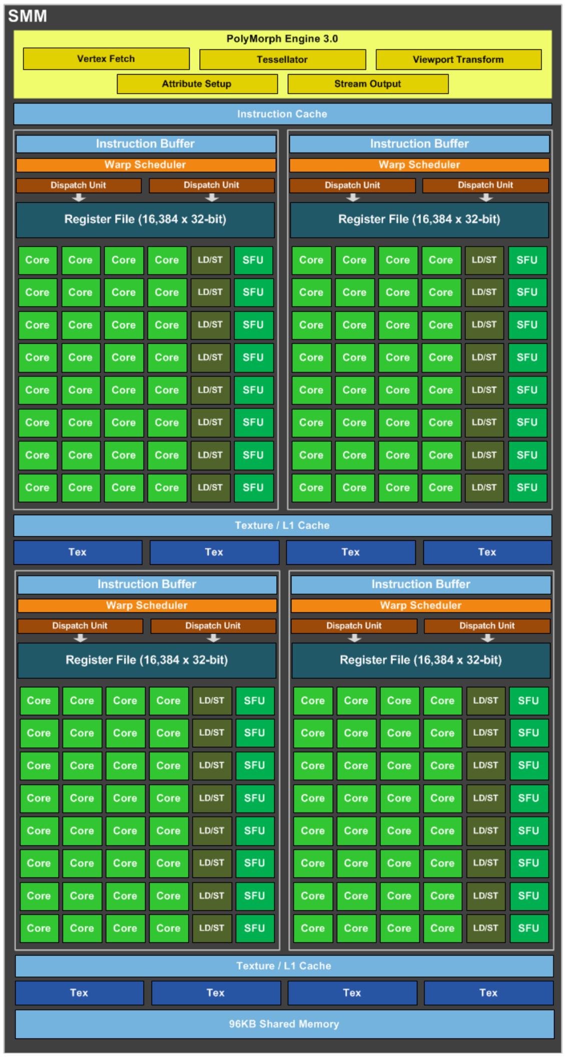


Figure 2.4: NVIDIA GTX 980 Streaming Multiprocessor [24].

Memory hierarchy

The cache hierarchy is smaller that the one we can find on a CPU architecture, furthermore, the cache size per thread is much less. Memory accesses for a given warp often access cache line which GPU architectures exploit by coalescing the memory access into one, thus improving memory bandwidth.

GPU architecture also counts with a number of other caches like: constant caches, for access optimization to parameters; texture cache, used for graphics; or even different memory space with the shared memory, a shared address space addressable by the warps in a given SM.

2.2.2 CUDA programming model

This section is going to use CUDA as a reference to cover programming model for GPUs, even thought it shares many characteristics with OpenCL.

The CUDA programming model is available for C, C++ and Fortran, it adds a number of features to the languages to be able to define tasks, or kernels, to execute on CUDA-enabled GPU devices. The initial version was released for the NVIDIA GPU microarchitecture Tesla, and many versions with added functionalities have come later on.

Kernels

A kernel is a GPU task programmed using the CUDA language extension. A kernel is composed of thread blocks (TB) or cooperative thread array (CTA), which is a set of threads that cooperate between themselves to execute the code of the kernel.

A kernel takes special parameters regarding the amount of threads to use, defining the number of thread blocks and thread block size. A side from these parameters it can take parameters like a normal function would do, although as the CPU and GPU memory space are independent, many parameters have to be moved to the GPU memory before execution of the kernel. The kernel execution can be launched from the CPU or from within the GPU itself by another kernel.

Threads and synchronization

Thread block have a number of threads designated at launch time, this threads share resources like the shared memory of a SM. Each individual thread has an index identifier of its own used to compute the offsets to access its own data.

The execution of a thread block is not guaranteed to be done simultaneously due to resource allocation limitations, to deal with situations when this can incur on synchronization problems, the programming languages provides synchronization barriers. The synchronization barriers only affect threads within a thread block, they do not synchronize different thread blocks of the same kernel instance.

Atomic instructions

The CUDA programming model supports a number of atomic instructions for global memory and shared memory, the data types supported are integer and float although not for all atomic operations. No mutex mechanism is provided by the programming model as, the execution of instruction of the warps in a predicated manner would not allow the usage of it within the threads of a given warp.

3 Evaluation methodology

3.1 NVIDIA CUDA toolkit

We use the NVIDIA CUDA toolkit version 7.5, which includes the nvcc compiler, that we use with -O3 optimization flag, and the NVIDIA Visual profiler, to measure the performance of our GPU Viterbi decoder described in Chapter 4 on a machine with a CPU with the parameters shown in Table 3.1a and a GPU with the parameters shown in Table 5.1b.

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

(a) CPU 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

(b) GPU parameters

Figure 3.1: Hardware configuration of our evaluation platform.

To be able to use the GPGPU-Sim simulator we also use the NVIDIA CUDA toolkit version 4.0 as it is the latest toolkit supported. We use it with the included nvcc compiler, with -O3 optimization flag, to compiler our GPU Viterbi decoder to run be able to run with the simulator.

3.2 Architectural Simulation

3.2.1 GPGPU-Sim

We use the GPGPU-Sim simulator [4] version 3.2.2 for the architectural performance characterization of our GPU Viterbi decoder with the hardware configuration described in Section 5.1. GPGPU-Sim is a cycle-level GPU performance simulator which models microarchitectures similar to contemporary GPU microarchitectures like NVIDIA's Fermi GPU microarchitecture.

3.2.2 GPUWattch

For the architectural power and energy characterization we used GPUWattch [13] which is an energy model based upon McPAT [17] integrated with GPGPU-Sim. GPUWattch accounts for the GPU specific architectural components by matching the GPGPU-Sim configuration and provides static and dynamic power estimates using performance counters provided by GPGPU-Sim.

3.2.3 GPUWattch power model problems

The power model implemented by GPUWattch showed high inaccuracy with some specific parameters when we used our own microarchitecture configuration which is described in Section 5.1. The inaccuracies affected both power and area estimations and the discrepancies with what would have been expectable and reasonable estimations were big enough that we had to modify the model as to be able to have acceptable estimations.

Area

Area estimations were off by a significant factor when increasing the floating point units (FPU) of the cores from 32 to 128, change proposed in Section 5.1. The area devoted to this component is summarized in Table 3.2, the GPUWattch row shows that the area devoted to the FPU represents a 80.61% of the total area of the GPU, an area disproportionately big.

	Area FPU	Total area FPU	Total area GPU	Percentage
GPUWattch	0.450896(*)	923.43	1145.44	80.61%
Synthesis tools	0.010566	21.50	243.50	8.83%

Figure 3.2: GPUWattch FPU Area model estimations (in mm^2).

We modified the model area parameters of the FPU with parameters for an FMA32 unit obtained with synthesis tools [5]. The area for a single FPU for a 28 nm technology was estimated by the model to be 0.90 mm^2, although for the computations a factor of 0.5 is applied (*), versus 0.010566 mm^2 estimated by the synthesis tools calculations, a discrepancy of 85x. In Table 3.2, the row Synthesis tools shows a much more seemingly reasonable numbers where the FPU represents the 8.83% of the total area of the GPU and a total area much more acceptable.

Static power

Static power estimations for the shared memory were way beyond a reasonable value as well as its percentage split of the total of the static power. The values reported by the model showed that the shared memory consumed 676.91 W of static power which represented a 96% of the total static power for the GPU. In this case we ignored the reported static power metrics for the memory as a correct value would have been very insignificant and either way the shared memory is not a target of the architectural evaluations.

3.3 Audio speech datasets

Regarding the datasets, we use the WFST from English language provided by the Kaldi toolset, which is created from a vocabulary of 125.000 words, and audio files from Librispeech corpus [18].

4
GPU Viterbi decoder

In this chapter we present our Viterbi search algorithm implementation developed for GPU architectures based on the Kaldi CPU Viterbi implementation and a GPU Viterbi implementation presented in [31]. This section is structured as follows. First, we review the Kaldi CPU Viterbi implementation, and its main components. Afterwards, we present the GPU Viterbi implementation with special focus on singular implementation characteristics. Finally, we evaluate our implementation.

4.1 Kaldi CPU Viterbi Algorithm

The Kaldi ASR suite provides an implementation of the Viterbi search algorithm for CPU which implements the search phase of the speech recognition process. Their implementation is divided in three main phases or components: Processing of epsilon arcs, Processing of non-epsilon arcs and Backtracking.

The algorithm implementation uses as input the following data:

- **Acoustic scores:** Contains the output scores of the audio processing done by the DNN for each frame of the input speech.
- **Words dictionary:** Contains the dictionary of words
- **WFST:** Represents the WFST state machine and contains the information about the states, and the arcs for each of them. The arcs contain the information about the input and output labels and the weight.

Token information track

To be able to retrieve the output sequence of words, the algorithm has to keep information about the visited tokens. For this purpose, a structure named **token buffer** holds information of all the visited states relative to the traversal cost, the output label of the state (which represents the word) and which was the best predecessor token (with best traversal cost).

To manage the expansion of the arcs into tokens in a given frame, the algorithm uses a hash table of tokens to easily access the active ones. With this structure it avoids checking all the states of the WFST at every frame.

Algorithm procedure

The algorithm starts in a default state or token and starts the processing the scores for each of the frames present on the acoustic scores. For each frame, first the algorithm processes the epsilon arcs and afterwards the non-epsilon. The processing consists on expanding the arcs of the active tokens generated on the previous frame and then updating or adding the new states visited to the active tokens structure for the following as well as with the tracking of the search information. The expansion is only done if the cost is below a pruning threshold value. When the algorithm has processed all the frames, the backtracking phase takes place traversing the search backwards, starting on the token with the best score and outputs the sequence of words recognized from the given acoustic scores.

4.1.1 Processing Epsilon Arcs

The epsilon arc expansion phase expands only the epsilon arcs of the active tokens visited on the non-epsilon phase. The expansion is pruned with a cutoff value of the lower (best) cost determined during the expansion of the non-epsilon arcs. The expansion of the arc adds to the cost of the token inserted, if it is not already an active token with lower, thus better cost. The expansion adds to the current accumulated cost the cost of traversing the arc in the WFST. No acoustic cost is accounted for.

Optionally, the expansion of the epsilon arcs can be done recursively with the use of a stack, which successively expands the epsilon arcs of the successor active tokens until there are no more epsilon arcs to expand.

4.1.2 Processing Non-Epsilon Arcs

The non-epsilon arc expansion phase separates the active tokens from one frame to the next one. This phase first determines the pruning cutoff based on the cost of the active token of the current frame, and computes a first approximation of the cutoff for the next frame checking the cost of the non-epsilon arc expansions of the token for which the current pruning cutoff is based on. After determining the current and next cutoffs, the expansion of the arcs begins. Using the current cutoff to prune the expansion of active tokens and the next cutoff to prune the expansion of the non-epsilon arcs to new tokens. Like in the expansion of epsilon arcs, the expansion adds to the current accumulated cost of the token the cost of traversing the arc as well as the acoustic score for the given state.

4.1.3 Backtracking

The Backtracking phase if run after the processing of the expansion of the arcs of the last frame. The algorithm selects the best token, the one with better cost, among the active ones. From this token the search graph is traversed from the last token to the first one using the information stored in the token buffer and retrieving the outputs labels which correspond to the sequence of words. The sequence of words obtained is the one that the Viterbi search algorithm has deemed to be most likely sequence for the provided audio speech.

4.2 GPU Viterbi implementation details

Our implementation is based on the state-of-the-art GPU version described in [9], including the optimizations for improving GPU performance presented in [8].

