J Sign Process Syst
DOI 10.1007/s11265-011-0617-7

Implementation of a High Throughput 3GPP Turbo Decoder on GPU

Michael Wu · Yang Sun · Guohui Wang · Joseph R. Cavallaro

Received: 27 January 2011 / Revised: 1 June 2011 / Accepted: 7 August 2011

Abstract Turbo code is a computationally intensive channel code that is widely used in current and upcoming wireless standards. General-purpose graphics processor unit (GPGPU) is a programmable commodity processor that achieves high performance computation power by using many simple cores. In this paper, we present a 3GPP LTE compliant Turbo decoder accelerator that takes advantage of the processing power of GPU to offer fast Turbo decoding throughput. Several techniques are used to improve the performance of the decoder. To fully utilize the computational resources on GPU, our decoder can decode multiple codewords simultaneously, divide the workload for a single codeword across multiple cores, and pack multiple codewords to fit the single instruction multiple data (SIMD) instruction width. In addition, we use shared memory judiciously to enable hundreds of concurrent multiple threads while keeping frequently used data local to keep memory access fast. To improve efficiency of the decoder in the high SNR regime, we also present a low complexity early termination scheme based on average extrinsic LLR statistics. Finally, we examine how different workload partitioning choices affect the error correction performance and the decoder throughput.

Keywords GPGPU · Turbo decoding · Accelerator · Parallel computing · Wireless · Error control codes · Turbo codes

M. Wu (✉) · Y. Sun · G. Wang · J. R. Cavallaro
Rice University, Houston, TX, USA
e-mail: mbw2@rice.edu

1 Introduction

Turbo code [1] has become one of the most important research topics in coding theory since its discovery in 1993. As a practical code that can offer near channel capacity performance, Turbo codes are widely used in many 3G and 4G wireless standards such as CDMA2000, WCDMA/UMTS, IEEE 802.16e WiMax, and 3GPP LTE (long term evolution). The inherently large decoding latency and complex iterative decoding algorithm have made it very difficult to be implemented in a general purpose CPU or DSP. As a result, Turbo decoders are typically implemented in hardware [2–8]. Although ASIC and FPGA designs are more power efficient and can offer extremely high throughput, there are a number of applications and research fields, such as cognitive radio and software based wireless testbed platforms such as WARPLAB [9], which require the support for multiple standards. As a result, we want an alternative to dedicated silicon that supports a variety of standards and yet delivers good throughput performance.

GPGPU is an alternative to dedicated silicon which is flexible and can offer high throughput. GPU employs hundreds of cores to process data in parallel, which is well suited for a number of wireless communication algorithms. For example, many computationally intensive blocks such as channel estimation, MIMO detection, channel decoding and digital filters can be implemented on GPU. Authors in [10] implemented a complete 2×2 WiMAX MIMO receiver on GPU. In addition, there are a number of recent papers on MIMO detection [11, 12]. There are also a number of GPU based LDPC channel decoders [13]. Despite the popularity of Turbo codes, there are few existing Turbo

Published online: 10 September 2011

decoder implementations on GPU [14, 15]. Compared to LDPC decoding, implementing a Turbo decoder on GPU is more challenging as the algorithm is fairly sequential and difficult to parallelize.

In our implementation, we attempt to increase computational resource utilization by decoding multiple codewords simultaneously, and by dividing a codeword into several sub-blocks to be processed in parallel. As the underlying hardware architecture is single instruction multiple data (SIMD), we pack multiple sub-blocks to fit the SIMD vector width. Finally, as an excessive use of shared memory decreases the number of threads that run concurrently on the device, we attempt to keep frequently used data local while reducing the overall shared memory usage. We include an early termination algorithm by evaluating the average extrinsic LLR from each decoding cycle to improve throughput performance in the high signal to noise ratio (SNR) regime. Finally, we provide both throughput and bit error rate (BER) performance of the decoder and show that we can parallelize the workload on GPU while maintaining reasonable BER performance.

The rest of the paper is organized as follows: In Sections 2 and 3, we give an overview of the CUDA architecture and Turbo decoding algorithm. In Section 4, we will discuss the implementation aspects on GPU. Finally, we will present BER performance and throughput results and analyses in Section 5 and conclude in Section 6.

2 Compute Unified Device Architecture (CUDA)

A programmable GPU offers extremely high computation throughput by processing data in parallel using many simple stream processors (SP) [16]. Nvidia's Fermi GPU offers up to 512 SP grouped into multiple stream multi-processors (SM). Each SM consists of 32 SP and two independent dispatch units. Each dispatch unit on an SM can dispatch a 32 wide SIMD instruction, a *warp* instruction, to a group of 16 SP. During execution, a group of 16 SP processes the dispatched *warp* instruction in a data parallel fashion. Input data is stored in a large amount of external device memory (>1GB) connected to the GPU. As latency to device memory is high, there are fast on-chip resources to keep data on-die. The fastest on-chip resource is registers. There is a small amount of 64KB fast memory per SM, split between user-managed shared memory and L1 cache. In addition, there is an L2 cache per GPU device which further reduces the number of slow device memory accesses.

There are two ways to leverage the computational power of Nvidia GPUs. Compute Unified Device Architecture [16] is an Nvidia specific software programming model, while OpenCL is a portable open standard which can target different many core architectures such as GPUs and conventional CPUs. These two programming models are very similar but utilize different model terminologies. Although we implemented our design using CUDA, the design can be readily ported into OpenCL to target other multi-core architectures.

In the CUDA programming model, the programmer specifies the parallelism explicitly by defining a kernel function, which describes a sequence of operations applied to a data set. Multiple thread-blocks are spawned on GPU during kernel launch. Each thread-block consists of multiple threads, where each thread is arranged on a grid and has a unique 3-dimensional ID. Using the unique ID, each thread selects a data set and executes a kernel function on the selected data set.

At runtime, each thread-block is assigned to an SM and executed independently. Thread-blocks typically are synchronized by writing to device memory and terminating the kernel. Unlike thread-blocks, threads within a thread-block, which reside on a single SM, can be synchronized through barrier synchronization and share data through shared memory. Threads within a thread-block execute in blocks of 32 threads. When 32 threads share the same set of operations, they share the same *warp* instruction and are processed in parallel in an SIMD fashion. If threads do not share the same instruction, the threads are executed serially.

To achieve peak performance on a programmable GPU, the programmer needs to keep the available computation resource fully utilized. Underutilization occurs due to horizontal and vertical waste. Vertical waste occurs when an SM stalls and cannot find an instruction to issue. And horizontal waste occurs when the issue width is larger than the available parallelism.

Vertical waste occurs primarily due to pipeline stalls. Stalls occur for several reasons. As the floating point arithmetic pipeline is long, register to register dependencies can cause a multi-cycle stall. In addition, an SM can stall waiting for device memory reads or writes. In both cases, GPU has hardware support for fine-grain multithreading to hide stalls. Multiple threads, or concurrent threads, can be mapped onto an SM and executed on an SM simultaneously. The GPU can minimize stalls by switching over to another independent *warp* instruction on a stall. In the case where a stall is due to memory access, the programmer can fetch frequently used data into shared memory to reduce memory access latency. However, as the number of concurrent threads is limited by the amount of shared

memory and registers used per thread-block, the programmer needs to balance the amount of on-chip memory resources used. Shared memory increases computational throughput by keeping data on-chip. However, excessive amount of shared memory used per thread-block reduces the number of concurrent threads and leads to vertical waste.

