TAMPERE UNIVERSITY OF TECHNOLOGY

Author(s)	Xianjun, Jiao; Canfeng, Chen; Jääskeläinen, Pekka; Guzma Vladimir; Berg, Heikki
Title	A 122Mb/s Turbo decoder using a mid-range GPU
Citation	Xianjun, Jiao; Canfeng, Chen; Jääskeläinen, Pekka; Guzma, Vladimir; Berg, Heikki 2013. A 122Mb/s Turbo decoder using a mid-range GPU. The 9th IEEE International Wireless Communications and Mobile Computing Conference (IWCMC 2013) July 1-5, 2013 1090-1094.
Year	2013
DOI	http://dx.doi.org/10.1109/IWCMC.2013.6583709
Version	Post-print
URN	http://URN.fi/URN:NBN:fi:tty-201309201352
Copyright	© 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

All material supplied via TUT DPub is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorized user.

A 122Mb/s Turbo Decoder using a Mid-range GPU

Jiao Xianjun, Chen Canfeng
Nokia Research Center
Beijing, China
Email: ryan.jiao@nokia.com

Pekka Jääskeläinen,
Vladimír Guzma
Tampere University of Technology, Finland
Email: pekka.jaaskelainen@tut.fi

Heikki Berg
Nokia Research Center
Tampere, Finland
Email: heikki.berg@nokia.com

***Abstract*—Parallel implementations of Turbo decoding has been studied extensively. Traditionally, the number of parallel sub-decoders is limited to maintain acceptable code block error rate performance loss caused by the edge effect of code block division. In addition, the sub-decoders require synchronization to exchange information in the iterative process. In this paper, we propose loosening the synchronization between the sub-decoders to achieve higher utilization of parallel processor resources. Our method allows high degree of parallel processor utilization in decoding of a single code block providing a scalable software-based implementation.**

The proposed implementation is demonstrated using a graphics processing unit. We achieve 122.8Mbps decoding throughput using a medium range GPU, the Nvidia GTX480. This is, to the best of our knowledge, the fastest Turbo decoding throughput achieved with a GPU-based implementation.

Keywords*—*Turbo decoder, Loose synchronization, Massively parallel computation, GPGPU, OpenCL

I. INTRODUCTION

Since its invention in 1993, Turbo coding has been a hot research topic both in academy and industry. Parallelization of the computationally intensive Turbo decoder [1] has been studied extensively. In addition to hardware-based Turbo decoders, software-based Turbo decoders have been implemented using *General Purpose Processors (GPP)* [2], *Digital Signal Processors (DSPs)* [3], [4], [5] and stream processors [6].

General Purpose computing capable *Graphics Processor Units (GPGPU)* [7] often consist of a large number of parallel processing elements with reduced dynamic scheduling support. Thus, the key challenge in utilizing the massive amount of parallel processing elements is to get the problem expressed in high enough number of parallel threads. Compared to DSP and FPGA, GPGPU has higher performance price ratio because of larger commercial market, thus performing computational intensive tasks in communication area by GPGPU are attracting more and more researchers.

There are several prior works on GPGPU turbo decoder implementations [8], [9], [10], [11]. They all have restricted the number of parallel *sub-decoders (SD)*/threads to mitigate the "edge effect" caused by the *Code Block (CB)* division, and, therefore, cannot effectively utilize all the processing elements available in GPUs. The highest reported single CB decoding throughput has been 7.97Mbps on Nvidia C1060 with 6 iterations [10]. By utilizing the Nvidia GTX470 GPU with 2048 simultaneously decoded CBs, the paper [11] presents a decoder with 25Mbps throughput using 6 iterations. However, in addition to the average throughput, also the decoding latency of a single CB has strict requirements. While the 25Mbps throughput is quite good, the increased latency in decoding of a single CB reduces its practicality.

In this paper, we demonstrate that the number of parallel SDs can be as high as the number of bits in a CB, while *Block Error Rate (BLER)* performance is maintained acceptable by adding more iterations. We also demonstrate that these SDs do not need to be synchronized strictly, and find out the asynchronous range of SDs by simulation. By loosening the synchronization requirements across the SDs, we can utilize all the processing elements of the GPU for single CB decoding. The resulting Turbo decoder can reach up to 122 Mbps throughput using a mid-range GPU.

II. OPENCL PROGRAMMING MODEL

Open Computing Language (OpenCL) [12] is a standard for heterogeneous parallel programming that is nowadays common in parallel programming, especially when targeting GPUs. OpenCL is used for defining the program of the proposed implementation.

In OpenCL a parallel program is split to *kernels*. Typically, multiple instances of one kernel are executed in parallel. The parallel instances are called *Work-Items (WI)*. The OpenCL programmer groups WIs into a number of *Work-Groups (WG)* while keeping in mind that only the WIs that belong to the same WG can synchronize with each other using the explicit synchronization primitives. WIs belonging to same WG access a *local* shared memory, and its consistency is guaranteed only at the points of the barrier or memory fence calls. The standard does not define synchronization between different WGs. That is, while WIs in different WGs can access a common shared *global* memory, global memory data consistency is guaranteed only after the kernel execution ends. [13]

In this work we use the Nvidia Fermi [14] GPGPU architecture as the experiment platform. Fermi consists of a number of *Streaming Multiprocessors (SM)* and each SM has a number of scalar cores. OpenCL WIs execute in the scalar cores. A WG runs in one SM to completion without switching to other SMs. One SM can execute multiple WGs in serial or parallel fashion depending on the number of hardware contexts available.

III. LOOSELY SYNCHRONIZED PARALLEL TURBO DECODING

A. Parallel Turbo Decoding

The principle of the Turbo encoder is depicted in Fig. 1. Picking the LTE Turbo code [15] as an example, $encoder1$ and

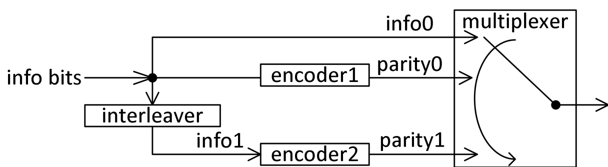

Fig. 1. Turbo encoder.

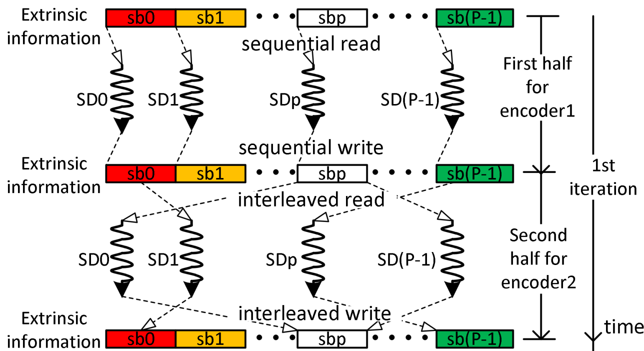

Fig. 2. Parallel turbo decoding.

$encoder2$ are the same *Recursive Systematic Convolutional (RSC)* encoders. If M information bits are fed into turbo encoder, M parity bits from $encoder1$ ($parity0$), M parity bits from $encoder2$ ($parity1$) and M systematic information bits ($info0$) will be sent out from encoder. At the receiver side, turbo decoder receives noise corrupted $info0$, $parity0$ and $parity1$. Noise corrupted $info1$ is acquired by interleaving $info0$ in the receiver.

A parallel turbo decoding process of [1] is depicted in Fig. 2. Assume that P SDs are used, so $info0,1$, $parity0,1$ are divided into P sub-blocks: $sb0$, $sb1$, ..., $sb(P-1)$ and are fed to P SDs. The decoding process is composed of several iterations. There first half of the iteration does trellis transversal for $encoder1$, second half for $encoder2$. Thus, $info0$ and $parity0$ are fed to first half and $info1$ and $parity1$ are fed to the second half.

Extrinsic information reflects the additional information over a known channel. This extrinsic information is refined throughout the iterative process by leveraging the known encoding polynomials and interleaving relationship. In the first iteration half, each extrinsic information sub-block is read sequentially by each SD and written sequentially after trellis transversal. In the second iteration half, each SD has to read corresponding extrinsic bit from interleaved address and write to the same address, due to the interleaver at the encoder side. After all iterations, transmitted bits are recovered by doing a hard decision on converged extrinsic information plus known channel information.

When doing a trellis transversal in the decoding process, each SD saves end states to the initializing start states of neighbor SD in next iteration. If the sub-block length is too short, the state metrics do not converge after trellis transversal. This "edge effect" leads to reduced BLER performance. Generally, at least 96 bits should be assigned to each SD to maintain an acceptable BLER, thus, the number of SDs cannot be arbitrary large. [1]

P	8	16	32	64	96	128	192	256
ITERs	6	6	6	6	6	7	7	8
P	384	512	768	1024	1536	2048	3072	6144
ITERs	9	10	12	15	20	26	35	65

TABLE I. NUMBER OF ITERATIONS (ITERS) NEEDED WHEN THE NUMBER OF SDS (P) GROWS.

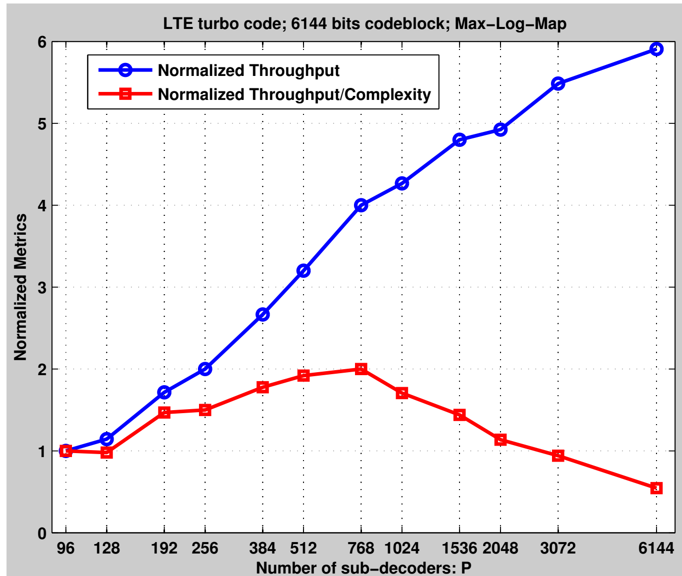

Fig. 3. Normalized speedup and efficiency versus number of SDs

In principle, all SDs need to synchronize with each other to exchange the latest extrinsic information between iteration halfs, and the number of parallel SDs is limited by the edge effect. These factors reduce the maximum utilized parallel SDs for an GPU-based implementation.

B. Effects of sub-decoder count

Thanks to the nature of the iterative decoding process, more iterations can be used to reduce the edge effect. Table I gives the number of iterations needed to get a BLER performance loss less than $0.2dB$ under different number of SDs for a 6144 bits LTE turbo CB with the Max-Log-Map algorithm [16].

In the extreme case, we can have 6144 SDs, each SD processing only 1 bit. Assuming that single iteration processing time of each SD is proportional to number of bits assigned, i.e. $6144/P$, overall decoding time will be proportional to $ITERs \times 6144/P$ (ITERs is the number of iterations). Single CB decoding throughput will be proportional to inverse of decoding time: $P/(ITERs \times 6144)$. The overall computational complexity is proportional to ITERs, because single iteration complexity is only related to CB length regardless of P. Fig. 3 shows the normalized throughput and 'throughput over complexity' versus P, where normalization is done for the case of $P = 96$, $ITERs = 6$. From Fig. 3, throughput increases monotonously with P, while efficiency (throughput/complexity) decreases after $P = 768$. This is caused by the number of necessary iterations growing faster when P is larger than 768, as shown in Table I.