The algorithm implementation uses the same input data as the CPU Kaldi version, but to be able to better use the hardware resources of the GPU architectures the structures are organized as structs of arrays (SoA) instead of arrays of structs (AoS). The objective of this change is to improve the memory bandwidth of consecutive threads when accessing consecutive memory positions, since the hardware resources will coalesce the memory accesses.

The GPU is the main component used throughout the whole execution of the Viterbi search. The CPU is only used for setting up kernel launch parameters, the final stage of a minimum reduction and the final stage of the reconstruction of sequence of words in the backtracking phase. Combined with the fact that all data structures used are kept on the GPU memory we avoid unnecessary memory transfers between the CPU and GPU memory

Intermediate data structures

The adaptation of the algorithm to the GPU requires changes to the internal structures and new additional ones:

- **Active Tokens:** Structure which holds the information of the active tokens in order to be able to retrieve the output sequence of words. The information covers the accumulated traversal cost, the best previous token and the output label of the token (which represents the word)
- **Tokens Table:** Structure that holds only the cost information for all the states of the WFST machine. This structure is needed due to synchronization problems.

- **Arcs Buffer:** Structure required to hold the frame information of the token expansion into arcs due to the breakdown of the algorithm into multiple kernels.

Algorithm breakdown into CUDA kernels

Following the CUDA programming model we breakdown the Viterbi search algorithm into multiple kernels which try to parallelism different parts of it in an efficient manner. The threads of the different kernel take the role of a token or of an arc depending on the task of the kernel and the data structures that it accesses with the objective to extract parallelism and do regular memory accesses.

The CUDA kernels implemented are described below:

- **Find minimum:** This kernel finds the token with the best cost. It computes the min reduction of the cost of the active tokens as well as providing the identifier for it. Each thread operates on a token. The reduction is partial due to synchronizations limitations between thread blocks, explained in Section 2.2.2. The final iteration of the reduction is computed in the CPU.
- **Compute cutoff:** This kernel computes the current cutoff and the next cutoff. Given the best token for a frame found in the **Find minimum** kernel, it computes the cost of the expansion of the arcs and selects the best one as the next cutoff. This kernel is only run by one thread, the alternative would mean memory transfers to be able to compute it in the CPU.
- **Compact Arcs:** This kernel does the memory space assignation for the expansion of the tokens into the arcs buffer. Each thread operates on a token and do the reserve of the arcs space atomically modifying the index size of the arcs buffer, the reserve is not done if the token cost is worse than the cutoff. No memory allocations are done, the allocations are done previously on the setup.
- **Fill Arcs:** This kernel fills the values of the arcs reserved on the kernel **Compact Arcs**. Each thread operates on an arc and accesses the information of the arcs on the WFST as well as the token accumulated traversal cost.
- **Expand Arcs:** This kernel computes the traversal cost of the arcs for the given frame and updates part of the information for the new active tokens. Each thread operates on an arc, the expansion is not done if the cost is worse than the next cutoff. Multiple arcs can lead to the same token with different traversal costs for this reason this kernel only updates the traversal cost with an atomicMin and reserves the space on the active token modifying atomically the index.
- **Expand Token Data:** This kernel updates the remaining part of the information of the active tokens. Each thread operates on an arc, only the arcs which lead to the same token and have matching traversing cost will compete to update the remaining information of the token with an atomic operation. This is due to not being able to use mutex synchronizations mechanisms, as explained in Section 2.2.2.

- **Reset Token Table:** This kernel resets the token table information. Each thread operates on an token, only the active tokens are cleared.

- **Backtracking:** This kernel computes the backtracking sequence of words identifiers. This kernel is only run by one thread, the alternative would mean memory transfers to be able to compute it in the CPU.

Implementation obstacles

Processing the Viterbi search algorithm in a GPU means processing the expansion throughout the WFST state machine, traversing tokens and arcs in a parallel way. Many of the arcs expansions for a given frame will lead to update the information of a given token multiple times. The information to update consist on the accumulated cost, identifier of the best token of the last frame and word identifier of the token. If the cost is the minimum, the other two need to be updated. This could be implemented with a mutex region over the destination token but this option is not available due to how the synchronization works on CUDA architectures, as explained in Section 2.2.2.

To overcome this synchronization problem, we split the process in two steps: In a first step we update atomically the token cost, establishing the lowest one (lower is better), and in a second step we update the remaining information if the cost matches with the lower established in the previous step. This approach still presents two obstacles to overcome. One being that the cost is represented as a float and CUDA does not support atomicMin over this type of data. The other being that still for the remaining information be require an atomic way to write it, and mutex are not an option.

To solve the first problem we use apply a series of modifications to the float data type to convert the data to a sortable integer [30] which transform the float in a way that allows us to compare the values as we would for an integer and get a correct comparison.This sortable integer allows us to use the atomicMin, later we can convert back the sortable integer to the original float.

The second problem is addressed using the atomic instruction AtomicExch which allows us to atomically modify 64 bits of memory.

4.2.1 Processing Epsilon Arcs

The epsilon arc expansion phase for the GPU is made up of the following kernels: First the expansion of the tokens to arcs is done by the kernels **Compact Arcs** and **Fill Arcs**, after the arcs are expanded to visit the new tokens with the kernels **Expand Arcs** and **Expand Token Data**, finally we need to reset the token table since we change frame and the active tokens are the new ones this is done by the kernel **Reset Token Table**. Some kernels are executed in both epsilon and non-epsilon phases, but the configuration selects the appropriate arcs to expand in this case epsilon arcs.

4.2.2 Processing Non-Epsilon Arcs

The non-epsilon arc expansion phase for the GPU is made up of the following kernels: First the selection of the best token among the active ones is computed by the kernel **Find minimum** as well as the CPU, after the cutoff values are computed by the kernel **Compute cutoff**. Finally, the expansion of the tokens to arcs is done by the kernels **Compact Arcs** and **Fill Arcs**, and the expansion of the arcs to new active tokens by the kernels **Expand Arcs** and **Expand Token Data**, same process as with the epsilon arcs phase, but configured to expand non-epsilon arcs.

4.2.3 Backtracking

The backtracking phase for the GPU is made up of the following kernels: First the selection of the best token among the active ones is computed by the kernel **Find minimum** as well as the CPU, and then the backtracking of the best likely path is done by the **Backtracking** kernel. Finally the CPU puts together the sequence of words to output.

4.3 Experimental evaluation

The machine where the experimental evaluation was done is described in Chapter 3, Figure 3.1. A series of audio speech samples, from the files described in Section 3.3, with length of 2 to 10 seconds were tested on both implementations of the Viterbi search algorithm, the Kaldi CPU Viterbi implementation and our GPU Viterbi implementation.

Performance comparison

The performance numbers obtained comparing the GPU version with the CPU version achieves an speedup between 9.93x to 17.9x in favour of the GPU version, as seen in Figure 4.1. The speedup, albeit low when taking into consideration the parallelism capabilities of a GPU in comparison with a CPU, it is similar to the one of other Viterbi GPU implementations where they obtain a speedup of 3.74x [31], although the hardware platforms are different. The decoder implementation is in real-time, meaning that the decoding time for the audio speech files is below the audio time of the files.

	Audio length (s)	GPU (s)	CPU (s)	GPU Speedup
Audio #1	3.98	0.08	1.14	14.25x
Audio #2	10.42	0.17	1.72	9.93x
Audio #3	3.26	0.07	1.04	14.76x
Audio #4	6.61	0.11	1.28	11.15x
Audio #5	2.66	0.06	1.08	17.90x

Figure 4.1: Comparison of execution time of Viterbi decoding algorithm between GPU and CPU.

The more time consuming kernel is the **Fill Arcs** taking a total of 29.3% of the execution, followed by kernel **Compact Arcs** at 22.4%. Figure 4.2 includes the percentage for all the kernels of the GPU Viterbi decoder.

Kernel	Percentage
Fill Arcs	29.3%
Compact Arcs	22.4%
Expand Arcs	18.5%
Expand Token Data	12.7%
Compute Cutoff	5.2%
Find Minimum	3.6%
Reset Token Table	3.9%
Backtracking	0.5%

Figure 4.2: Percentage of execution time of the GPU Viterbi decoder per kernel.

Regarding power dissipation, the profiling tools indicate a total consumption of 70.9 W.

Our GPU Viterbi search algorithm implementation is functional and performs similar, even better than other implementations for GPU, meeting real-time requirements for speech recognition systems. This implementation is our baseline Viterbi decoder used for the gpu architecture characterization detailed on the following chapters; Chapter 5 and Chapter 6.

5 Architectural characterization

In this chapter we describe the baseline architecture configuration used with GPGPU-Sim to characterize our speech recognition GPU Viterbi decoder presented in Chapter 4. This chapter is organized as follows: First we present the baseline architecture configuration which is used to obtain the metrics in the rest of the chapter which serves as a baseline configuration to the changes applied to the architecture on the following Chapter 6. Afterwards, we show and study the performance metrics, power and area measurements of the execution of our GPU Viterbi implementation on the described architecture for the GPGPU-Sim.

5.1 Baseline architecture

Having in mind the objective to evaluate the hardware requirements on GPU architectures of the GPU Viterbi Decoder we use the simulation tools described in Section 3.2. For this matter, we require a GPU hardware architecture configuration, GPGPU-Sim provides a number of configurations, the more recent being the one emulating the NVIDIA GTX 480. Since our objective was to evaluate state-of-the-art GPU hardware, we put together a configuration which resembles the high-end GPU hardware NVIDIA GTX 980.

The more relevant hardware configuration parameters are listed in Figure 5.1. The simulated GPU architecture is made up of 16 Streaming multiprocessor (SM), or cores, with a technology of 28 nm running at 1.28 GHz. Each SM can run up to 2048 concurrent threads, and can execute up to 4 warps simultaneously, each warp executing an instruction on up to 32 threads. In consonance to the number of warp schedulers and warp size, each SM has 128 Execution Units (EU) and 128 Special Function Units (SFU), and a single Load/Store issue (LD/ST). Finally, each SM has a total of 65536 shader registers which amount to a total of 32 registers per maximum concurrent threads and a shared memory of 96 KB.

Streaming Multiprocessors	16
Technology	28 nm
Frequency	1.28 GHz
Concurrent Threads/SM	2048
Warp size	32 Threads
Warp schedulers/SM	4
EU, SFU	128, 128 (per core)
LD/ST	1 (per core)
Shader Registers	65536
Shared memory	96 KB

(a) General parameters

Memory Channels	4
L2 MC subpartitions	2
L1 inst cache	2 KB (4 sets, 128 line size, 4 assoc)
L1 data cache	32 KB (64 sets, 128 line size, 4 assoc)
L2 cache subpartition	256 KB (256 sets, 128 line size, 8 assoc)
L2 cache total	2 MB (256 KB * 2 L2 MC subpartitions * 4 MC)

(b) Memory hierarchy parameters

Figure 5.1: GPGPU-Sim GTX 980-like configuration parameters.

Memory hierarchy information is much harder to come up with since there is no public information, to our knowledge, of the specific configuration of the different cache levels. Some of the configurations are based on the experimental findings by others [20], but the tools we used imposed restrictions on how the caches could be configured, so we settled for close but reasonable configurations.

The simulated GPU architecture features 4 memory channels (MC) each with 2 L2 subpartitions, by GPGPU-Sim memory hierarchy design. The instructions L1 cache was unmodified from the NVIDIA GTX 480, as we deem it irrelevant. The data L1 cache is 32 KB in size, 64 sets, 4 associative, 32 mshr entries. An L2 subpartition is 256 KB in size, 256 sets, 4 associative, 32 mshr entries. The whole L2 cache is made up of 8 L2 subpartitions, 2 in each MC.

