Turbo Decoding on Tailored OpenCL Processor

Heikki Kultala, Otto Esko, Pekka Jääskeläinen, Vladimír Guzma, Jarmo Takala
Tampere University of Technology
Tampere, Finland
Email: firstname.lastname@tut.fi

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

Tommi Zetterman, Heikki Berg
Radio Systems Laboratory
Nokia Research Center
Espoo, Finland
Email: firstname.lastname@nokia.com

***Abstract*—Turbo coding is commonly used in the current wireless standards such as 3G and 4G. However, due to the high computational requirements, its software-defined implementation is challenging. This paper proposes a static multi-issue exposed datapath processor design tailored for turbo decoding. In order to utilize the parallel processor datapath efficiently without resorting to low level assembly programming, the turbo decoder is implemented using OpenCL, a parallel programming standard for heterogeneous devices. The proposed implementation includes only a small set of Turbo-specific custom operations to accelerate the most critical parts of the algorithm. Most of the computation is performed using general-purpose integer operations. Thus, the processor design can be used as a general-purpose OpenCL accelerator for arbitrary integer workloads as well. The proposed processor design was evaluated both by implementing it using a Xilinx Virtex 6 FPGA and by ASIC synthesis using 130 nm and 40 nm technology libraries. The implementation achieves over 63 Mbps Turbo decoding throughput on a single low-power core. According to the ASIC synthesis, the maximum operating clock frequency is 344 MHz/1 050 MHz (130 nm/40 nm).**

I. Introduction

Turbo code has been widely used in wireless communication standards such as CDMA2000, WCDMA, LTE and LTE-A. Around the time the turbo code was invented, its high decoding complexity and latency made it impossible to be used in real time communications. Parallel turbo decoding [1], [2] broke the limitation of sequential traversal across all trellis stages, improving the decoding throughput dramatically. This enabled more and more standards to adopt the turbo code as their *Forward Error Correction (FEC)* scheme.

Because of the high computational complexity, turbo decoding in mobile baseband processors has been usually implemented with customized fixed-function hardware blocks [3]. However, increasing fraction of the radio signal processing functions in baseband are being implemented using software running in *Digital Signal Processors (DSP)*. The benefits of the so-called *Software Defined Radio (SDR)* include the fast time to market, multi-standard flexibility and the ability to update the functionality after the product manufacture.

In this work we propose a Turbo decoder-optimized clustered OpenCL accelerator processor based on the *Transport Triggered Architecture (TTA)* design paradigm. The proposed design includes only a small set of Turbo-specific custom operations to accelerate the most critical parts of the algorithm. Most of the computation is performed using general-purpose integer operations using the OpenCL parallel programming standard [4]. Thus, the processor design can be used as a general-purpose OpenCL accelerator for other integer workloads as well.

978-1-4673-2480-9/13/$31.00

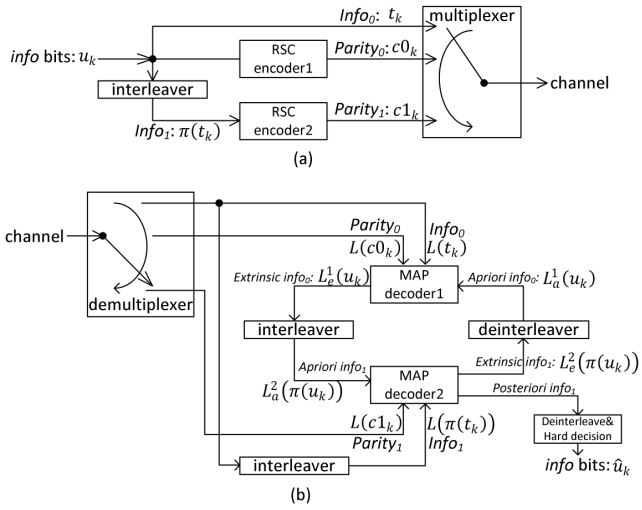

Fig. 1. Structures of (a) turbo encoder and (b) turbo decoder.

