A GPU Implementation of a MAP Decoder for Synchronization Error Correcting Codes

Johann A. Briffa

Abstract—In this paper we present a parallel implementation of a MAP decoder for synchronization error correcting codes. For a modest implementation effort, we demonstrate a considerable decoding speedup, up to two orders of magnitude even on consumer GPUs. This enables the analysis of much larger codes and worse channel conditions than previously possible, and makes applications of such codes feasible for software implementations.

Index Terms—Insertion-Deletion Correction, MAP Decoder, Forward-Backward Algorithm, CUDA, GPU

I. INTRODUCTION

While most coding schemes focus on the correction of substitution errors, the problem of correcting synchronization errors has seen a recent increase in interest [1]. A key development has been the concatenated scheme by Davey and MacKay [2], where a random binary marker sequence is used by the decoder to determine synchronization. A sparse representation of the message is then added to this marker sequence. We presented a maximum a-posteriori (MAP) decoder for a generalized construction in [3] with the same asymptotic complexity as the Davey-MacKay decoder. Improved encodings were later presented in [4]. A significant difficulty that remains for widespread use of these codes is their decoding complexity. MAP decoding is achievable but time-consuming, with complexity dependent on the channel conditions. Initial attempts at speeding up the decoding either result in sub-optimal decoding [2] or are based on a speed/memory trade-off [5]. In both cases the effect on decoding performance either was not analyzed in detail, or is only known for particular conditions. An alternative approach is to look at a parallelization of the MAP algorithm on a graphics processing unit (GPU) using CUDA [6].

Other MAP decoders have been implemented on GPUs; the closest to our work are for convolutional codes (as used in turbo codes) [7], [8] or for general hidden Markov models [9]. However, there are fundamental differences in our application which require a different approach. First, the state-space in turbo codes is fixed and generally very small (e.g. [8] uses an 8-state trellis). In our case, the state space can easily run into hundreds of states for poor channel conditions. Second, the cited work is for binary codes; our application requires a non-binary decoder, adding another complexity dimension. Finally, the branch metric for convolutional codes is trivial to compute,

so is generally recomputed as needed. In our case this requires a separate forward pass, and dominates complexity.

This paper starts with general notation and a system description in Section II, followed by a definition of the MAP decoder and its complexity in Section III. In Section IV we consider the problem of parallelizing the decoder and analyze the scalability and expected speedup of this solution in Section V. Practical results follow in Section VI for two GPU systems, with conclusions drawn in Section VII.

II. PRELIMINARIES

We use the channel model of [2]. At time t , one bit enters the channel, and one of three events may happen: insertion with probability P_i where a random bit is output; deletion with probability P_d where the input is discarded; or transmission with probability $P_t = 1 - P_i - P_d$. A substitution occurs in a transmitted bit with probability P_s . In the case of insertion the channel remains at time t and is subject to the same events again, otherwise we proceed to time $t + 1$, ready for another input bit. We define the *drift* S_t at time t as the difference between the number of transmitted bits and the number of received bits before the events of time t are considered.

For any sequence $\mathbf{z}$ we denote arbitrary subsequences as $\mathbf{z}_a^b = (z_a, \dots, z_{b-1})$, where $\mathbf{z}_a^a = ()$ is an empty sequence. The juxtaposition of $\mathbf{y}$ and $\mathbf{z}$ is denoted by $\mathbf{y}\|\mathbf{z}$. For $n, q, N \in \mathbb{N}$ we assume a message sequence $\mathbf{D}_0^N = (D_0, \dots, D_{N-1})$, where each symbol $D_i \in \mathbb{F}_q$ is mapped to a codeword of length $n \geq 2^q$ by an encoding $C_i : \mathbb{F}_q \hookrightarrow \mathbb{F}_2^n$ for $i = 0, \dots, N - 1$. Therefore the message $\mathbf{D}_0^N$ is encoded as $\mathbf{X}_0^\tau = C_0(D_0) \parallel \dots \parallel C_{N-1}(D_{N-1})$, where $\tau = nN$. Note that a different mapping may be used for each symbol index; in the Davey-MacKay scheme this is obtained as the addition of the sparse representation of D_i and the section of the random marker sequence corresponding to the symbol at index i . This is transmitted over the channel, resulting in the received sequence $\mathbf{Y}_0^\rho$, where in general $\rho \neq \tau$.