Although shared memory can improve performance of a program significantly, there are several limitations. Shared memory on each SM is banked 16 ways. It takes one cycle if 16 consecutive threads access the same shared memory address (broadcast) or none of the threads access the same bank (one to one). However, a random layout with some broadcast and some one-to-one accesses will be serialized and cause a stall. The programmer may need to modify the memory access pattern to improve efficiency.

Horizontal waste occurs when there is an insufficient workload to keep all of the cores busy. On a GPU device, this occurs if the number of thread-blocks is smaller than the number of SM. The programmer needs to create more thread-blocks to handle the workload. Alternatively, the programmer can solve multiple problems at the same time to increase efficiency. Horizontal waste can also occur within an SM. This can occur if the number of threads in a thread-block is not a multiple of 32. For this case, the programmer needs to divide the workload of a thread-block across multiple threads if possible. An alternative is to pack multiple sub-problems into one thread-block as close to the width of the SIMD instruction as possible. However, packing multiple problems into one thread-block may increase the amount of shared memory used, which leads to vertical waste. Therefore, the programmer may need to balance horizontal waste and vertical waste to maximize performance.

As a result, it is a challenging task to implement an algorithm that keeps the GPU cores from idling–we need to partition the workload across cores, use shared memory effectively to reduce device memory accesses, while ensuring a sufficient number of concurrently executing thread-blocks to hide stalls.

3 Turbo Decoding Algorithm

Turbo decoding is an iterative algorithm that can achieve error performance close to the channel capacity. A Turbo decoder consists of two component decoders and two interleavers, which is shown in Fig. 1. The Turbo decoding algorithm consists of multiple passes through the two component decoders, where one iteration consists of one pass through both decoders.

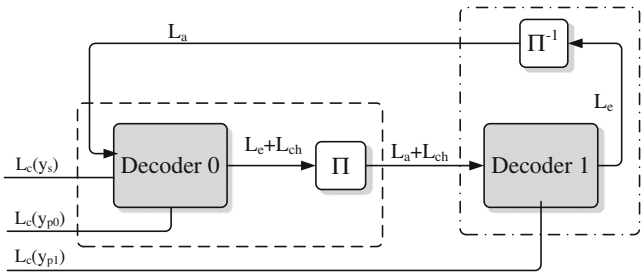


Figure 1 Overview of Turbo decoding.

Although both decoders perform the same sequence of computations, the decoders generate different log-likelihood ratios (LLRs) as the two decoders have different inputs. The inputs of the first decoder are the deinterleaved extrinsic log-likelihood ratios (LLRs) from the second decoder and the input LLRs from the channel. The inputs of the second decoder are the interleaved extrinsic LLRs from the first decoder and the input LLRs from the channel.

Each component decoder is a MAP (maximum *a posteriori*) decoder. The principle of the decoding algorithm is based on the BCJR or MAP algorithms [17]. Each component decoder generates an output LLR for each information bit. The MAP decoding algorithm can be summarized as follows. To decode a codeword with N information bits, each decoder performs a forward trellis traversal to compute N sets of forward state metrics, one α set per trellis stage. The forward traversal is followed by a backward trellis traversal which computes N sets of backward state metrics, one β set per trellis stage. Finally, the forward and the backward metrics are merged to compute the output LLRs. We will now describe the metric computations in detail.

As shown by Fig. 2, the trellis structure is defined by the encoder. The 3GPP LTE Turbo code trellis has eight states per stage. In the trellis, for each state in the trellis, there are two incoming paths, with one path for

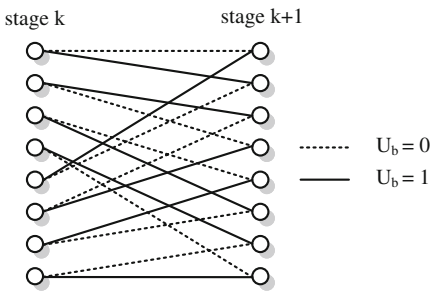


Figure 2 3GPP LTE Turbo code trellis with eight states.

$u_b = 0$ and one path for $u_b = 1$. Let s_k be a state at stage k, the transition probability is defined as:

$$\gamma_k(s_{k-1}, s_k) = \left(L_c\left(y_k^s\right) + L_a\left(y_k^s\right)\right) u_k + L_c\left(y_k^p\right) p_k, \quad (1)$$

where u_k, the information bit, and p_k, the parity bit, are dependent on the path taken (s_{k+1}, s_k). $L_c(y_k^s)$ is the systematic channel LLR, $L_a(y_k^s)$ is the a priori LLR, and $L_c(y_k^p)$ is the parity bit channel LLR at stage k.

During the forward traversal, the sets of state metrics are computed recursively as the next set of state metrics is dependent on the current set of state metrics. The forward state metric for a state s_k at stage k, $\alpha_k(s_k)$, is defined as:

$$\alpha_k(s_k) = \max^*_{s_{k-1} \in K}(\alpha_{k-1}(s_{k-1}) + \gamma(s_{k-1}, s_k)), \quad (2)$$

where K is the set of paths that connects a state in stage $k-1$ to state s_k in stage k.

After the forward traversal, the decoder performs a backward traversal to compute the backward state metrics recursively. The backward state metric for state s_k at stage k, $\beta_k(s_k)$, is defined as:

$$\beta_k(s_k) = \max^*_{s_{k+1} \in K}(\beta_{k+1}(s_{k+1}) + \gamma(s_{k+1}, s_k)). \quad (3)$$

After computing β_k, the state metrics for all states in stage k, we compute two LLRs per trellis state. We compute one state LLR per state s_k, $\Lambda(s_k|u_k = 0)$, for the incoming path that is connected to state s_k which corresponds to $u_k = 0$. In addition, we also compute one state LLR per state s_k, $\Lambda(s_k|u_b = 1)$, for the incoming path that is connected to state s_k which corresponds to $u_k = 1$. The state LLR, $\Lambda(s_k|u_b = 0)$, is defined as:

$$\Lambda(s_k|u_b = 0) = a_{k-1}(s_{k-1}) + \gamma(s_{k-1}, s_k) + \beta_k(s_k), \quad (4)$$

where the path from s_{k-1} to s_k with $u_b = 0$ is used in the computation. Similarly, the state LLR, $\Lambda(s_k|u_b = 1)$, is defined as:

$$\Lambda(s_k|u_b = 1) = a_{k-1}(s_{k-1}) + \gamma(s_{k-1}, s_k) + \beta_k(s_k), \quad (5)$$

where the path from s_{k-1} to s_k with $u_b = 1$ is used in the computation.

To compute the extrinsic LLR for u_k, we perform the following computation:

$$L_e(k) = \max^*_{s_k \in K}(\Lambda(s_k|u_b = 0)) - \max^*_{s_k \in K}(\Lambda(s_k|u_b = 1)) - L_a(y_k^s) - L_c(y_k^s), \quad (6)$$

where K is the set of all possible states and $\max^*()$ is defined as $\max^*(S) = ln(\sum_{s \in S} e^s)$.