II. Software architecture

Turbo encoder and decoder structures are depicted in Fig. 1. At encoder side, information bits and its interleaved version are encoded by two identical Recursive Systematic Convolutional (RSC) encoders. K Systematic information bits $info_0 = \{t_0, ...t_k, ...t_{K-1}\}$, parity bits $parity_0 = \{c0_0, ...c0_k, ...c0_{K-1}\}$ (from the first encoder) and parity bits $parity_1 = \{c1_0, ...c1_k, ...c1_{K-1}\}$ (from the second encoder) are multiplexed and sent to the channel. At the decoder side, noise corrupted $\{info_0, parity_0\}$ and $\{info_1, parity_1\}$ ($info_1$ produced by interleaving $info_0$) are fed into two *Maximum a Posteriori (MAP)* decoders. Extrinsic information on systematic bits is exchanged between two MAP decoders via interleaver/deinterleaver in the iterative decoding process. In the second half (MAP decoder2) of the last iteration, estimates of the bits are produced based on hard decision of posteriori information.

A. Fixed Point Max-Log-Map Algorithm

The *Max-Log-Map* algorithm [1], [5] is used to calculate extrinsic information in each MAP decoder. The algorithm inputs are *Log-Likelyhood Ratio (LLR)* on the channel systematic bit $L(t_k)$, the parity bit $L(c_k)$ and the priori systematic bit

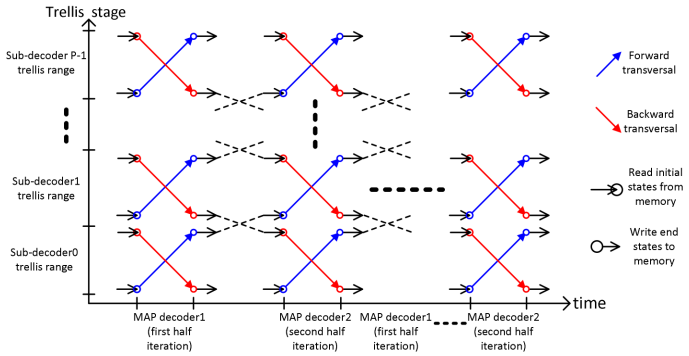

Fig. 2. Trellis progresses and state metrics read/written.

$L_a(u_k)$ from the other MAP decoder. The algorithm output is extrinsic LLR $L_e(u_k)$ on the systematic bit, which is calculated by:

$$L_e(u_k) = (\max_{u_k=1}\{\alpha_{k-1}(s_{k-1}) + (2c_k - 1)L(c_k) + \beta_k(s_k)\} - \max_{u_k=0}\{\alpha_{k-1}(s_{k-1}) + (2c_k - 1)L(c_k) + \beta_k(s_k)\})/2$$

where s_k is the state index at stage k; $\alpha_{k-1}(s_{k-1})$ and $\beta_k(s_k)$ are forward and backward state metrics; $c_k \in \{0, 1\}$ is parity bit generated by state transition $s_{k-1} \to s_k$; $\max_{u_k=x}\{\cdot\}$ finds the maximum value from all possible state transitions $s_{k-1} \to s_k$ driven by input $u_k = x$. A scale factor 0.7 [5] is used on $L_e(u_k)$ before exchanging it with the other MAP decoder. The forward and backward state metrics are calculated recursively by:

$$\alpha_k(s_k) = \max_{s_{k-1}}\{\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1} \to s_k)\}$$
$$\beta_k(s_k) = \max_{s_{k+1}}\{\beta_{k+1}(s_{k+1}) + \gamma_k(s_k \to s_{k+1})\}$$

Branch metric $\gamma_k(s_{k-1} \to s_k) = (2u_k - 1)(L(t_k) + L_a(u_k)) + (2c_k - 1)L(c_k)$, where $u_k, c_k \in \{0, 1\}$ are state transition related systematic and parity bits.

We used 8-bit signed numbers (char type in the OpenCL C language) as the basic type. Modulo selection on maximum metric [6], [7] is used to avoid overflow detection in trellis traversal. In order to avoid ambiguity, branch metric γ_k is saturated into $[-14, 14]$. By using the fixed point parameters above, the code word error rate performance loss in terms of required signal to noise ratio is only 0.5 dB compared to a floating point implementation.

B. Parallel OpenCL Program Design

Multiple subdecoders are used to process different portions of input bits concurrently, and there is one subdecoder running per one OpenCL Work Item (WI). All the WIs run in the same OpenCL Work Group and can use local memory for communication. All data except constants and the input and output buffers are put into the local or the private memory.

In each subdecoder, forward and backward traversal are performed concurrently [8]. The initial state metrics of the subdecoder are taken from the ending state of adjacent MAP subdecoder in the previous iteration [8]. The trellis-time progressing of subdecoders is depicted in Fig. 2 assuming P subdecoders.