III. THE MAP DECODER

The MAP decoder of [3] calculates the *a posteriori* probability $L_i(D)$ of having encoded symbol $D \in \mathbb{F}_q$ in position i for $0 \leq i < N$, given the entire received sequence, using:

$$L_i(D) = \frac{1}{\lambda_N(\rho - \tau)} \sum_{m', m} \sigma_i(m', m, D) \quad (1)$$

$$\lambda_i(m) = \alpha_i(m) \beta_i(m) \quad (2)$$

$$\sigma_i(m', m, D) = \alpha_i(m') \gamma_i(m', m, D) \beta_{i+1}(m) \quad (3)$$

$$\alpha_i(m) = \sum_{m', D} \alpha_{i-1}(m') \gamma_{i-1}(m', m, D) \quad (4)$$

Manuscript received October 31, 2012; revised January 25, 2013. Parts of this research have been carried out using computational facilities procured through the European Regional Development Fund, Project ERDF-080.

J. A. Briffa is with the Dept. of Computing, University of Surrey, Guildford GU2 7XH, England. Email: j.briffa@surrey.ac.uk

$$\beta_i(m) = \sum_{m', D} \beta_{i+1}(m') \gamma_i(m, m', D) \quad (5)$$

$$\gamma_i(m', m, D) = \Pr\{D_i = D\} R(\mathbf{Y}_{ni+m'}^{n(i+1)+m} | C_i(D)) \quad (6)$$

where the prior probabilities are denoted by $\Pr\{D_i = D\}$. The state transition metric is calculated using

$$R(\hat{\mathbf{y}}|\mathbf{x}) = \dot{\alpha}_n(\dot{\mu} - n), \text{ where} \\ \dot{\alpha}_t(m) = \sum_{m'} \dot{\alpha}_{t-1}(m') \cdot Q(\hat{\mathbf{y}}_{t-1+m'}^{t+m} | x_{t-1}), \quad (7)$$

$\dot{\mu}$ is the length of $\hat{\mathbf{y}}$, n is the length of $\mathbf{x}$, and $Q(\mathbf{y}|x)$ can be directly computed from $\mathbf{y}$, x and the channel parameters:

$$Q(\mathbf{y}|x) = \begin{cases} P_d & \text{if } \mu = 0 \\ \left(\frac{P_s}{2}\right)^{\mu-1} (P_t P_s + \frac{1}{2} P_i P_d) & \text{if } \mu > 0, y_{\mu-1} \neq x \\ \left(\frac{P_s}{2}\right)^{\mu-1} (P_t \bar{P}_s + \frac{1}{2} P_i P_d) & \text{if } \mu > 0, y_{\mu-1} = x, \end{cases}$$

where μ is the length of $\mathbf{y}$ and $\bar{P}_s = 1 - P_s$.

Since the set of all possible states is unbounded for the channel considered, a practical implementation has to take sums over a finite subset, chosen so that only the least likely states are omitted. We denote the number of states required for a segment of length T bits by M_T . Therefore, computations (1), (4), and (5) have a state space of size M_T while computation (7) has a state space of size M_n . Similarly, the number of drift changes considered over a single bit depends on the number of consecutive insertions, and is denoted by I_T for a segment of length T bits. This limits the connectivity of consecutive states. The precise determination of the size of the state space is beyond the scope of this paper, and is left for a separate work.

The α and β computations as given in (4) and (5) have a very wide numerical range. However, the decoder needs only the relative values of $L_i(D)$ for different D , so we can safely normalize the α and β metrics as they are computed. For example, for α the computation (4) is changed to:

$$\alpha_i(m) = \frac{\alpha'_i(m)}{\sum_{m'} \alpha'_i(m')} \quad (8)$$

$$\alpha'_i(m) = \sum_{m', D} \alpha_{i-1}(m') \gamma_{i-1}(m', m, D) \quad (9)$$

A similar argument applies for the computation of β and $\dot{\alpha}$.

For typical codeword sizes and channel conditions it is sufficient to compute (7) at single precision (`float`). The use of double precision (`double`) is indicated for the remaining equations, particularly for large N and poor channel conditions. This has been verified by simulation, comparing the decoding error rate with a reference multi-precision implementation [10]. Therefore `float` performance has a dominant effect on computing the γ metric, while `double` performance is dominant for the remaining metrics.

The asymptotic complexity of the decoder is given by $O(NnqM_\tau M_n^2 I_n)$, where N , n , and q depend only on the code parameters while M_τ , M_n , and I_n also depend on the channel conditions [3]. It was argued in [2] that capping the number of consecutive insertions to two causes minimal loss of decoding performance; the same cap was also used in [3], [11]. However, more advanced constructions require a higher cap or lifting the restriction [4].