5.2 Performance characterization

The characterization of the baseline architecture is done with the tools introduced in Section 3.2, and audio samples from the files described in Section 3.3.

Execution time

The execution time to decode an audio file of 200 ms is of 2.79 ms, in the same order of magnitude than what we obtain, 3.5 ms decoding time, for the same file on our evaluation platform described in Section 3.1.

Regarding the split of the time execution among the kernels, it is similar but different than the one we saw on Section 4.3 for the evaluation done on an NVIDIA GTX 980. The most significant is the kernel **Expand Arcs** with 25.92% and the second is **Compact Arcs** with 20.62%. Figure 5.2 includes the complete list. This differences are due to the architectural differences between our evaluation GPU platform and the GPGPU-Sim simulator.

Kernel	Percentage
Expand Arcs	25.92%
Compact Arcs	20.62%
Reset Token Table	18.27%
Expand Token Data	14.95%
Fill Arcs	12.50%
Compute Cutoff	5.43%
Find Minimum	2.11%
Backtracking	0.21%

Figure 5.2: Percentage of execution time of the GPU Viterbi decoder per kernel on the baseline architecture for GPGPU-Sim.

Our baseline architecture can achieve maximum ipc of 2048, this is 32 instructions per warp, 4 warps per core, 16 cores in total. The average ipc obtained with our GPU Viterbi decoder is of 84.6, a shy 4% of ipc efficiency. On the next section we explore the cause of this.

Issue rate

The maximum issue rate obtainable by or baseline architecture is of 64 warps per cycle, this is 4 warp schedulers per core, 16 cores in total. Our GPU Viterbi decoder has an issue rate of 3.26, which is an issue slot utilization of 5.1%. As with the ipc, it is very low, to see why we analysed the distribution of issue cycles.

Every cycle the GPU has 64 issue slots potentially available, we could be issuing 64 warps each cycle. But a issue slot can be unavailable due to many reasons: **Idle**, which is when the issue slot is free but does not have any warp to issue due to stalls in the Front-End of the processor; **RAW**, which happens when there are warp available but have dependencies with other uncompleted ones; **Stall**, due to unavailable Executing Units (EU); and **SM Off**, when the GPU is not using a particular SM due to not having a kernel to execute. Moreover, when the issue slot is available and has an available warp to issue, it can happen that this warp is fully utilized, executing 32 instructions, or not when executing less then 32, which affects the ipc negatively.

In the Figure 5.3 we can see the distribution of issue slots for our Viterbi decoder. As we already saw the issue slot utilization is of 5.1%, the majority 4.1% being warp with 32 threads, and the 1% left being warps with less than 31 threads. The most significant cause of issue slot under utilization is the Stalls at 45.99%; in our case mainly due to stalls in the LD/ST unit. The idle Front-End also is a significant percentage of the total 20.53%, as well as the SM Off at 17.63% which indicates an important under utilization of significant hardware resources. Finally, RAW dependencies between warp instructions are the less important factor to the issue slot utilization.

Idle	RAW	Stall	SM Off	W1-W31	W32
20.53%	10.76%	45.99%	17.63%	0.99%	4.10%

Figure 5.3: Issue distribution of GPU Viterbi decoder on baseline architecture.

The LD/ST unit stalls on GPGPU-Sim can be due to two factors: Reservation stalls, due to not enough miss status holding registers (mshr) to hide the latency of the memory accesses; and Replay of LD/ST, due to the re-execution of the instructions to do all the memory accesses of a warp when this has low memory coalescing. On our application the main factor is the reservation stalls by two thirds and the replay of LD/ST by one third.

Caches behaviour

Regarding the behaviour of the caches Figure 5.4 summarizes the main metrics. The L1 instructions cache miss ratio is very insignificant, we have to consider than in a GPU environment we are executing more instructions per memory access. The L1 data miss ratio is considerably high, this is due to the low locality that the Viterbi decoder experiences when traversing the states and arcs of the WFST state machine. Finally, the L2 miss ratio is lower but quite significant.

Cache	Accesses	Misses	Miss ratio
L1 Inst	6,300,305	104,798	2%
L1 Data	4,018,787	2,486,302	62%
L2	2,964,068	818,144	28%

Figure 5.4: Caches behaviour of GPU Viterbi decoder on baseline architecture.

Memory coalescing

As seen previously in the issue slot distribution, LD/ST stalls is the biggest reason of non utilization of the issue slots. The memory replays are a big factor of this under utilization, for this reason it is important to measure the memory coalescing of the application. The maximum achievable is a memory coalescing of memory accesses the size of the warp size, 32 in our baseline architecture. The higher the memory coalescing, higher use of a memory access is done per warp, better bandwidth efficiency. If the memory coalescing is low and a warp generates many memory accesses, the given instruction will use more bandwidth.

Our GPU Viterbi decoder on the baseline architecture achieves an overall memory coalescing of 7.5, lower than what the architecture allows. Several kernels achieve lower numbers like the

Compact Arcs, at 2.8 or Expand Arcs, at 5.03, which non surprisingly are the most time consuming ones. This low coalescing is to be expected due to the traversal of the states and arcs of the WFST state machine that does not allow for consecutive accesses to memory.

5.3 Power and Energy consumption

The power consumption of our GPU Viterbi implementation on our baseline architecture reaches a total of 92.89 W, of it 66.77 W account for dynamic power and 26.12 W for static power. The power estimation is similar to the one obtained using the NVIDIA GTX 980, which gave an estimation of 70.9 W, the difference is signification but it is to be expected due to the complexity of this estimations. For the speech audio file tested for this measurements, the energy consumption is at 0.25 J, which is used as a reference in the rest of the section.

Energy distribution per component

Figure 5.5 provides a breakdown on the distribution of the total energy consumption, static and dynamic, by component. The most significant being the Core at 38.34% followed by the dram memory at 24.82%.

Component	Percentage
Cores	38.34%
L1	6.03%
L2	7.97%
MC	2.21%
NOC	3.78%
Dram	24.82%
Other	16.85%

Figure 5.5: Energy component distribution of GPU Viterbi decoder on baseline architecture.

Energy distribution per kernel

Figure 5.6 provides a breakdown on the distribution of the total energy consumption, static and dynamic, by kernel. The energy consumption is well distributed among the more time consuming kernels. It is important to note that the kernel Reset Token Table includes the first reset, where the whole token table is reset. This is the reason of the high percentage on the energy consumption split, but the following iterations do not have the same energy consumption.

Kernel	Percentage
Find Minimum	1.75%
Compute Cutoffs	3.52%
Compact Arcs	17.47%
Fill Arcs	15.81%
Expand Arcs	18.97%
Expand Token Data	10.54%
Reset Token Table	31.76%
Backtracking	0.14%

Figure 5.6: Energy kernel distribution of GPU Viterbi decoder on baseline architecture.

5.4 Area study

Regarding the area estimation for our baseline architecture, the GPU is measured at a total of 243.64 mm^2. The area is lower than similar commercial counterparts, but the model does not provide estimation for graphics specific hardware which can explain the differences.

Component	Percentage
Cores	88.50%
L2	6.16%
NoC	0.01%
MC	5.33%

Figure 5.7: GPU area split on baseline architecture.

Figure 5.7 provides a breakdown of the GPU area. Almost the entirety of the area is devoted to the Cores. The biggest components of the core are the Load Store Unit (shared memory, data cache...) at 41.5% and the Execution Unit (Register file, ALUs, FPUs...) at 43.5% of the core area.

6 Design space exploration

In this chapter we present the experimental results of our design space exploration of several configurations parameters on our baseline architecture configuration for the GPGPU-Sim presented in Section 5.1 and we evaluate its impact on our speech recognition GPU Viterbi decoder presented in Chapter 4. This chapter is organized as follows: First we explore the memory hierarchy configuration, followed by the impact of warp concurrent numbers and number of schedulers and finally the effects of frequency scaling of the GPU, for each section we analyse a number of performance, energy and area metrics relevant to it. Finally, we propose a modified version of our baseline architecture which deals with our previous findings.

6.1 Memory Hierarchy

As we have already covered in Chapter 5, the memory hierarchy is the biggest limiting factor in the utilization of the issue slot which at the same time underutilizes the architectural resources of our GPU. At the same time the cache miss ratio is very significant, which is relevant for a memory intensive application like the Viterbi decoder. With this issues in mind we tested and evaluated the viability of several architectural modifications to our baseline architecture to try to mitigate them.

The modifications include:

- Increase the size of data L1 and L2, independently and together, as well as their number of mshr to evaluate the impact on the miss ratio of L1 and the capabilities of the mshr to hide the memory stalls due to reservation fails. The changes test L1 sizes from 32 KB to 2 MB, and L2 subpartition sizes from 256 KB to 16 MB, mshr increased accordingly.
- Increase the number of memory controllers, while applying the previous modification, to

provide higher bandwidth and see its effect on the performance of the GPU. The changes test 4, 8, 16 and 32 memory controllers with 2 L2 subpartitions each. The L2 subpartitions size is decreased accordingly to the increase of memory controllers.

6.1.1 Performance impact

Performance is directly affected by the increase of the L1 size, reaching a speedup of 1.18x on the execution time with maximum size, this is due to the reduction on L1 misses. On the contrary, when increasing the L2 size we do not observe such improvement on the performance nor any degradation impact. Figure 6.1 shows the increase on performance, each bar increases size of L1 and L2 together. Furthermore, increasing the number of MC also has an additive benefit: reaching a speedup of 1.52x with 8 MC, 1.76x with 16 MC and up to 1.84x with 32 MC, all speedups with maximum cache sizes.

This results indicates that the L2 is not constraining and that the bottlenecks regarding the memory hierarchy are located at the L1, where the increase in size reduces capacity misses, and at memory controllers where more parallelism in processing the memory petitions benefit the performance.

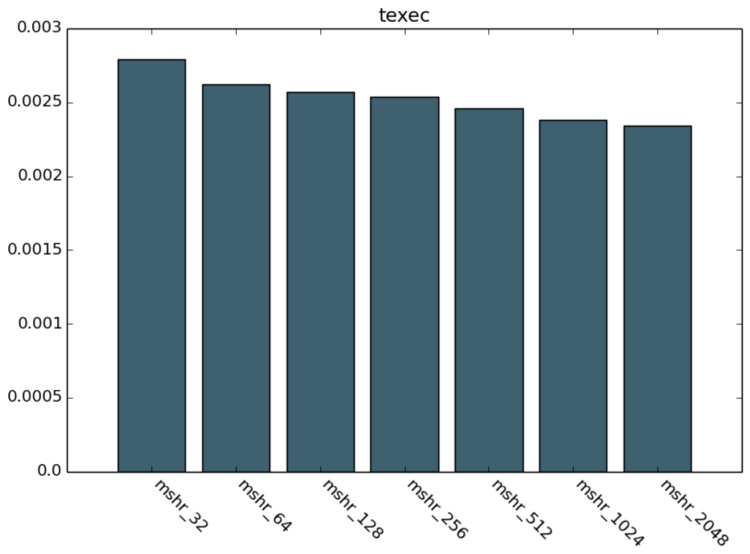


Figure 6.1: Execution time variation when increasing both L1 and L2 cache sizes (seconds).

The IPC goes in hand with the increase on performance, as the number of instructions stays the same. The baseline average is at 84.6 ipc which increases to 100.73 ipc when increasing both L1 and L2 and reaching up to 155.93 ipc when factoring modifying the number of MC.