$L(c_k)$, $L(t_k)$ and $L_a(u_k)$ are divided into P subblocks (of size $N = K/P$) to be fed into P parallel subdecoders.

In the beginning of the kernel code, the data is rearranged to achive coalesced memory accesses, i.e., each WI accesses memory from the form of $L[Nd + t]$ (stride) to $L[tP + d]$ (block), where $d \in \{0, 1, ...P - 1\}$ is subdecoder/WI index returned by function get_local_id(0). $t \in \{0, 1, ...N - 1\}$ is the trellis stage index used in the traversal/loop inside of a WI. Besides input data rearrangement, the intermediate memory used in the calculation is also arranged in block manner. This arrangement ensures that the adjacent WIs access adjacent memory addresses and therefore vector memory access is easily achieved. Interleaving $info_0$ elements to get $info_1$ (also in coalesced form) is also performed in all WIs. These conversions are integrated to the routines which copy data from the global memory input buffers to the local memory. Corresponding reverse arrangement and interleaving are merged into the copy back routines at the end of the kernel code. Interleaving addresses are pre-calculated during compile time into constant tables. In addition, the state metrics of the subdecoders are initialized appropriately.

Both the forward and the backward trellis traversals of length N are done simultaneously in each WI. Traversal is split into two loops (both of size $N/2$). Before the first loop, the initial state metrics needed by the forward and the backward traversals are read from the local memory. In the first loop, the forward and the backward state metrics are calculated and stored in WI private memory recursively from stage 0 to $(N/2) - 1$ and stage $N - 1$ to $N/2$, respectively. In the second loop, besides the forward and backward recursive state metrics calculation (stage $N/2$ to $N - 1$ and stage $(N/2) - 1$ to 0 respectively), extrinsic LLRs are calculated according to new calculated state metrics and stored state metrics in the first loop, and stored into local memory in an interleaved manner to have second half iteration access a priori LLR in a streaming manner. After the second loop, a priori state metrics of forward and backward are saved into local memory which will be read by adjacent WIs in the next iteration.

The second half iteration has almost the same processing with the first half iteration except that extrinsic LLRs are stored in a non-interleaved manner to have the first half iteration access a priori LLR in streaming manner in the next iteration. Because of the conflict-free character of LTE *Quadratic Permutation Polynomial (QPP)* interleaver, accessing extrincs LLRs is either dynamically coalesced or statically vectorized (explained in the next section).

III. Processor Architecture

The processor architecture was tailored using the *TTA-based Co-design Environment (TCE)* tools and its retargetable OpenCL compiler [9], [10] based on the *Transport Triggered Architecture (TTA)* paradigm [11]. In transport triggered processors, the datapath buses are exposed to the programmer: the processor is programmed by scheduling the data transfers that take place. Actual operations (e.g., arithmetic or memory operations) are executed when a transport is made to specific "trigger port" of a function unit implementing the operation.

Compared to traditional "operation-programmed" *Very Long Instruction Word (VLIW)* architectures, where the instruc-

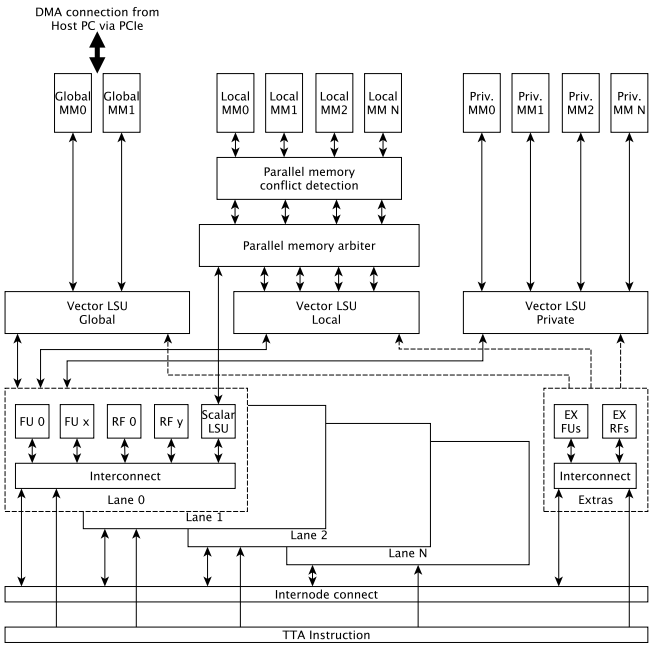

Fig. 3. Processor architecture and the memory hierarchy. Per-lane vector *Load Store Unit (LSU)* data connections are only drawn from *Lane 0* for clarity. The data memory is divided into three separate address spaces matching the OpenCL standard: global, local and private. Each address space is implemented with multiple parallel 32-bit wide memory modules (MM).

