Graphics processing unit implementation and optimisation of a flexible maximum a-posteriori decoder for synchronisation correction

Johann A. Briffa

Department of Computing, University of Surrey, Guildford GU2 7XH, England
E-mail: J.Briffa@surrey.ac.uk

Abstract: In this paper, the author presents an optimised parallel implementation of a flexible maximum a-posteriori decoder for synchronisation error correcting codes, supporting a very wide range of code sizes and channel conditions. On mid-range GPUs the author demonstrates decoding speedups of more than two orders of magnitude over a central processing unit implementation of the same optimised algorithm, and more than an order of magnitude over the author's earlier GPU implementation. The prominent challenge is to maintain high parallelisation efficiency over a wide range of code sizes and channel conditions, and different execution hardware. The author ensures this with a dynamic strategy for choosing parallel execution parameters at run-time. They also present a variant that trades off some decoding speed for significantly reduced memory requirement, with no loss to the decoder's error correction performance. The increased throughput of their implementation and its ability to work with less memory allow us to analyse larger codes and poorer channel conditions, and makes practical use of such codes more feasible.

1 Introduction

The problem of correcting synchronisation errors has recently seen an increase in interest [1]. We believe this is because of two factors: recent applications for such codes, where traditional techniques for synchronisation cannot be applied, and the feasibility of decoding because of improvements in computing resources. A recent application is for bit-patterned media [2, 3], where written-in errors can be modelled as synchronisation errors. Bit-patterned media is of great interest to the magnetic recording industry because of the potential increase in writing density. Another example is robust digital watermarking, where a message is embedded into a media file and an attacker seeks to make the message unreadable. An effective attack is to cause loss of synchronisation; synchronisation-correcting codes have been successfully applied to resist these attacks in speech [4] and image [5] watermarking.

Most practical decoders for synchronisation correction work by extending the state space of the underlying code to account for the state of the channel (which represents the synchronisation error). This increases the decoding complexity significantly, particularly under poor channel conditions where the state space is necessarily larger. Although optimal decoding is achievable, the complexity involved remains a barrier for wider adoption. The problem is even more pronounced when these codes are part of an iteratively decoded construction.

A key practical synchronisation-correcting scheme is the concatenated construction by Davey and MacKay [6], where the inner code tracks synchronisation on an unbounded random insertion and deletion channel. We presented a maximum a-posteriori (MAP) decoder for a generalised construction of the inner code in [7] and improved encodings in [8]. In [9], we presented a parallel implementation of our maximum a-posteriori (MAP) decoder on a graphics processing unit (GPU) using NVIDIA's Compute Unified Device Architecture (CUDA) [10]. This resulted in a decoding speedup of up to two orders of magnitude, depending on code parameters and channel conditions. Since that work we have also presented a number of additional improvements to the MAP decoder algorithm [11], resulting in a speedup of over an order of magnitude in a serial implementation, as we shall show. Unfortunately, these algorithmic improvements change the proportion of time spent computing the various equations, so that a straightforward application of the algorithm improvements to our

earlier GPU implementation does not yield the expected speedup. A more careful parallelisation strategy is required, which we discuss in this paper.

Other GPU implementations of the forward–backward algorithm (on which our MAP decoder is based) have significant fundamental differences from this paper. Recent work on turbo decoders [12–14] is limited to the parallel implementation of specific binary code constructions. Furthermore, turbo decoders work with a small state space of a fixed size, and the branch metric is trivial to compute, so is generally recomputed as needed. These factors simplify the problem considerably, making it much easier to optimise a parallel implementation. In contrast, we consider the problem of efficiently parallelising a flexible MAP decoder implementation, where: (a) the code size is variable, (b) codes are non-binary and the implementation works with variable alphabet size, (c) the state space is variable in size, depending on channel conditions, and can easily run into hundreds of states for poor channels and (d) the branch metric computation requires a lattice traversal and represents a significant fraction of the overall complexity. Owing to these variables, hard optimisations cannot be done, so our challenge is to plan things to work well with suitable run-time decisions. Since various optimisation decisions are taken at run-time, this also allows us to automatically cater for different hardware.

In this paper, we present an optimised parallel implementation of the MAP decoder with the algorithmic improvements of [11]. Two variants of this algorithm are implemented: a direct implementation where intermediate metrics are stored for the decoding of a whole frame, and a reduced-memory implementation where some intermediate metrics are recalculated for the backward pass. This considerably reduces the memory footprint of the decoder, which is a particularly important consideration for a GPU implementation, at the cost of decoding speed. In optimising the implementation, we consider the use of GPU on-chip memory to reduce memory transfers and to improve access patterns. We also introduce a dynamic strategy for choosing how each function is parallelised, including the number of threads to use, in order to optimise the efficiency and usage of the GPU compute cores.

The approach presented here can be generalised to other implementations of the forward–backward algorithm, such as those used in turbo decoding and in applications of hidden Markov models. The techniques we propose here are particularly useful in

similarly flexible implementations, commonly found in simulators and other research tools. In addition, we discuss a number of techniques for handling device limitations and optimising the parallelisation parameters dynamically; these may be relevant to other researchers working with CUDA.