Regarding the issue rate, we observe the same increase. From a baseline issue rate of 3.26 to 6.01 when applying all the modifications. The utilization increases but still is it only at 9.4%. Figure 6.2 shows the change in issue slots distribution when increasing L1 and L2 sizes. As we can see, the increase on the issue slot utilization is not due to an increase in useful cycles, represented by

W1-W31 and W32 bars, but due to a reduction of useless cycles, mainly the cycles wasted waiting for memory.

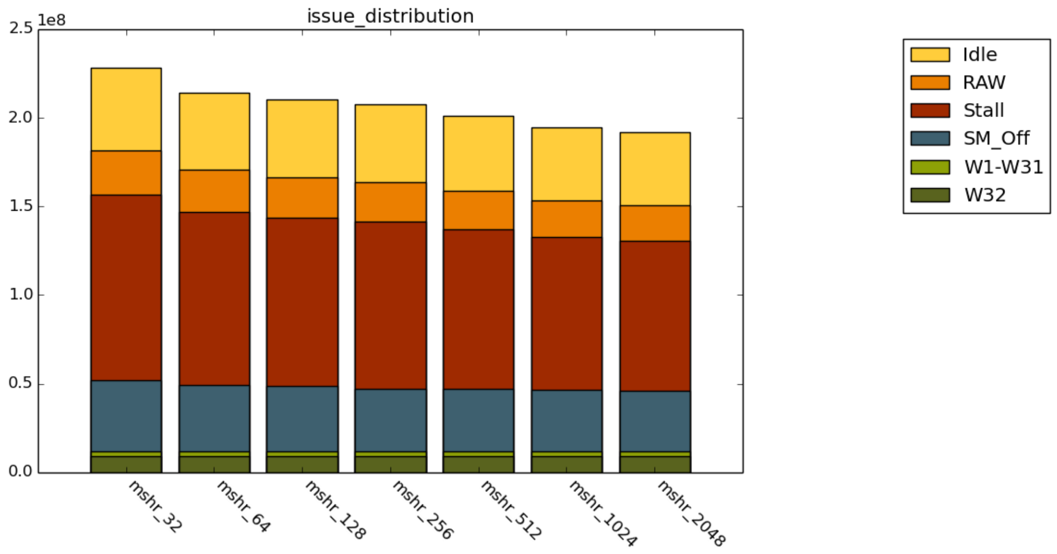


Figure 6.2: Issue slot distribution when increasing both L1 and L2 cache sizes.

Figure 6.3 shows the reduction of L1 cache miss rate from 0.62 to 0.48 due to the L1 size increase which has a direct effect on the performance. This reduction removes L2 accesses. It is interesting to note that when increasing the size of the L2 alone, the number of misses reduces to a size in which the we eliminate all the capacity misses, which can be seen in Figure 6.4. As stated previously, unlike with the L1, this reduction does not provide performance gains.

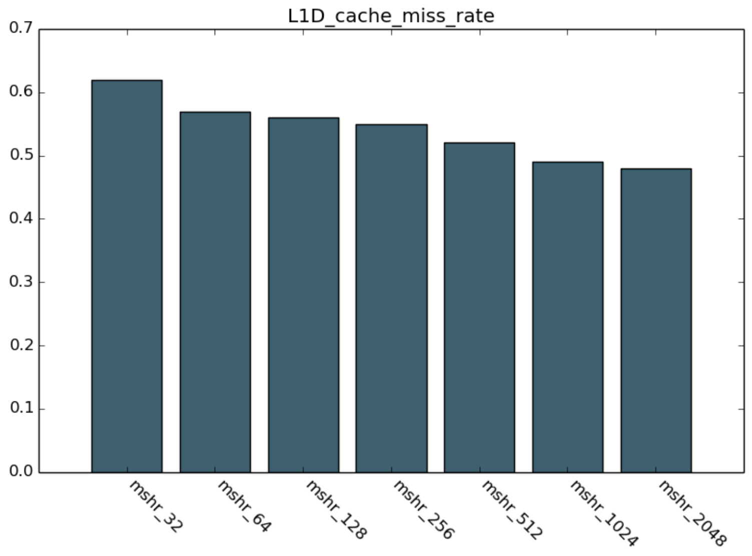


Figure 6.3: L1 data cache miss rate when increasing its size.

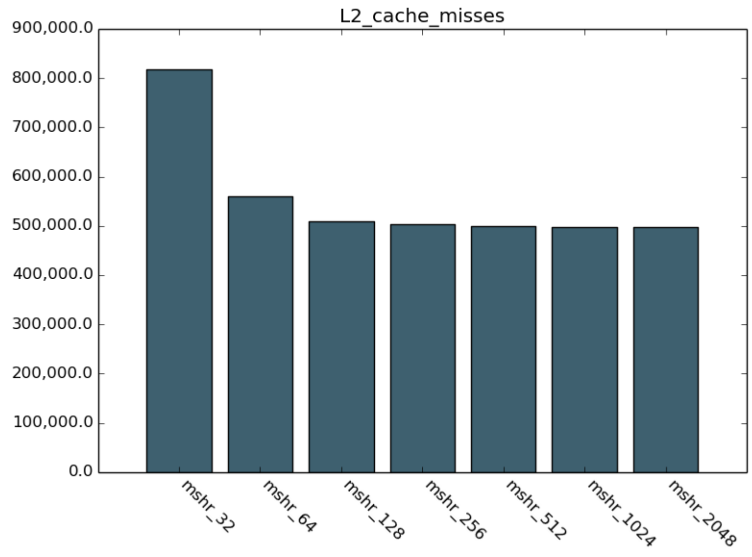


Figure 6.4: L2 data cache misses when increasing its size.

6.1.2 Power and energy impact

Power and energy metrics increase exponentially when increasing the cache sizes, especially after quadrupling the size, which make this sizes configurations infeasible. Power increases from 92.89 W to 689.55 W, when increasing L1 and L2 sizes and reaching up to 937.80 W with 32 MC, even thought the addition of MC reduces the energy consumption by providing performance benefit. The power increase is due to dynamic power, being the main factor the L1, followed by the L2. Figure 6.5 shows the exponential increase in power and its split between static and dynamic power when increasing both L1 and L2 sizes.

It is especially interesting that we find a reduction in power, from 92.89 W to 86.79 W, when increasing by a factor of two the size of the L2 alone, the reduction is also seen in energy consumption from 0.25 J to 0.24 J. Figure 6.6 shows the distribution between components of the energy consumed, the reduction is due to an initial reduction on the consumption of the dram due to the drop in L2 misses seen in Figure 6.4, after this point the L2 consumption increases and hides the reduction of energy consumption by the dram.

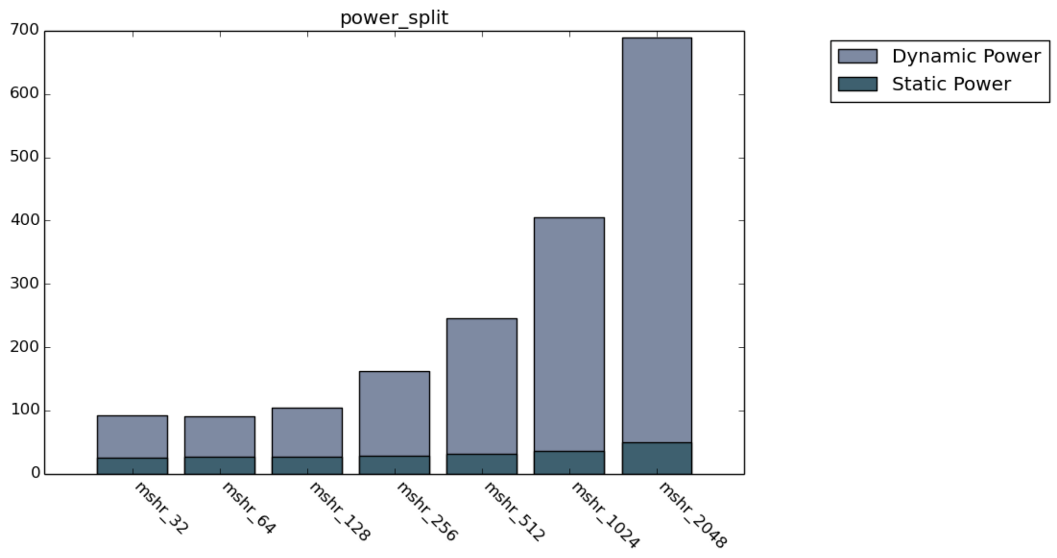


Figure 6.5: Power split, dynamic and static, when increasing both L1 and L2 cache sizes (Watts).

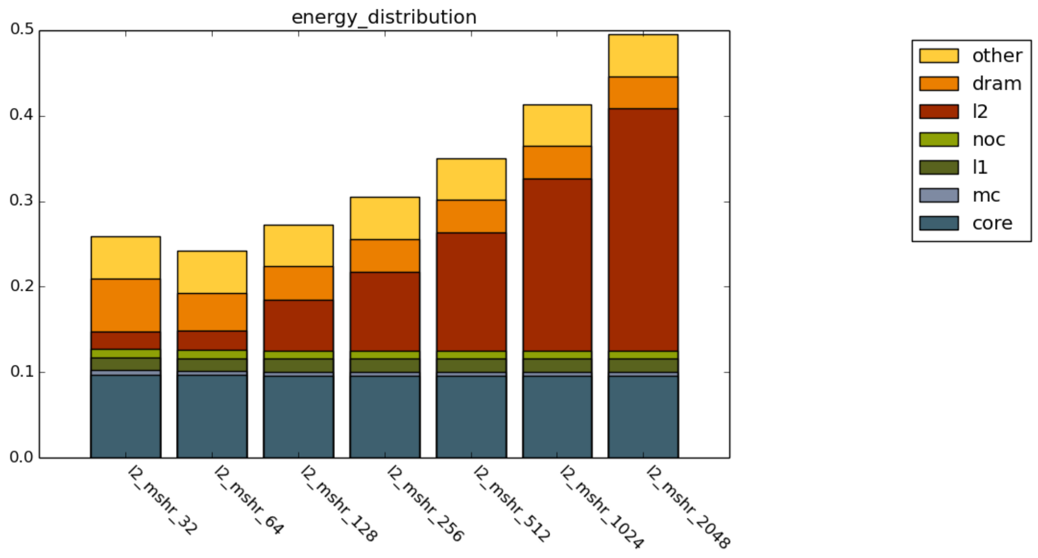


Figure 6.6: Energy consumption per component when increasing only the L2 size (Joules).

6.1.3 Area impact

As seen with the power and energy, area also experiences an exponential increase especially after quadrupling the size of the caches. Area increases from 243.64 mm^2 to 1550.63 mm^2 when increasing L1 and L2 sizes, the increase of MC to 32 also increases, but not as dramatically than the caches, to a total of 1820.74 mm^2. Figure 6.7 shows the increase of area produced by the increase of L1 and L2 size.

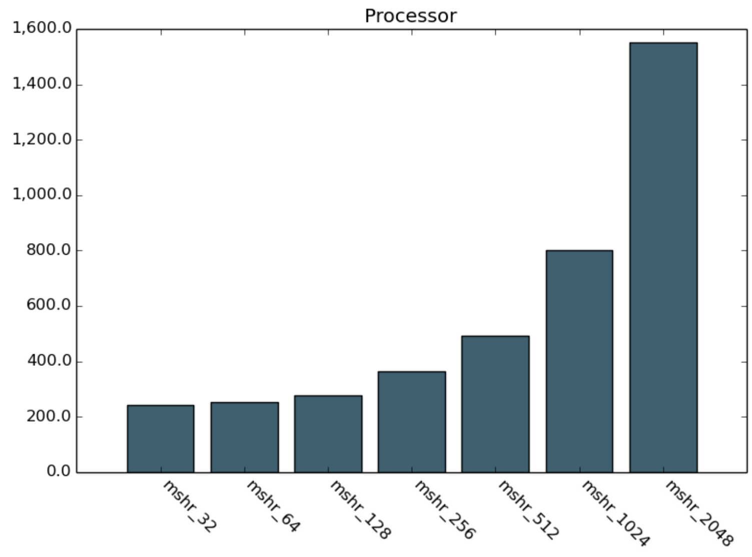


Figure 6.7: GPU area increase when increasing L1 and L2 sizes (mm^2).