tion set specifies operations, and data transfers occur as part of operations, the TTA programming model has the benefit that the register file bypasses are explicitly programmed ("software bypassing"), and all the operands of operations do not have to be read in the same clock cycle. Similarly, the computed results do not have to be read to the destination register file on the same cycle they are produced, and the result write to a register can be totally omitted of the result is bypassed directly to some another operation. This allows using smaller register files with less read and write ports [12].

Because in TTA processors the register files and function units are fully decoupled from the rest of the architecture due to the customizable interconnection network, it is easy to design new processors in a "component based" manner.

A. Clustered Static Multi-Issue Exposed Datapath

The high-level structure of the proposed processor resembles a common clustered-VLIW architecture with nine clusters. Each cluster contains a set of function units, register files and interconnect buses as listed in Table I. Eight of these clusters are symmetric "lanes" dedicated to executing code in different OpenCL work items in a data parallel fashion. The lanes can execute code from multiple work items if there are more work items than clusters, in order to get higher utilization of the resources. All the resources in all of the clusters are explicitly programmed with a single wide instruction to maximize the utilization potential of the datapath resources. The architecture along with its memory hierarchy (explained in the next section) is illustrated in Fig. 3.

Each lane contains multiple function units and register files with adequate connectivity for an additional degree of

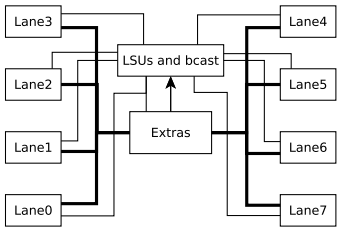

Fig. 4. The clusters and their connectivity.

instruction-level parallelism. There is no hard limit on how many work items can be scheduled on one lane. However, the amount of live variables in comparison to the available general purpose registers introduce the practical upper limit: If all the private scalar variables do not fit into the registers of the lane, they have to be spilled into the memory and the performance drops dramatically.

The ninth cluster (called the "extras cluster") is for executing code common to all work items, such as calculating memory addresses for vector memory operations, calculating loop indices and handling control flow operations. There are two interconnect buses connecting the extras cluster into the lane clusters. Four lane clusters share each bus, and a separate *broadcast unit* was added for transferring the data that is the same for all work items from the extras cluster to all the lanes.

Multiple datapath interconnect topologies were explored during the design of the processor before settling to the one depicted in Fig. 4. The reasoning behind the chosen topology is that there is no expected data traffic between the different lanes. OpenCL work items are by definition independent, thus their data should reside within a lane. All the Inter-cluster traffic is between a lane cluster and the extras cluster, or from the extras cluster to some or all of the lanes. A "star topology" around the extras cluster was originally selected as the basis for the interconnect as it allows fast transfers from the extras cluster to the lanes and from the lanes to the extras. A full star topology would have added eight transport buses just for the lane interconnections, and most of the time these buses would have stayed unused. Simulations showed that the "split bar topology" that shares one bus between four lane interconnects did not cause any performance degradation, but allowed the processor to have six buses less than in a full star topology.

TABLE I. RESOURCES IN THE CLUSTERS OF THE PROCESSOR. (*) = SHARED WITH LANE 0 (**) = ONE SHARED WITH LANE 0. ALL SHARED WITH VECTOR LSU.

Resource	lanes	extras	total
Integer register files (2rd+2wr ports, 16 regs each)	4	4	36
Boolean register files (1rd+1wr port, 2 regs each)	0	1	1
Integer ALUs	2	3	19
Adders/subtractors	2	3	19
Multiplier	1	0	8
Turbo-specific Special Function Units	2	0	16
Pack/Unpack Function Units	2	0	16
Vector LSU data ports	3	3(*)	24
Scatter-gather/scalar LSU	1	3(**)	10
Transport bus	8	10	76

B. Memory Architecture

The OpenCL standard defines an abstract memory model hierarchy with multiple address spaces for OpenCL devices.