The decoding algorithm described above requires the completion of N stages of α before the backward traversal. This is very sequential and requires a large amount of memory to store N stages of α before the start of β computation. There are several ways to distribute the workload and memory storage across multiple decoders to decrease decoding latency. Typically, the incoming codeword can be broken up into multiple sub-blocks. Each sub-block is processed independently and in parallel. As the starting state metrics of the forward and backward traversal are unknown for the sub-blocks, we assume a uniform distribution for the starting state metrics. As each sub-block may not have an accurate starting metric, decoding each sub-block independently will result in error correction performance loss. There are a number of ways to recover the performance loss due to this edge effect. One is next iteration initialization, where we forward the forward and backward state metric between iterations. As shown in Fig. 3, the last α computed for sub-block i can be forwarded to the sub-block $i+1$. Similarly, the last backward metric computed by partition $i+1$ can be forwarded to sub-block i. By forwarding the metrics among sub-blocks between iterations, these sub-blocks will have more accurate starting metrics for the next decoding iteration. Another common alternative is to perform the sliding window algorithm with training sequence [18], where the ith sub-block starts the forward and the backward traversal w samples earlier. This allows different sub-blocks to start at more accurate forward and backward metrics.

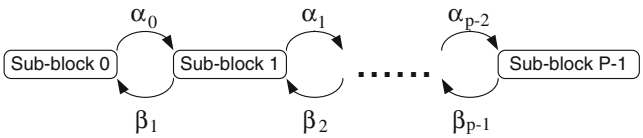


Figure 3 Next iteration initialization.

4 Implementation of Turbo Decoder on GPU

We implemented a parallel Turbo decoder on GPU. Instead of spawning one thread-block per codeword to perform decoding, a codeword is split into P sub-blocks and decoded in parallel using multiple thread-blocks. The algorithm described in Section 3 maps very efficiently onto an SM since the algorithm is very data parallel. As the number of trellis states is eight for the 3GPP compliant Turbo code, the data parallelism of this algorithm is eight. However, the minimum number of threads within a *warp* instruction is 32. Therefore, to reduce horizontal waste, we allow each thread-block

to process a total of 16 sub-blocks from 16 codewords simultaneously. Therefore the number of threads per thread-block is 128, which enables four fully occupied *warp* instructions. As P sub-blocks may not be enough to keep all the SMs busy, we also decode N codewords simultaneously to minimize the amount of horizontal waste due to idling cores. We spawn a total of $\frac{NP}{16}$ thread-blocks to handle the decoding workload for N codewords. Figure 4 shows how threads are partitioned to handle the workload for N codewords.

In our implementation, the inputs of the decoder, LLRs from the channel, are copied from the host memory to device memory. At runtime, each group of eight threads within a thread-block generates output LLRs given the input for a codeword sub-block. Each iteration consists of a pass through the two MAP decoders. Since each half iteration of the MAP decoding algorithm performs the same sequence of computations, both halves of an iteration can be handled by a single MAP decoder kernel. After a half decoding iteration, thread-blocks are synchronized by writing extrinsic LLRs to device memory and terminating the kernel. In the device memory, we allocate memory for both extrinsic LLRs from the first half iteration and extrinsic LLRs from the second half iteration. For example, the first half iteration reads a priori LLRs and writes extrinsic LLRs interleaved. The second half iteration reads a priori LLRs and writes extrinsic LLRs deinterleaved. Since interleaving and deinterleaving permute the input memory addresses, device memory access becomes random. In our implementation, we prefer sequential reads and random writes over random reads and sequential writes as device memory writes are non-blocking. This increases efficiency as the kernel does not need to wait for device memory writes to complete to proceed. One single kernel can handle input and output reconfiguration easily with a couple of simple conditional reads and writes at the beginning and the end of the kernel.

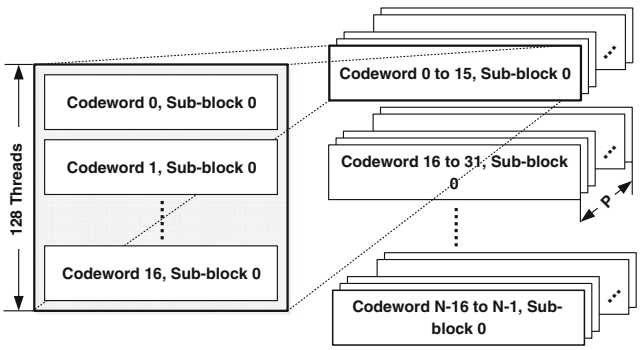


Figure 4 To decode N codewords, we divide each codeword into P sub-blocks. Each thread-block has 128 threads and handles 16 codeword sub-blocks.

In our kernel, we need to recover performance loss due to edge effects as the decoding workload is partitioned across multiple thread-blocks. Although a sliding window algorithm with training sequence can be used to improve the BER performance of the decoder, it is not implemented. The next iteration initialization technique improves the error correction performance with much smaller overhead. In this method, the α and β values between neighboring thread-blocks are exchanged through device memory between iterations.

The CUDA architecture can be viewed as a specific realization of a multi-core SIMD processor. As a result, although the implementation is optimized specifically for Nvidia GPUs, the general strategy can be adapted for other many-core architectures with vector extensions. However, many other vector extensions such as SSE and AltiVec do not support transcendental functions which lead to greater throughput difference between max-log-MAP and full-log-MAP implementations.

The implementation details of the reconfigurable MAP kernel are described in the following subsections.

4.1 Shared Memory Allocation

If we partition a codeword with K information bits into P partitions, we need to compute $\frac{K}{P}$ stages of α before we can compute β. If we attempt to cache the immediate values in shared memory, per partition, we will need to store $\frac{8K}{P}$ floats in shared memory. As we need to minimize vertical waste by decoding multiple codewords per thread-block, the amount of shared memory is quadrupled to pack 4 codewords into a thread-block to match the width of a *warp* instruction. Since we only have 48KB of shared memory which is divided among concurrent thread-blocks on an SM, we will not be able to have many concurrent threads if P is small. For example, if $K = 6{,}144$ and $P = 32$, the amount of shared memory required by α is 24KB. The number of concurrent threads is only 64, leading to vertical waste as we cannot hide the pipeline latency with concurrent running blocks. We can reduce the amount of shared memory used by decreasing P. This, however, can reduce error correction performance. Therefore, we need a better strategy for managing shared memory instead of relying on increasing P.

Instead of storing all α values in shared memory, we can spill α into device memory each time we compute a new α. We only store one stage of α during the forward traversal. For example, suppose α_{k-1} is in shared

memory. After calculating α_k using α_{k-1}, we store α_k in device memory and replace α_{k-1} with α_k. During LLR computation, when we need α to compute $\Lambda_k(s_k|u_b = 0)$ and $\Lambda_k(s_k|u_b = 1)$, we fetch α directly into registers. Similarly, we store one stage of β_k values during the backward traversal. Therefore, we do not need to store β into device memory. In order to increase thread utilization during extrinsic LLR computation, we save up to eight stages of $\Lambda_k(s_k|u_b = 0)$ and eight stages of $\Lambda_k(s_k|u_b = 1)$. We can reuse shared memory used for LLR computation, α, and β. Therefore, the total amount of shared memory per thread-block, packing 16 codewords per thread-block, is 2,048 floats or 8KB. This allows us to have 768 threads running concurrently on an SM while providing fast memory access most of the time.