This paper starts with definitions and a summary of earlier work in Section 2, including descriptions of the coding scheme, channel and the MAP decoder algorithm. Our contributions start in Section 3 where we consider the challenges for an efficient parallel implementation of the MAP decoder. This is followed with an initial parallel implementation and a reduced-memory variant in Section 4. In Section 5 we ensure that the parallel decoder works within hardware constraints and improve efficiency when code parameters are less than ideal. We analyse the practical performance on three GPU systems in Section 7, followed by final results in Section 7 and conclusions in Section 8.

2 Preliminaries

2.1 Coding scheme

In this paper, we are concerned with the coding scheme of [9, 11], which we summarise below. The encoding is defined by the sequence $\mathcal{C} = (C_0, \ldots, C_{N-1})$, which consists of the constituent encodings $C_i : \mathbb{F}_q \hookrightarrow \mathbb{F}_2^n$ for $i = 0, \ldots, N-1$, where n, q, $N \in \mathbb{N}$, $2^n \geq q$ and $\hookrightarrow$ denotes an injective mapping. For any sequence z, denote arbitrary subsequences as $z_a^b = (z_a, \ldots, z_{b-1})$, where $z_a^a = ()$ is an empty sequence. Given a message $\boldsymbol{D}_0^N = (D_0, \ldots, D_{N-1})$, each C_i maps the q-ary message symbol $D_i \in \mathbb{F}_q$ to codeword $C_i(D_i)$ of length n. That is, $\boldsymbol{D}_0^N$ is encoded as $\boldsymbol{X}_0^{nN} = C_0(D_0) \parallel \cdots \parallel C_{N-1}(D_{N-1})$, where $\boldsymbol{y} \parallel \boldsymbol{z}$ is the juxtaposition of $\boldsymbol{y}$ and $\boldsymbol{z}$.

The above encoding is normally used as an inner code to correct synchronisation errors, serially concatenated with a conventional outer code to correct residual substitution errors. In such a construction, the MAP decoder's posterior probabilities are used to initialise the outer decoder. Such a construction can be iteratively decoded by setting the prior symbol probabilities of the MAP decoder with extrinsic information from the previous pass of the outer decoder.

2.2 Channel model

As in [9, 11], we consider the binary substitution, insertion and deletion (BSID) channel, an abstract random channel with unbounded synchronisation and substitution errors. This channel was originally presented in [15] and more recently used in [6–8, 16, 17] and others. 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. After an insertion, the channel remains at time t and is subject to the same events again; otherwise it proceeds 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 received bits and the number of transmitted bits before the events of time t are considered. As in [6], the channel can be seen as a Markov process with the state being the drift S_t. It is helpful to see the sequence of states as a trellis diagram, observing that there may be more than one way to achieve each state transition.

2.3 MAP decoder

We summarise here the optimised MAP decoder of [11], which we are concerned with parallelising. The decoder uses the standard forward–backward algorithm for hidden Markov models. We assume a message sequence $\boldsymbol{D}_0^N$, encoded to the sequence $\boldsymbol{X}_0^\tau$, where $\tau = nN$. The sequence $\boldsymbol{X}_0^\tau$ is transmitted over the BSID channel, resulting in the received sequence $\boldsymbol{Y}_0^\rho$, where, in general, ρ is not equal to τ. To avoid ambiguity, we refer to the message

sequence as a 'message' of size N and the encoded sequence as a frame of size τ. We calculate the APP $L_i(D)$ of having encoded symbol $D \in \mathbb{F}_q$ in position i for $0 \leq i < N$, given the entire received sequence, using

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

$$\text{where } \lambda_i(m) = \alpha_i(m)\beta_i(m) \tag{2}$$

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

and $\alpha_i(m)$, $\beta_i(m)$ and $\gamma_i(m', m, D)$ are the forward, backward and state transition metrics, respectively. Note that strictly, the above metrics depend on $\boldsymbol{Y}_0^\rho$, but for brevity we do not indicate this dependence in the notation. The summation in (1) is taken over the combination of m', m, being, respectively, the drift before and after the symbol at index i. The forward and backward metrics are obtained recursively using

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

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

Initial conditions are given by $\alpha_0(m)$ and $\beta_N(m)$, set as the prior probabilities of the frame boundaries. Finally, the state transition metric is defined as

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

where $C_i(D)$ is the n-bit sequence encoding D and $R(\dot{\boldsymbol{y}}|\boldsymbol{x})$ is the probability of receiving a sequence $\dot{\boldsymbol{y}}$ given that $\boldsymbol{x}$ was sent through the channel (we refer to this as the receiver metric). The a priori probability $\Pr\{D_i = D\}$ is determined by the source statistics, which we generally assume to be equiprobable so that $\Pr\{D_i = D\} = 1/q$. In iterative decoding, the prior probabilities are set using extrinsic information from the previous pass of an outer decoder.

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. For a transmitted segment of length T bits, we denote the range of states considered by the upper and lower limits m_T^+, m_T^-, respectively, for a state space of size $M_T = m_T^+ - m_T^- + 1$. Therefore the α and β metrics are defined over a state space of size $M_\tau = m_\tau^+ - m_\tau^- + 1$, whereas the γ metric is defined over a state space of size M_τ for the initial drift m' and a state space of size $M_n = m_n^+ - m_n^- + 1$ for the drift change $m - m'$. The precise determination of the size of the state space is considered in [11].