6.1.4 Summary

The memory hierarchy changes incur in a significant increase in performance of 1.84x and utilization increase of up to 9.4%, while increasing exponentially the power up to 1009%, and energy up to 640%, as well as area up to 747%.

Due the severe overheads in power and area, we cannot maximize the cache sizes nor increase too much the number of MC, but an increase by a factor of 2x or 4x, for the caches, and a 8 MC, provide a good balance between performance gains and small power and area overheads. The best configuration candidate is a factor of 2x since we even see a reduction of energy, which is more valuable than a small performance gain obtained by using a factor 4x.

With a factor of 2x for the caches and 8 MC, the performance gain is 1.37x, power increase by 12.31%, energy reduction by 15.38%, and area increase by 13.31%.

6.2 Concurrent Warps & Warp Schedulers

The utilization of the GPU is very low so we could try to remove architectural resources from it by reducing the capabilities of the cores, do a resize of the cores, and due to the low utilization we would likely obtain reduction in area and power consumption and a slim degradation of the performance. With this idea in mind we tested and evaluated the viability of several architectural modifications to our baseline architecture to try to obtain these benefits.

The modifications include:

- Scaling down of the number of concurrent warps supported as well as the hardware resources associated with it, and weigh the benefits and drawbacks. The changes test warp concurrency numbers from 8 to 32.
- Reduction of the number of warp issue slots, or warp issue schedulers, and number of execution units in the cores, to see if has a big impact on the performance. The changes test 4, 3 and 2 warp issue slots and scale the execution units from 128 to 32.

6.2.1 Performance impact

The number of concurrent warps does not have an impact on the performance of the application until below the 16 concurrent warps, this is so because the thread block of the application is of 8 warps, thus below 16 concurrent warps it cannot run two thread blocks on the same core, degrading the performance to a 17%. Figure 6.8 shows the performance degradation starting at 15 concurrent warps.

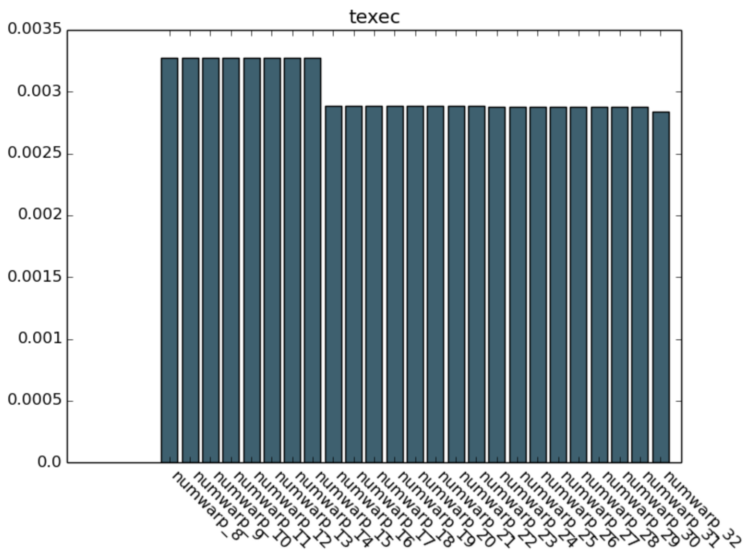


Figure 6.8: Execution time for the variation of number of concurrent warps (seconds).

Regarding the reduction of warp issue slots and execution units has a minimal effect on performance as it can be seen in Figure 6.9, which confirms the hypotheses that as the GPU utilization is low, the reduction of its resources would not hinder the performance significantly.

The IPC goes in hand with the decrease on performance, as the number of instructions stays the same. The baseline average is at 84.6 ipc which drops to 72.08 ipc when modifying the number of concurrent warps and is almost unaltered for the scheduler and execution units changes.

Regarding the issue rate, we observe the same pattern seen with performance and IPC. From a baseline issue rate of 3.26 to 2.77 when modifying the number of concurrent warps and almost unaltered for the scheduler and execution unit changes.

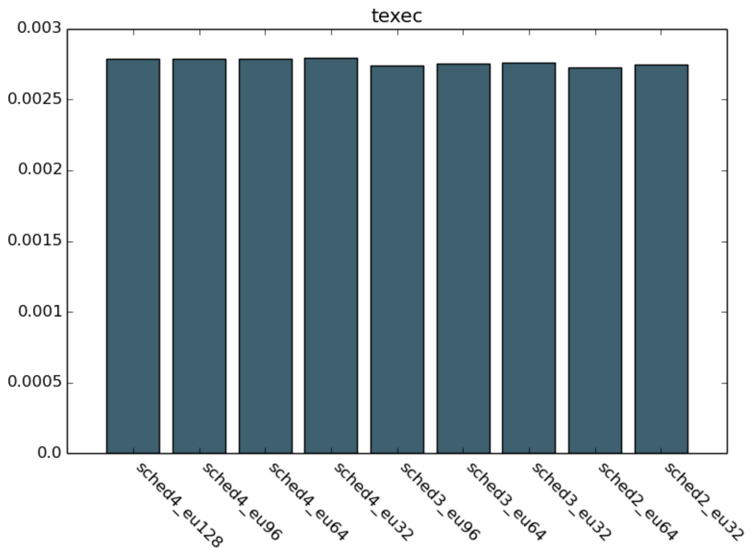


Figure 6.9: Execution time for the variation of warp issue schedulers and execution units (seconds).

More interesting are the results observed for the issue slots distribution. For the changes on the number of concurrent warps, the utilization drops from 5.1% to 4.3%. At the same time, as we can see on Figure 6.10, the distribution shifts from the memory stalls being the most relevant factor of no utilization of the issue slot to idle and raw dependencies. This is due to the fact what with less concurrent warps, the bottleneck is at the pipeline, we do not have enough warps to hide the dependencies between instructions and as a consequence we never get to saturate the memory hierarchy.

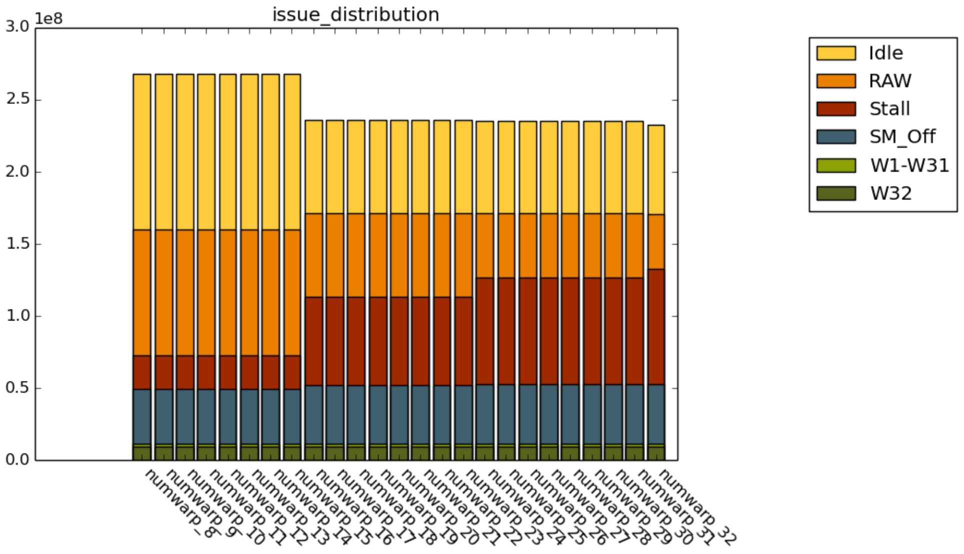


Figure 6.10: Issue slot distribution for the variation of number of concurrent warps.

On the other hand, for the changes of the scheduler and execution units, we see an important increase in utilization when reducing the number of warp scheduler slots: 6.8% utilization with three schedulers and 10.4% with two schedulers, Figure 6.11 shows this evolution. This is due to the fact that we are removing warp issue slots and thus reducing the amount of available warp issue slots on a given cycle, from the original 64 to 48 and 32 with three and two schedulers respectively, since the execution time is similar we observe a reduction of overall warp issue slots.

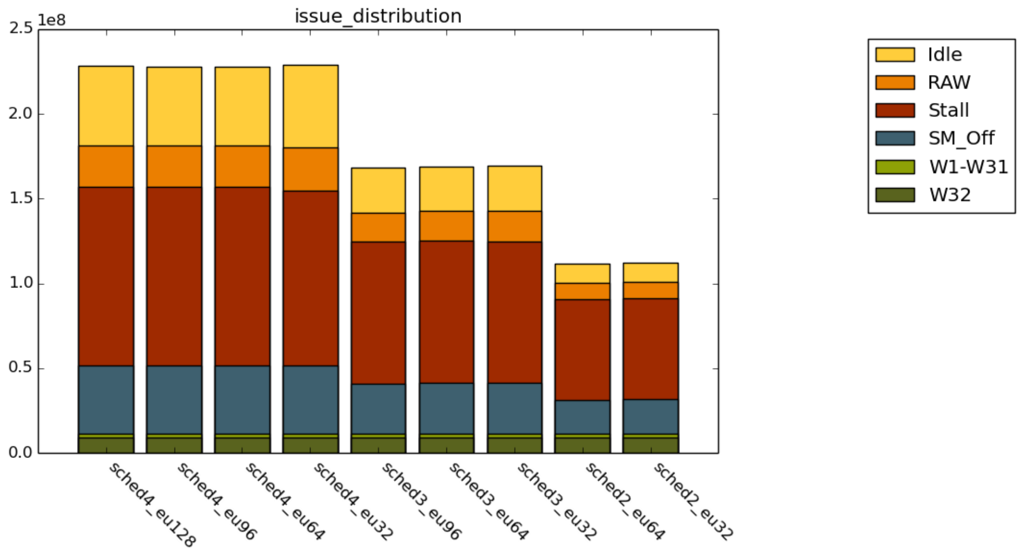


Figure 6.11: Issue slot distribution for the variation warp issue schedulers and execution units.

Finally, the impact of the tests on the caches is minimal. For the reduction of number of concurrent warps we see a small reduction of less than 5% of the L1 data accesses and misses, which leads to a reduction of the same proportion of L2 accesses, this reductions is due to more re-utilization of the cache lines as the number of concurrent warps lowers. For the reduction of scheduler and execution units, the caches are unaltered.

6.2.2 Power and energy impact

The reduction of concurrent warps has an impact on the dynamic power, decreasing the overall power consumption from 92.89 W to 82.48 W, a reduction of 12.6%. This power reduction only takes place for less than 16 concurrent warps as seen on Figure 6.12. Despite the power reduction, energy consumption increases due to longer execution from 0.25 J to 0.27 J, an increase of 8%.