4.2 Forward Traversal

During the forward traversal, eight cooperating threads decode one codeword sub-block. The eight cooperating threads traverse through the trellis in locked-step to compute α. There is one thread per trellis level, where the jth thread evaluates two incoming paths and updates $\alpha_k(s_j)$ for the current trellis stage using α_{k-1}, the forward metrics from the previous trellis stage $k-1$. Equation 2 computes $\alpha_k(s_j)$. The computation, however, depends on the path taken (s_{k-1}, s_k). The two incoming paths are known a priori since the connections are defined by the trellis structure as shown in Fig. 2. Table 1 summarizes the operands needed for α computation. The indices of the α_k are stored as a constant. Each thread loads the indices and the values $p_k|u_b = 0$ and $p_k|u_b = 1$ at the start of the kernel. The pseudo-code for one iteration of α_k computation is shown in Algorithm 1. The memory access pattern is very regular for the forward traversal. Threads access values of α_k in different memory banks. There are no shared memory conflicts in either case, that is memory reads and writes are handled efficiently by shared memory.

Algorithm 1 Thread i computes $\alpha_k(i)$

$a_0 \leftarrow \alpha_k(s_{k-1}|u_b = 0) + L_c(y_k^s) * (p_k|u_b = 0)$
$a_1 \leftarrow \alpha_k(s_{k-1}|u_b = 1) + (L_c(y_k^s) + L_a(k)) + L_c(p_k^s)(p_k|u_b = 1)$
$\alpha_k(i) = \max^*(a_0, a_1)$
write $\alpha_k(i)$ to device memory
SYNC

4.3 Backward Traversal and LLR Computation

After the forward traversal, each thread-block traverses through the trellis backward to compute β. We assign one thread to each trellis level to compute β, followed by computing Λ_0 and Λ_1 as shown in Algorithm 2. The indices of β_k and values of p_k are summarized in Table 2. Similar to the forward traversal, there are no shared memory bank conflicts since each thread accesses an element of α or β in a different bank.

Algorithm 2 Thread i computes $\beta_k(i)$ and $\Lambda_0(i)$ and $\Lambda_1(i)$

Fetch $\alpha_k(i)$ from device memory
$b_0 \leftarrow \beta_{k+1}(s_{k+1}|u_b = 0) + L_c(y_k^s) * (p_k|u_b = 0)$
$b_1 \leftarrow \beta_{k+1}(s_{k+1}|u_b = 1) + (L_c(y_k^s) + L_a(k)) + L_c(p_k^s)(p_k|u_b = 1)$
$\beta_k(i) = \max^*(b_0, b_1)$
SYNC
$\Lambda_0(i) = \alpha_k(i) + L_p(i)p_k + \beta_{k+1}(i)$
$\Lambda_1(i) = \alpha_k(i) + (L_c(k) + L_a(k)) + L_p(s_k)p_k + \beta_k(i)$

After computing Λ_0 and Λ_1 for stage k, we can compute the extrinsic LLR for stage k. However, there are 8 threads available to compute the single LLR, which introduces parallelism overhead. Instead of computing one extrinsic LLR for stage k as soon as the decoder computes β_k, we allow the threads to traverse through the trellis and save eight stages of Λ_0 and Λ_1 before performing extrinsic LLR computations. By saving eight stages of Λ_0 and Λ_1, we allow all eight threads

Table 1 Operands for α_k computation.

	$u_b = 0$		$u_b = 1$	
Thread id (i)	s_{k-1}	p_k	s_{k-1}	p_k
0	0	0	1	1
1	3	1	2	0
2	4	1	5	0
3	7	0	6	1
4	1	0	0	0
5	2	1	3	1
6	5	1	4	1
7	6	0	7	0

Table 2 Operands for β_k computation.

	$u_b = 0$		$u_b = 1$	
Thread id (i)	s_{k+1}	p_k	s_{k+1}	p_k
0	0	0	4	0
1	4	1	0	0
2	5	1	1	1
3	1	0	5	1
4	2	0	6	1
5	6	1	2	1
6	7	1	3	0
7	3	0	7	0

to compute LLRs in parallel efficiently. Each thread handles one stage of Λ_0 and Λ_1 to compute an LLR. Although this increases thread utilization, threads need to avoid accessing the same bank when computing an extrinsic LLR. For example, eight elements of Λ_0 for each stage are stored in eight consecutive addresses. Since there are 16 memory banks, elements of even stages Λ_0 or Λ_1 with the same index would share the same memory bank. Likewise, this is true for even stages of Λ_0. Hence, sequential accesses to Λ_0 or Λ_1 to compute an extrinsic LLR result in four-way memory bank conflicts. To alleviate this problem, we permute the access pattern based on thread ID as shown in Algorithm 3.

Algorithm 3 Thread i computes $L_e(i)$

$\lambda_0 = \Lambda_0(i)$
$\lambda_1 = \Lambda_1(i)$
for $j = 1$ to 7 **do**
  $index = (i + j)\&7$
  $\lambda_0 = \max^*(\lambda_0, \Lambda_0(index))$
  $\lambda_1 = \max^*(\lambda_1, \Lambda_1(index))$
  $L_e = \lambda_1 - \lambda_0$
  Compute write address
  Write L_e to device memory
end for

4.4 Early Termination Scheme

Depending on SNR, a Turbo decoder requires a variable number of iterations to achieve satisfactory BER performance. In the mid and high SNR regime, the Turbo decoding algorithm usually converges to the correct codeword with a small numbers of decoding iterations. Therefore, fixing the number of decoding iterations is inefficient. Early termination schemes are widely used to accelerate the decoding process while maintaining a given BER performance [19–21]. As the average number of decoding iterations is reduced by using an early termination scheme, early termination can also reduce power consumption. As such, early termination schemes are widely used in low power high performance Turbo decoder designs.

There are several major approaches to implement early termination: hard-decision rules, soft-decision rules, CRC-based rules and other hybrid rules. Hard-decision rules and soft-decision rules are the most popular early termination schemes due to low complexity. Compared to hard-decision rules, soft-decision rules provide better error correcting performance than other low complexity early termination algorithms. Therefore, we implement two soft-decision early termination schemes: minimum LLR threshold scheme and average LLR threshold scheme.

The stop condition of the minimum LLR scheme can be expressed by:

$$\min_{1 \le i \le N} |LLR_i| \ge T, \tag{7}$$

in which we compare the minimum LLR value with a pre-set threshold T at the end of each iteration. If the minimum LLR value is greater than the threshold, then the iterative decoding process is terminated.

The stop condition of the average LLR scheme can be represented by:

$$\frac{1}{N} \sum_{1 \le i \le N} |LLR_i| \ge T. \tag{8}$$

where N is the block length of the codeword and T is the pre-set threshold.