The receiver metric $R(\dot{\boldsymbol{y}}|\boldsymbol{x})$ is computed using a recursion over a lattice as follows. The required lattice has $n + 1$ rows and $\dot{\mu} + 1$ columns, where $\dot{\mu}$ is the length of $\dot{\boldsymbol{y}}$ and n is the length of $\boldsymbol{x}$. Each horizontal path represents an insertion with probability $P_i/2$, each vertical path is a deletion with probability P_d, whereas each diagonal path is a transmission with probability $P_t P_s$ if the corresponding elements from $\boldsymbol{x}$ and $\dot{\boldsymbol{y}}$ are different or $P_t(1 - P_s)$ if they are the same. Let $F_{i, j}$ represent the lattice node in row i and column j. Then the lattice computation in the general case is defined by the recursion

$$F_{i,j} = \frac{1}{2} P_i F_{i,j-1} + P_d F_{i-1,j} + \dot{Q}(\dot{y}_j | x_i) F_{i-1,j-1} \tag{7}$$

which is valid for $i < n$ and where $\dot{Q}(y|x)$ can be directly computed

from y, x and the channel parameters:

$$\dot{Q}(y|x) = \begin{cases} P_{\mathrm{t}}P_{\mathrm{s}}, & \text{if } y \neq x \\ P_{\mathrm{t}}(1 - P_{\mathrm{s}}), & \text{if } y = x \end{cases} \qquad (8)$$

Initial conditions are given by

$$F_{i,j} = \begin{cases} 1, & \text{if } i = 0, \; j = 0 \\ 0, & \text{if } i < 0 \text{ or } j < 0 \end{cases} \qquad (9)$$

The last row is computed differently as the channel model does not allow the last event to be an insertion. In this case, when $i = n$, the lattice computation is defined by

$$F_{n,j} = P_{\mathrm{d}}F_{n-1,j} + \dot{Q}(\dot{y}_j \,|x_n)F_{n-1,j-1} \qquad (10)$$

Finally, the required receiver metric is obtained from this computation as $R(\dot{y}|\boldsymbol{x}) = F_{n,\mu}$.

Observe that for a given $\boldsymbol{x}$, the receiver metric $R(\dot{y}|\boldsymbol{x})$ needs to be determined for all subsequences $\dot{y}$ within the drift limit considered. Therefore, for a given symbol D and initial drift m' in (6), the lattice computation is only done once for the largest drift change $m - m'$ that needs to be considered. The required values for the remaining values of m are then also available in the last row of the lattice.

Note also that the horizontal distance of a lattice node from the main diagonal is equivalent to the channel drift for the corresponding transmitted bit. For the transmitted sequence of n bits considered, we can take advantage of this by limiting the lattice computation to paths within a fixed corridor of width M_n around the main diagonal.

Finally, note that the α and β metrics are normalised as they are computed to avoid exceeding the limits of floating-point representation. Specifically, for α the computation (4) is changed to

$$\alpha_i(m) = \frac{\alpha_i'(m)}{\sum_{m'} \alpha_i'(m')} \qquad (11)$$

where $\alpha_i'(m) = \sum_{m', D} \alpha_{i-1}(m')\gamma_{i-1}(m', \; m, \; D) \qquad (12)$

A similar argument applies for the computation of β. In addition, the receiver metric is computed at single precision (i.e. 32-bit floating point) whereas the remaining equations use double precision (i.e. 64-bit floating point).

2.4 CUDA notation

In this paper, we follow the usual CUDA notation, which we summarise here for convenience. For further detail, the reader is referred to [10]. CUDA defines a general-purpose parallel programming model for a hybrid system with a 'host' central processing unit (CPU) and an attached GPU 'device' (or more than one). The device contains the GPU chip, organised as a number of 'multiprocessors' with a fixed number of compute cores each and off-chip memory. Each multiprocessor also contains a fixed amount of on-chip 'shared' memory, accessible by all compute cores in the multiprocessor. Off-chip memory is accessible by all GPU threads through 'global' memory variables in read/write mode or in read-only mode through 'constant' or 'texture' memory constructs.

Every function executed on the GPU is called a 'kernel'; this is run as a 'grid' of equally shaped 'blocks' of parallel threads, as specified by the execution configuration. To avoid ambiguity, we avoid using the term 'block' for any other purpose. Each block of threads executes on the same multiprocessor in groups of threads called 'warps'. Threads in a given warp start execution at the same

address but are free to branch and execute independently (i.e. 'diverge'). However, highest efficiency is achieved when there is no divergence within a warp. Note that more than one block may be 'resident' in a given multiprocessor if sufficient resources (i.e. registers and shared memory) are available. This increases the number of warps available to the scheduler, and may be used to hide latency.

3 Challenges for parallel implementation

3.1 Effect of changes to receiver metric computation

As compared with our previous GPU implementation, the optimised decoder summarised in Section 2.3 changes to the way the receiver metric is computed. Of particular relevance to this paper, in the optimised decoder (a) for a given $\boldsymbol{x}$, we simultaneously compute $R(\dot{y}|\boldsymbol{x})$ for all subsequences of $\dot{y}$ and (b) we replace the trellis-based forward pass of [7] with a more efficient corridor-constrained lattice implementation. Together, these changes result in a decrease in complexity (for computing the receiver metric) of up to three orders of magnitude, depending on channel conditions.

To demonstrate the effect on the overall decoding speed, we repeat the simulations of [9] using a serial CPU implementation of the improved decoder. Results comparing the overall decoding speed, for the same codes and on the same computer, are shown in Fig. 1. Under moderate channel conditions, it can be seen that the improved decoder is more than ten times faster over a wide range of message sizes N and alphabet sizes q. Speedup improves for poorer channel conditions.

Such a significant change means that the receiver metric computation can no longer be considered to dominate the overall complexity. This is particularly so for a GPU implementation, where we expect a higher speedup in computing the γ metric since this has greater data parallelism. Therefore in parallelising this improved algorithm, we also have to pay particular attention to an efficient parallel computation of the α and β metrics, which have internal dependencies.
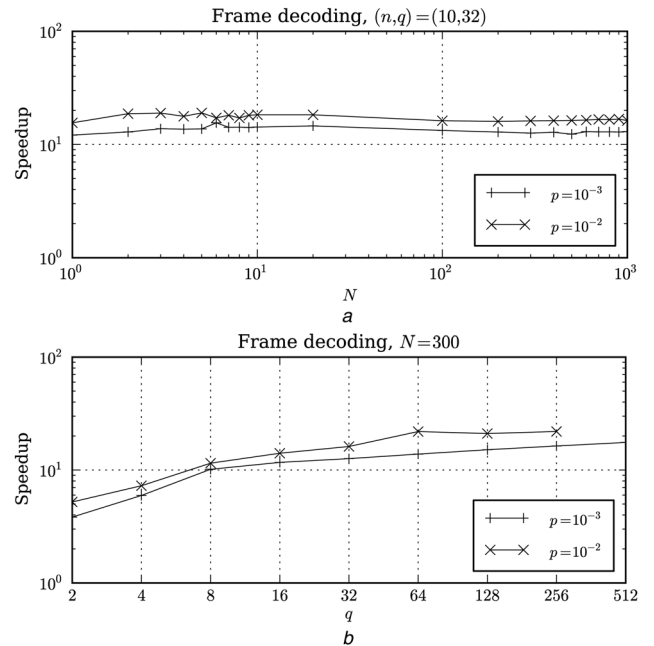

Fig. 1 *Decoding speedup of a serial CPU implementation for lattice corridor batch computation (new) over individual trellis computation (old) of receiver metric, at different channel conditions $p: = P_{\mathrm{i}} = P_{\mathrm{d}}$; $P_{\mathrm{s}} = 0$ for a range of*
a Message size N
b Alphabet size q

3.2 Effect of implementation flexibility

The GPU implementation of this MAP decoder also faces challenges not considered in similar parallel implementations for turbo codes [12–14]. In our case, the flexibility of the MAP decoder and the nature of the channel we are concerned with pose additional difficulties for parallelisation, as noted in Section 1. The reference CPU implementation is intended for simulation applications, and consequently supports a wide range of code parameters. Specifically, the CPU implementation supports: (a) arbitrary codeword length $n \geq 1$, (b) channel symbols from an arbitrary finite alphabet, as long as the alphabet is defined in an additive Abelian group, (c) messages defined over an arbitrary finite alphabet of size $q \geq 2$, as long as each message symbol is representable by a codeword, that is, $q \leq s^n$ for a channel symbol alphabet of size s, (d) arbitrary message length $N \geq 1$, (e) arbitrary state space limits $m_n^- \leq 0$, $m_n^+ \geq 0$, $m_\tau^- \leq m_n^-$ and $m_\tau^+ \geq m_n^+$, chosen dynamically based on channel conditions, (f) arbitrary received sequence length ρ, as long as the drift at the end falls within the chosen state space limits, that is, $m_\tau^- \leq \rho - \tau \leq m_\tau^+$, (g) arbitrary specification of constituent encodings C_i, (h) arbitrary thresholds to avoid pursuing low-probability trellis paths (not considered in this paper) and (i) lazy computation of the γ metric (not considered in this paper; potentially useful with path thresholding).

Additionally, the implementation supports the independent choice of real number types for the computation of the forward–backward algorithm and the inner lattice traversal. On the CPU implementation, supported types include single and double precision IEEE floating-point numbers, as well as GNU multi-precision numbers [18]. This allows the user to trade off arithmetic speed and storage requirements for accuracy, and the comparison of results with an exact implementation. The implementation also allows the user to choose between different algorithms to compute the receiver metric. These include the constrained-corridor lattice algorithm of Section 2.3, an unconstrained version of the algorithm and the trellis-based algorithm used in our earlier work.

All these variables pose a number of particular challenges for an efficient parallel implementation. First of all, it is important for the parallel implementation to support the same range of parameters as the CPU implementation. This can be a problem when functions are parallelised across one of these variables, as the required range may exceed constraints of the parallel architecture. For example, in our earlier GPU implementation, q was limited by hardware constraints and results were given for alphabet sizes up to $q = 512$. Another expected problem is that the parallelisation efficiency depends on the code parameters. With so many variables and such a wide range for each, it becomes particularly problematic to ensure high parallelisation efficiency under all conditions. Unlike our earlier GPU implementation, in this paper we propose specific steps to improve efficiency under suboptimal conditions. This variability also has an effect on scheduling kernel issues across multiple streams, as the time taken by individual kernels depends on the code parameters. On most current hardware, the ideal issue order for different kernels that can execute concurrently depends on their relative timings [19]. When such decisions had to be taken in this paper, we have favoured larger codes, where the speedup is of greater objective benefit.

A more subtle problem is that functions which have many distinct execution paths tend to require a higher register count. In a parallel implementation, this can reduce the upper limit for parallel threads, as each multiprocessor will have a finite register bank to share. Furthermore, the compiler may reduce register usage by 'spilling' some values to global memory, for a very significant increase in latency. This problem can be alleviated in part by using C++ templates [20, 21]. Each template instance is effectively independent, so that the corresponding execution path decisions are taken at compile time. This reduces register requirements, which is of great benefit in the parallel implementation; although also useful for the CPU implementation, the difference in execution speed is minimal. However, this solution is not without its cost: the use of templates increases the compiler's work in proportion to the number of combinations of template parameters. This can increase object code size and compilation time considerably. Furthermore, for a CUDA compilation with current tools, this also presents a constraint because of the limit on constant memory use within the same compilation unit. The solution we have applied to this problem was to divide the instantiation of the various combinations of template parameters into separate units. That is, the class methods were specified in a separate implementation header, imported by a number of source files, each of which instantiated an independent subset of all required template parameter combinations.

Finally, it is worth realising that the requirements of a CPU implementation can be at odds with those of a GPU implementation. For example, the sequence of constituent encodings $\mathcal{C}$ is held in memory as a vector of N constituent encodings C_i, each of which is a vector of q codewords, where each codeword is in turn a vector of n channel symbols. This is ideal for a CPU implementation, allowing each symbol to be accessed by indirect addressing, and also allowing references to individual codewords or codebooks to be specified naturally. On the GPU, however, such an organisation is problematic because it requires Nq independent memory allocations and memory transfers, each of which is an expensive operation. It is more efficient to reorganise the table on the CPU, copying the data into a single flat array and using that structure on the GPU. This avoids changing the canonical data structure on the CPU, which would have wider repercussions, and is the solution we have adopted in this paper.

4 Initial parallel implementation

4.1 Global storage

The most straighforward implementation of the MAP decoder considers it as consisting of four functions, one for computing each of the γ, α, β and L metrics. These depend on each other, dictating the order of computation: specifically, α and β depend on γ, whereas L depends on α, β and γ. Therefore γ needs to be computed first, followed by α and β (in any order, as these are independent of each other), and finally L is computed. This methodology, which we refer to as 'global' storage, assumes that there is sufficient memory to store the α, β and γ metrics.

4.1.1 γ metric: Starting with the computation of the γ metric (6), we have already observed in [9] that the computation is independent for each of i, D, m' and m. Furthermore, as noted in Section 2.3, for any given i, D and m', we only need to perform the lattice computation for the largest drift change $m - m'$ to be considered. It makes sense, therefore, to compute and store the γ metric for the whole valid range of m, for any given i, D and m'.

To make the best use of the available data parallelism, we initially use block coordinates (i, m') for grid size $N \times M_\tau$, and threads with coordinate D for block size q. This increases the parallelism by a factor M_τ with respect to our earlier GPU implementation, where the γ computation kernel had a grid size N. This increased parallelism is particularly useful when global storage is not possible, as we shall see in Section 4.2. Each thread performs an independent lattice computation and determines the γ metric for the whole valid range of m (i.e. M_n entries).

As in our earlier GPU implementation, we store the four-dimensional γ matrix as a flat array in global memory. However, we change the indexing order, so as to have index D innermost. This allows the threads in a warp to access a contiguous range of memory; this access is fully coalesced as long as the initial address and the transaction size are a multiple of 128 bytes. Since q is always a power of two, this is guaranteed for any $q \geq 16$ with double precision storage. This effect is particularly valuable since

there is no warp divergence, as the lattice traversal is identical for all D. This index is followed by $m - m'$, so that consecutive accesses by the same thread are as close to each other as possible, maximising cache re-use for small q.

To minimise global memory access and avoid register spilling into local memory, each thread holds the current lattice row being computed in shared memory. This requires an array of size $n + m_n^+ + 1$ single precision numbers per thread, dynamically allocated on kernel launch. The lattice computation algorithm is rewritten accordingly.

4.1.2 α metric: The α metric computation can be divided into two main operations: the main computation (12), followed by normalisation (11). We have already observed in [9] that the main computation at index i and state m depends on normalisation at index $i - 1$. For all m', while normalisation at index i for state m depends on computation at index i for all m'. In addition, for any given i, computation and normalisation are independent for different values of m.

Parallelisation strategy is the same as our earlier GPU implementation, where a separate kernel call is required for the main computation at each i, followed by a separate kernel to perform normalisation at that i. This is necessary because the only way to synchronise across a grid is the completion of a kernel call [10].

The main computation at i uses block coordinate m for a grid size of M_τ and threads with coordinate D for block size q, where each thread computes the corresponding partial summation over m'. The final result is then computed from these partial sums using a parallel summation across the threads in the block, using shared memory to communicate between threads. This requires a shared memory array of q double precision values per block. Normalisation requires two steps: computing the sum of all α_i' and dividing each α_i' by this sum. Both are most easily parallelised across a single block of M_τ threads. This uses only one multiprocessor, but greatly facilitates implementation in a function which corresponds to a very small proportion of the overall computation time.

In contrast with our earlier GPU implementation, we extract the initialisation of the α metric at $i = 0$ to a separate kernel. Since initialisation consists simply of setting each of M_τ values, this is implemented as a single block of M_τ threads. Although this change may not seem very significant, it slightly simplifies the main computation, keeping register usage to a minimum. In turn, this allows us to maximise occupancy by allowing more resident kernels.

The α metric is stored as a two-dimensional array in global memory, with the state index m innermost. This speeds up access in the alpha metric computation kernel, where each thread needs to read the metric values at all states for the previous index $i - 1$. The speedup is achieved by copying the row at index $i - 1$ to shared memory, requiring a shared memory array of M_τ double precision values per block. This memory copy can be done in parallel across the block, so that global memory access is coalesced. The use of a two-dimensional array ensures this by correctly aligning each row.

4.1.3 β metric: A similar argument applies to the computation of β, except that now the main computation at index i depends on normalisation at index $i + 1$. Furthermore, α and β can be computed concurrently as there is no data dependency between them; this can be achieved using streams on devices that support concurrent kernel execution.

Recall that the computation of α and β requires a number of consecutive kernel calls each. Specifically, for α, this sequence consists of the intialisation kernel, normalisation at $i = 0$, computation at $i = 1$, normalisation at $i = 1$ and repeating computation and normalisation for increasing i until $i = N$. Since each kernel depends on the completion of the preceding one, this dependency is best expressed by issuing the kernels in the same stream. For β, the sequence is the same but the index order is reversed (i.e. initialisation and normalisation for $i = N$, followed by computation and normalisation pairs

for decreasing i from $i = N - 1$ to $i = 0$). The independence of the β kernels from the α kernels is expressed by issuing these in a second stream.

Unfortunately, hardware limitations in Fermi and initial Kepler devices (GK104 architecture, for compute capability 3.0) cause additional complication. Since these devices have 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 [19]. Since the kernels for α and β at each index have the same complexity, we avoid this problem by using a breadth-first launch order, as follows. Issue first the initialisation of $\alpha_{i=0}$ in stream one and of $\beta_{i=N}$ in stream two, followed by the normalisation of $\alpha_{i=0}$ in stream one and of $\beta_{i=N}$ in stream two. This is followed by the computation of $\alpha_{i=1}$ in stream one and of $\beta_{i=N-1}$ in stream two, and the normalisation of $\alpha_{i=1}$ in stream one and of $\beta_{i=N-1}$ in stream two. This is repeated, incrementing i for α and decrementing for β. Concurrent execution improves device utilisation when the grid size for a single kernel call is small. Note that this problem does not exist in the latest Kepler architecture (GK110), which has 32 compute engine queues [22].

4.1.4 L metric: Finally, the L computation (1) is independent across i, D. As in our earlier GPU implementation, we parallelise this across blocks with index i for a grid size N and threads with index D for block size q. In this paper, however, we also avoid multiple global memory reads and ensure coalesced memory access by first copying the required rows at α_i and β_{i+1} to shared memory. This requires two shared memory arrays of M_τ double precision values per block. Each γ value is only read once, and this is done in an order that ensures coalesced access.

4.2 Local storage

Unfortunately, 'global' storage is only possible when there is sufficient memory to store the α, β and γ metrics. The required storage capacity increases with increasing N, n, q and poorer channel conditions (since this increases the required state space). To illustrate this, consider a system with a moderate message size $N = 210$ and a range of alphabet sizes q; we plot the required metric storage memory in Fig. 2 over a range of channel conditions. For ease of comparison, horizontal lines are included at values of 1 GiB and 2 GiB, corresponding to common per-CPU core or per-GPU device limits for metric storage. It can be readily seen that these limits are reached at moderate to low channel error rates for larger alphabet sizes, and also at high channel error rates for moderate alphabet sizes.

This problem can be resolved by dividing the computation of γ_i across i and observing that each of α_i, β_i and L_i depend only on a single index for γ. Specifically, (a) α_i depends on α_{i-1} and γ_{i-1}, (b) β_i depends on β_{i+1} and γ_i and (c) L_i depends on α_i, β_{i+1} and
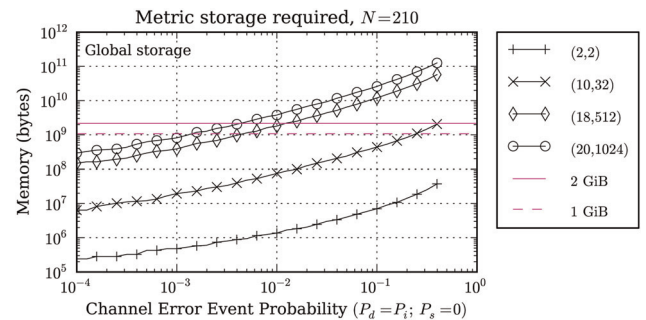

Fig. 2 *Metric storage requirements for a moderate message size $N = 210$ and half-rate codewords (n, q), over a range of channel conditions. It is assumed that all metrics are stored as double precision values.*

γ_i. Since the order of computation of α_i and β_i is enforced by their internal dependencies, each γ_i has to be computed at least twice. We avoid computing it a third time by first completing the α metric computation, then combining the computation of L with that of β. This is shown graphically in Fig. 3.

This mode requires us to divide the γ and L kernels to compute values at a single index i. Therefore the γ compute kernel now uses block coordinate m' for grid size M_τ. Similarly, the L compute kernel now only uses a single block. In both cases, i is passed as a kernel parameter. This significantly reduces device util-isation, particularly under good channel conditions where M_τ is relatively small.

We can mitigate this problem using concurrent execution by issuing multiple kernels, as follows. Consider first the computation of the α metric, requiring kernel calls as shown in Fig. 3a. In this figure, each row corresponds to the computation of a particular row of the α metric, with index i. Each kernel within a given row depends on the previous kernel, whereas the compute kernel also depends on the normalisation kernel of the previous row. The former dependency can be expressed by issuing the three kernels of a given row in the same stream; the latter dependency can be expressed using CUDA events [19]. Such an arrangement allows compute kernels for the γ metric to execute concurrently with other kernels from previous rows, increasing device usage.

In this way, kernel issues could be divided over $N+1$ different streams. Observe, however, that independent storage space is required for the γ metric at each index that may be computed inde-pendently. This makes it impractical to use a separate stream for

each possible index, in which case the amount of space required is the same as for global mode. Instead, we limit the number of streams to a depth of four, used on a rotating basis. This limit was chosen empirically; others have already observed that it is dif-ficult to have more than four kernels running concurrently [19].

The use of streams raises the question of how best to order kernel issues, as considered for the concurrent computation of α and β in Section 4.1. In this case, the problem is somewhat more complex, as different streams may be executing kernels of different sizes and durations. Additionally, the use of events to synchronise between streams requires that the event on stream i is recorded before the wait on that event is issued on stream $i+1$. Now for the event to be recorded, all prior kernels on that stream need to be issued (al-though not necessarily executed). Effectively, this means that for stream i we need to issue the following sequence consecutively: wait for event completion on stream $i-1 \rightarrow$ compute $\alpha_i \rightarrow$ normal-ise $\alpha_i \rightarrow$ record event on stream i. All this has to be issued before we can repeat the issue sequence for the next stream. This require-ment forbids us from issuing all kernels across the four streams in breadth-first order. Instead, we issue the γ computation kernels in breadth-first, and then issue the above sequences for each stream in the order required. This maximises concurrency of execution for the γ computation kernels, but limits concurrency of the remain-ing kernels to the combination of computation and normalisation of α (this applies to Fermi and first-generation Kepler devices).

A similar argument applies for the computation of the β and L metrics, requiring kernel calls as shown in Fig. 3b. Other than the order of index traversal, the only difference in this case is the pres-ence of the computation kernel for the L metric. The simplest way to deal with this is to issue this kernel after the normalisation kernel in the same row. This automatically satisfies its dependencies, which are the same as for the compute kernel for the β metric in the same row. It also allows the β computation and normalisation to occur first, minimising delays to subsequent rows. The L computa-tion kernel has a significant duration but only has one block; this inefficiency is easily hidden by concurrent kernels from subsequent rows. In this case, we again issue the γ computation kernels breadth-first, and then issue the remaining sequence for each stream in the order required. We include the L computation in this sequence rather than issuing it separately in breadth-first order. This allows that kernel to be concurrent with the β kernels rather than only with itself, making more efficient use of the available multiprocessors.

A summary of the kernels required for the initial parallel imple-mentation described so far, for global and local storage, is given in Table 1. It can be seen that the block size is equal to q or M_τ.