IV. PARALLEL IMPLEMENTATION

The MAP decoder consists of four functions, one for computing each of the γ , α , β , and L matrices. These depend on each other, dictating the order of computation. We follow the usual CUDA notation, where a *block* is the collection of threads executing on the same multiprocessor, and a *grid* is the set of equally-shaped blocks in a kernel call.

Starting with the γ computation (6), which needs to be performed first, it can be seen that computation is independent for each of i , D , m' , m . This facilitates a data-parallel implementation across any of these variables. For the expected ranges of i and D it is natural to consider using a grid size N and block size q . We store the four-dimensional γ matrix as a flat array in global memory, with indices m' and $m - m'$ innermost. This allows each thread to access a contiguous range of memory sequentially, while separate threads access regions that are far away from each other. This maximizes cache re-use as long as the number of concurrent threads is not greater than the number of cache lines.

The α and β computations pose greater difficulty due to their recursive nature and the need to normalize. Considering the pre-normalization computation (9), there is a clear data dependency across the range of i ; however, for any given i the computation is independent for different values of m . The expected range of m depends on N , n , and channel error rate, and can be as high as several hundred. This makes m a natural candidate for parallelization across blocks, for a grid size of M_τ . Further, the summation over D can be separated across threads, with the partial summation over m' computed independently for each D , and the final result computed from these partial sums. This gives a block size of q .

Due to data dependency, for a given i all $\alpha'_i(m)$ must be computed before $\alpha_i(m)$ can be determined. The only way to synchronize across a grid is the completion of a kernel call [6], so a separate call is required for each i to compute (9). After each call, a separate kernel is required to perform normalization (8). This requires two steps: computing the sum of all α'_i , and dividing each α'_i by this sum. Both can be parallelized across a single block of M_τ threads.

A similar argument applies to the computation of β . Further, α and β can be computed concurrently as there is no data dependency. On devices supporting concurrent kernel execution, this can be achieved using streams, but is complicated by Fermi hardware limitations. A new kernel is only dispatched if preceding kernels in the same stream have completed. Since Fermi has only one compute engine queue, if any stream has more than one kernel scheduled consecutively, the issuer will stall until the last kernel in the sequence is dispatched. The kernels for α and β at each index have the same complexity, so a successful strategy is as follows: schedule the computation of $\alpha_{i=0}$ in stream one and of $\beta_{i=N}$ in stream two, followed by the normalization of $\alpha_{i=0}$ in stream one and of $\beta_{i=N}$ in stream two. This is repeated, incrementing i for α and decrementing for β . Concurrent execution improves device utilization when the grid size for a single kernel call is small.

Finally, the L computation (1) is independent across i , D . Similarly to the computation of γ , we parallelize this with a

Table I
SUMMARY OF KERNELS USED IN PARALLEL IMPLEMENTATION.

Kernel	Grid size	Block size	Calls	Complexity
Compute γ	N	q	1	$O(nM_nI_n)$
Compute α, β	M_τ	q	$N + 1$	$O(M_n)$
Normalize α, β	1	M_τ	$N + 1$	$O(M_\tau)$
Compute L	N	q	1	$O(M_\tau M_n)$

grid size N and block size q .

V. ANALYSIS

A summary of the kernels used is given in Table I, where it can be seen that the block size is equal to q or M_τ . Now the device architecture imposes a hardware limit on the maximum block size [6], limiting the alphabet sizes and state space supported by our implementation. Current code constructions fall well within these limitations, as we will see in Section VI. We also list in the table the complexity of computations for a single thread, assuming $M_n \gg I_n$.

The effectiveness of the parallelization depends on the utilization at various levels. At device level, utilization for a single kernel depends on the grid size: ideally this is a multiple of N_{SM} , the number of streaming multiprocessors (SM). At a SM level, determining the utilization is rather more complicated. Threads in a block are grouped into warps, and instructions are issued at warp level: ideally the block size should be a multiple of the warp size W . However, maximizing throughput also depends on having enough resident warps, which is limited by the compute capability and by the number of registers required per thread. The number of arithmetic operations that can be executed in parallel depends on the number of cores per SM, N_C , and on the precision. Furthermore, global memory access has a high latency, so minimizing data transfers between global memory and the SM is important. It is also important to organize any global memory access into optimal patterns to make use of memory access coalescing.