The simulation results show that for multi-codeword parallel Turbo decoding, the variation among minimum LLR values for different codewords is very large. Since each thread-block decodes 16 sub-blocks from 16 codewords simultaneously, we can only terminate the thread-block if all 16 codewords meet the early termination criteria. Therefore, the minimum LLR values are not accurate metrics for early termination for our implementation. The average LLR value is more stable so that the average LLR scheme is implemented for this parallel Turbo decoder. As mentioned in the previous sub-section, during the backward traversal and LLR computation process, eight stages of Λ_0 and Λ_1 are saved in the memory. After Λ_0 and Λ_1 are known, the eight threads are able to compute LLRs using these saved Λ_0 and Λ_1 values in parallel. Therefore, to compute the average LLR value in one codeword, each thread can track the sum of the LLRs when going through the whole trellis. In the end of the backward traversal, we combine all eight sums of LLRs and compute the average LLR value of the codeword. Finally, this average LLR value is compared with a pre-set threshold to determine whether the early termination condition is met. The detailed algorithm is described in Algorithm 4.

Another challenge is that it is difficult to wait for hundreds of codewords to converge simultaneously and terminate the decoding process for all codewords at the same time. Therefore, a tag-based scheme is employed. Once a codeword meets the early termination condition, the corresponding tag is marked and this codeword will not be further processed in the later iterations. After all the tags are marked, we stop the iterative decoding process for all the codewords. By using a tag-based early termination scheme, the decoding

Algorithm 4 Early termination scheme for thread i

```
Compute the codeword ID C_id
if tag[C_id]==1 then
    Terminate the thread i
end if
Forward traversal
for all Output LLR L_e During Backward traversal do
    Sum(i)+ = L_e
end for
if threadId==0 then
    Average = (1/8) Σ_{j=1}^{8} Sum(j)
end if
```

throughput can be significantly increased in the mid and the high SNR regime.

4.5 Interleaver

An interleaver is used between the two half decoding iterations. Given an input address, the interleaver provides an interleaved address. This interleaves and deinterleaves memory writes. In our implementation, a quadratic permutation polynomial (QPP) interleaver [22], which is proposed in the 3GPP LTE standards was used. The QPP interleaver guarantees bank free memory accesses, where each sub-block accesses a different memory bank. Although this is useful in an ASIC design, the QPP interleaver is very memory I/O intensive for a GPU as the memory write access pattern is still random. As inputs are stored in device memory, random accesses result in non-coalesced memory writes. With a sufficient number of threads running concurrently on an SM, we can amortize the performance loss due to device memory accesses through fast thread switching. The QPP interleaver is defined as:

$$\Pi(x) = f_1 x + f_2 x^2 \pmod{N}. \tag{9}$$

The interleaver address, $\Pi(x)$, can be computed on-the-fly using Eq. 9. However, direct computation can cause overflow. For example, 6143^2 can not be represented as a 32-bit integer. Therefore, the following equation is used to compute $\Pi(x)$ instead:

$$\Pi(x) = (f_1 + f_2 x \pmod{N}) \cdot x \pmod{N} \tag{10}$$

Another way of computing $\Pi(x)$ is recursively [6], which requires $\Pi(x)$ to be computed before we can compute $\Pi(x+1)$. This is not efficient for our design as we need to compute several interleaved addresses in parallel. For example, during the second half of the iteration to store extrinsic LLR values, eight threads need to compute eight interleaved addresses in parallel. Equation 10 allows efficient address computation in parallel.

Although our decoder is configured for the 3GPP LTE standard, one can replace the current interleaver function with another function to support other standards. Furthermore, we can define multiple interleavers and switch between them on-the-fly since the interleaver is defined in software in our GPU implementation.

4.6 max* Function

We support the Full-log-MAP algorithm as well as the Max-log-MAP algorithm [23]. Full-log-MAP is defined as:

$$\overset{*}{\max}(a, b) = max(a, b) + ln(1 + e^{-|b-a|}). \tag{11}$$

The complexity of the computation can be reduced by assuming that the second term is small. Max-log-MAP is defined as:

$$\overset{*}{\max}(a, b) = max(a, b). \tag{12}$$

As was the case with the interleaver, we can compute $max^*(a, b)$ directly. We support Full-log-MAP as both natural logarithm and natural exponential are supported on CUDA. However, logarithm and natural exponentials take longer to execute on the GPU compared to common floating operations, e.g. multiply and add. Therefore we expect throughput loss compared to Max-log-Map.

5 BER Performance and Throughput Results

We evaluated the accuracy of our decoder by comparing it against a reference standard C language implementation. To evaluate the BER performance and throughput of our Turbo decoder, we tested our Turbo decoder on a Windows 7 platform with 8GB DDR2 memory running at 800 MHz and an Intel Core 2 Quad Q6600 processor running at 2.4Ghz. The GPU used in our experiments is the Nvidia GeForce GTX 470 graphic card, which has 448 stream processors running at 1.215GHz with 1280MB of GDDR5 memory running at 1,674 MHz.

5.1 Decoder BER Performance

Our decoder can divide a codeword into P sub-blocks. Since our decoder processes eight stages in parallel to compute LLRs, we support a P value if the length of the corresponding sub-blocks is divisible by eight. We expect that the number of sub-blocks per codeword

affects the overall decoder BER performance as larger P introduces more edge effects.

For our simulation, the host computer first generates random 3GPP LTE Turbo codewords. After BPSK modulation, input symbols are passed through the channel with AWGN noise, the host generates LLR values based on the received symbol which are fed into the Turbo decoder kernel running on the GPU. For these experiments, we tested our decoder with the following P values, $P = 1, 32, 64, 96, 128$ for a 3GPP LTE Turbo code with $N = 6144$. In addition, we tested both Full-log-MAP as well as Max-log-MAP with the decoder performing six decoding iterations.

Figure 5 shows the BER performance of our decoder using Full-log-MAP, while Fig. 6 shows the BER performance of our decoder using Max-log-MAP.

In both cases, BER of the decoder becomes worse as we increase P. The BER performance of the decoder is significantly better when Full-log-MAP is used. Furthermore, we see that larger P can offer reasonable performance. For example, when $P = 96$, where each sub-block is only 64 stages long, the decoder provides BER performance that is within 0.1dB of the performance of the optimal case ($P = 1$). For parallelism of 32, the decoder provides BER performance that is close to the optimal case.

5.2 Decoder Throughput

5.2.1 Maximum Throughput

The value P affects the throughput performance as it controls the number of thread-blocks spawned at runtime. To find the maximum throughput this decoder can achieve, we use an extremely large workload, a batch of 2,048 codewords, to ensure there are a sufficient number of thread-blocks for all possible P values. As the decoding time is linearly dependent on the number of trellis stages traversed, varying P and K do not significantly affect the decoder throughput provided there is a sufficient workload to keep the cores busy. The decoder's maximum throughput only depends on the number of iterations performed, max* and the interleaver method used. We vary these parameters and measure throughput of the decoder using event management in the CUDA runtime API. The throughput of the decoder is summarized in Table 3. We see that the throughput of the decoder is inversely proportional to the number of iterations performed. The throughput of the decoder after m iterations can be approximated as T_0/m, where T_0 is the throughput of the decoder after one iteration. Although the throughput of Full-log-MAP is slower than Max-log-MAP as expected, the difference is small. However, Full-log-MAP provides significant BER performance improvement.