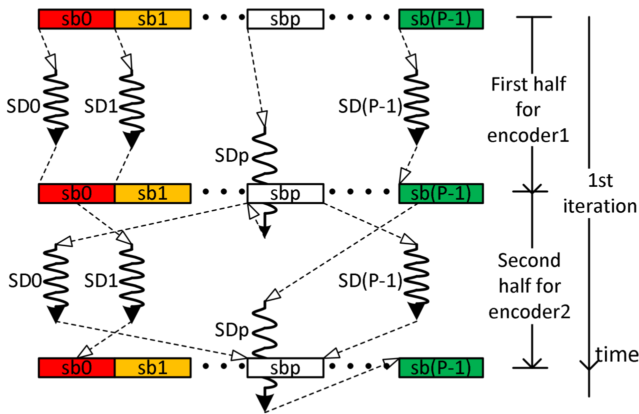

Fig. 4. Asynchronous parallel turbo decoding.

C. Loosely synchronized parallel turbo decoding

In the OpenCL programming model, all WIs that require synchronization have to belong to the same WG, which can typically utilize only one processor core (a SM in the Fermi), regardless of how many cores there are available. Thus, increasing the number of WIs in a single WG doesn't help after the scalar cores (or processing elements) of the single processor core are fully utilized. However, if multiple WIs used for decoding a single CB could be distributed into multiple WGs, and thus executed in multiple SMs in parallel, the resources of multiple SMs could be used for improving the decoding performance.

Fig. 4 depicts asynchronous parallel turbo decoding, where multiple SDs used for decoding single CB are grouped into many WGs. In Fig. 4, SDp does not get its extrinsic values written into the global memory in time, thus $SD0$ in the second half, and $SD(P-1)$ in the first half of next iteration use the old values. Because of the iterative nature of turbo decoding as well as the information passed (essentially bit probabilities), this kind of relaxation can be allowed. The old values are still bit probabilities, albeit less accurate ones, as they are produced in the previous iterations.

In order for the loose synchronization based implementation to work, the implementation platform has to fulfill two assumptions: 1) All the global memory updates are made visible to all WGs *eventually* during the kernel execution, and 2) There is mutual progress along the WGs participating in the decoding. That is, all SDs across all WGs experience progress, although not strictly synchronized.

In order to see the effect of receiving the results from SDs in another WG late, assume that each WG may be late N steps after the starting point at probability Pa. Fig. 5 shows a simulation result on the tolerance of "step lateness". Tolerance means that if delay steps of information is larger than the value in Y axis, BLER performance loss will be more than $0.2dB$ compared with the case of $Pa = 0$ and $steps = 0$. Labels CL, SD, WG and $ITER$ mean the length of the CB, total number of SDs, number of WGs, and number of iterations, respectively. By adding more iterations the decoder tolerates more asynchronous steps.

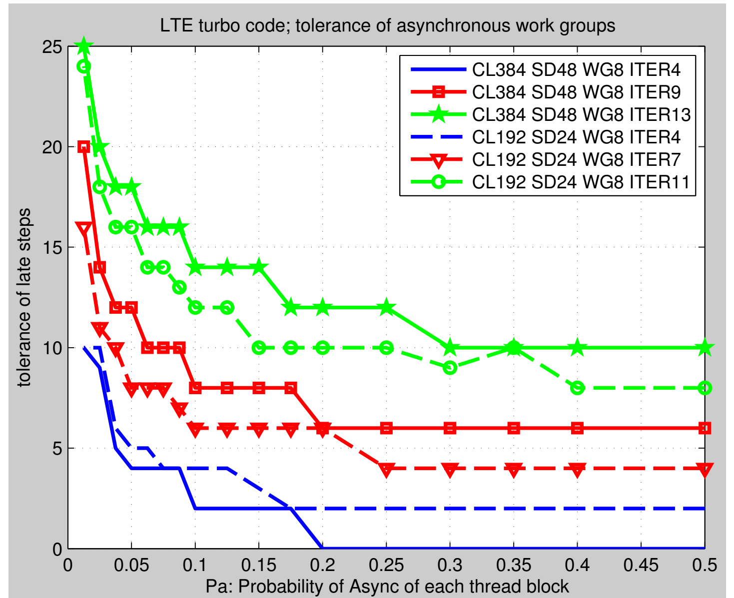

Fig. 5. Tolerances of asynchronous work groups.

IV. Experimental results

Nvidia GTX480 GPU and its OpenCL 1.1 SDK was used for testing the loosely synchronized Turbo decoding approach. Max-Log-Map algorithm [16], with fixed point width of 32bits, was chosen to decode a 6144 bits LTE turbo CB. Though GPU has float point units, fixed point is used to get future portability for low power fixed point processor. Only global memory is used in the implementation. At the beginning of turbo decoding, info, parity, extrinsic and stake data are re-organized to ensure coalesced global memory accesses.

In total, $P \times 2$ work items are launched to decode a single CB using a 2D work space. P is the number of SDs, the second dimension (with size 2) was used for a single SD to perform forward and backward trellis transversal concurrently (see Fig. 7b in [1]). Barrier is put in the cross point of forward and backward transversal to synchronize 2 WIs in the same SD. We did not attempt to parallelize the 8 states metric or 16 branch metric calculations like in the previous GPGPU turbo decoder implementations [8], [9], [10], [11]. This is because 8 or 16 WIs for a single SD have to use synchronization, the overheads of which we wanted to avoid.

Table II shows a throughput comparison of different GPGPU implementations. The numbers for prior works were normalized to 6 iterations. We use a little bit better GPU GTX480 compared to main target GTX470, and they have the same Fermi architecture. Main differences are number of streaming processors (15 vs 14) and clock rate (1.215GHz vs 1.4GHz). For a fairer comparison, a normalized metric is given in the last column by considering the dominant limiting factor. Single CB throughputs (first 7 rows) are normalized by $throughput/(Clock \times MemBW/1000)$, because operational resources won't dominate decoding speed (processor clock rate and memory bandwidth are key factors) in this "lack-of-workload" situation . Multiple CBs results (last 5 rows) are normalized by $throughput/(PeakGFLOPs/1000)$, because in this case we want to exhaust all operational resources by

Nvidia GPU	Nbr cores	Clock [GHz]	Mem BW [GB/s]	Peak GFLOPs	Codeblock size [bits]	[Mbps]	Norm. metric
ION [8]	16	1.1	25.6	35	WCDMA, 5K	0.378	13.4
GF8600GTS [8]	32	1.45	32	139.2	WCDMA, 5K	0.485	10.5
GF8600GTS512 [8]	X	1.63	64	416	WCDMA, 5K	0.958	9.18
C1060 [8]	240	1.3	102.4	933.1	WCDMA, 5K	1.75	13.15
GF9800GX2 [9]	128	1.5	64	576	LTE, 6K	2.1	21.9
C1060 [10]	240	1.3	102.4	933.1	LTE, 6K	7.97	59.9
Proposed GTX480	480	1.4	177.4	1345	LTE, 6K	25	100.7
GTX470 [11]	448	1.215	133.9	1088.6	LTE, 2048*6K	25	23
Proposed GTX480	480	1.4	177.4	1345	LTE, 8*6K	73.6	54.7
Proposed GTX480	480	1.4	177.4	1345	LTE, 12*6K	87.1	64.7
Proposed GTX480	480	1.4	177.4	1345	LTE, 96*6K	119.1	88.6
Proposed GTX480	480	1.4	177.4	1345	LTE, 1536*6K	122.8	91.3

TABLE II. THROUGHPUTS OF VARIOUS GPGPU TURBO DECODERS INCLUDING THE PROPOSED ONE.

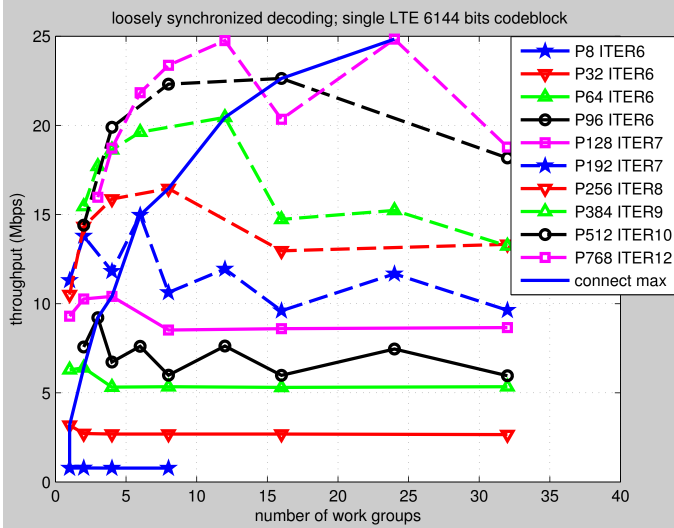

Fig. 6. Throughput of the proposed LTE Turbo decoder by number of work-items and work-groups.

throwing plenty of CBs.

Different number of SDs per WG were tested to see the performance effects. When utilizing multiple WGs, we expect to utilize multiple SMs which update global memory without consistency guarantees. Fig. 6 shows the throughput of decoding a single LTE 6144 bits CB with different number of WGs and SDs.

Fig. 6 confirms that the loose synchronization enables increased utilization of the GPU resources: higher the number of SDs and WGs, higher the throughput. The last curve, "connect max", connects the maximum point of each of the curves. The maximum speedup is achieved when $P/(numberofWGs) = 32$, which is due to Nvidia GTX480 architecture having 32 cores in a single SM. The speedup from additional WGs saturates at about 15 WGs due to the GPU having 15 SMs that execute the WGs.

Picking the $P = 64$ case as a reference [1], [10], [11], the BLER performance was compared against $P = 768$, $24WGs$ and 12 iterations (Fig. 6 "P768 ITER12" vs "P64 ITER6"). The $Eb/N0$ gap was only $0.2dB$, while observing the four time increase in throughput.

The performanace was tested also with multiple CBs. In the extreme case only one WG for each CB decoding was used to avoid intra SMs coupling. There are plenty of loads and no reason to introduce much relationship between SMs. Typical results were $119.1Mbps$ for $C = 96$ and $122.8Mbps$ for $C = 1536$, where C is the number of concurrently decoded CBs. From the Table II our Turbo decoder achieves $100.7/59.9 = 1.68$ and $91.3/23 = 4$ times speedup compared with prior best results both in single and multiple CB decoding, respectively.

It is worth noting, that our throughput result 87.1Mbps in Table II of decoding concurrent 12 CBs (each CB with P=192 and 6 WGs) exceeds the required 75Mbps of LTE R10 uplink categories 1-7 and downlink categories 1-4, 6 (2 layers), 7 (2 layers) for the case of one transport block per 1ms TTI. Also notice that 73.6Mbps in Table II of decoding concurrent 8 CBs (each CB with P=192 and 6 WGs) approaches the required 75Mbps. There are about twelve 6144 CBs in a transport block [17], making the proposed Turbo decoder practical.

V. PREVIOUS WORK

The common aspects in the proposed work and other recent GPGPU Turbo decoder research [8], [9], [10], [11] are the use of a Log-Map or Max-Log-Map based algorithm, and a parallel Turbo decoding scheme. Our contribution to the state-of-the-art is as follows: 1) Much higher (6144 vs. 192) number of SDs can be utilized to decode a *single* CB, 2) use of "loose synchronized" WGs to decode a single CB for improved task parallelism, 3) the first GPGPU Turbo decoder which meets the LTE throughput requirement, thus is approaching hardware solutions in this respect.

VI. CONCLUSION

In this paper, a loosely synchronized GPGPU implementation was proposed for Turbo decoder. The use of loose synchronization allows achieving higher utilization of the GPGPUs parallel resources. The tolerance of asynchronous steps among multiple OpenCL WGs was measured to be in the acceptable range.

The implementation allows a large number of parallel SDs to be utilized to perform Turbo decoding. The SDs can be grouped into multiple non-synchronized WGs, which can execute freely in multiple SMs in parallel.

The OpenCL 1.1 based Turbo decoder implementation was tested in a mid-range NVidia GPU, producing a throughput that meets the requirements of LTE uplink category 5 and downlink category 4, for the first time in a GPGPU-based Turbo decoder implementation.

Optimizing memory accessing pattern and observing loosely synchronized execution in real hardware will be the future work.

Acknowledgements. Author Pekka Jääskeläinen was funded by the Academy of Finland (funding decision 253087).

REFERENCES

[1] Sun Y. and Cavallaro J. R., "Efficient hardware implementation of a highly-parallel 3GPP LTE/LTE-advance turbo decoder," *Integration, the VLSI Journal*, vol. 44, pp. 305–315, Sept. 2011.

[2] Valenti M. and Sun J., "The UMTS Turbo Code and an Efficient Decoder Implementation Suitable for Software-Defined Radios," *International Journal of Wireless Information Networks*, vol. 8, no. 4, pp. 203–215, Oct. 2001.

[3] Liang Zhang and Yubai Li, "Implementing and Optimizing a Turbo Decoder on a TI TMS320C64x Device," in *International Conference on Computational Problem-Solving (ICCP)*, Oct. 2011, pp. 401–404.

[4] T. Ngo and I. Verbauwhede, "Turbo codes on the fixed point DSP TMS320C55x," in *IEEE Workshop on Signal Processing Systems (SIPS)*, 2000, pp. 255–264.

[5] Myoung-Cheol and In-Cheol Park, "SIMD Processor-Based Turbo Decoder Supporting Multiple Third-Generation Wireless Standards," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 7, pp. 801–810, Oct. 2007.

[6] Huili Guo et.al., "High performance turbo decoder on CELL BE for WiMAX System," in *International Conference on Wireless Communications and Signal Processing*, Nov. 2009, pp. 1–5.

[7] J. Nickolls and W.J. Dally, "The GPU Computing Era," *Micro, IEEE*, vol. 20, no. 2, pp. 56–69, Mar. 2010.

[8] Marilyn Wolf Dongwon Lee and Hyesoon Kim, "Design Space Exploration of the Turbo Decoding Algorithm on GPUs," in *CASES*, 2010, pp. 217–226.

[9] Dhiraj Reddy Nallapa Yoge and Nitin Chandrachoodan, "GPU Implementation of a Programmable Turbo Decoder for Software Defined Radio Applications," in *International Conference on VLSI Design*, Jan. 2012, pp. 149–154.

[10] Michael Wu et.al., "Implementation of a 3GPP LTE Turbo Decoder Accelerator on GPU," in *IEEE Workshop on Signal Processing Systems (SIPS)*, Oct. 2010, pp. 192–197.

[11] Guohui Wang Michael Wu, Yang Sun and Joseph R. Cavallaro, "Implementation of a High Throughput 3GPP Turbo Decoder on GPU," *J Sign Process Syst, Springer*, Sept. 2011.

[12] Benedict Gaster et.al., *Heterogeneous Computing with OpenCL*, Elsevier, Aug. 2011.

[13] Khronos Group, *OpenCL Specification v1.1r36*, Sept. 2010.

[14] NVIDIA Corporation, "FERMI Compute Architecture White Paper," http://www.nvidia.com/object/fermi-architecture.html.

[15] ETSI, *LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding (3GPP TS 36.212 version 10.5.0 Release 10)*, Mar. 2012.

[16] J. Vogt and A. Finger, "Improving the max-log-MAP turbo decoder," *Electronic Letters*, vol. 36, pp. 1937–1939, 2000.

[17] ETSI, *LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); User Equipment (UE) radio access capabilities (3GPP TS 36.306 version 10.4.0 Release 10)*, Jan. 2012.