For the problem considered, the parallelization efficiency depends on the device specifications and the code parameters; for some kernels it also depends on the channel conditions. Consider first the γ computation, which has the highest overall computational cost. Device usage is optimal when N is a multiple of N_{SM} , and close to optimal for $N \gg N_{\text{SM}}$. Under these conditions, we get an ideal speedup equal to N_{SM} . When $N < N_{\text{SM}}$, the gain is equal to N . For `float`, the arithmetic throughput is equal to N_C operations per clock cycle per SM. Ideally, therefore, the block size q is a multiple of both N_C and W . Under these conditions, and if latency is completely hidden, we could expect a speedup equal to N_C . For $q < N_C$, we could expect at best a speedup equal to q , and only under ideal conditions. A similar argument can be applied for the computation of L , except for the use of `double` and a lower potential throughput. Arguments for the computation of α and β are further complicated by the kernel call overhead.

VI. RESULTS

We determine GPU performance on two Fermi consumer devices. CPU timings are given for a serial implementation,

Table II
HARDWARE SPECIFICATIONS FOR CPU AND GPU SYSTEMS.

	Opteron 2431	GTX 480	GT 520
Processors $\times$ cores	2×6	15×32	1×48
Core Speed (GHz)	2.412	1.401	1.62
Memory (MiB)	32 768	1 536	1 024
Peak GFLOPs (<code>float</code>)	2.412 (scalar)	672.48	77.76
Peak GFLOPs (<code>double</code>)	2.412 (scalar)	336.24	6.48

Figure 1. (a) Time to compute γ for $q = 32$ and $p = 10^{-3}$ over a range of N , normalized by a factor $NqnM_nI_n$, and (b) ratio of timings on CPU as compared with GTX 480 and GT 520.

used as a reference for validation of results and speedup comparisons. Hardware specifications are summarized in Table II. The choice of codebook has no impact on the decoding speed, so we always use the construction of [2]. However, we do not cap the number of consecutive insertions and do not perform any path truncation. This considers decoder complexity under worst-case conditions, as needed for more advanced constructions. The SM cache has been kept at the default configuration, where on-chip memory is allocated as 48 KiB of shared memory and 16 KiB of L1 cache. This is equivalent to 128 lines of cache for each SM.

We consider first the time to compute the γ metric over a range of N , for fixed q and channel conditions $p := P_i = P_d$. To analyze the effect on parallelization efficiency, we plot the computation time normalized by the expected complexity in Figure 1a. Note that the normalized CPU time is not constant as one would expect. This indicates that the expected complexity is missing some details. This is not surprising, as the complexity expressions consider only the floating-point arithmetic, ignoring the computational overhead of loop handling or memory access. For the GPU timings, note that maximum efficiency is achieved at $N = 8$ for the GT 520 and close to $N = 100$ for the GTX 480. This means that as the grid size increases beyond the number of SMs on the device, performance continues to improve as the additional blocks allow the device to hide latency more efficiently, until we reach

Figure 2. (a) Time to compute γ for $N = 300$ and $p = 10^{-3}$ over a range of alphabet sizes, normalized by a factor $NqnM_nI_n$, and (b) ratio of timings on CPU as compared with GTX480 and GT520.

eight blocks per SM, the maximum number of resident blocks.

The GPU to CPU speedup under the same conditions is shown in Figure 1b, together with the peak speedup for a compute-bound problem on each device. On the GTX480 for $N \geq 100$ the speedup tops out at more than 100 $\times$ for $p = 10^{-4}$; this compares favourably with the theoretical peak of 278.8 $\times$. At a higher $p = 10^{-2}$ the speedup decreases to around 70 $\times$, indicating a loss of efficiency for larger state spaces. This is most likely due to the increased access to global memory. The GT520 peak performance is achieved at $N = 8$, dropping at $N = 9$, and increasing up to $N = 16$; this corresponds to under-utilization of the device in the last batch, as expected. It is interesting to note that even such a low-end device can achieve a real performance gain of almost 10 $\times$, which compares well with the theoretical peak of 32.2 $\times$.

We next vary q for fixed N and channel conditions, as shown in Figure 2a. Again, the CPU time increases worse than the complexity expression anticipates. The GTX480 reaches maximum efficiency at $q = 128$, equivalent to four warps per block, despite a grid size equal to twenty blocks per SM. The larger block size leads to an increased occupancy and better latency hiding. Consistent with this, a similar analysis for $N = 15$ shows performance improvement beyond $q = 128$.