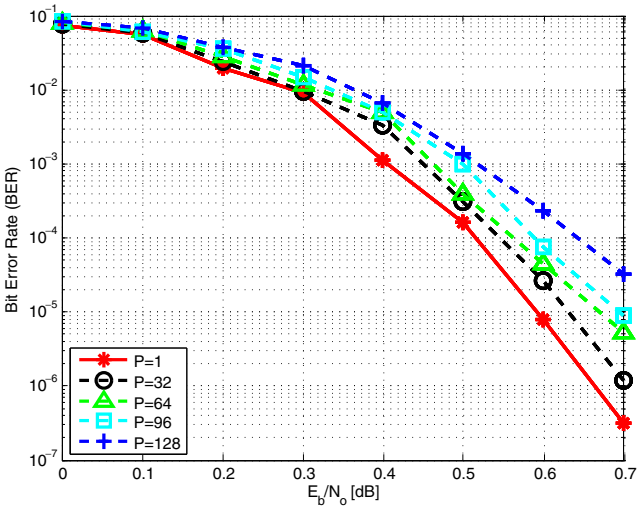


Figure 5 BER performance (BPSK, Full-log-MAP).

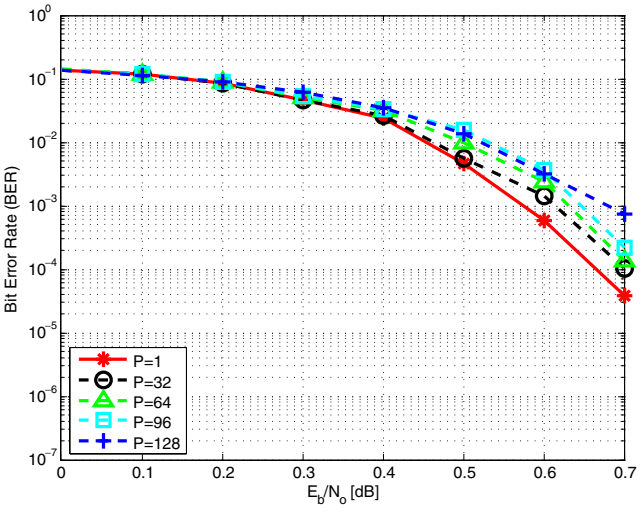


Figure 6 BER performance (BPSK, Max-log-MAP).

Table 3 Maximum decoder throughput.

Iteration	Max-log-MAP (Mbps)	Full-log-MAP (Mbps)
1	95.36	89.79
2	61.08	57.14
3	44.99	42.07
4	35.57	33.14
5	29.45	27.54
6	25.13	23.31
7	21.92	20.26
8	19.41	18.00
9	17.44	16.19

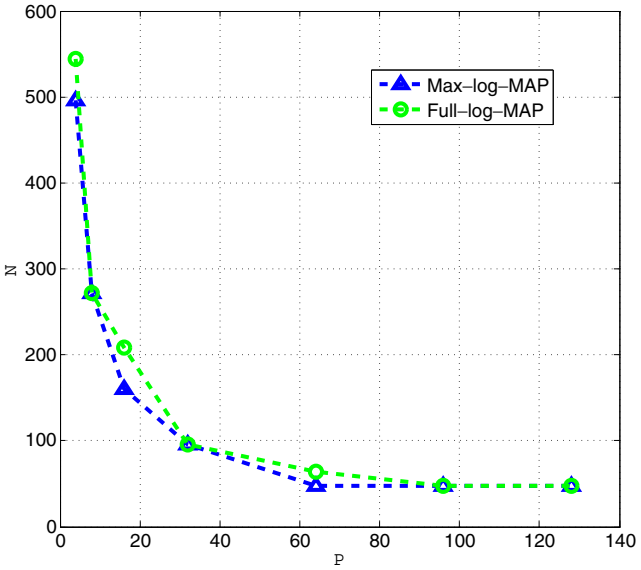


Figure 7 Number of codewords (N) versus number of sub-blocks (P).

5.3 Number of Sub-blocks vs. Number of Codewords

In the previous section, we found the maximum decoder throughput by feeding a very large workload into the decoder. For workloads with fewer codewords, P affects the throughput performance as P controls the number of thread-blocks spawned at runtime. A workload of N codewords will spawn $\frac{NP}{16}$ thread-blocks since each thread-block processes 16 sub-blocks for 16 codewords at the same time. As the GPU runs many concurrent threads to keep the SM busy and hide stalls through thread switching, we expect that larger P configurations, which spawn more thread-blocks per codeword, will require smaller N to approach maximum throughput.

To show how P affects the number of codewords required to achieve high throughput, we set a target throughput and vary N in steps of 32 for various values of P until the decoder's throughput exceeded the target throughput. In these experiments, we set $K = 6{,}144$, the number of decoding iterations to 5. For Max-log-MAP decoder, we set a target throughput of 27Mbps. Similarly, we set a target throughput of 24Mbps for Full-log-MAP.

As shown in Fig. 7, the trends are similar for both cases. As expected, as a larger P spawns more thread-blocks per codeword, larger P offers better throughput performance. There is a trade-off between decoding latency and error correction performance. Although larger P offers lower latency, larger P provides poorer error correction performance. Simulations show that the case of $P = 32$ seems to provides balanced performance. This particular configuration provides good error correction performance while requiring a reasonable size workload to achieve high throughput.

5.4 Throughput with Early Termination

To accelerate the decoding process, we implement early termination by using the average LLR rule according to Algorithm 4. The computation of average LLR is performed in the CUDA kernel. As the cost of the tag checking is small, tag checking is done in the host code. A simulation-based analysis is performed to determine a threshold value. In these simulations, the average LLR values are computed when the decoding process converges to the correct codeword. Based on the simulation results, a threshold of $T = 40$ is selected to guarantee that the BER is below 10^{-5}. To get better BER performance, a higher threshold T can be used.

Figure 8 shows the throughput results when the early termination scheme is employed. The maximum number of iterations is set to 16. As the SNR goes higher, the average number of iterations needed to reach the given BER level is reduced, so the decoding throughput is increased. The simulation results also show that the throughput for $E_b/N_0 = 0.5$ dB is higher than that for $E_b/N_0 = 0$ dB although their average number of iterations are the same (both are 16). This result matches our expectation for the tag-based early termination algorithm. As mentioned in Section 4.4, the tag-based early termination algorithm stops the decoding process for the already converged codeword, so even with the same average number of iterations the amount of computations is significantly reduced under these circumstances.

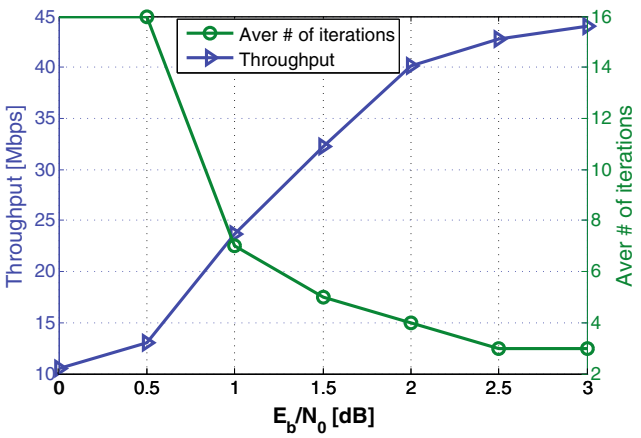


Figure 8 Throughput and average number of iterations when the early termination scheme is used.