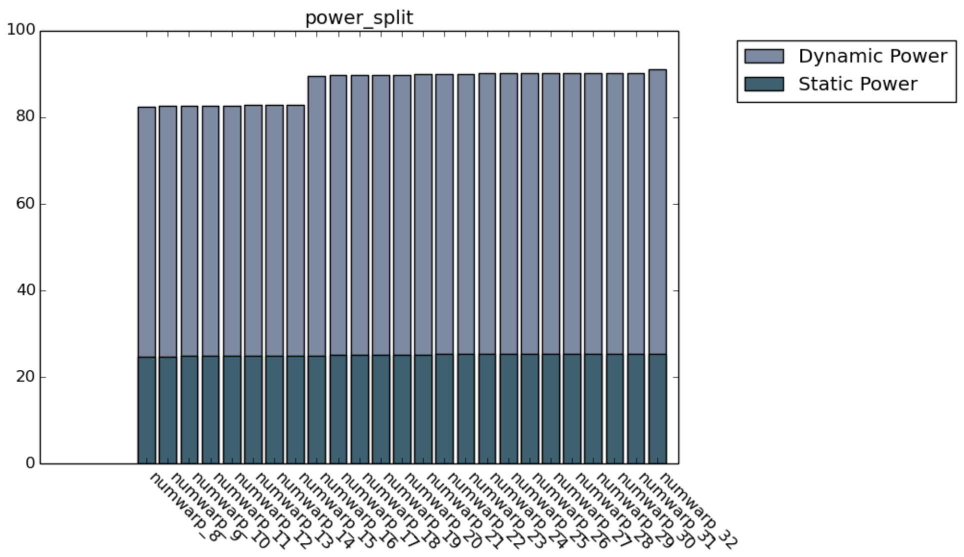


Figure 6.12: Power split, dynamic and static, for the variation of number of concurrent warps (Watts).

Regarding the reduction of warp schedulers and execution units, we observe a decrease in power, static and dynamic, as well as a decrease in energy consumption.

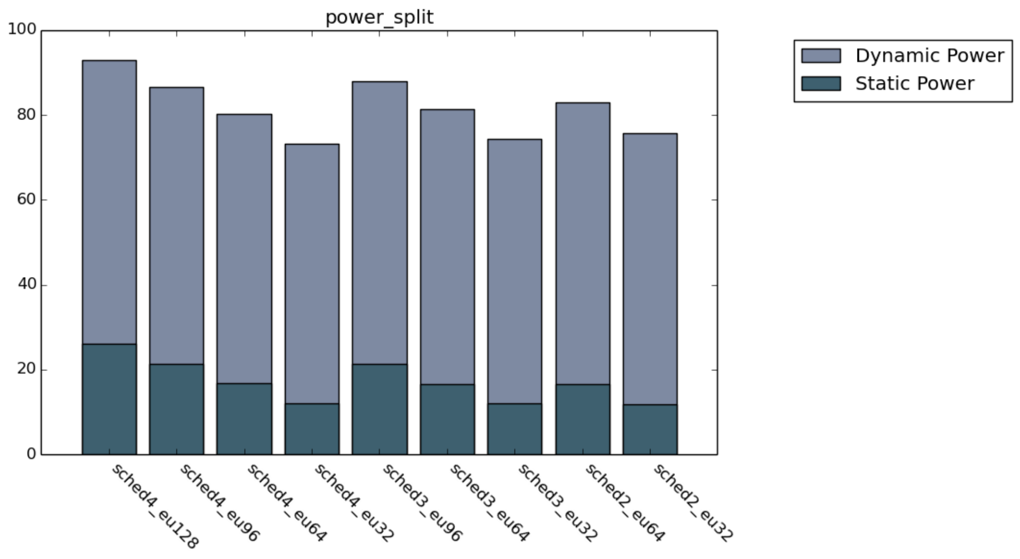


Figure 6.13: Power split, dynamic and static, for the variation warp issue schedulers and execution units (Watts).

The maximum power reduction we obtain it with 4 schedulers and 32 execution units, from a total of 92.89 W to 73.20, a reduction of 21.19%. On Figure 6.14 we can see the power reduction for the different configurations, the cause of the reduction are the reduction of execution units, whereas the removal of scheduling units slightly increases the dynamic power.

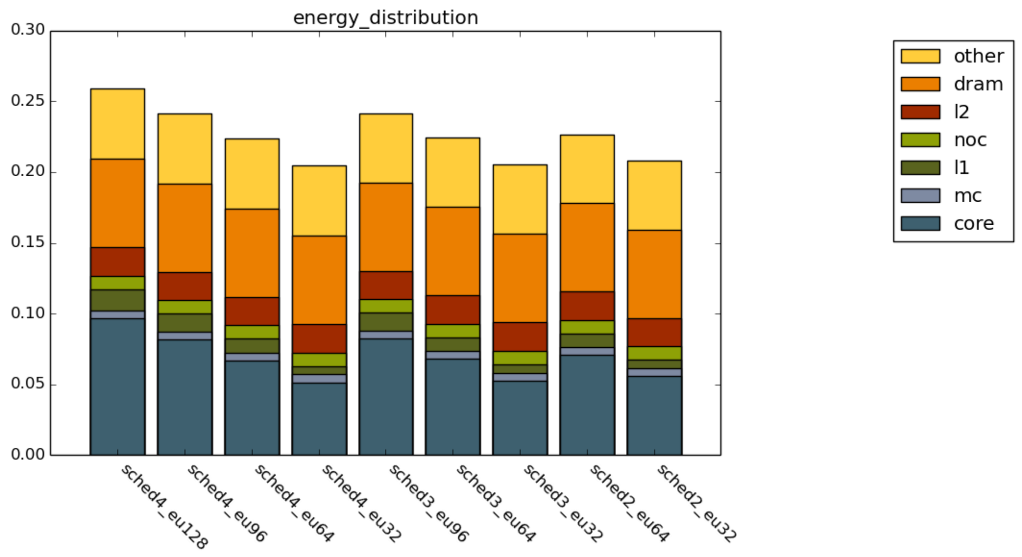


Figure 6.14: Energy distribution per component for the variation warp issue schedulers and execution units (Joules).

The energy reduces from 0.25 J to a maximum of 0.20 J, for the same configuration where we observe the lowest power consumption. On Figure 6.14 we see the energy consumption split between components, the changes target core components, where we see the main impact reduction, but we also see a reduction in L1 energy consumption.

6.2.3 Area impact

The change of concurrent warps affects the area by scaling the register file according to the maximum supported. The impact of this scaling on the area reduces it from 243.64 mm^2 to 214.79 mm^2, a reduction of 11.84%.

The removal of warp schedulers and execution units has a bigger impact on the area: from 243.64 mm^2 to 193.53 mm^2, a shrinkage of 20.56%. Figure 6.15 shows the variation of area of the components of the whole execution units of a given core. The ALU, FPU and Complex ALU are the components which contribute to this area reduction.

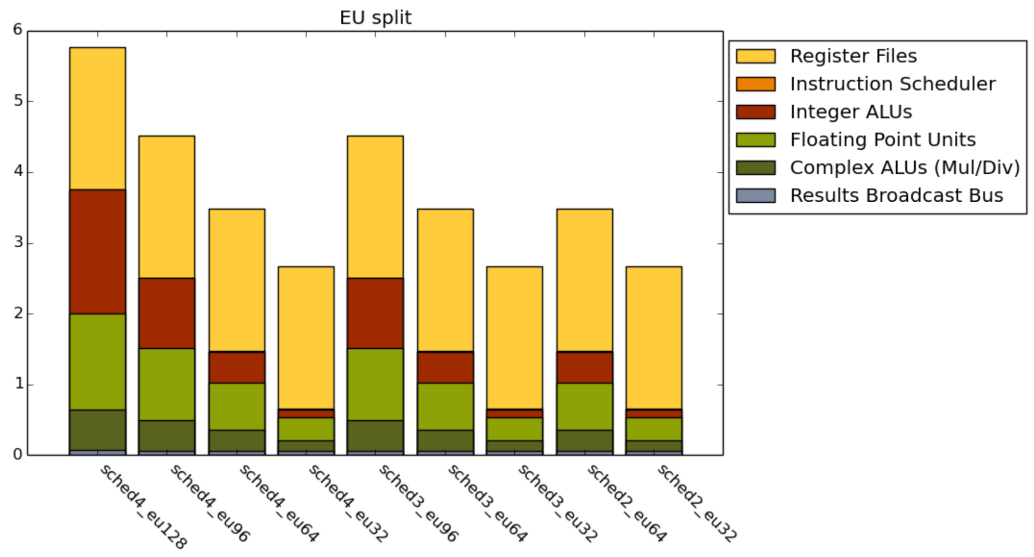


Figure 6.15: GPU area for the execution units per component for the variation warp issue schedulers and execution units (mm^2).

6.2.4 Summary

The core resize exploration incur in a small degradation in performance of up to 17% an utilization increase of up to 10.4%, while reaching a power reduction of up to 21.19%, and energy reduction up to 20%, as well as area shrinkage up to 20.56%.

The best configuration candidate seems: a reduction to 16 concurrent warps, before the drop in performance which allows us to benefit from a small power and area increase; and 2 warp issue schedulers with 32 execution units, which provide the maximum issue slot utilization increase, close to the best power and energy reduction and maximum area reduction.

With 16 concurrent warps the performance degrades by 3.31%, with power dissipation reduction of 3.62%, energy increase by 3.34% and area reduction by 10.65%. On the other hand, with the scheduling resize, we observe a performance increase of 1.6%, warp issue utilization up to 10.4%, power dissipation reduction of 18.43%, energy reduction by 16.8% and area reduction by 20.56%.

6.3 Frequency

The main bottleneck is still the memory, we cannot provide sufficient bandwidth to hide the latency for the Viterbi decoder, and the GPU wastes cycles of execution waiting for memory to fetch the information. We could try to scale down the frequency of the GPU which could lead to energy savings if the degradation of the performance is not significant.

The modifications include:

- Scaling down of frequency of the GPU to study the trade-off of energy savings and performance degradation. The changes test a range of frequencies from 1.28 GHz to 0.28 GHz, in steps of 100 MHz.

6.3.1 Performance impact

Performance is greatly impacted by the slowdown of frequency. While decreasing the frequency from 1.28 Ghz to 0.28 GHz the application goes from 2.79 ms to 5.4 ms, a slowdown of 93.54%. Figure 6.16 shows the exponential decrease in performance for this architectural exploration.

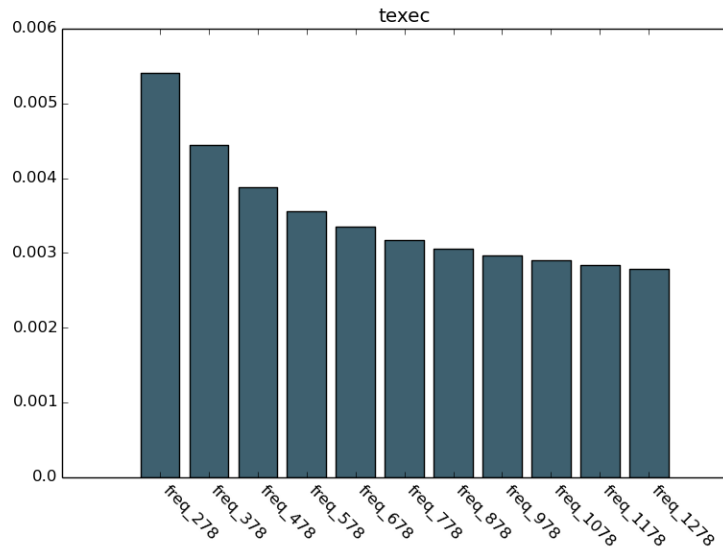


Figure 6.16: Execution time for the variation frequency (seconds).

The IPC experiences a less strong exponential increase, starting at 84.6 ipc for the baseline and reaching up to 200.66 ipc, an increase of 237%, with the slowest frequency configuration. The reason behind this increase is the reduction of cycles of execution from 3,566,151 to 1,503,566 cycles. The decrease in cycles is due to the fact that most of the cycles where wasted waiting for memory petitions. By increasing the frequency we make every cycle larger which lowers the performance of the execution units but not the memory petitions, and thus it hides them with less cycles.

Regarding the issue rate, we observe the same increase. From a baseline issue rate of 3.26 to a 7.73 for the lowest frequency. The issue slot utilization increases greatly as seen on the Figure 6.17 which shows the issue distribution of the architectural exploration. From a baseline issue slot utilization of 5.1% to a 12.1%. The decrease of frequency manages to hide many of the cycles of no utilization of the issue slot, especially the ones related to memory stalls but also the idle ones.

Finally this architectural exploration does not have any impact on the L1 and L2 caches.

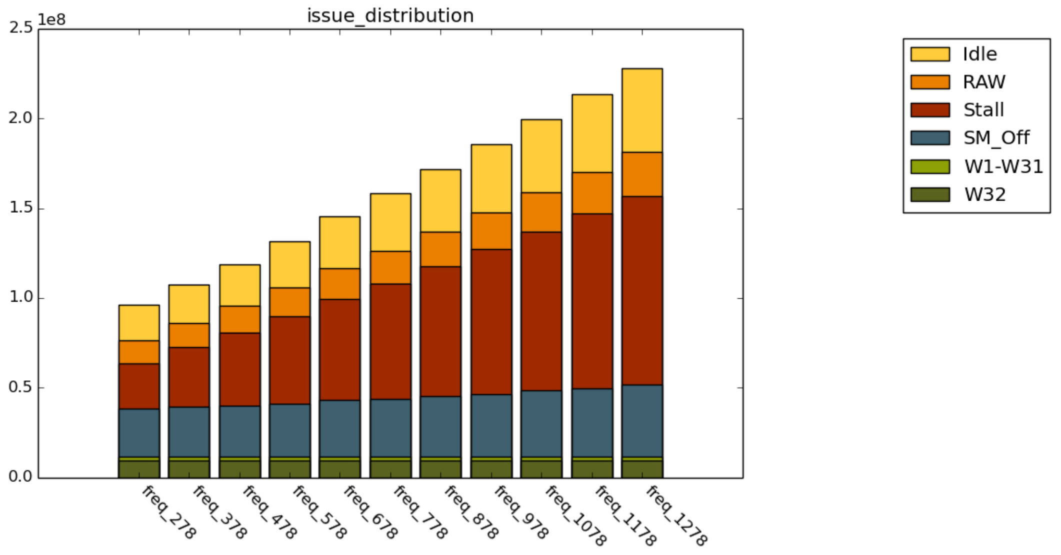


Figure 6.17: Issue slot distribution for the variation frequency.

6.3.2 Power and energy impact

Figure 6.18 shows the decrease on frequency lowers the dynamic power, and the overall requirement of the baseline architecture from 92.89 W to 70.67 W, a reduction of 23.92%. Unfortunately the performance slowdown increases the overall energy from 0.25 J to 0.38 J, an increase of 52%.

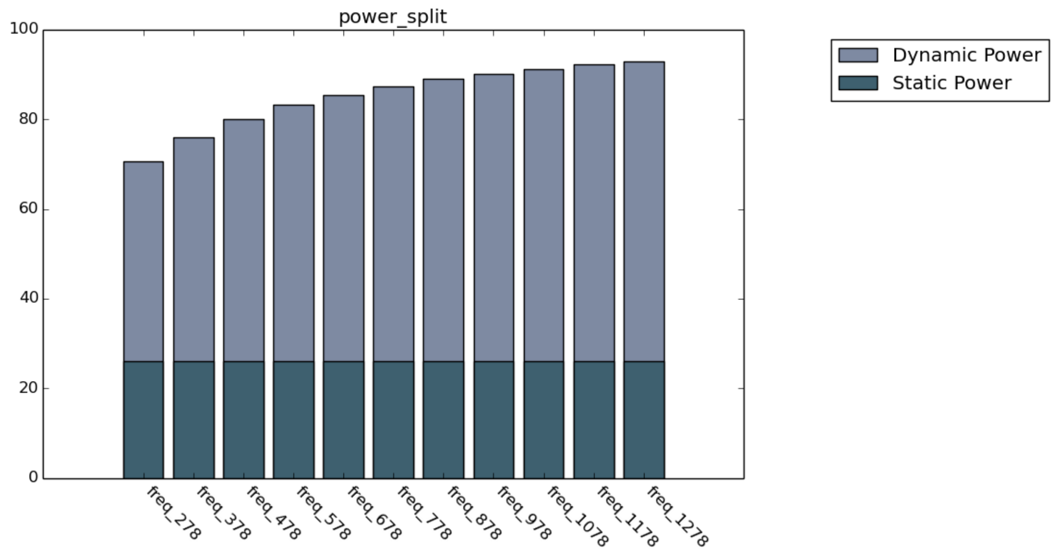


Figure 6.18: Power split, dynamic and static, for the variation of frequency (Watts).

6.3.3 Area impact

The area of the GPU baseline architecture is left unaffected by this architectural exploration.

6.3.4 Summary

The frequency scaling incur in a significant degradation in performance of 93.54% an utilization increase of up to 12.1%, while reaching a power reduction of up to 23.92%, and energy reduction up to 52%. The area is left unaltered.

Performance is not a limitation for our architecture, we could consider lowering to a maximum the frequency, but the energy consumption increases quite significantly. The considered candidate is a drop in frequency to around half the baseline frequency, 678 Mhz in this experimental run, which is before we observe an exponential increase of execution time and energy consumption.

With a frequency of 678 MHz, the performance degrades by 19.97%, power consumption decreases by 7.95% and energy increases to 14.5%. The area is left unaltered.

6.4 Improved architecture

In this section we apply the previously gathered information of the different architectural explorations to propose a modified version of the baseline architecture with the intention to fit better the requirements of speech recognition algorithms on GPU architectures.

The modifications over the baseline architecture include:

- L1 data size increased by a factor of 2. 64 KB (64 sets, 128 line size, 8 assoc)
- L2 total size increased by a factor of 2. Subpartitions maintain same size 256KB (256 sets, 128 line size, 8 assoc), the increase in size is due to the number of memory channels.
- A total of 8 memory channels.
- 16 concurrent warps.
- 2 warp issue scheduler.
- 32 execution units per core.
- Frequency at 639 MHz, half of the original.

6.4.1 Performance impact

The performance of our improved architecture is worsened, as seen in Figure 6.19, the application goes from 2.79 ms to 3.1 ms, an slowdown of 11.1%. The contribution factors are the changes on the memory hierarchy, the increase in L1 and L2 cache size as well as the increase of memory controllers, which contribute positively and the decrease in frequency which overcome the gains previously achieved. Overall the slowdown is relatively small and after all the application is reaches real-time speech recognition.

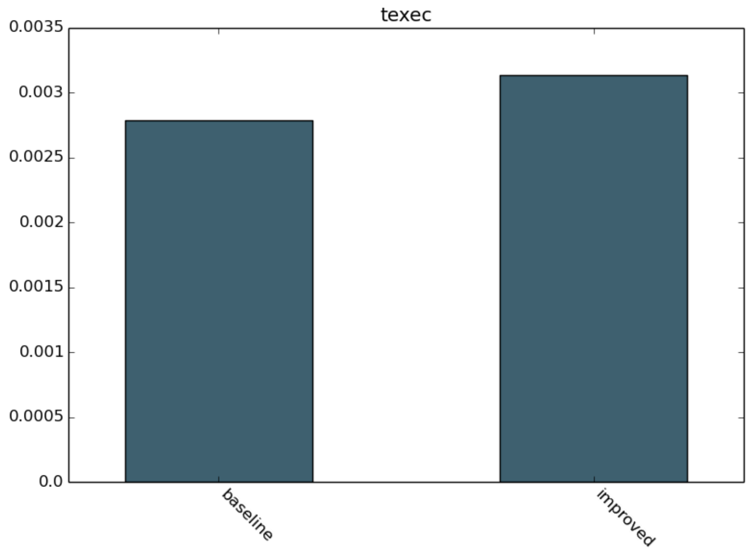


Figure 6.19: Execution time change for the improved architecture (seconds).

The IPC of the application increases, from 84.6 ipc to 150.63 ipc, an increase of 78%. The increase is caused by the reduction of cycles caused by the increase in performance of memory hierarchy, which remove execution cycles, as well as the increase in frequency which also remove execution cycles.

Regarding the issue rate, we observe the same increase. From a baseline issue rate of 3.26 to 5.8. As with the increase in IPC, the contributor factors are the memory hierarchy changes and the frequency.

The issue slot utilization has a big increase, from a baseline of 5.1% to 18.1%, the biggest contributor is the warp issue scheduler reduction. Figure 6.20 shows the issue slot distribution, in it we see a reduction of the percentage of issue slot cycles of all the factors that do not contribute to the performance, and the overall utilization gain stated previously.

Finally, the L1 data and L2 see a reduction on miss rate of, from 62% to 57% and from 28% to 20%, respectively. The memory hierarchy changes are the reason for this improvement.

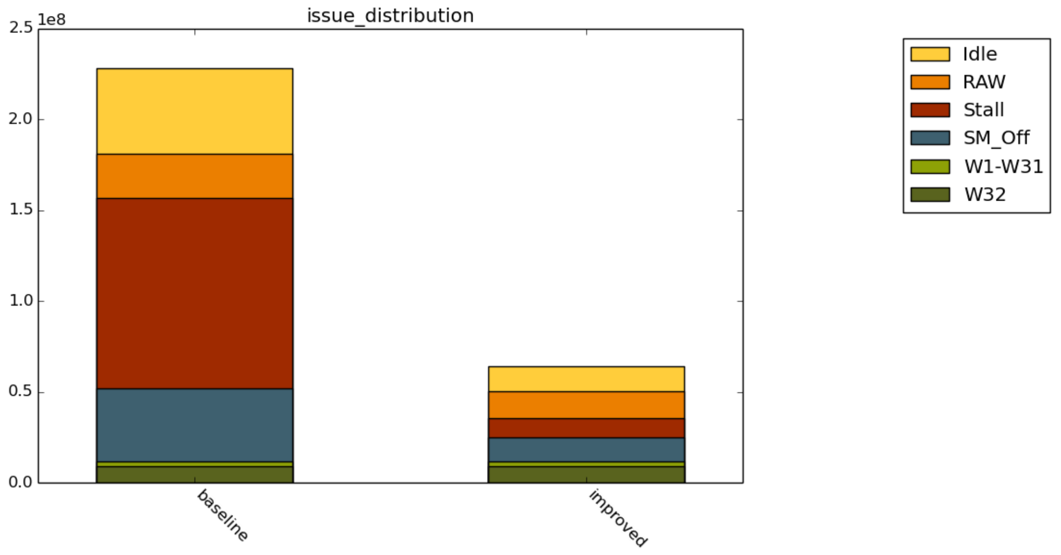


Figure 6.20: Issue slot distribution for the improved architecture.

6.4.2 Power and energy impact

Power and energy consumptions are reduced as a result of the architectural improvements. On Figure 6.21 we can see the decrease of power, both static and dynamic, from a baseline of 92.89 W to 63.46 W, a reduction of 31.6%. The changes on the memory hierarchy have a null effect on the power reduction, but the reduction of concurrent warps and warp issue schedulers, especially as well as the reduction of frequency contribute to this gain.

