Beyond Gbps Turbo Decoder on Multi-Core CPUs

Adrien Cassagne∗†, Thibaud Tonnellier∗, Camille Leroux∗, Bertrand Le Gal∗, Olivier Aumage† and Denis Barthou†
∗IMS Lab, Bordeaux INP, France
†Inria / Labri, Univ. Bordeaux, INP, France

Abstract—This paper presents a high-throughput implementation of a portable software turbo decoder. The code is optimized for traditional multi-core CPUs (like x86) and it is based on the Enhanced max-log-MAP turbo decoding variant. The code follows the LTE-Advanced specification. The key of the high performance comes from an inter-frame SIMD strategy combined with a fixed-point representation. Our results show that proposed multi-core CPU implementation of turbo-decoders is a challenging alternative to GPU implementation in terms of throughput and energy efficiency. On a high-end processor, our software turbo-decoder exceeds 1 Gbps information throughput for all rate-1/3 LTE codes with \( K < 4096 \).

I. INTRODUCTION

Turbo codes [1] are widely used as the channel coding component in digital communication standards, such as the LTE wireless specification [2]. Dedicated hardware architectures of turbo-decoders are usually mapped on custom Silicon in order to reach high energy-efficiency and high-throughput [3]–[6]. In [6] a turbo-decoder ASIC is demonstrated to reach 1.01 Gbps while consuming 0.7 nJ per decoded bit \((K = 6144 and 6 iterations)\). However, dedicated hardware turbo-decoders lack flexibility. This will become especially true in the future 5G mobile networks where baseband processing will be virtualized and implemented on centralized cloud platforms [7], [8]. Software implementations will then offer flexibility and short development cycles to the channel decoder design, at the cost of lower throughputs and higher energy consumptions. In such a context, the channel coding functions must be analyzed and optimized to be efficiently mapped on general purpose processors [7], [8]. In the case of turbo coding, some work has been initiated to devise efficient software implementations of turbo decoders, mostly focusing on the LTE standard.

In [9]–[18], turbo decoders were implemented on GPU targets to benefit from their computing power in order to comply with the LTE required throughputs. This was made possible by exploiting the parallelism within the turbo decoding process \((intra-frame parallelism)\). An alternative is to process several codewords on distinct computation resources \((inter-frame parallelism)\). For instance, in [15], a throughput of 122.8 Mbps was reached for a code dimension \( K = 6144 \) and 6 decoding iterations on a GPU device. One should notice that this throughput is approximately one decade lower than the dedicated hardware implementation in [6] while the power consumption is one to two decades higher. Indeed, despite the large amount of parallelism in a GPU, it is not obvious to feed every processing unit with data and some non-negligible time is spent in memory accesses. This leads to an inefficient use of the hardware resources and to high energy consumption.

An intermediate solution between dedicated hardware and a GPU is the use of general purpose processors (GPP). Multi-core devices provide high performance computation capabilities while consuming noticeably less power. Thanks to a large set of processor cores it becomes possible to implement computation intensive applications such as channel decoding with a lower energy consumption in comparison to GPU targets. In [16], [19], [20], software turbo decoders are implemented on GPP targets with \( intra-frame \) parallelism, similar to hardware-oriented strategies. Unlike these works, we propose to investigate the exclusive use of \( inter-frame \) parallelism. A generic and portable software turbo decoder has been designed and ported on several GPP targets. Experimental results show that inter-frame parallelism allows a more efficient use of CPU resources and our software turbo-decoder outperforms existing implementations in terms of throughput and energy efficiency. Moreover, it exceeds 1 Gbps information throughput on a high-end CPU, making multi-core CPU a compelling alternative to GPU for channel decoding processing in cloud-based Random Access Network (RAN) [7], [8].

The remainder of this paper is organized as follows. Section II presents the turbo code decoding algorithm that was implemented in this work. Section III shows the benefit of inter-frame parallelism for multi-core implementations. Section IV details the optimized implementation of the turbo-decoder. Section V presents experiments and comparison with related works in the field.

II. OVERVIEW OF THE TURBO DECODING PROCESS

The turbo-decoding process is an iterative process in which two soft input soft output (SISO) decoders exchange extrinsic information. Each SISO decoder uses the channel information and \( a \ priori \) extrinsic information to compute \( a \ posteriori \) extrinsic information. The \( a \ posteriori \) information becomes the \( a \ priori \) information for the other SISO decoder and is exchanged via interleaver/deinterleaver.

In turbo-coding, the two component codes are convolutional codes, the associated decoding modules perform the BCJR or forward-backward algorithm [21] which is optimal for the maximum a posteriori (MAP) decoding of convolutional codes. In order to calculate the extrinsic information for a bit, a BCJR SISO decoder first computes the probability that a trellis transition occurred during the encoding process. The branch
metrics associated with states $s_i^k$ and $s_j^{k+1}$ are computed as:
\[
\gamma(s_i^k, s_j^{k+1}) = 0.5(L_{sys}^k + L_{a}^k)u^k + 0.5(L_{p}^{k+1}).
\]
(1)

Here, $L_{sys}^k$ and $L_a^k$ are the systematic channel LLR and the a-priori LLR for $k^{th}$ trellis section, respectively. In addition, the parity LLRs for the $k^{th}$ trellis step are $L_p^k = L_p^0$ for MAP decoder 0 and $L_p^k = L_p^1$ for MAP decoder 1. We do not need to evaluate the branch metric $\gamma(s_i^k, s_j^{k+1})$ for all 16 possible branches, as there are only four different branch metrics: $\gamma_0^k = 0.5(L_{sys}^k + L_b^k + L_p^k)$, $\gamma_{1}^k = 0.5(L_{sys}^k + L_b^k - L_p^k)$, $\gamma_{0}^k$, and $\gamma_{2}^k$. After that, the SISO decoder computes forward and backward recursions over the trellis representation of the convolutional code. In this work, we use the Enhanced max-log-MAP algorithm [22], [23]. For each state $j$ of section $k$ of the trellis, the forward ($\alpha$) and backward ($\beta$) metrics are computed as follows:
\[
\alpha_j^{k+1} = \max_{i \in F}\{\alpha_i^k + \gamma(s_i^k, s_j^{k+1})\}
\]
(2)
\[
\beta_j^k = \max_{i \in F}\{\beta_i^{k+1} + \gamma(s_i^k, s_j^{k+1})\}.
\]
(3)

Then, the extrinsic information for each bit at position $k$ is:
\[
L_e^k = \max_{\{s_k, s_{k+1}\} \in U}\{\alpha_k^k + \beta_j^k + \gamma(s_k^k, s_j^{k+1})\} - \max_{\{s_k, s_{k+1}\} \in U}\{-\alpha_k^k + \beta_j^k + \gamma(s_k^k, s_j^{k+1})\} - L_{sys}^k - L_a^k.
\]
(4)

Finally, $L_e^k$ is scaled by a fixed factor of 0.75.

III. PARALLELISM ANALYSIS

a) Intra-frame versus inter-frame parallelism: A Turbo decoder is in charge of decoding a large set of frames. Two strategies are then possible to speedup the decoding process. i) Intra-frame parallelism: the decoder exploits the parallelism within the turbo-decoding process by executing concurrent tasks during the decoding of one frame. ii) inter-frame parallelism: several frames are decoded simultaneously.

In the perspective of a hardware implementation, the intra-frame approach is efficient [24] because the area overhead resulting from parallelization is lower than the speedup. On the contrary, the inter-frame strategy is inefficient, due to the duplication of multiple hardware turbo-decoders. The resulting speedup comes at a high cost in term of area overhead.

In the perspective of a software implementation, the issue is different. The algorithm is executed on a programmable non-modifiable architecture. The degree of freedom lies in the mapping of the different parallelizable tasks on the parallel units of the processor. Modern multi-core processors support Single Program Multiple Data (SPMD) execution. Each core includes Single Instruction Multiple Data (SIMD) units. The objective is then to identify the parallelization strategy suitable for both SIMD and SPMD programming models. In the literature, intra-frame parallelism is often mapped on SIMD units while inter-frame parallelization is usually kept for multi-threaded approaches (SPMD). In [16], [20], multiple trellis-state computations are performed in parallel in the SIMD units. In [9]–[18], [20], the decoded frame is split into sub-blocks that are processed in parallel in the SIMD units. An alternative approach is to process both SISO decoding in parallel but it requires additional computations for synchronization and/or impacts on error-correction performance [24]. However, for all these approaches a part of the computation of the BCJR decoder remains sequential, bounding the speedup beyond the capabilities of SIMD units. Inter-frame parallelism has been proposed in [9], [10], [16], [20]. Multiple codewords are decoded in parallel, it improves the memory access regularity and the usage rate of SIMD units. The speedup is no longer bounded by the sequential parts, all removed, but this comes at the expense of an increase in memory footprint and latency.

In this work, we focus on the inter-frame parallelization and show that the use of this approach allows some register-reuse optimizations that are not possible in the intra-frame strategy.

b) Inter-frame parallelism on multi-core CPUs: The contribution of this work is to propose an efficient mapping of multiple frames on the CPU SIMD units (inter-frame strategy): the decoding of $M$ frames is vectorized. Before the decoding process can be launched, this new approach requires to: (a) buffer a set of $M$ frames and (b) reorder the input LLRs in order to make the SIMDization efficient with memory aligned transactions (see Fig. 1). Similarly, a reversed-reordering step has to be performed at the end of the decoding process. These reordering operations are expensive but they make the complete decoding process very regular and efficient for SIMD parallelization. Moreover, reordering is applied only once, independently of the number of decoding iterations.

Algorithm 1 Standard BCJR implementation

1: for all frames do
2: for $k = 0$; $k < K$; $k = k + 1$ do
3: $\gamma^k \leftarrow$ computeGamma($L_{sys}^k, L_p^k, L_e^k$) $\triangleright$ Sequential loop
4: $\alpha^0 \leftarrow$ initAlpha() $\triangleright$ Parallel loop
5: for $k = 1$; $k < K$; $k = k + 1$ do
6: $\alpha^k \leftarrow$ computeAlpha($\alpha^{k-1}, \gamma^{k-1}$) $\triangleright$ Sequential loop
7: $\beta^{K-1} \leftarrow$ initBeta() $\triangleright$ Parallel loop
8: for $k = K - 2$; $k \geq 0$; $k = k - 1$ do
9: $\beta^k \leftarrow$ computeBeta($\beta^{k+1}, \gamma^k$) $\triangleright$ Parallel loop
10: for $k = 0$; $k < K$; $k = k + 1$ do
11: $L_e^k \leftarrow$ computeExtrinsic($\alpha^k, \beta^k, \gamma^k$) $\triangleright$ Parallel loop
Algorithm 2: Loop fusion BCJR implementation

1: for all frames do  
2:   \( \gamma_0 \leftarrow \text{initAlpha}() \)  
3:   for \( k = 1; k < K; k = k + 1 \) do  
4:       \( \gamma^{k-1} \leftarrow \text{computeGamma}(\alpha^{k-1}, \gamma^{k-1}, \beta^{k-1}, L^{k-1}) \)  
5:       \( \alpha^k \leftarrow \text{computeAlpha}(\alpha^{k-1}, \gamma^{k-1}) \)  
6:       \( \beta^{K-1} \leftarrow \text{initBeta}() \)  
7:       \( L^{K-1} \leftarrow \text{computeExtrinsic}(\alpha^{K-1}, \beta^{K-1}, \gamma^{K-1}) \)  
8:   for \( k = K - 2; k \geq 0; k = k - 1 \) do  
9:       \( \beta^k \leftarrow \text{computeBeta}(\beta^{k+1}, \gamma^k) \)  
10:      \( \gamma^k \leftarrow \text{computeExtrinsic}(\alpha^k, \beta^k, \gamma^k) \)  
11:  \( \gamma^k \leftarrow \text{computeExtrinsic}(\alpha^k, \beta^k, \gamma^k) \)

In the proposed implementation, the inter-frame parallelism is used to fill the SIMD units of the CPU cores. Algorithm 1 illustrates the traditional implementation of the BCJR (used for the *intra-frame* vectorization). The inter-frame strategy makes the outer loop on the frame parallel (through vectors). This means all computations inside this loop operate on SIMD vectors instead of scalars, and the inner loops can be turned into sequential loops on SIMD vectors. This gives the opportunity for memory optimizations, through loop fusion. The initial 4 inner loops are merged into 2 loops. Algorithm 2 presents this loop fusion optimization. This makes possible the scalar promotion of \( \beta_j \) (no longer an array), since it can be directly reused from the CPU registers. In this version, the SIMD are always stressed.

On a multicore processor, each core decodes \( M \) frames using its own SIMD unit and \( T \) threads are activated, a total of \( M \times T \) frames are therefore decoded simultaneously with the inter-frame strategy. Theoretically, this SIMD parallelization strategy provides an acceleration up to a factor, \( T \), with \( T \) cores. Large memory footprint, exceeding L3 cache capacity may reduce the effective speedup, as shown in Section V.

IV. IMPLEMENTATION OF THE DECODER

The presented decoder implementation is available in the AFF3CT\(^1\) software [25]. The use of C++ templates associated to our generic SIMD library enables the same source code to be compiled using different formats (32-bit \texttt{float}, 16-bit \texttt{short}, and 8-bit \texttt{char}) and different SIMD instructions (SSE, AVX and NEON), providing possible trade-offs between SIMDization, throughput and error-correction performance.

\(\textit{a) Fixed-point representation:} \) Nowadays on x86 CPUs, there are large SIMD registers: SSE/NEON are 128 bits wide and AVX are 256 bits wide. The number of elements that can be vectorized depends on the SIMD length and on the data format: \( n_{\text{elem}} = \text{sizeof(SIMD)} / \text{sizeof(data)} \). Therefore, the key for a large parallelism is to work on short data.

As there is no floating-point support for 16-bit and 8-bit data, a fixed-point representation is used. The AWGN channel soft information is quantized as follows: \( y_{s,v}^k = \Psi(2^v \cdot y^k \pm 0.5) \), with \( y^k \) the current floating-point value from the channel, \( s \) the number of bit of the quantized number, including \( v \) bits for the fractional part and the saturation

\(1\)AFF3CT is an Open-source software (MIT license) for fast forward error correction simulations, see http://aff3ct.github.io
from registers in order to compute the next $\beta_j^{k-1}$ values. The $\beta_j^k$ metrics are then never stored in the memory.

V. EXPERIMENTS AND RESULTS

The experiments have been conducted on three different x86-based processors detailed in Table I. A mid-range processor (P2) is used for comparison with similar CPU targets in the literature [16], [19], [20] while the two high-end processors (P1 and P3) are used for comparison with GPU-based turbo-decoder implementations. Indeed, P1 and P3 have a number of cores that is similar to the number of Streaming Multiprocessors (SM) inside a GPU. Moreover, the code has been compiled on Linux (Ubuntu 14.04 LTS) with the GNU compiler (version 4.8) and with the -Ofast -funroll-loops -msse4.1/-mavx2 flags.

a) BER/FER performance: Fig. 2 shows the decoding performance of the proposed software turbo-decoder for the $K = 6144$ rate-1/3, LTE-specified turbo-code. The decoding performance of a floating-point decoder is provided as a reference. Unlike [16], the proposed 16-bit implementation does not degrade the decoding performance. The 8-bit version of our decoder shows a 0.15dB degradation. The limited precision of 8-bit format together with early saturation inside the decoder are responsible for this small performance loss.

b) Throughput performance: Fig. 3 shows the evolution of the information throughput depending on the code dimension $K$. This experiment was conducted on P2 and P3 (both have Haswell architectures). The throughput tends to increase linearly with the number of cores (up to 24 cores) except in AVX mode where a performance drop can be observed when $K > 4096$. The reason is that the AVX instructions use vectors $2 \times$ wider than those used by SSE instructions and the inter-frame strategy loads twice the number of frames to fill these vectors. Thus, for $K > 4096$, in AVX, the memory footprint exceeds the 3 cache optimal occupancy and the performance is driven by the RAM bandwidth. Then, as $K$ increases the number of RAM accesses increases and there is not enough memory bandwidth to feed all the cores. This explains the decreasing throughput for $K > 4096$, in AVX mode. Nonetheless, on P3 target, the throughput exceeds 1Gb/s for all codes with $K < 4096$.

Fig. 4 shows the energy consumed by the processor to decode one information bit ($E_d$) of the codes using SSE and AVX instructions, on the P2 CPU target. For small codewords ($K = 1024$) it is more energy efficient to resort to AVX. But this is not so clear on larger codewords ($K = 6144$) since with 3/4 cores, the code using SSE outperforms the AVX one.

Table II shows a performance comparison with related works. The variety of CPU/GPU targets and algorithmic parameters allows to show some global emerging trends. When comparing to similar CPU targets [16], [20], the proposed im-

\begin{table}[h]
\begin{center}
\caption{Specifications of the Target Processors.}
\begin{tabular}{|c|c|c|c|}
\hline
CPU & P1: Xeon E5-2650 & P2: Core i7-4960HQ & P3: Xeon E5-2680v3 \\
\hline Intel Arch. & Ivy Bridge Q1’12 & Haswell Q4’13 & Haswell Q2’14 \\
\hline Cores/Freq. & 8 cores, 2.8-3.6 GHz & 4 cores, 2.6-3.8 GHz & 12 cores, 2.5-3.3 GHz \\
\hline L2/L3 & 20MB L3 & 6MB L3 & 20MB L3 \\
\hline TDP & 95 W & 47 W & 120 W \\
\hline
\end{tabular}
\end{center}
\end{table}

\begin{figure}[h]
\begin{center}
\includegraphics[width=0.8\textwidth]{ber_fer}
\caption{Bit Error Rate (BER) and Frame Error Rate (FER) of the decoder for $K = 6144$ (6 iters) and $R = 1/3$. Enhanced max-log-MAP algorithm (scaling factor = 0.75). BPSK modulation and AWGN channel were used.}
\end{center}
\end{figure}

\begin{figure}[h]
\begin{center}
\includegraphics[width=0.8\textwidth]{throughput}
\caption{Information throughput depending on $K$ for various number of cores and SIMD instruction types. 6 iterations, 8-bit fixed-point.}
\end{center}
\end{figure}

\begin{figure}[h]
\begin{center}
\includegraphics[width=0.8\textwidth]{energy}
\caption{Energy-per-bit ($E_d$) depending on the number of cores and the instruction types. 6 iterations, 8-bit fixed-point. The throughput and power measurements were conducted on P2 with the Intel Power Gadget tool.}
\end{center}
\end{figure}
TABLE II

<table>
<thead>
<tr>
<th>Work</th>
<th>Year</th>
<th>Platform</th>
<th>Arch.</th>
<th>TDP Watts</th>
<th>Cores or SM</th>
<th>Freq. GHz</th>
<th>Algorithm</th>
<th>Prec. bit</th>
<th>SIMD length</th>
<th>Inter level</th>
<th>K</th>
<th>Iters</th>
<th>Decoding performances</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>2013</td>
<td>Tesla C2050</td>
<td>Fermi</td>
<td>240</td>
<td>12</td>
<td>1.50</td>
<td>ME-MAP</td>
<td>52</td>
<td>384</td>
<td>11644</td>
<td>5</td>
<td>10.05</td>
<td>1.097</td>
<td>1.000</td>
</tr>
<tr>
<td>[3]</td>
<td>2013</td>
<td>Tesla C2075</td>
<td>Fermi</td>
<td>240</td>
<td>14</td>
<td>1.35</td>
<td>EML-MAP</td>
<td>32</td>
<td>128</td>
<td>11644</td>
<td>5</td>
<td>10.05</td>
<td>1.089</td>
<td>1.914</td>
</tr>
<tr>
<td>[4]</td>
<td>2013</td>
<td>GTX 470</td>
<td>Fermi</td>
<td>240</td>
<td>16</td>
<td>1.42</td>
<td>ME-MAP</td>
<td>52</td>
<td>384</td>
<td>11644</td>
<td>5</td>
<td>10.05</td>
<td>1.084</td>
<td>1.458</td>
</tr>
<tr>
<td>[5]</td>
<td>2013</td>
<td>GTX 560 Ti</td>
<td>Fermi</td>
<td>116</td>
<td>0</td>
<td>1.80</td>
<td>EML-MAP</td>
<td>32</td>
<td>128</td>
<td>11644</td>
<td>6</td>
<td>10.04</td>
<td>2.012</td>
<td>1.682</td>
</tr>
<tr>
<td>[7]</td>
<td>2013</td>
<td>GTX 580</td>
<td>Kepler</td>
<td>192</td>
<td>8</td>
<td>0.70</td>
<td>EML-MAP</td>
<td>52</td>
<td>384</td>
<td>11644</td>
<td>6</td>
<td>10.05</td>
<td>1.075</td>
<td>1.914</td>
</tr>
<tr>
<td>[8]</td>
<td>2013</td>
<td>Tesla K20</td>
<td>Kepler</td>
<td>222</td>
<td>0</td>
<td>1.10</td>
<td>ME-MAP</td>
<td>52</td>
<td>384</td>
<td>11644</td>
<td>6</td>
<td>10.05</td>
<td>1.072</td>
<td>1.914</td>
</tr>
<tr>
<td>[9]</td>
<td>2013</td>
<td>GPU-AI K20</td>
<td>Kepler</td>
<td>222</td>
<td>0</td>
<td>1.10</td>
<td>ME-MAP</td>
<td>52</td>
<td>384</td>
<td>11644</td>
<td>6</td>
<td>10.05</td>
<td>1.072</td>
<td>1.914</td>
</tr>
</tbody>
</table>

VI. CONCLUSION

Future communication standards will make use of cloud-based RAN. In such a context, channel decoding processing will be mapped on programmable targets. In this work, we investigate the use of inter-frame parallelism for optimized software implementation of turbo decoders. Our results show that CPUs are competitive solutions in terms of throughput and energy consumption. The proposed software decoder exceeds 1 Gbps on a high-end CPU.

ACKNOWLEDGMENT

This work was supported by a grant overseen by the French National Research Agency (ANR), ANR-15-CE25-0006-01.

REFERENCES