5.5 Architecture Comparison

Table 4 compares our decoder with other programmable Turbo decoders. Compared to the general purpose processors and the multi-core DSP based solutions from [5, 24–27], our decoder with $P = 32$ compares favorably in terms of throughput and BER performance. For example, compared to a custom designed SIMD processor from [5], our solution shows both a flexibility advantage by supporting both Full-log-MAP and Max-log-MAP algorithms and a throughput advantage by supporting 15 times the data rate. This is expected as our device has significantly more computational resources than general purpose processors and multi-core DSPs. In addition, we can support both the Full-log-MAP algorithm and the sub-optimal Max-log-MAP algorithm while most other solutions only support the sub-optimal Max-log-MAP algorithm.

There are two recent papers on Turbo decoder on GPU [14, 15]. Both of these implementations try to increase computational throughput by reducing device memory accesses by saving α values in shared memory. However, the amount of shared memory per SM is limited. As a result, we need to divide a codeword into many sub-blocks to reduce the amount of shared memory required by each thread-block. Dividing a long codeword into many small sub-blocks improves throughput but reduces the error correction performance. An alternative is to divide a long codeword into a few sub-blocks. This requires a large amount of shared memory per thread-block. As a result, we cannot pack multiple sub-blocks in a thread-block and cannot have many concurrent threads to hide pipeline stalls, leading to significant horizontal and vertical waste which reduce decoder throughput. In [14], we kept the design to eight threads per thread-block, which supports sub-block length up to 192 stages. As the underlying instructions are 32 wide SIMD instructions, cores are used only at most $\frac{1}{4}$th of the time with this design.

In this paper, we took a more a more balanced approach to shared memory usage. Since α values are stored in device memory and fetched into shared memory when needed, shared memory is not a limitation and is not dependent on P. As a result, we can spawn more concurrent threads to hide stalls and pack multiple sub-blocks in a thread-block to meet the SIMD instruction width. In this paper, we pack multiple sub-blocks from 16 codewords onto the same thread-block. We have 128 threads per thread-block which can fully utilize the width of the SIMD instructions, minimizing vertical waste. As a result, our present solution is significantly faster while requiring fewer number of sub-blocks per codeword to achieve high performance.

To understand the impact of architecture change and code redesign between [14] and this paper, we benchmarked our original max-log-MAP decoder in [14] on Nvidia GTX470 for 5 decoding iterations. For P = 96, we achieved a throughput of 11.05 Mbps. Compared to the throughput performance of our original design on Telsa C1060, the improvement is approximately two times faster. The improvement is expected as there are 1.87 times more cores on Nvidia GTX470 and the introduction of L1 and L2 cache. The throughput performance of the proposed design on GTX470 is approximately 2.67 times faster than the original design on GTX470. This reflects the improvement we achieved with the redesign. Although our new design packs multiple sub-blocks to meet the SIMD instruction width, we do not achieve four times the throughput. This is due to two reasons. First, although we fully utilize the SIMD instruction width, the number of instructions needed is not four times smaller than the number of instructions needed in the original design. Compared to our original implementation, the number of instructions for our new design increase as extra load and store instructions are needed to move data between shared memory to device memory. Using the profiler, we noticed that the number of issued instructions of the new design is only 46.5% of the original design. Second, as data are fetched from device memory in the proposed implementation. There are cache misses which increase the execution time.

Table 4 Our GPU based decoder vs other programmable Turbo decoders.

Work	Architecture	MAP algorithm	Throughput	Iter.
[24]	Intel Pentium 3	Log-MAP and Max-log-MAP	366 Kbps/51 Kbps	1
[25]	Motorola 56603	Max-log-MAP	48.6 Kbps	5
[25]	STM VLIW DSP	Log-MAP	200 Kbps	5
[26]	TigerSHARC DSP	Max-log-MAP	2.399 Mbps	4
[27]	TMS320C6201 DSP	Max-log-MAP	500 Kbps	4
[5]	32-wide SIMD	Max-log-MAP	2.08 Mbps	5
[15]	Nvidia C1060	Max-log-MAP	2.1 Mbps	5
[14]	Nvidia C1060	Log-MAP and Max-log-MAP	6.77/5.2 Mbps	5
This work	Nvidia GTX470	Log-MAP and Max-log-MAP	29.45/27.54 Mbps	5

6 Conclusion

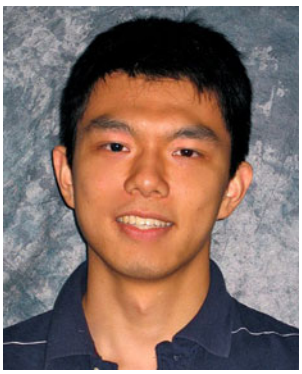
In this paper, we presented a 3GPP LTE compliant Turbo decoder implemented on GPU. We divide the workload across cores on the GPU by dividing the codeword into many sub-blocks to be decoded in parallel and by decoding multiple codewords at the same time. In addition, we improve efficiency by allowing a thread-block to decode multiple codeword at the same time. We use shared memory to speed up device memory access. However, we do not store all immediate data on-chip to increase the number of concurrently running threads. The implementation also ensures computation is completely parallel for each sub-block. As different sub-block sizes can lead to BER performance degradation, we presented how both BER performance and throughput are affected by sub-block size. We show that our decoder provides high throughput even though the Full-log-MAP algorithm is used. This work will enable the implementation of a high throughput decoder completely in software on a GPU.

Acknowledgements This work was supported in part by Renesas Mobile, Texas Instruments, Xilinx, and by the US National Science Foundation under grants CNS-0551692, CNS-0619767, EECS-0925942 and CNS-0923479.