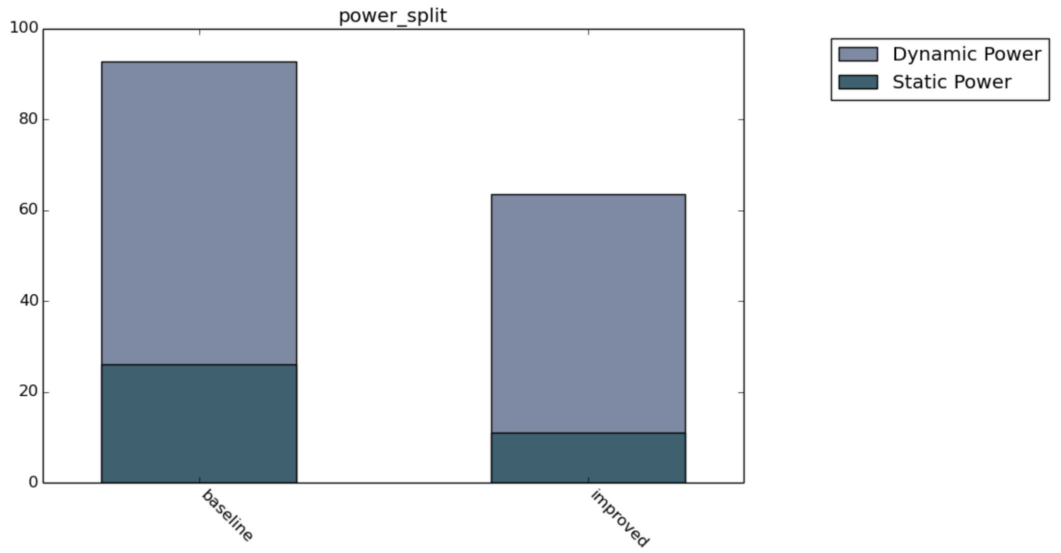


Figure 6.21: Power split, dynamic and static, for the improved architecture (Watts).

On Figure 6.22 we can see the energy reduction from a baseline of 0.25 J to 0.19 J, a reduction of 24%. All architectural modifications go in the direction of reducing the energy consumption but the frequency slowdown, which is the only factor which effectively by increasing the execution time increases the energy consumption.

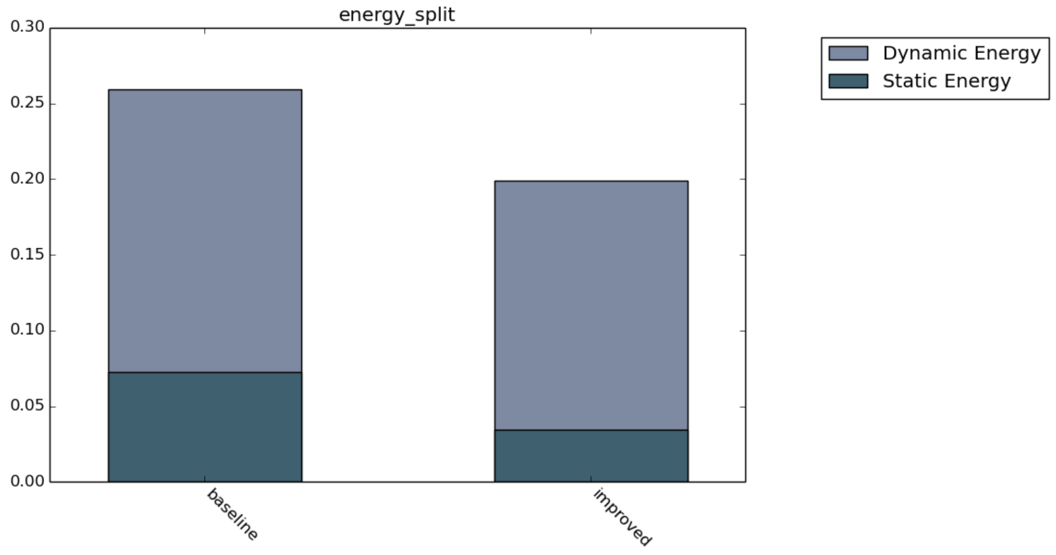


Figure 6.22: Energy split, dynamic and static, for the improved architecture (Joules).

6.4.3 Area impact

The architectural improvements reduce the area from a baseline of 243.64 mm^2 to 199.87 mm^2, a shrinkage of 17.96%. In Figure 6.23 we can see the area reduction. The memory hierarchy improvements increase significantly the area for the L1, L2 and MC, but then the changes on concurrent warps and warp issue schedulers overcompensate by reducing the execution unit of the cores.

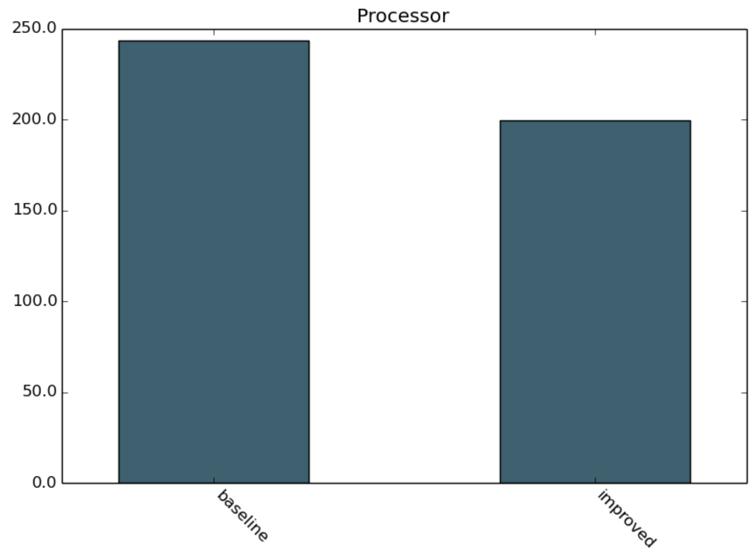


Figure 6.23: GPU area for the improved architecture (mm^2).

6.4.4 Summary

The improved architecture incur in a small degradation in performance of up to 11.1% an utilization increase of up to 18.1%, while reaching a power reduction of up to 31.6%, energy reduction up to 24%, as well as area shrinkage up to 17.96%.

7 Conclusions

Automatic speech recognition systems are a decisive method of interaction with hardware devices, currently and in the future. Speech recognition workloads can be highly parallelized, a characteristic that GPUs are able to exploit greatly whereas CPUs cannot do it as extensively. Nonetheless, GPUs are not perfectly well-suited for the characteristics of this workloads, as the irregular memory accesses are the key performance limiting factor, however they easily exceed the real-time requirement for automatic speech recognition systems.

The aim of this work was to characterize the limiting factors of state-of-the-art GPU architectures for automatic speech recognition systems and provide insights on GPU architectures more well-suited for this type of workloads.

Throughout this work, we have implemented the Viterbi decoder algorithm for GPUs architectures, which is used for the dominant phase of the speech recognition audio processing. Our GPU Viterbi algorithm implementation achieves an speedup between 9.93x to 17.9x when comparing it to CPU implementations, and performs better than other GPU implementations. Nevertheless, this results are far from reaching the full parallelization capabilities of GPU in front of CPU architectures.

We have explored a number of architectural modifications on a simulated state-of-the-art GPU architecture, used as a baseline architecture, with the aim to extract more performance, reduce power and energy requirements and shrink processor area. The architectural modifications include: the increase of the memory hierarchy capabilities of GPU architectures to relieve this observed bottleneck; the resize of core architectural resources to increase the utilization of the GPU resources, while reducing power, energy and area requirements; and reduction of frequency with the objective to increase resources utilization factors and reduce power requirements and the expense of performance degradation.

Finally, we propose an improved GPU architecture which incorporates changes of the previous explorations of architectural modifications. The improved architecture incurs in a performance degradation of 11.1% while increasing the utilization factor of the architectural resources to 18.1%, achieving a power reduction of 31.6%, an energy reduction of 24% and area shrinkage of 17.96%. The resulting GPU architecture, even though it degrades performance, still meets real-time speech recognition requirements and overall is an architecture more well-suited for the characteristics of automatic speech recognition workloads.

Bibliography

[1] X. Anguera, S. Bozonnet, N. Evans, C. Fredouille, G. Friedland, and O. Vinyals. Speaker diarization: A review of recent research. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(2):356–370, Feb 2012.

[2] https://en.wikipedia.org/wiki/Siri.

[3] 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.*, volume 5, pages 872–875, Apr 1980.

[4] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[5] N. Brunie, F. de Dinechin, and B. de Dinechin. A mixed-precision fused multiply and add. In *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 165–169, Nov 2011.

[6] Shaunak Chatterjee and Stuart Russell. A temporally abstracted Viterbi algorithm. *Uai11*, 2011.

[7] 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 *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, Dec 2014.

[8] Jike Chong. Efficient Automatic Speech Recognition on the GPU By : Jike Chong. 2010.

[9] Jike Chong, Ekaterina Gonina, Youngmin Yi, and Kurt Keutzer. A Fully Data Parallel WFST-based Large Vocabulary Continuous Speech Recognition on a Graphics Processing Unit. pages 2–5.

[10] http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[11] 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 *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, June 2015.

[12] https://en.wikipedia.org/wiki/Google_Now.

[13] http://www.gpgpu-sim.org/gpuwattch/.

[14] R. Hsiao, M. Fuhs, Q. J. Y. Tam, I. Lane, and T. Schultz. Handbook of natural language processing and machine translation. page 496–504, 2011.

[15] Adam Louis Janin. *Speech Recognition on Vector Architectures.* PhD thesis, Berkeley, CA, USA, 2004. AAI3165422.

[16] Parallel Processing Letters, World Scientific, Publishing Company, Andrew Lumsdaine, Bruce Hendrickson, Jonathan Berry, Sandia National, Laboratories Albuquerque, Revised January, and Bernard Tourancheau. Challenges in parallel graph processing. 2007.

[17] 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 *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec 2009.

[18] http://www.openslr.org/12/.

[19] Xie Lingyun and Du Limin. Efficient viterbi beam search algorithm using dynamic pruning. In *Signal Processing, 2004. Proceedings. ICSP '04. 2004 7th International Conference on*, volume 1, pages 699–702 vol.1, Aug 2004.

[20] Xinxin Mei, Xiaowen Chu, and Senior Member. Dissecting GPU Memory Hierarchy through Microbenchmarking. pages 1–14.

[21] https://en.wikipedia.org/wiki/Cortana_(software).

[22] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88, 2002.

[23] U. Nallasamy, I. Lane, M. Fuhs, M. Noamany, Y. Tam, Q. Jin, and T. Schultz. Handbook of natural language processing and machine translation. page 535–540, 2011.

[24] http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

[25] https://www.khronos.org/opencl/.

[26] Daniel Povey, Arnab Ghoshal, Nagendra Goel, Mirko Hannemann, Yanmin Qian, Petr Schwarz, Jan Silovsk, and Petr Motl. The Kaldi Speech Recognition Toolkit.

[27] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[28] Johan Schalkwyk, Doug Beeferman, Françoise Beaufays, Bill Byrne, Ciprian Chelba, Mike Cohen, Maryam Kamvar, and Brian Strope. *"Your Word is my Command": Google Search by Voice: A Case Study*, pages 61–90. Springer US, Boston, MA, 2010.

[29] https://www.skype.com/en/features/skype-translator/.

[30] http://stereopsis.com/radix.html.

[31] Kisun You, Jike Chong, Youngmin Yi, Ekaterina Gonina, Christopher J. Hughes, Yen Kuang Chen, Wonyong Sung, and Kurt Keutzer. Parallel scalability in speech recognition: Inference engines in large vocabulary continuous speech recognition. *IEEE Signal Processing Magazine*, 2009.