The *global* address space is shared among work groups and the host processor. The *local* memory is shared between work items within a single work group, and the *private* memory contains the private variables of the work items. The mapping of these address spaces to actual memories is left to the implementation, which allows device specific optimizations. The memory bandwidth utilization can be maximized by carefully deciding the address space mapping and writing the application to take the full advantage of it, or vice versa. [4]

The memory architecture of the proposed processor is divided into three independent address spaces according to the OpenCL address space division as depicted in Fig. 3. In our test setup, a personal computer host had DMA access to the global memory via PCI Express bus. The local and the private memories were not directly accessible from the processor host.

In order to satisfy the high bandwidth requirements of the turbo decoder, vector memory operations were implemented to access each address space. A common way to increase memory bandwidth without expensive multiport memories is to use a parallel memory architecture consisting of multiple parallel single port memory modules [13]. In our architecture, the number of 32-bit parallel memory modules per local and private address spaces matches the number of lane clusters. This setup allows each lane to have conflict free 32-bit access to local and private memories in parallel per clock cycle in a conflict free situation. The width of the global memory is 64-bit, i.e. two memory modules, which matches the data width of the x4 PCI Express bus used in the test setup. The access to global memory is performed with an interleaving vector LSU which also supports scalar operations. This vector LSU can perform a maximum of 64-bit memory access per clock cycle. Wider accesses are internally pipelined and interleaved.

Private memory vector LSU is interfaced with the same number of memory modules as there are lane clusters. Like the global memory vector LSU, the private memory vector LSU assumes low order scheme and provides vector accesses for multiples of 32-bit words. Scalar accesses are also supported. The differentiating feature compared to the global memory vector LSU is that the private memory vector LSU implements wider vector accesses without interleaving.

The access to the local address space differs from the other address spaces. As seen in the Fig. 3, there are scalar LSUs and a vector LSU connected to the the local memory modules through a memory arbiter and a parallel memory conflict detection hardware. The purpose of the memory arbiter is to serialize the potential concurrent accesses from the local memory vector LSU and the scalar LSUs. That is, the memory hierarchy supports both dynamically coalesced and statically scheduled parallel memory accesses.

The conflict detection hardware is a refined version of the implementation proposed by Tanskanen et al. [13] This hardware unit includes runtime memory access conflict detection and conflict resolution logic which are used to provide scatter-gather access to the parallel memory using the scalar LSUs. The most important new features in the updated hardware unit include support for subword accesses and merging of non-conflicting subword writes to a same bank into a single write operation.

The parallel memory conflict detection hardware also includes the data and the address crossbars which simplify the design of the local memory vector LSU. This vector LSU has to only take care of the address calculations for the vector access and the sign extension of the loaded data. The reason for including the local memory vector LSU lies in the static address calculations: whenever the vector access can be determined during the compile time it saves the lane clusters the burden of calculating all of the addresses explicitly.

C. Mapping the Algorithm to the Memory Architecture

Ideally, the address calculations for the vector LSUs are performed in the extras cluster. For this reason, the vector LSU address ports are only connected to the extras cluster. The address port connections are drawn with dashed arrows in Fig. 3. Each of the input and the output data ports are connected to a single lane, e.g. , the second ports are connected to the second lane cluster. The first ports are the exception to this rule as they are also connected to the extras cluster in addition to the first lane cluster. This allows extras cluster to perform scalar accesses through the vector LSU.

All the other memory operations in the inner loops except the extrinsic values which are used to transfer data between the first half (MAP decoder1) and the second half (MAP decoder2) of the turbo decoder are trivially compile-time vectorizable. The interleaving, on the other hand, makes the memory addresses dynamic and prevent vectorization of these memory operations. Each extrinsic value in both halves is written once per iteration, but read twice per iteration, once for forward metrics and one for backward metrics. By storing the first decoder extrinsic buffer in an interleaved format, and the second decoder extrinsic buffer in a non-interleaved format, all stores to both extrinsic stores are dynamic (scatter) stores without bank conflict (character of the LTE QPP interleaver), and all the loads from both buffers become streaming loads which can be vectorized and perfectly parallelized. As loads make up two thirds of the total memory operations, this saves address calculation operations.

In order to utilize the full bandwidth of the load-store units, four 8-bit values are packed together in SIMD-like fashion on many data buffers. This allows accessing 32 8-bit values with one 256-bit vector memory operation.