References

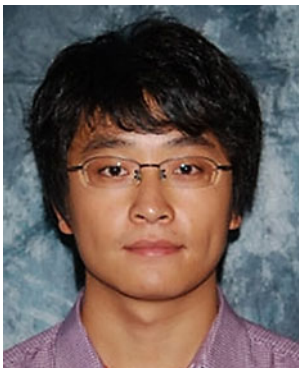
1. Berrou, C., Glavieux, A., & Thitimajshima, P. (1993). Near Shannon limit error-correcting coding and decoding: Turbo-codes. In *IEEE international conference on communication*.
2. Garrett, D., Xu, B., & Nicol, C. (2001). Energy efficient turbo decoding for 3G mobile. In *International symposium on low power electronics and design* (pp. 328–333). ACM.
3. Bickerstaff, M., Davis, L., Thomas, C., Garrett, D., & Nicol, C. (2003). A 24Mb/s Radix-4 LogMAP turbo decoder for 3GPP-HSDPA mobile wireless. In *IEEE Int. Solid-State Circuit Conf. (ISSCC)*.
4. Shin, M., & Park, I. (2007). SIMD Processor-based turbo decoder supporting multiple third-generation wireless standards. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 15*, 801–810.
5. Lin, Y., Mahlke, S., Mudge, T., Chakrabarti, C., Reid, A., & Flautner, K. (2006). Design and implementation of turbo decoders for software defined radio. In *IEEE workshop on signal processing design and implementation (SIPS)*.
6. Sun, Y., Zhu, Y., Goel, M., & Cavallaro, J. R. (2008). Configurable and scalable high throughput turbo decoder architecture for multiple 4G wireless standards. In *IEEE international conference on Application-Specific Systems, Architectures and Processors (ASAP)* (pp. 209–214).
7. Salmela, P., Sorokin, H., & Takala, J. (2008). A programmable Max-Log-MAP turbo decoder implementation. *Hindawi VLSI Design* (pp. 636–640).
8. Wong, C.-C., Lee, Y.-Y., & Chang, H.-C. (2009). A 188-size 2.1 mm^2 Reconfigurable turbo decoder chip with parallel architecture for 3GPP LTE system. In *Symposium on VLSI circuits* (pp. 288–289).
9. Amiri, K., Sun, Y., Murphy, P., Hunter, C., Cavallaro, J.R., & Sabharwal, A. (2007). WARP, a unified wireless network testbed for education and research. In *MSE '07: Proceedings of the 2007 IEEE international conference on microelectronic systems education*.
10. Kim, J., Hyeon, S., & Choi, S. (2010). Implementation of an SDR system using graphics processing unit. *IEEE Communications Magazine, 48*(3), 156–162.
11. Wu, M., Sun, Y., & Cavallaro, J. R. (2009). Reconfigurable real-time MIMO detector on GPU. In *IEEE 43rd Asilomar conference on signals, systems and computers (ASILOMAR'09)*.
12. Nylanden, T., Janhunen, J., Silvén, O., & Juntti, M. J. (2010). A GPU implementation for two MIMO-OFDM detectors. In *International conference on embedded computer systems (SAMOS)* (pp. 293–300).
13. Falcão, G., Silva, V., & Sousa, L. (2009). How GPUs can outperform ASICs for fast LDPC decoding. In *ICS '09: Proceedings of the 23rd international conference on supercomputing* (pp. 390–399).
14. Wu, M., Sun, Y., & Cavallaro, J. (2010). Implementation of a 3GPP LTE turbo decoder accelerator on GPU. In *Signal Processing Systems (SIPS)* (pp. 192–197).
15. Lee, D., Wolf, M., & Kim, H. (2010). Design space exploration of the turbo decoding algorithm on GPUs. In *International conference on compilers, architectures and synthesis for embedded systems* (pp. 214–226).
16. NVIDIA Corporation, CUDA compute unified device architecture programming guide (2008). Available: http://www.nvidia.com/object/cuda_develop.html
17. Bahl, L., Cocke, J., Jelinek, F., & Raviv, J. (1974). Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory, IT-20*, 284–287.
18. Naessens, F., Bougard, B., Bressinck, S., Hollevoet, L., Raghavan, P., der Perre, L. V., & Catthoor, F. (2008). A unified instruction set programmable architecture for multi-standard advanced forward error correction. In *IEEE workshop on Signal Processing Systems(SIPS)*.
19. Hagenauer, J., Offer, E., & Papke, L. (1996). Iterative decoding of binary block and convolutional codes. *IEEE Transactions on Information Theory, 42*, 429–445.
20. Shao, S. L. R., & Fossorier, M. (1996). Two simple stopping criteria for turbo decoding. *IEEE Transactions on Information Theory, 42*, 429–445.
21. Matache, A., Dolinar, S., & Pollara, F. (2000). Stopping rules for turbo decoders. In *JPL TMO Progress Report* (pp. 42–142).
22. Sun, J., & Takeshita, O. (2005). Interleavers for turbo codes using permutation polynomials over integer rings. *IEEE Transactions on Information Theory, 51*, 101–119.
23. Robertson, P., Villebrun, E., & Hoeher, P. (1995). A comparison of optimal and sub-optimal MAP decoding algorithm operating in the log domain. In *IEEE Int. Conf. Commun.* (pp. 1009–1013).
24. Valenti, M., & Sun, J. (2001). The UMTS turbo code and a efficient decoder implementation suitable for software-defined radios. *International Journal of Wireless Information Networks, 8*(4), 203–215.
25. Michel, H., Worm, A., Munch, M., & Wehn, N. (2002). Hardware software trade-offs for advanced 3G channel coding. In *Proceedings of design, automation and test in Europe*.

26. Loo, K., Alukaidey, T., & Jimaa, S. (2003). High performance parallelised 3GPP turbo decoder. In *IEEE personal mobile communications conference* (pp. 337–342).
27. Song, Y., Liu, G., & Yang, H. (2005). The implementation of turbo decoder on DSP in W-CDMA system. In *International conference on wireless communications, networking and mobile computing* (pp. 1281–1283).



Michael Wu received his B.S. degree from Franklin W. Olin College in May of 2007 and his M.S. degree from Rice University in May of 2010, both in Electrical and Computer Engineering. He is currently a Ph.D candidate in the E.C.E department at Rice University. His research interests are wireless algorithms, software defined radio on GPGPU and other parallel architectures, and high performance wireless receiver designs.

Yang Sun received the B.S. degree in Testing Technology & Instrumentation in 2000, and the M.S. degree in Instrument Science & Technology in 2003, both from Zhejiang University, Hangzhou, China. From 2003 to 2004, he worked at S3 Graphics Co. Ltd. as an ASIC design engineer, developing 3D Graphics Processors (GPU) for computers. From 2004 to 2005, he worked at Conexant Systems Inc. as an ASIC design engineer, developing Video Decoders for digital satellite-television set-top boxes (STBs). He is currently a Ph.D student in the Department of Electrical and Computer Engineering at Rice University, Houston, Texas. His research interests include parallel algorithms and VLSI architectures for wireless communication systems, especially forward-error correction (FEC) systems. He received the 2008 IEEE SoC Conference Best Paper Award, the 2008 IEEE Workshop on Signal Processing Systems Best Paper Award (Bob Owens Memory Paper Award), and the 2009 ACM GLSVLSI Best Student Paper Award.



Guohui Wang received his B.S. in Electrical Engineering from Peking University, Beijing, China, in 2005, and M.S. in Computer Science from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2008. Currently, he is a Ph.D. student in Department of Electrical and Computer Engineering, Rice University, Houston, Texas. His research interests include VLSI signal processing for wireless communication systems and parallel signal processing on GPGPU.

Joseph R. Cavallaro received the B.S. degree from the University of Pennsylvania, Philadelphia, Pa, in 1981, the M.S. degree from Princeton University, Princeton, NJ, in 1982, and the Ph.D. degree from Cornell University, Ithaca, NY, in 1988, all in electrical engineering. From 1981 to 1983, he was with AT&T Bell Laboratories, Holmdel, NJ. In 1988, he joined the faculty of Rice University, Houston, TX, where he is currently a Professor of electrical and computer engineering. His research interests include computer arithmetic, VLSI design and microlithography, and DSP and VLSI architectures for applications in wireless communications. During the 1996–1997 academic year, he served at the National Science Foundation as Director of the Prototyping Tools and Methodology Program. He was a Nokia Foundation Fellow and a Visiting Professor at the University of Oulu, Finland in 2005 and continues his affiliation there as an Adjunct Professor. He is currently the Director of the Center for Multimedia Communication at Rice University. He is a Senior Member of the IEEE. He was Co-chair of the 2004 Signal Processing for Communications Symposium at the IEEE Global Communications Conference and General/Program Co-chair of the 2003, 2004, and 2011 IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP).