# HSPA+/LTE-A Turbo Decoder on GPU and Multicore CPU

Michael Wu, Guohui Wang, Bei Yin, Christoph Studer, and Joseph R. Cavallaro

Dept. of Electrical and Computer Engineering, Rice University, Houston, TX
e-mail: {mbw2, gw2, by2, studer, cavallar}@rice.edu

***Abstract*—This paper compares two implementations of reconfigurable and high-throughput turbo decoders. The first implementation is optimized for an NVIDIA Kepler graphics processing unit (GPU), whereas the second implementation is for an Intel Ivy Bridge processor. Both implementations support max-log-MAP and log-MAP turbo decoding algorithms, various code rates, different interleaver types, and all block-lengths, as specified by HSPA+ and LTE-Advanced. In order to ensure a fair comparison between both implementations, we perform device-specific optimizations to improve the decoding throughput and error-rate performance. Our results show that the Intel Ivy Bridge processor implementation achieves up to 2× higher decoding throughput than our GPU implementation. In addition our CPU implementation requires roughly 4× fewer codewords to be processed in parallel to achieve its peak throughput.**

## I. Introduction

Turbo codes are capacity-approaching channel codes that can be decoded at high throughput and low power using dedicated hardware accelerators. Hence, turbo codes are used in a large number of cellular wireless standards, such as 3GPP HSPA+ [1] and LTE-Advanced [2]. Recently, a number of software-based wireless testbeds have been developed to demonstrate the feasibility of software-based real-time communication systems on general purpose processors [3]–[5]. Implementations on such architectures are attractive for multiple reasons. First, they are inherently flexible (with respect to code rates, block lengths, etc.) and capable of supporting multiple standards. Second, they use off-the-shelf components, without the need for dedicated hardware accelerator blocks.

Although turbo codes offer superior error-rate performance over convolutional codes, turbo decoding requires higher computational complexity [6]. To meet the throughput requirements of existing wireless standards, turbo decoding is typically carried out with specialized hardware accelerators, such as ASIC designs [7]–[9] or FPGA implementations [10]. As a consequence, SDR systems such as the ones in [3]–[5] rely on convolution codes instead of turbo codes to avoid the high complexity of channel decoding. The use of convolutional codes, however, results in inferior error-correction performance (compared to turbo codes). In addition, LTE-Advanced, specifies the use of turbo codes for both uplink and downlink. Hence, corresponding SDR receiver designs necessitate the development of software-based turbo decoder solutions that are capable of achieving high throughput.

This work was supported in part by Broadcom and by the US National Science Foundation under grants CNS-1265332, ECCS-1232274, EECS-0925942 and CNS-0923479. We would also like to thank Nvidia and Intel for their generous hardware donations.

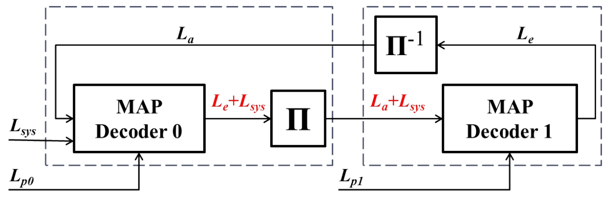


Figure 1. High-level structure of a rate-1/3 3GPP turbo decoder.

*Contributions:* In this paper, we evaluate the performance of turbo decoder implementations on two different high performance programmable processors, namely on a quad-core Intel i7-3770K (Ivy Bridge) and a Nvidia GeForce GTX 680 (Kepler GK104). We design two parallel turbo decoder implementations with a similar feature set. Both proposed implementations support HSPA+ and LTE-Advanced and take advantage of unique features of both platforms. In particular, we perform a variety of device specific optimizations, such as the use of linear-MAP approximation on the CPU and the use of shuffle instructions on the GPU, to maximize the throughput and/or to improve the error-rate performance. We conclude by comparing the throughput of both implementations.

## II. Overview of Turbo Decoding

The high-level structure of a rate-1/3 3GPP turbo decoder is shown in Figure 1. The turbo decoder consists of two concatenated component decoders exchanging soft information in terms of the log-likelihood ratio (LLR) for each transmitted information bit through an interleaver (denoted by $\prod$) and a deinterleaver (denote by $\prod^{-1}$). HSPA+ uses intra-row and inter-row permutations to generate the interleaver addresses [1], whereas LTE-Advanced (LTE-A) uses a quadratic permutation polynomial (QPP) interleaver [11]. The component decoders are the same for HSPA+ and LTE-A.

### A. Algorithm outline

Turbo decoding is carried out in multiple iterations (denoted by $I$) where each iteration consists of two component decoding phases. In each phase, a component decoder performs maximum a-posteriori (MAP) decoding using the BCJR algorithm [12], which generates so-called extrinsic LLRs given the LLRs obtained by the detector and a-priori LLRs obtained from the other component decoder. The BCJR algorithm consists of one forward and one backward traversal on a trellis, which is defined by the underlying code. Specifically, to decode a codeword of $N$ information bits, the BCJR algorithm performs the following steps: (i) In the forward traversal step,

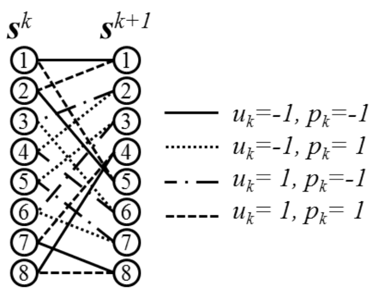


Figure 2. Structure of the 3GPP trellis. There are 8 states per trellis step and one step per transmitted information bit. The vector $\mathbf{s}^k$ consists of all state metrics at trellis step $k$. The values $u_k$ and $p_k$, are the $k^{\text{th}}$ information bit and the parity bit (both $\pm 1$) respectively.

it iteratively computes $N$ sets of forward state metrics for each transmitted information bit. (ii) In the backward traversal step, it iteratively computes $N$ sets of backward state metrics for each transmitted information bit. To compute the extrinsic LLRs, the BCJR algorithm then combines the forward and backward state metrics.

### *B. Branch-metric computation*

HSPA+ and LTE-Advanced both operate on a 8-state trellis, which is illustrated in Figure 2. Let $s_j^{k+1}$ be the $j^{\text{th}}$ state associated with information bit $k+1$. There are two incoming branches into state $s_j^{k+1}$. Each incoming branch is associated with values $u_k$ and $p_k$, the $k^{\text{th}}$ information bit and the parity bit (both $\pm 1$), respectively. The branch metrics associated with states $s_i^k$ and $s^{k+1}$ are computed as follows:

$$\gamma\left(s_i^k, s_j^{k+1}\right) = 0.5(L_{sys}^k + L_a^k)u_k + 0.5(L_p^k p_k).$$

Here, $L_{sys}^k$ and $L_a^k$ are the systematic channel LLR and the a-priori LLR for $k^{\text{th}}$ trellis step, respectively. In addition, the parity LLRs for the $k^{\text{th}}$ trellis step are $L_p^k = L_{p0}^k$ for MAP decoder 0 and $L_p^k = L_{p1}^k$ for MAP decoder 1. Note that we do not need to evaluate the branch metric $\gamma\left(s_k, s_{k+1}\right)$ for all 16 possible branches (see Figure 2), as there are only four different branch metrics: $\gamma_k^0 = 0.5(L_{sys}^k + L_a^k + L_p^k)$, $\gamma_k^1 = 0.5(L_{sys}^k + L_a^k - L_p^k)$, $-\gamma_k^0$, and -$\gamma_k^1$.

### *C. Forward and backward state metric computation*

The forward state metrics can be computed iteratively from trellis step to trellis step. The forward state metrics of step $k{+}1$ correspond to the vector $\boldsymbol{\alpha}^{k+1} = [\alpha_1^{k+1}, \ldots, \alpha_8^{k+1}]$, where the $j^{\text{th}}$ forward state metric $\alpha_j^{k+1}$ only depends on two forward state metrics of stage $k$. These state metrics are computed by

$$\alpha_j^{k+1} = \max\nolimits_{i \in F}^* \left\{\alpha_i^k + \gamma(s_i^k, s_j^{k+1})\right\} \qquad (1)$$

where the set $F$ contains the two indices of the states in step $k$ connected to state $s_j^{k+1}$ (as defined by the trellis). The $\max^*\{\cdot\}$ operator is defined as

$$\max{}^*\{a, b\} = \max\{a, b\} + \log\left(1 + \exp(-|a - b|)\right), \qquad (2)$$

where $\log\left(1 + \exp(-|a - b|)\right)$ is a correction term. For the max-log approximation, we approximate max* by $\max^*(a, b) \approx \max(a, b)$. In this case, one can scale the extrinsic LLRs by a factor of 0.7 to to partially recover the error-rate performance loss induced by the approximation (see, e.g., [8], [13] for additional details).

Computation of the backward state metrics is similar to that of the forward trellis traversal in (1). The vector of backward state metrics, denoted by $\boldsymbol{\beta}^k = [\beta_1^k, \ldots, \beta_8^k]$, is computed as

$$\beta_j^k = \max\nolimits_{i \in B}^* \left\{\beta_i^{k+1} + \gamma(s_j^k, s_i^{k+1})\right\}. \qquad (3)$$

Here, the set $B$ contains the indices of states in step $k+1$ connected to state $s_j^k$ as defined by the trellis.

### *D. LLR computation*

After the forward and backward iterations have been carried out, the extrinsic LLRs for the $k^{\text{th}}$ bit are computed as

$$\begin{aligned} L_e^k = &\max\nolimits_{\{s_k, s_{k+1}\} \in U^1}^* \left\{\alpha_i^k + \beta_j^{k+1} + \gamma\left(s_i^k, s_j^{k+1}\right)\right\} \\ &- \max\nolimits_{\{s_k, s_{k+1}\} \in U^{-1}}^* \left\{\alpha_i^k + \beta_j^{k+1} + \gamma\left(s_i^k, s_j^{k+1}\right)\right\} \\ &- L_{sys}^k - L_p^k, \end{aligned}$$

where the sets $U^1$ and $U^{-1}$ designate the set of states connected by paths where $u_k = 1$ and the set of states connected by paths where $u_k = -1$, respectively.

## III. Turbo Decoder Implementations

At a high level, both the Ivy-Bridge and Nvidia Kepler architectures can be viewed as multi-core SIMD processors. For the Intel CPU, we explicitly deploy SIMD instructions via Intel intrinsics to vectorize the MAP decoding algorithm. For the NVIDIA GPU, we used CUDA [14] to parallelize the workload. The CUDA compiler can easily vectorize the GPU computations, since in the CUDA programming model, threads execute the same set of computations, just on different input data.

To achieve high decoding throughput, the BCJR algorithm outlined in Section II needs to be vectorized. We deploy the vectorization scheme put forward in [15], which vectorizes the BCJR algorithm into SIMD operations on vectors with eight 8 bit elements to accelerate a UMTS turbo decoder on an Analog Devices DSP. In the following subsections, we compare and contrast our turbo decoder implementations for the Intel CPU and the NVIDIA GPU.

### *A. SIMD data types*

For the quad-core Intel Ivy-Bridge processor, each core can execute SIMD instructions, supporting operations on various vector data types. Most hardware implementations of turbo decoders carry out fixed point calculations and use 10 bit-to-12 bit precision to achieve an error-rate performance close to a floating point implementation [7]–[9]. To achieve high throughput on the CPU, while maintaining good error-rate performance, we used vectors with eight 16 bit integer elements.

The targeted Nvidia GTX 680 consists of 8 Next Generation SM (SMX) units, where each SMX unit is roughly equivalent to an Intel Ivy-Bridge core. An SMX unit can issue multiple 1024 bit SIMD instructions in parallel, where each instruction operates on vectors with 32 elements each having 32 bit. The architecture is optimized for single-precision floating-point operations (integer operations can be up to $6\times$ slower). As a result, we used single-precision floating point operations in our GPU implementation. Since the computations for turbo-decoding consists of operations on vectors with 8 elements, we also decode at least 4 codewords in parallel to ensure a full utilization of the $1024$ bit SIMD instruction.

### B. Memory allocation

Among all possible codes in HSPA+ and LTE-A, the longest code is the LTE codeword with $K = 6144$ information bits. The length of the rate-$1/3$ encoded data is $18444$ bit, which is the largest amount of memory required among all codewords.

For our CPU implementation, we store all data as $16$ bit values. The implementation requires $48$ KB for input/output LLRs, and $96$ KB for forward state metrics. Since the amount of L2 cache per core is $256$ KB, all data fits into the cache.

On the GPU, shared memory, a small amount of memory (48KB per SMX) managed using explicit load and store instructions, can be used to cache data locally. Unfortunately, we cannot store data in the shared memory. This is because we decode at least 4 codewords in parallel to ensure the full utilization of the 1024 bit SIMD instruction and requires at least $4\times$ the amount of storage, which outstrip the amount of available shared memory. Therefore, we store the input/output LLRs and forward state metrics in the device's memory, which has high access latency, reducing the throughput of the design.

### C. Multi-mode interleaver lookup table

To support HSPA+ and LTE-A, we need to support both interleaver types. Generating the HSPA interleaver addresses is rather complicated [1]. To achieve high throughput, we decided to generate lookup tables which contain all possible interleaved and deinterleaved memory addresses instead of computing the addresses on-the-fly. For the Intel architecture, we store the lookup table in memory and rely on the fact that the entries in the lookup table will be cached. For the Nvidia architecture, we explicitly copy the correct entries of the lookup table at the start of the decoding iteration into constant memory, a small amount of read-only cache available on the GPU; this enables efficient lookup table accesses.

### D. Max$^*$ operation

For the max-log approximation, we simply omit the correction term of max$^*$ operator, as defined in (2), and approximate max$^*$ as a simple max operation followed by scaling the extrinsic LLRs by a factor of 0.7. For both the CPU and GPU implementations, the max-log approximation corresponds to a vertical (element-wise) max instruction.

Since the GPU supports element-wise logarithm and exponential functions, we can implement the log-MAP algorithm directly. Overall, the log-MAP algorithm requires one vertical max instruction, one vector subtraction, one vector absolute value, one call to the element-wise log function, one call to element-wise exponential function, and one vector addition.

The Intel architecture does not support scalar or vector fixed-point logarithm or exponential functions. We therefore approximate the correction term $c(a,b) = \log\left(1+\exp(-|a-b|)\right)$ with a piece-wise linear function as $c(a,b) = \max\{0.25(2.77-|a-b|),0\}$ [16]. We then add this correction term to $\max\{a,b\}$. This approximation requires 6 additional instructions compared to the max-log approximation: one vector subtraction, one vector absolute value, one vector shift, one vector maximum, and one vector addition.

### E. Branch-metric computation

In order to compute all branch metrics for every state $k$, we need to compute four branch metrics, $\gamma_k^0$ and $\gamma_k^1$ and the negated versions, $-\gamma_k^0$ and $-\gamma_k^1$ (see Section II-B), which requires scalar operations only.

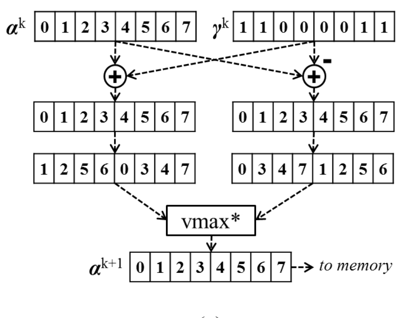
To parallelize the workload on the CPU and GPU, we fetch 8 consecutive systematic channel LLRs, 8 consecutive parity LLRs and 8 consecutive a priori LLRs at a time. We then compute the branch metrics, $\{\gamma_{k+1}^1,\ldots,\gamma_{k+8}^1\}$ and $\{\gamma_{k+1}^0,\ldots,\gamma_{k+8}^0\}$, in parallel. Finally, we compute the negated versions. In total, the branch metric computation requires two vector additions and three vector subtractions.

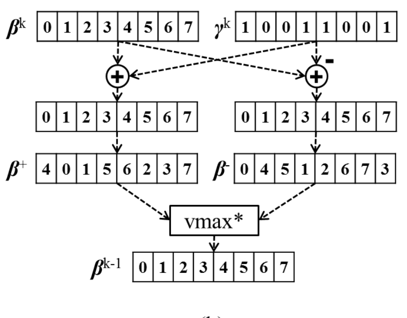
### F. Forward and backward traversal

Figures 3(a) and 3(b) depict the vectorized implementation of the forward and backward state-metric computation units in (1) and (3), respectively. Compared to the implementation in [15], we rearranged the placement of shuffles (data exchange between SIMD lanes) to increase the instruction-level parallelism (ILP). This approach does not increase the number of required shuffle operations and is beneficial for the Intel architecture, since multiple shuffles can be executed in parallel. Figure 3(c) depicts the computations used to generate the extrinsic LLRs, where $\boldsymbol{\beta}^+$ and $\boldsymbol{\beta}^-$ are intermediate values computed while computing $\boldsymbol{\beta}^k$ (see Figure 3(b)).

On the Intel architecture, the $\boldsymbol{\alpha}^k$ and $\boldsymbol{\beta}^k$ computations can be implemented very efficiently using intrinsics. The vector $\boldsymbol{\gamma}^k$ is constructed using one shuffle instruction. The rest of the $\boldsymbol{\alpha}^k$ and $\boldsymbol{\beta}^k$ computation consists of two vector additions, two $128$ bit shuffles and two element-wise max$^*$ operations. Since we use $16$ bit for the forward state metrics, these metrics can overflow. Hence, we re-normalize the metrics by subtracting $\alpha^k(1)$ from $\boldsymbol{\alpha}^k$ [9]. To reduce the number of instructions required by this re-normalization step, we normalize the state metric only every 8 trellis steps. As a result, the overhead of renormalization is low, requiring three additional instructions (extract the first element, broadcast, and vector subtract) every 8 trellis steps. The same normalization scheme is used during the backward state metric computation phase.

In our previous work [17], we emulated shuffle instructions with shared memory load and store instructions on the Nvidia architecture. One new feature of Kepler is that it explicitly supports shuffle instructions. We therefore replaced the shared memory operations with shuffles. As a result, the number of instructions for $\boldsymbol{\alpha}^k$ and $\boldsymbol{\beta}^k$ computation is similar to that of the CPU implementation. Since all computations are carried out in floating point arithmetic, the forward and backward state metrics do not need to be normalized.


(a)


(b)

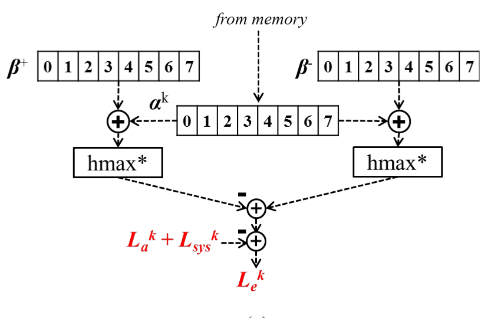

(c)

Figure 3. (a) Vectorized computation of $\boldsymbol{\alpha}^{k+1}$ for the 3GPP turbo code. The block vmax* implements the vectorized element-wise max* operator. (b) Vectorized computation of $\boldsymbol{\beta}^{k-1}$ for the 3GPP turbo code. (c) Vectorized LLR computation for the 3GPP turbo code. The block hmax* reduces 8 elements in the input vector to one element using the max* operator.

## G. LLR computation

As shown in Figure 3(c), the LLR computation consists of two vector addition instructions, two hmax* functions, and a few scalar operations. The function hmax* reduces the elements to one element using the max* operator; this can be accomplished using a tree reduction, which requires 3 shuffle instructions and 3 max* operations on the CPU.

Unfortunately, a tree reduction does not use all lanes of the SIMD instruction. To increase computational efficiency, we buffer the inputs to hmax* for 8 consecutive stages in shared memory column wise, instead of evaluating hmax* one stage at a time. We then apply the element-wise max* operation row-wise to find the minimum in each column.

## H. Multi-codeword decoding

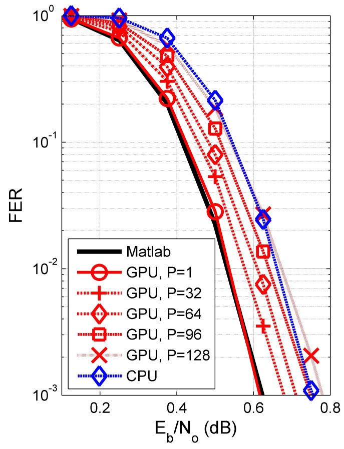
Transmitted frame for both LTE-A and HSPA+ consists of multiple codewords. Since both CPU and GPU are multi-core processors, we parallelize the workload across all available cores to maximize the peak decoding throughput.

For the CPU, such a parallelization is straightforward. We assigned at least one codeword per core using OpenMP to maximize the core utilization and, hence, the throughput.

The GPU heavily relies on multi-threading to hide pipeline stalls and memory-access stalls. A corresponding implementation requires a large number of independent sets of instructions [14]. As a result, we assigned a large number of codewords to each SMX using CUDA. To reduce the number of codewords needed to achieve high throughput on the GPU, we employed a windowed decoding approach, which divides a codeword into $P$ sections (or windows) and processes these sections in parallel [17]. Since the forward and backward state metrics are computed from the first trellis stage to the last stage (in a continuous fashion), we exchange the forward and backward state metrics among different windows between the iterations. This approach reduces the performance loss associated with parallel processing. Nevertheless, there is a tradeoff between the number of windows and the decoding latency, which we will discuss in the following section.

## IV. Implementation Results

We now show our implementation results on an Intel Core i7-3770K, a quad core Ivy Bridge processor, and a Nvidia GTX 680, a Kepler GK104 GPU. In our experiments, we used the Intel C++ Compiler 14.0 and the CUDA 5.5 toolkit. In the following, we first compare the error-rate performance in terms of the frame error rate (FER) with a floating point reference decoder [8], and then, we compare the throughput of our optimized turbo decoder implementations.


(a)

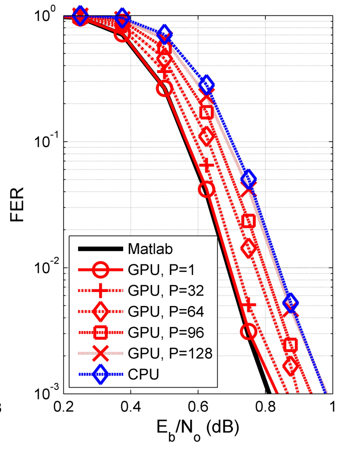

(b)

Figure 4. FER performance of (a) log-MAP decoding and (b) max-log-MAP decoding for $K = 6144$ and 6 decoding iterations.

## A. FER performance

We evaluate the FER in an additive white Gaussian noise (AWGN) channel. For the following experiments, we used the longest LTE code-word, which consists of $K = 6144$ information bits, and a code rate of $1/3$. Since the GPU turbo decoder supports windowed decoding, we consider a different number of windows (all of equal size), where the number of windows is $P \in \{1, 32, 64, 96, 128\}$.

Figure 4(a) compares the FER of the CPU decoder using the linear approximation detailed in Section II-B and the GPU decoder using log-MAP decoding. Figure 4(b) compares the FER of all turbo decoders using the max-log-MAP algorithm.

As expected, the use of 16 bit precision for the CPU implementation leads to a small loss ($\sim$0.14 dB for the max-log-MAP case) in terms of FER. The linear approximation results in only a very small FER degradation ($\sim$0.12 dB). For the GPU implementation, the FER performance with $P = 1$ matches the

Table I
PEAK THROUGHPUT FOR THE $K = 6144$, RATE-1/3 LTE TURBO CODE.

| | CPU (Mbps) | | GPU (Mbps) | |
|---|---|---|---|---|
| $I$ | max-log-MAP | linear log-MAP | max-log-MAP | log-MAP |
| 4 | 122.6 | 62.3 | 54.2 | 55.2 |
| 6 | 76.2 | 40.0 | 37.0 | 35.5 |
| 8 | 55.6 | 30.8 | 30.0 | 27.8 |
| 10 | 46.9 | 25.1 | 24.9 | 23.7 |

reference (floating-point) implementation; for larger window sizes $P$, the FER performance degrades only slightly. We note that the same behavior applies to the HSPA+ codes, which are not shown here for the sake of brevity.

## B. Peak decoding throughput

Table I shows the peak throughput of our CPU and GPU implementations. The results for HSPA+ are similar; for log-MAP algorithm and 6 decoding iterations, we achieved 41.6 Mbps and 36.7 Mbps on CPU and GPU respectively.

As expected, the throughput for both implementation is inversely proportional to the number of decoding iterations $I$. The CPU implementation appears to be instruction bounded as additional instructions increase the runtime proportionally. The number of instructions required to compute a forward state metric is 7 using the max-log approximation, whereas the use of the linear log-MAP approximation requires 6 additional instructions. As a result, the number of instructions required for linear log-MAP is $2\times$ of max-log-MAP. For 6 decoding iterations, the throughput of the linear log-MAP approximation is 40 Mbps. The throughput of the linear log-MAP approximation is approximately $2\times$ lower than the throughput of max-log-MAP implementation, which is 76.2 Mbps.

For the GPU implementation, we found the instructions per cycle (IPC) is low, typically $\sim$1. Using the Nvidia profiler, we found that instructions operands are often unavailable (due to memory latency), leading to a low execution unit utilization. As a result, additional instructions (that do not require additional memory access) can execute on the idling execution units and thus do not significantly degrade the throughput. Hence, the throughput of log-MAP is not significantly slower than that of the max-log-MAP algorithm on the GPU.

For the CPU implementation, a workload of 8 parallel codewords is sufficient to reach the peak throughput. For the GPU implementation, significantly more codewords need to be processed in parallel. Given a codeword, increasing the number of windows, $P$, does not increase peak throughput as the number of computation and memory transactions required to process the codeword stays the same. Increasing $P$, however, is an effective way of reducing the number of codewords required to reach peak throughput. This trend is similar to our previous implementation [17]. On the Kepler architecture, we require 512 codewords for $P = 1$, 16 codewords for $P = 32$ and above, to reach the peak performance.

In summary, our GPU implementation is up to $2\times$ slower than that of our CPU implementation for the max-log approximation, and only $1.2\times$ slower for the optimal log-MAP algorithm. For the Nvidia Kepler architecture, the maximum IPC is 6-to-7, while the achieved IPC is $\sim$1. The achieved IPC is low as operands of the instructions are often not ready due to memory access latency, leading to low execution unit utilization and low throughput. Coupled by the fact that CPU is clocked much faster than the GPU, the CPU implementation was able to outperform the GPU implementation.

We emphasize that it is possible to further improve the decoding throughput on the GPU. For example, we can reduce the number of memory accesses via data compression, which reduces the time for which execution units wait for operands. Nevertheless, the CPU implementation seems to be better suited for SDR applications since we achieve higher throughput with $2\times$ fewer number of parallel codewords, which significantly reduces the latency of the entire system.

## REFERENCES

[1] A. Vosoughi, G. Wang, H. Shen, J. R. Cavallaro, and Y. Guo, "Highly scalable on-the-fly interleaved address generation for UMTS/HSPA+ parallel turbo decoder," in *IEEE ASAP*, June 2013, pp. 356–362.

[2] The 3rd Generation Partnership Project (3GPP), "Evolved universal terrestrial radio access (E-UTRA); multiplexing and channel coding, Tech. Spec. 36.212 Release-11," 2012.

[3] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker, "Sora: High performance software radio using general purpose multi-core processors," in *USENIX NSDI*, Apr. 2009, pp. 75–90.

[4] C. R. Berger, V. Arbatov, Y. Voronenko, F. Franchetti, and M. Puschel, "Real-time software implementation of an IEEE 802.11 a baseband receiver on Intel multicore," in *IEEE ICASSP*, May 2011, pp. 1693–1696.

[5] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker, "Soft-LTE: A software radio implementation of 3GPP Long Term Evolution based on Sora platform," in *Demo in ACM MobiCom 2009*, Sept. 2009.

[6] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo-codes," *IEEE Trans. on Commun.*, vol. 44, pp. 1261 – 1271, Oct. 1996.

[7] C. Benkeser, A. Burg, T. Cupaiuolo, and Q. Huang, "Design and optimization of an HSDPA turbo decoder ASIC," *IEEE JSSC*, vol. 44, no. 1, pp. 98–106, Jan. 2009.

[8] C. Studer, C. Benkeser, S. Belfanti, and Q. Huang, "Design and implementation of a parallel turbo-decoder ASIC for 3GPP-LTE," *IEEE JSSC*, vol. 46, no. 1, pp. 8–17, Jan. 2011.

[9] Y. Sun and J. R. Cavallaro, "Efficient hardware implementation of a highly-parallel 3GPP LTE/LTE-advance turbo decoder," *INTEGRATION, the VLSI journal*, vol. 44, no. 4, pp. 305–315, Sept. 2011.

[10] *Xilinx Corporation, 3GPP LTE turbo decoder v2.0*, 2008. [Online]. Available: http://www.xilinx.com/products/ipcenter/DO-DI-TCCDEC-LTE.htm

[11] A. Nimbalker and Y. W. Blankenship and B. K. Classon and K. T. Blankenship, "ARP and QPP interleavers for LTE turbo coding," in *IEEE WCNC*, Apr. 2008, pp. 1032–1037.

[12] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. on Inf. Theory*, vol. IT-20, pp. 284–287, Mar. 1974.

[13] J. Vogt and A. Finger, "Improving the max-log-MAP turbo decoder," *IET Electron. Letters*, vol. 36, no. 23, pp. 1937–1939, 2000.

[14] *NVIDIA Corporation, CUDA Compute Unified Device Architecture Programming Guide*, 2008. [Online]. Available: http://www.nvidia.com/object/cuda_develop.html

[15] K. Loo, T. Alukaidey, and S. Jimaa, "High Performance Parallelised 3GPP Turbo Decoder," in *IEEE Personal Mobile Commun. Conf.*, Apr. 2003, pp. 337–342.

[16] J.-F. Cheng and T. Ottosson, "Linearly approximated log-MAP algorithms for turbo decoding," in *IEEE VTC Spring*, vol. 3, May 2000, pp. 2252–2256.

[17] M. Wu, Y. Sun, G. Wang, and J. R. Cavallaro, "Implementation of a high throughput 3GPP turbo decoder on GPU," *J. of Signal Process. Syst.*, vol. 65, no. 2, pp. 171–183, Nov. 2011.