The corresponding GPU to CPU speedup is shown in Figure 2b. As expected the GPU speedup increases with alphabet size; this is useful, as for a codebook of size (n, q) compute time increases linearly in n and q while for a given code rate q increases exponentially with n . Results for the complete frame decoding time are consistent with those for γ , which is known to dominate the decoding time.

VII. CONCLUSIONS

We have presented a parallel implementation of a MAP decoder for synchronization error correcting codes, resulting in a speedup of around 100 $\times$. This allows us to analyse larger

codes and worse channel conditions than previously possible, and makes applications of these codes more feasible.

Results are given for the Fermi architecture; Kepler [12] devices have more cores per multiprocessor, usually at a lower clock rate. However, double throughput per multiprocessor for current consumer devices has not increased in comparison with Fermi devices. Per multiprocessor, on such devices we expect to see a considerable performance improvement at single-precision, as long as the device is filled, and some drop in performance at double-precision (due to the slower clock). The point at which the device is fully utilized for single-precision is expected to occur at larger N and q .

A number of aspects of this parallelization bear further analysis. It may be possible to use shared memory to reduce global memory transfers, improving performance. One may also improve performance for small q (and large N) by computing multiple indexes in a single block. At the other end, the current limit on q can be overcome by splitting the alphabet over different blocks. Our implementation assumes that there is sufficient memory to pre-compute the γ metric; this can be overcome by computing γ incrementally as needed by the computations for α , β , and L . This impacts performance as each γ metric needs to be computed more than once. All this is the subject of further work.

ACKNOWLEDGMENT

The author would like to thank Prof. Ing. V. Buttigieg and Dr S. Wesemeyer for helpful discussions and comments.

REFERENCES

- [1] H. Mercier, V. Bhargava, and V. Tarokh, "A survey of error-correcting codes for channels with symbol synchronization errors," *IEEE Commun. Surveys Tuts.*, vol. 12, no. 1, pp. 87–96, First Quarter 2010.
- [2] M. C. Davey and D. J. C. MacKay, "Reliable communication over channels with insertions, deletions, and substitutions," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 687–698, 2001.
- [3] J. A. Briffa, H. G. Schaathun, and S. Wesemeyer, "An improved decoding algorithm for the Davey-MacKay construction," in *Proc. IEEE Intern. Conf. on Commun.*, Cape Town, South Africa, May 23–27, 2010.
- [4] V. Buttigieg and J. A. Briffa, "Codebook and marker sequence design for synchronization-correcting codes," in *Proc. IEEE Intern. Symp. Inform. Theory*, St. Petersburg, Russia, Jul. 31–Aug. 5, 2011.
- [5] X. Jiao and M. A. Armand, "Interleaved LDPC codes, reduced-complexity inner decoder and an iterative decoder for the Davey-MacKay construction," in *Proc. IEEE Intern. Symp. Inform. Theory*, St. Petersburg, Russia, Jul. 31–Aug. 5, 2011, pp. 742–746.
- [6] *NVIDIA CUDA C Programming Guide*, NVIDIA Corporation, Oct. 2012, version 5.0.
- [7] D. Lee, M. Wolf, and H. Kim, "Design space exploration of the turbo decoding algorithm on GPUs," in *Proc. Intern. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, 2010, pp. 217–226.
- [8] M. Wu, Y. Sun, G. Wang, and J. Cavallaro, "Implementation of a high throughput 3GPP turbo decoder on GPU," *Journal of Signal Processing Systems*, vol. 65, no. 2, pp. 171–183, 2011.
- [9] J. Li, S. Chen, and Y. Li, "The fast evaluation of hidden Markov models on GPU," in *IEEE Intern. Conf. on Intelligent Computing and Intelligent Systems (ICIS)*, 2009, vol. 4, Nov. 2009, pp. 426–430.
- [10] T. Granlund, *GNU MP: The GNU Multiple Precision Arithmetic Library*, Free Software Foundation, Dec. 2012, edition 5.1.0. [Online]. Available: <http://gmplib.org/gmp-man-5.1.0.pdf>
- [11] J. A. Briffa and H. G. Schaathun, "Improvement of the Davey-MacKay construction," in *Proc. IEEE Intern. Symp. on Inform. Theory and its Applications*, Auckland, New Zealand, Dec. 7–10, 2008, pp. 235–238.
- [12] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, NVIDIA Corporation, 2012, version 1.0.