THE A* SPEECH RECOGNITION SYSTEM ON PARALLEL ARCHITECTURES

Patrick Cardinal[1,2], *Gilles Boulianne*[1] *and Pierre Dumouchel*[1,2]

[1]Centre de Recherche Informatique de Montréal (CRIM),Montréal, Canada
[2]École de Technologie Supérieure, Montréal, Canada
Email: {patrick.cardinal,pierre.dumouchel,gilles.boulianne}@crim.ca

ABSTRACT

The speed of modern processors has remained constant over the last few years but the integration capacity continues to follow Moore's law and thus, to be scalable, applications must be parallelized. In addition to the main CPU, almost every computer is equipped with a Graphics Processors Unit (GPU) which is in essence a specialized parallel processor. This paper explore how performance of speech recognition systems can be enhanced by using the A* algorithm which allows better parallelization over the Viterbi algorithm and a GPU for the acoustic computations in large vocabulary applications. First experiments with a "unigram approximation" heuristic resulted in approximatively 8.7 times less states being explored compared to our classical Viterbi decoder. The multi-thread implementation of the A* decoder combined with GPU for acoustic computation led to a speed-up factor of 5.2 over its sequential counterpart and an improvement of 5% absolute of the accuracy over the sequential Viterbi search at real-time.

1. INTRODUCTION

Large vocabulary automatic speech-recognition is a computationally intensive task. Most speech recognizers run under a sequential implementation that cannot take advantage of modern processors with multi-core technology. In order to exploit this power, a parallel speech recognition system must be implemented.

Other architectures specialized in parallel computations can be used as a coprocessor to outperform the speeds offered by a modern CPU alone. In fact, almost every modern-day computer contains such a device: modern graphic cards incorporate a specialized processor called Graphics Processing Unit (GPU). A GPU is mainly a Single Instruction, Multiple Data (SIMD) parallel processor that is computationally powerful, while being quite affordable. Over the years, the GPU has evolved into a flexible processor.

A noteworthy technological advance was achieved in 2007, when NVidia and ATI introduced the unified architecture. This greatly enhanced the flexibility and usability of the GPU, to the extent that it is becoming a mainstream alternative for general purpose calculations.

The two major time consuming components in speech recognition are the acoustic likelihood computation and the optimal path search. The first component takes 30%-70% of total time. This calculation involves mostly arithmetic operations than can be computed by a dot product. This allows an efficient implemention in a SIMD (Single Instruction Multiple Data) parallel architecture such as SSE registers or a graphic processor (GPU) [1].

The search component consumes most of the remaining time. The classic way to perform the decoding uses the Viterbi algorithm. This algorithm is simple and straightforward to implement. It is nonetheless difficult to achieve an efficient parallelized version of the Viterbi algorithm on a classical multi-core computer. The main reason is that only 1% of the states are active at each frame and these are scattered in memory. This situation adds to the well established difficulty of having to search a sparse graph on a parallel architecture of the Intel processor type [2].

A parallel implementation of a speech recognition system is presented by Phillips *et al.* [3]. Their system builds the transducer on the fly during the decoding process. They have obtained a performance of 0.8x real-time on a 16 CPU computer for the North American Business News (NAB) database. This is a speed-up of 4.87 compared to 3.8x real-time on a single CPU.

Parihar *et al.* implement the parallelization of the search component of a lexical-tree based speech recognizer [4]. In this work, lexical-tree copies are dynamically distributed among the cores to ensure a good load balancing. This results in a speed-up of 2.09 over a serialized version on a Core i7 quad (4 cores) processor. The speed-up is limited by the memory architecture.

In [5], Ishikawa *et al.* implemented a parallel speech recognition system in a cellphone using a 3-core processor. The system was divided in 3 steps, one for each core. They reported a speed-up factor of 2.6 but their approach is not scalable since involved steps are not easily parallelizable.

In [6], Chong *et al.* implemented a WFST (weighted finite state transducer) and LLM (linear lexical model) speech recognizer in a GPU (240 arithmetic units). A GPU offers a highly parallel architecture which is better suited for sparse graph searches. They report speed increases of up to 10.96 for the LLM and 13.72 for the FST model.

This paper presents results of using the A* search algorithm in which a GPU is used for the acoustic computations in a large vocabulary speech recognition parallel system. The A* approach has previously been applied to speech recognition by [7]. It divides the search operation into two steps. The first step is the computation of a heuristic that yields an estimate of the cost for reaching the final state from any given state in the graph. The second step is a best-first search driven by the heuristic. The advantage of this approach is that the heuristic can be constructed to allow an efficient computation in parallel. The search itself is still difficult to parallelize, but it can be reduced by using a good heuristic since, in this case, a smaller number of states will be explored.

2. A* DECODER

2.1. Background work

We had already parallelized the classical Viterbi algorithm in this previous work [1]. The set of active states is divided into n subsets, where n is the number of threads dedicated to the state expansion process. However, some transitions lead to states belonging to another thread. Updating these states in parallel can lead to data incoherency. To circumvent this problem, we keep duplicate destination state information and merge them after all states have been expanded. Although this solution implies an overhead, it is much faster than the use of mutexes.

At the end of the expansion process, the best score is found and state pruning is applied in parallel. Then, surviving states of each subset are merged together to create the set of active states for the next iteration. The merging step is performed sequentially.

The results have revealed a speed-up factor of only 1.3 on a core2 quad processor over a single processor system. This result is mainly due to the sparsity of the active states in memory caused by the beam pruning which leads to a misuse of the memory architecture. Since we cannot know which states will be used in advance, it is very difficult to overcome this problem.

If, at the outset, we disposed of a smaller graph, allowing a exhaustive search, it would be much easier to implement a parallel version of it. This is the idea behind the use of the A* algorithm for which the heuristic is represented by a FST.

2.2. A* algorithm

Unlike the time synchronous Viterbi algorithm, the A* algorithm is a best-first scheme, that implies a scoring procedure to explore the most promising states. The score of a state is

$$Score(q) = g(q,t) + h(q', t+1) + cost(q, q')$$

where $g(q,t)$ is the score for reaching state q from the initial one at time t, h is the heuristic score that gives an estimation of the cost for reaching a final state from the adjacent state q' at time $t+1$ and $cost(q,q')$ is the cost for going to q' from q. A heuristic is said to be admissible if, for every state, it underestimates the real cost for reaching the final state. In that case, the A* algorithm is optimal. A pseudocode of the A* algorithm is shown in Algorithm 1. For simplicity, epsilon transition handling has not been illustrated in this algorithm.

The input of the algorithm is the HCLG recognition network composed of HMMs (H), triphone context dependency (C), lexicon (L) and a trigram backoff language model (G). This network is represented by a $\mathcal{WFST} = (Q, i, F, \Sigma_i, \Sigma_o, E, \lambda, \rho)$ where Q is a set of states, $i \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, Σ_i is the input alphabet of the automaton (distributions), Σ_o is the output alphabet of the automaton (words), $E \subset Q \times \Sigma_i \times \Sigma_o \times \mathbb{R} \times Q$ is the set of transitions, $\lambda : i \to \mathbb{R}$ is the initial weight function and $\rho : F \to K$ is the final weight function. The second input is the heuristic function $h : q, t \to \mathbb{R}$ which gives the estimated cost for reaching a final state from state q at time t.

$openList \leftarrow \{((i, \lambda, 0), heuristic(i, 0))\}$
$closedList \leftarrow \emptyset$
while $openList \neq \emptyset$ **do**
    `// Extract state with lowest score`
    $(q, t, g) \leftarrow openList.Extract()$
    $closedList \leftarrow closedList \cup (q, t)$
    if $q \in F$ *and* $t = numFrames$ **then**
        `// Best path found`
        ExitSearch()
    end
    foreach $(q, \sigma_i, \sigma_o, w, q') \in E[q]$ **do**
        if $(q', t+1) \notin closedList$ **then**
            $g' \leftarrow g + obsCost(\sigma_i, t) + w$
            $h \leftarrow heuristic(q', t+1)$
            $entry \leftarrow (q', t+1, g')$
            $score \leftarrow g' + h$
            $openList \leftarrow openList \cup \{(entry, score)\}$
        end
    end
end

Algorithm 1: *The A* algorithm*

2.3. Unigram Langage Model Heuristic

In our implementation, the heuristic is also represented by a WFST. The heuristic costs are computed by performing backward Viterbi decoding. The heuristic FST must be small enough to allow for an exhaustive search. In our experiments, it is built with the same models as that of the recognition network, with the exception that the trigram language model is replaced by a unigram model derived from the trigram. The resulting FST is small enough to be exhaustively and efficiently decoded.

Note that application of the Viterbi algorithm on the heuristic is simpler and faster than on the recognition network because no backpointers need to be kept to retrace the best state sequence. Moreover, since all states are explored at each frame, they reside in contiguous memory locations for optimal cache usage.

2.4. Mapping Recognition FST States to Heuristic States

Recall that A* search uses the heuristic cost given by the function $h(q_r, t)$, where q_r is a recognition FST state. In essence, this function performs a lookup in the Viterbi treillis computed on the heuristic. Thus, we need to know which state (q_h, t) in the heuristic is equivalent to (q_r, t). A mapping between states of the heuristic and those of the recognition FST must thus be discovered.

To establish this mapping, we can use the FST composition as described by Mohri [8]. The inverted (input and output symbols swapped) heuristic FST is composed with the recognition FST. A state in the composed FST is a pair $s_{hr} = (q_h, q_r)$ where q_h and q_r are, respectively, states of the heuristic and recognition FST. The existence of a state (q_h, q_r) implies that at least one path from i_h to q_h in the heuristic FST has the same distribution sequence than a path from i_r to q_r in the recognition FST. Since the composed FST is connected, there is also a path from q_h to a final state of the heuristic FST that has the same distribution sequence than a path from q_r to a final state of the recognition FST. Consequently, both states are considered to be equivalent. Note that the FST resulting from the composition is not used, only the list of state pairs is useful. In addition, this mapping is computed offline.

2.5. Block Processing

Data structures required for implementing A* are more complex than the simple array used in a Viterbi decoder. The A* algorithm always explores the most promising path first. For efficiency, paths are stored in a binary heap for which the three main operations (insertion, extraction and decrease key) are in O(log n). However, the algorithm needs to know if a node is already in the heap before inserting it. Since searching a node in a heap is O(n), a hash table is used to keep track of nodes in the open list. Moreover, since we don't want to explore the same node more than one time, we use a closed list of nodes already explored which is also implemented with a hash table. For efficiency, there is an open list (hash table) and a closed list per frame.

In addition to complex data structures, the number of nodes to explore grows as the square of the number of frames as is the case with the Viterbi algorithm. To circumvent both problems, a block approach has been implemented as follows. The heuristic is first computed for Δ frames. Then, the A* search is performed on the $\Lambda < \Delta$ first frames. The search stops when a node at time Λ with a cost (path cost + heuristic cost) larger than the best cost added to a user value (beam) is extracted from the open List.

The window is then advanced of Λ frames. The process is applied until the end of audio is reached. In order to save computation time, several consecutive searches can be done with one heuristic computation as shown by Figure 1.

This approach is equivalent to the beam pruning in the Viterbi algorithm. In order to limit the number of nodes in the open list, nodes outside the beam are not inserted in it.

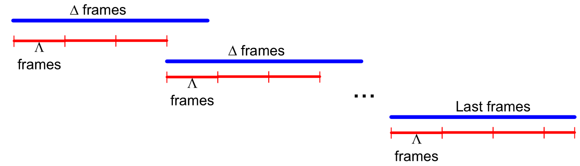

Fig. 1. *A* search by block*

3. ACOUSTIC LIKELIHOOD COMPUTATION

GPU is a SIMD parallel processor that is specialized in graphical rendering, which involves a high rate of linear algebra operations. Thus, using linear algebra-based acoustic likelihood calculations should lead to a more efficient use of the GPU hardware. The acoustic likelihood for a Gaussian mixture model is defined as :

$$b_j(\vec{o}_t) = \sum_{c=1}^{C_j} \alpha_{jc} \frac{1}{\sqrt{(2\pi)^d |\Sigma_{jc}|}} e^{-\frac{1}{2}(\vec{o}_t - \vec{\mu}_{jc})' \Sigma_{jc}^{-1} (\vec{o}_t - \vec{\mu}_{jc})}$$

where $b_j(\vec{o}_t)$ is the probability that distribution j generates the d-dimensional observation vector $\vec{o}$ at time t, C_j is the number of Gaussians in the distribution j, α_{jc} is the weight of Gaussian c in distribution j, μ_{jc} and Σ_{jc} are the mean vector and the covariance matrix of Gaussian c in distribution j. This equation can be expressed as:

$$b_{jc}(\vec{o}_t) = (h_{jc} + u_{jc}\vec{o}_t + v'_{jc}\vec{o}_t^2)$$

where:

$$\begin{aligned} h_{jc} &= \ln \alpha_{jc} - \frac{1}{2}\ln((2\pi)^d |\Sigma_{jc}|) - \frac{1}{2}\vec{\mu}'_{jc}\Sigma_{jc}^{-1}\vec{\mu}_{jc} \\ u_{jc} &= \vec{\mu}'_{jc} \cdot \Sigma_{jc}^{-1} \\ v_{jc} &= Diag(-\frac{1}{2}\Sigma_{jc}^{-1}) \end{aligned}$$

This computation can be accomplished by a dot-product of the following two vectors in which subscripts designating the distribution component have been omitted for clarity:

$$\begin{aligned} \vec{obs} &= (\bar{1}, o_1, o_2, \cdots, o_n, o_1^2, o_2^2, \cdots, o_n^2) \\ \vec{M} &= (h, \mu_1\sigma_{11}^{-1}, \cdots, \mu_n\sigma_{nn}, -\frac{1}{2}\sigma_{11}^{-1}, \cdots, -\frac{1}{2}\sigma_{nn}^{-1}) \end{aligned}$$

where $\bar{1}$ is the identity element of multiplication. The likelihood of a distribution is defined as :

$$\ln b_j(\vec{o}_t) = \bigoplus_{c=1}^{C_j} (\vec{obs} \cdot \vec{M}_{jc})$$

where $\bigoplus$ is the logarithmic addition and is defined as $\ln(e^x + e^y)$. In this form, the computation of acoustic probabilities is perfectly suitable for a GPU since each distribution can be independently computed in parallel, and the results rest upon basic dot product operations.

3.0.1. CUDA development framework

We have implemented the acoustic computation module in CUDA, a development framework for NVidia graphic cards [9]. The CUDA framework shows the graphic card as a parallel coprocessor for the CPU. The development language is C with some extensions.

A program in the GPU is called a kernel and many of them can be concurrently launched. A kernel is made up of configurable amounts of blocks, each of which consists in a configurable amount of threads.

At execution time, each block is assigned to a multiprocessor. More than one block can be assigned to a given multiprocessor. Blocks are divided in groups of 32 threads called warps. In a given multiprocessor, 16 threads (half-warp) are executed at the same time. A time slicing-based scheduler switches between warps to maximize the use of available resources.

There are two kinds of memory. The first is the global memory which is accessible by all multiprocessors. Since this memory is not cached, it is important to ensure that the read/write memory accesses by a half-warp are coalesced in order to improve the performance. The texture memory is a small part of the global memory which is cached. The texture memory can be efficient when there is locality in data.

The second kind of memory is the shared memory which is internal to multiprocessors and is shared within a block. This memory, which is a lot faster than the global memory, can be seen as user-managed cache. This memory is divided into banks in such a way that successive 32-bit words are in successive banks. To be efficient, it is important to avoid conflicting accesses between threads. Conflicts are resolved by serializing accesses; this incurs a performance drop proportional to the number of serialized accesses.

3.0.2. Kernel for acoustic calculation

As described above, the likelihood of a given mixture is the logarithmic addition of dot-products for each component of the mixture. This operation can be implemented as a reduction algorithm[10] which uses the addition as reduction operator, except for the last C_j number of operations, for which the logarithmic addition is used to complete the reduction.

In our implementation, the computation of a mixture likelihood is computed by one-block of threads. Consequently, the number of launched blocks is the number of distributions in the acoustic model. Each block contains 256 threads.

For efficiency, the observation vector $\vec{obs}$ is copied C_j times. As a result, it is the same length as a distribution vector. There is thus a direct correspondence between its elements and those of $\vec{M}$, thus avoiding index calculations.

Moreover, to ensure efficiency of the reduction process and coalescing access to the global memory, the model vector $\vec{M}$ is reorganized at the distribution level. It's organized such that the C_j first elements are the constants,

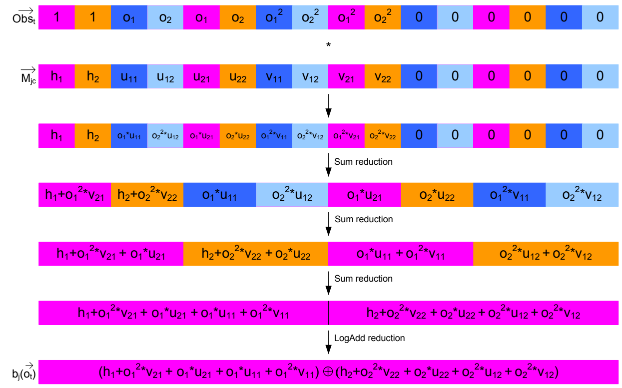

Fig. 2. *Reduction algorithm applied to the acoustic computation.*

followed by the $\mu_1\sigma_{11}^{-1}$ value of each component and so on. Figure 2 shows an example of the reduction algorithm applied in this context. In this figure, u_{xc} and v_{xc} denote the $\mu_x\sigma_{xx}^{-1}$ and $-\frac{1}{2}\sigma_{xx}$ values of component c.

Note that the observation vector has also been reorganized in the same way to ensure consistency.

Since the global memory latency is high, it is efficient to use data in shared memory as most as possible. In the case of likelihood computation, it is achieved by computing many frames at the same time. Thus, instead of computing likelihoods frame per frame, each block compute 7 frames for each distribution. The number 7 has been determined the most efficient after experimental results. This number could be different in other GPU since it depend of the amount of available shared memory.

4. EXPERIMENTS

4.1. Experimental Setup

The baseline system for comparison is a WFST-based speech recognition system developed at CRIM and tuned for speaker-independent transcription of broadcast news.

The acoustic model has been trained with 171 hours coming from French television programs in Quebec. The programs are a mix of weather, news, talk shows, etc. that have been transcribed manually. The acoustic parameters consist of 12 MFCCs plus the energy component, corresponding first and second derivatives, for a total of 39 features. The model contains 4600 distributions of 32 and 128 Gaussians with diagonal covariance matrices.

The language model has been trained with text from a French local newspaper (La Presse, 93 million words) and the acoustic training set's textual transcripts (2.1 million words). Both the unigram and trigram language models use the same vocabulary of 59624 words.

The CPU used is a Intel Core i7 quad at 2.9 GHz with 8 GB of RAM. Acoustic computations use the SSE registers. On the baseline version, required acoustic likelihoods are computed on-demand. This optimization is not possible with the A* algorithm since all likelihoods are used for computing the heuristic.

Algorithm	Computation time (seconds)	# of explored nodes	Accuracy
Viterbi	2069	2 459 801 548	68.67 %
A* (1 Thread)	6134	283 041 383	70.01%
A* (4 Threads)	2497	283 041 383	70.01%

Table 1. *Viterbi vs A* for real time settings.*

The GPU used is the NVidia GeForce GTX295. This card contains 2 GPU of 240 cores each. However, we have not been able to efficiently use the second GPU. Consquently, only one GPU has been used in all experimentations.

For all experiments involving the A* algorithm, the heuristic length Δ has been set to 500 frames. A* search is performed on $\Lambda = 20$ frames with a lookahead of 100 frames. Thus, for each block of heuristic scores, 20 A* searches are performed.

The test set is made up of 44 minutes (2625 seconds) of audio with a duration between 32 and 50 seconds.

4.2. A* Speech Recognition on Multi-Core Processors

We firstly experimented how a A* based speech recognition system perform on multi-core processors compared to the classical Viterbi approach.

4.2.1. Comparaison with the Classical Viterbi

Table 1 shows the performance of our A* decoder compared to the classical Viterbi decoder. The experiment has been done with 32 Gaussian component distributions.

The main advantage of the Viterbi decoder comes from the fact that it computes only 29% of all likelihoods since they are computed on-demand. This allows the Viterbi decoder to perform very well in real time as shown by the results.

In the case of the A* decoder, results show that the sequential implementation is slower than the Viterbi decoder. This is mainly due to the acoustic likelihood computation which accounts for 64% of the total time. Recall that all likelihoods must be computed since they are needed for the heuristic. The heuristic computation itself accounts for 27% of the total time.

However, the 4 thread version, with a speed-up of 2.46, achieves real-time. This performance could be improved if more threads were available.

Note that the number of explored nodes is about 8.7 times smaller in the A* decoder. This is the reason why the search itself account for only 7% of the total computation time.

Figure 3 shows results of a second experiment ran with a 128 Gaussian component acoustic model distribution. In this scenario, the real-time accuracy of the Viterbi decoder drops to around 65%, even if only 16% of acoustic likelihoods are computed, search time has to be limited by a low beam value.

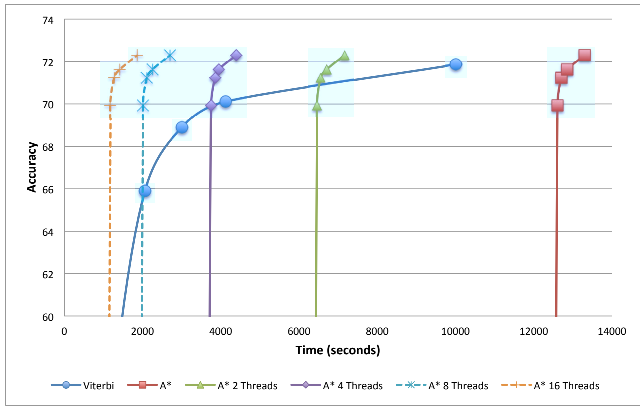

Fig. 3. *128 Gaussian components A* decoder accuracy vs execution time. Dashed lines are projections.*

The A* decoder cannot achieve real time with only 4 threads even with a speed-up of 3 times over its sequential counterpart. However, projection (represented by dashed lines) to 8 threads show that real-time can be reached with an accuracy of 71.62%. With 16 threads real-time accuracy would be 72.29%.

Note that for an accuracy of 72%, the A* decoder with 4 threads is 2.27 times faster than the Viterbi decoder. It would be 3.7 and 5.34 times faster with 8 and 16 threads.

4.2.2. Parallelization of Heuristic Computation

As described earlier, the heuristic computation operates in 2 steps: computation of acoustic likelihoods and computation of heuristic costs. These steps take more than 91% of the total search time. Table 2 shows how the computation time can be decreased by using multi-core architectures. Experiments have been conducted with 128 Gaussians acoustic models on the whole test set.

Step	Computation time 1 thread	4 threads	speed-up factor
Acoustic likelihoods	10659 sec	2913 sec	3.7x
Heuristic costs	1512 sec	495 sec	3.1x

Table 2. *Heuristic computation speed-up.*

The first line of Table 2 shows that computation of acoustic likelihoods parallelizes very well in a multi-core processor with a speed-up approaching the theorical maximum of N, where N is the number of cores.

Note that this parallelization could also be applied in the classical Viterbi decoder. However, the improvement will not be as significant since likelihoods are computed on-demand and only a subset of the distributions are used.

Note that epsilon transition expansions, which take approximatively 8.5% of the Viterbi computation time, are not parallelized.

We believe that the theorical maximum is not being reached on this part because of a misuse of the memory architecture, and that an optimisation of the data structures will enhance performances on multi-core processors.

4.3. GPU Performance

Table 3 shows how the acoustic computation time can be decreased by using a GPU. Experiments have been conducted with 128 Gaussians acoustic models on the whole test set.

Architecture	Time (seconds)	Speed-up
CPU 1 core	10659	–
CPU 4 cores	2913	3.7x
GPU 240 cores	964	11x

Table 3. *GPU for acoustic likelihoods computation.*

The two first lines recap the results shown in Table 2. The results show that the use of a GPU leads to a speed-up of 11x over a single CPU core using SSE registers. Note that time for transferring observation from computer memory to GPU global memory is included. Since acoustic model is transferred in the GPU memory only one time at the beginning, this time is not included.

4.4. Combining GPU with the Multi-Core Processor

Our main experiment consists in combining computation of acoustic likelihoods on the GPU and A* search on the multi-core processor. Figure 4 shows the results of this experiment with a 128 Gaussian component acoustic model.

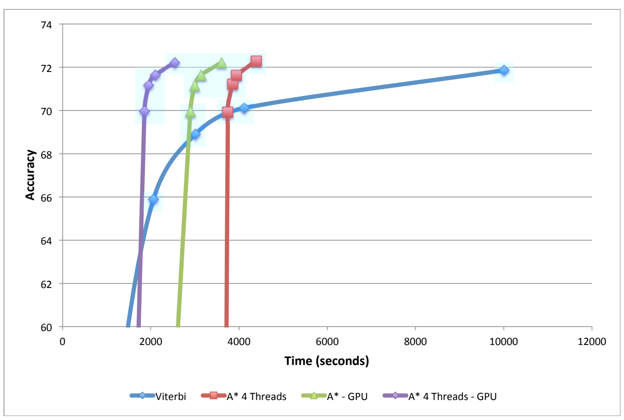

Fig. 4. *A* with GPU decoder accuracy vs execution time.*

In this scenario, the A* decoder now achieves real-time when a 4 core CPU is used. Combined use of the GPU led to a speed-up factor of 5.2 and improved the accuracy at real-time by approximatively 5% absolute over the classical Viterbi implementation.

5. CONCLUSION

This paper presented our current work on the parallelization of speech recognition systems using the A* algorithm. A WFST constructed from a unigram provides an admissible and efficient heuristic making the number of explored states by the A* algorithm 8.7 times smaller compared to the Viterbi algorithm in a real-time scenario. The A* search itself is less than 7% of the total computation time.

Results also show that using 4 cores in a multi-threaded implementation of the heuristic computation led to an overall speed-up factor of 3. The first and more time consuming step is the computation of acoustic likelihoods which parallelization reduced by a factor of 3.7. Computation of heuristic costs was only reduced by a factor of 3, but better reductions should be possible with better data structures.

Finally, using a GPU for acoustic computation in combination with the multi-threaded A* decoder led to an overall speed-up of 5.2. This speed-up allowed an improvement of the accuracy at real-time of 7% absolute.

6. REFERENCES

[1] P. Cardinal, P. Dumouchel, and G. Boulianne, “Parallel Architectures in Speech Recognition,” *In proceedings of Interspeech*, 2009.

[2] A.Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, “Challenges in Parallel Graph Processing,” *Parallel Processing Letters*, pp. 5–20, 2007.

[3] S. Phillips and A. Roggers, “Parallel speech recognition,” *International Journal of Parallel Programming*, 1999.

[4] N. Parihar, R. Schluter, D. Rybach, and E. A. Hansen, “Parallel Lexical-tree Based LVCSR on Multi-core Processors,” *In proceedings of Interspeech*, 2010.

[5] S. Ishikawa, K. Yamabana, R. Isotani, and A. Okumura, “Parallel LVCSR Algorithm for Cellphone-Oriented Multicore Processors,” in *The IEEE International Conference on Acoustics, Speech and Signal Processing*, 2006.

[6] J. Chong, E. Gonina, K. You, and K. Keutzer, “Exploring Recognition Network Representations for Efficient Speech Inference on Highly Parallel Platforms,” *In proceedings of Interspeech*, 2010.

[7] P. Kenny, R. Hollan, G. Boulianne, H. Garudadri, M. Lennig, and D. O’Shaugnessy, “An A* Algorithm for Very Large Locabulary Continuous Speech Recognition,” in *The IEEE International Conference on Acoustics, Speech and Signal Processing*, 1992, vol. 1.

[8] M. Mohri, F.C.N. Pereira, and M. Riley, “Weighted finite-state transducers in speech recognition,” in *Proceedings of the ISCA Tutorial and Research Workshop, Automatic Speech Recognition: Challenges for the new Millenium (ASR2000)*, 2000.

[9] CUDA, “http://www.nvidia.com/object/cuda_home.html,” .

[10] T. Sumanaweera and D. Liu, “Medical Image Reconstruction with the FFT,” in *GPU Gems 2*. 2005, Addison Wesley.