D. Special Function Units

Recursive state metrics calculation is a group of operations consisting of consecutive sum and max, or difference and max operations. There are eight of these operation pairs running four times for every iteration of each produced bit. A *Special Function Unit (SFU)* that can execute these operations quickly was added to the design. A similar SFU was proposed in [14] and [15]. Our implementation does not include the branch metric γ value calculation, which is done by general purpose integer FUs outside SFU, so the same custom operation can be used for both the forward and the backward metrics.

Using 8-bit arithmetic precision allows additional optimization for this SFU: the input and output data can be kept in a 4x8b SIMD format, which reduces the need for I/O ports in the function unit, register read and write ports in the register files and the number of programmable transport buses. The SFU implements saturating arithmetic on the branch metric

γ in order to avoid the ambiguity of modulo selection. For constructing and deconstructing the 4x8b SIMD words, we added the PACK and UNPACK operations.

Extrinsic LLRs calculation is handled by another SFU. This SFU uses the same 4x8b SIMD format to read 17 input values (8 forward state metrics, 8 backward state metrics and 1 parity LLR) by using only 5 input ports. The metric data can stay in the SIMD format whole time from the state metrics calculations to the Extrinsic LLRs.

IV. Evaluation

The processor design space was explored by varying the number of lane clusters, the subdecoder count, and the set of custom operations (SFUs) to include. For all experiments, aggressive compiler-based loop unrolling was used, the instruction memory size limiting the unroll threshold. Instruction memory sizes of 2 048 and 4 096 instructions words were benchmarked, though 4 096 is assumed to be too big for realistic implementation.

Table II shows the throughput with different number of processor lanes and subdecoder counts with the clock speed scaled down to 100 MHz for the LTE 6 144 bits long codeblock with 6 decoder iterations. The numbers are produced using the instruction cycle accurate simulator of the TCE toolset which does not model the memory conflicts. The FPGA implementation verified the number of conflicts to be insignificant. The minor conflicts came from copy in and back routines in the beginning and the end of the kernel, where the stride-block arrangement and interleaving-deinterleaving are merged.

TABLE II. Throughput with different options. Scaled down to 100MHz operating frequency.

Lanes	Subdecoders	Performance (2k imem.)	Performance (4k imem.)
2	2	1 087 kbps	1 108 kbps
	4	1 539 kbps	1 583 kbps
	8	1 633 kbps	1 700 kbps
4	4	2 128 kbps	2 157 kbps
	8	2 889 kbps	3 024 kbps
	16	3 561 kbps	3 887 kbps
8	8	4 058 kbps	4 136 kbps
	16	6 344 kbps	6 717 kbps
	32	6 149 kbps	7 253 kbps

Table II shows that with a small number of lanes, multiple subdecoders in one vector lane gave better performance than a single subdecoder in a vector lane. There are two reasons for this. Firstly, the subdecoders have limited instruction-level parallelism and having multiple of them allows better utilization of all the processor hardware resources. Second, there are too many data dependencies between successive loop iterations that the loop unrolling does not expose enough parallelism between the unrolled iterations. With the final 8-lane machine with instruction memory size of 2048, however, 16 subdecoders gives better performance than 32 subdecoders. This is because the smaller work groups allow more aggressive unrolling while still keeping the instruction memory size under the desired 2 048 instruction limit. With bigger instruction memory the program scales much better with 32 subdecoders than 16 subdecoders, and a speed of over 70 kbps/MHz can be reached.

The processor design was first implemented with a Virtex 6 SX FPGA. This FPGA implementation was used to verify that the design is reasonable for a real hardware implementation and to get the accurate number of stall cycles due to memory bank conflicts. The final FPGA implementation consumed 72 % of the slices and 22 % of the on-chip RAM blocks of the chip. Executing the fastest version that fits in 2 048 instructions on the FPGA revealed there were total of $4\ 151$ stall cycles during the execution resulting in a total of $101\ 023$ cycles. This means that the stall cycles increased the cycle count by 4.3 %, resulting in performance of approximately 60.82 kbps/MHz. Thus, the FPGA implementation running at 65 MHz reaches performance of 3 953 kbps.

The processor core was also synthesized for 130 nm and 40 nm ASIC processes. On the 130 nm process the synthesis tools reported that 344 MHz clock speed would be achievable, which results in the throughput of 20.92 Mbps for the whole turbo decoder. The size of the processor core size was $703\ 088$ gates. Out of these the turbo-specific SFUs consumed $51\ 643$ gates, which is 7.3% of the total gate count of the processor core. On the 40nm process the maximum clock speed of $1\ 050$ MHz was achived which results in the maximum decoding throughput of 63.86 Mbps. Dynamic power consumption was simulated to be 261 mW for the core only, not including memories.

V. Related work

In the recent years, there have been promising efforts on *Software Defined Decoders (SDD)* on DSPs [19]–[21] but to our knowledge a solution which meets the demanding and contradicting high throughput and low power consumption requirements of mobile devices hasn't been proposed.

Several SDD implementations that exploit the vast amounts of parallelism available in *Graphics Processing Units (GPU)* have been proposed recently [18], [22]–[24]. Of these so called *General Purpose computing on Graphics Processing Units (GPGPU)* efforts, the best we could find achieves 7.97 Mbps throughput when decoding a single LTE turbo 6 144 bits codeblock using an Nvidia C1060 GPU [18]. The C1060 has total of 240 lanes on 10 processor cores and a 1.3 GHz clock rate. In [24], 25 Mbps thoughput is achieved by decoding 2 048 codeblocks in parallel on an Nvidia GTX470 GPU. However, this result is not comparable to the one proposed in this paper because decoding multiple codeblocks in parallel does not help in reducing the decoding latency of a single codeblock, and the 187.8W power consumption of the C1060 GPU rules out its use in mobile devices.

Salmela et al. proposed a high-performance turbo decoder which also uses a TTA-based processor design [15]. Its throughput of 22.7 Mbps on 130 nm ASIC technology is only slightly faster than with the proposed design (20.92 Mbps) on the same technology. Their software, however, is implemented with manually optimized assembly.

Shahabuddin et al. have also proposed a turbo decoder TTA processor [14]. Their processor is programmed using the high-level C language. The authors report a throughput of 31.78 Mbps for a *single iteration*, which translates to 5.30 Mbps for the full decode with 6 iterations, thus produces significantly lower throughput than our proposal.

Vogt et al. proposed a turbo decoder running on a customized SIMD-based processor [16]. Their design achieves

TABLE III. PERFORMANCE COMPARISON WITH OTHER TURBO DECODERS

Category	Ref.	HW Implementation	SW language	clock rate (MHz)	performance (Mbps)	Normalized (kbps/MHz)
ASIP	[14]	TTA Processor	C	200	5.30	26.50
	[15]	TTA Processor on 130 nm	Assembly	277	22.7	81.92
	[16]	SIMD ASIP on 65nm		400	17	42.5
	[16]	SIMD ASIP on Virtex 4		109	4.6	42.2
	[17]	FlexFEC(SIMD ASIP on 65nm)		320	23.3	72.91
		Proposed on Virtex 6 FPGA	OpenCL	65	3.95	60.82
		Proposed on 130nm	OpenCL	344	20.92	60.82
		Proposed on 40nm	OpenCL	1 050	63.86	60.82
GPU	[18]	Nvidia C1060	CUDA	1 300	7.97	6.13

only slightly lower througput per clock rate than ours. However, although on FPGA they can reach higher clock speeds than our proposed design, their design cannot reach as high clock speeds on ASIC as ours. The general purpose programmability, however, is assumed to be similar to ours.

Naessens et al. proposed a turbo decoder running on a customized 96-lane SIMD-based processor called FlexFEC [17]. The authors report a throughput of 140 Mbps for a *single iteration*, which translates to 23.33 Mbps for the full decode with 6 iterations while running at only 320 MHz. Their design achieves slightly higher throughput per cycle than ours, but it cannot reach as high clock frequencies than our design, and as it only has 10-bit datapaths and memories, the general purpose programmability of their design is assumed to be lower than ours, though they have demonstrated that that they can execute also LDPC codes in addition to turbo codes.

VI. CONCLUSIONS

We showed that an OpenCL-programmable clustered single core TTA processor with a limited set of custom operations can reach the latency requirements of Category 4 LTE and well over third of the maximum bandwidth required by LTE. The proposed processor is designed to be scalable to multiple cores when decoding multiple codeblocks, thus a three-core version of the design could be used to implement turbo decoding on a SDR-based LTE implementation, while the same processor can be used for other integer workloads. The ASIC implementation synthesized on a 40 nm process achieves the maximum clock frequency of 1 050 MHz which provides the decoding throughput of 63 Mbps with a single core. To the best of our knowledge, this throughput is better than any mostly software-based implementation of the turbo decoder.

Acknowledgements. The work has been financially supported by Academy of Finland (funding decision 253087).

REFERENCES

[1] A. J. Viterbi, "An intuitive justification and a simplified implementation of the map decoder for convolutional codes," *IEEE Journal on Selected Areas in Communications*, vol. 16, pp. 260–264, Feb. 1998.

[2] J.-M. Hsu, "A parallel decoding scheme for turbo codes," in *IEEE Int. Symp. Circ. Syst.*, 1998, vol. 4, pp. 445–448.

[3] E. Tell, A. Nilsson, and D. Liu, "A low area and low power programmable baseband processor architecture," in *Int. Workshop System-on-Chip for Real-Time Applications*, 2005, pp. 347–351.

[4] Khronos Group, *OpenCL Specification v1.2r15*, Nov. 2011.

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

[6] A.P. Hekstra, "An alternative to metric rescaling in viterbi decoders," *IEEE Transactions on Communications*, vol. 37, pp. 1220–1222, 1989.

[7] A. Worm, H. Michel, G. Kreiselmaier, M. Thul, and N. Wehn, "Advanced implementation issues of turbo-decoders," in *Proc. Int. Symp. Turbo-Codes and Related Topics*, 2000, pp. 351–354.

[8] 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, pp. 305–315, Sept. 2011.

[9] P.O. Jääskeläinen, C.S. de La Lama, P. Huerta, and J.H. Takala, "OpenCL-based design methodology for application-specific processors," in *Int. Conf. Embedded Comput. Syst.*, July 2010, pp. 223 –230.

[10] O. Esko, P. Jääskeläinen, P. Huerta, C. S. de La Lama, J. Takala, and J. Ignacio Martinez, "Customized exposed datapath soft-core design flow with compiler support," *International Conference on Field Programmable Logic and Applications*, pp. 217–222, 2010.

[11] H. Corporaal and M. Arnold, "Using Transport Triggered Architectures for embedded processor design," *Integrated Computer-Aided Eng.*, vol. 5, no. 1, pp. 19–38, 1998.

[12] J. Hoogerbrugge and H. Corporaal, "Register file port requirements of Transport Triggered Architectures," in *Proc. Annual Int. Symposium Microarchitecture*, Nov. 30 - Dec. 2 1994, pp. 191–195.

[13] J.K. Tanskanen, T. Pitkänen, R. Mäkinen, and J. Takala, "Parallel memory architecture for TTA processor," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, vol. 4599 of *Lecture Notes in Computer Science*, pp. 273–282. Springer Berlin Heidelberg, 2007.

[14] S. Shahabuddin, J. Janhunen, and M. Juntti, "Design of a transport triggered architecture processor for flexible iterative turbo decoder," in *Proc. Wireless Innovation Forum Conf. Wireless Commun. Tech. Software Defined Radio*, 2013, pp. 16–21.

[15] P. Salmela, H. Sorokin, and J. Takala, "A programmable max-log-map turbo decoder implementation," *VLSI Design*, vol. 2008, 2008.

[16] T. Vogt and N. Wehn, "A reconfigurable application specific instruction set processor for convolutional and turbo decoding in a sdr environment," in *Proc. Design Automation Test in Europe*, 2008, pp. 38–43.

[17] F. Naessens et al., "A 10.37 mm2 675 mw reconfigurable ldpc and turbo encoder and decoder for 802.11n, 802.16e and 3gpp-lte," in *VLSI Circuits (VLSIC), 2010 IEEE Symposium on*, 2010, pp. 213–214.

[18] M. Wu, Y. Sun, and J.R. Cavallaro, "Implementation of a 3GPP LTE Turbo Decoder Accelerator on GPU," in *IEEE Workshop on Signal Processing Systems*, Oct. 2010, pp. 192–197.

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

[20] L. Zhang and Y. 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.

[21] M.-C. Shin and I.-C. 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.

[22] D. Lee, M. Wolf, and H. Kim, "Design Space Exploration of the Turbo Decoding Algorithm on GPUs," in *CASES*, 2010, pp. 217–226.

[23] D.R.N. Yoge and N. Chandrachoodan, "GPU Implementation of a Programmable Turbo Decoder for Software Defined Radio Applications," in *International Conference on VLSI Design*, Jan. 2012, pp. 149–154.

[24] M. Wu, Y. Sun, G. Wang, and J. R. Cavallaro, "Implementation of a High Throughput 3GPP Turbo Decoder on GPU," *J Sign Process Syst, Springer*, Sept. 2011.