5 Advanced considerations

5.1 Handling resource limits

The initial parallel implementation described previously assumes that it is possible to issue kernels with the given block sizes: q for the γ, α, β and L computation kernels, and M_τ for the remaining
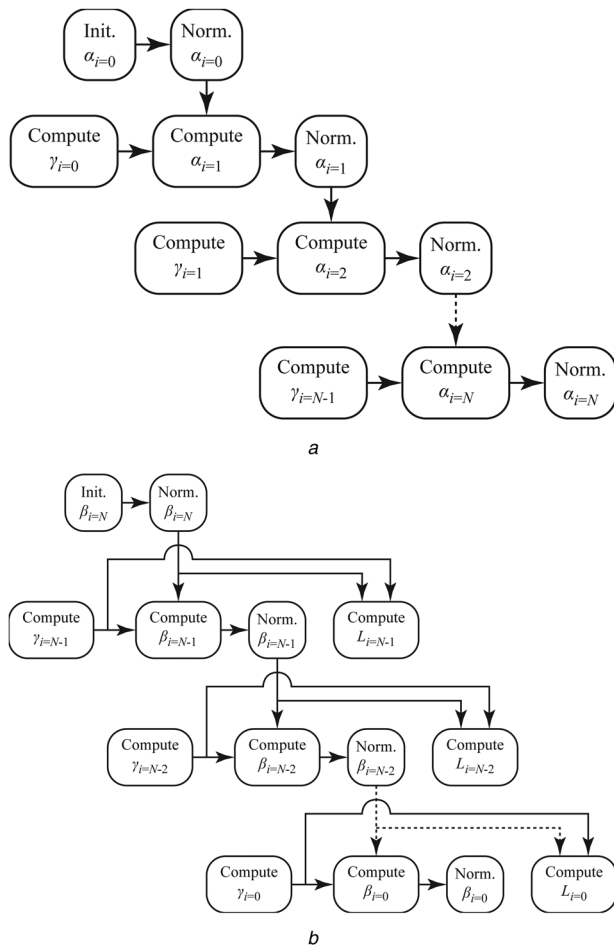

Fig. 3 *Graphical representation of the interdependencies and the sequence of kernel calls required to compute*
a α metric
b β and L metrics in local storage mode

Table 1 Summary of the kernels required for the initial parallel implementation, for global and local storage

Kernel	Storage mode	Grid size	Block size	Call
Compute γ	global	$N \times M_\tau$	q	1
	local	$1 \times M_\tau$	q	N
Initialise α, β	both	1	M_τ	1
Compute α, β	both	M_τ	q	N
Normalise α, β	both	1	M_τ	$N+1$
Compute L	global	N	q	1
	local	1	q	N
Compute Φ_T	both	1	M_τ	2

kernels. However, this may not be possible because of a number of constraints: (a) the device limit on the maximum number of threads per block, (b) register pressure, which depends on both the number of registers needed per thread and the number of registers available per multiprocessor and (c) shared memory pressure, which depends on the requirements for a given block size and the available shared memory per multiprocessor. Limits for a device depend on its compute capability, and can be queried by the implementation at run-time [10].

To illustrate the shared memory requirements, consider again a system with $N = 210$ and a range of alphabet sizes q; we plot the required shared memory per block in Fig. 4 over a range of channel conditions. For ease of comparison, horizontal lines are included at values of 48 KiB and 24 KiB. The former corresponds to the maximum amount of shared memory per multiprocessor on Fermi and Kepler devices; this is the limit for a kernel launch to succeed. The latter corresponds to half this value, allowing two resident blocks per multiprocessor. Observe how the requirements for the γ computation kernel exceed device limits for larger q. Other kernels only approach device limits when channel conditions are poor; limits are exceeded if N is sufficiently large.

A kernel launch that violates any of these limits will fail at run-time. For the four computation kernels, we solve this problem by dynamically determining a suitable block size at run-time, and adapting the kernel implementation to work with a block size that is not equal to q. We choose a block size equal to the smaller of q or the largest allowed multiple of the warp size, taking into account device limits, register pressure and the shared memory required for a given thread count. We denote this block size by B_x^γ for the γ computation kernel, $B_x^{\alpha\beta}$ for the α and β computation kernels and B_x^L for the L computation kernel. For the γ, α and β computation kernels, if the block size is less than q, each thread loops through values of D equal to its thread index plus multiples of the block size. For the L computation kernel, we adopt a different approach: for $B_x^L < q$, we extend the grid size by a factor $G_y^L = \lceil q/B_x^L \rceil$, dividing the computation of the range of q across

Fig. 4 *Shared memory requirements per block for the γ, α, β and L computation kernels, assuming a block size of q, for a moderate message size $N = 210$ and half-rate codewords (n, q), over a range of channel conditions. It is assumed that internal metrics for the γ kernel are stored as single precision values, whereas those for the other kernels are stored as double precision values.*

multiple blocks. This has the advantage of keeping the same parallelisation in a kernel which normally has a small grid size. In both cases, unless q is an exact multiple of the block size chosen, some threads will have no work to do. In the common case where q is a power of two, any large q is also a multiple of the warp size (32 for current hardware); in this case there will be no warp divergence, so any effect on performance should be limited to the loss in latency hiding.

5.2 Improving occupancy

At the other end of the scale, we had suggested in [9] that one may improve performance for small q (and large N) by computing multiple indexes in a single block. This can potentially improve multiprocessor occupancy, and therefore reduce latency in memory-limited kernels. It is worth noting here that occupancy does not depend only on the block size, but also on the number of resident blocks per multiprocessor [23]. In turn, this depends on register pressure and shared memory pressure. Therefore it is advantageous to increase the block size as long as this does not increase register and shared memory requirements proportionally.

Consider first the γ metric computation. A suitable strategy for increasing block size is to aggregate the work done in multiple blocks; rather than aggregating multiple indexes i, however, we aggregate multiple states m'. This has the advantage of allowing the aggregation to happen in both global and local storage methodologies. At the limit, one could attempt to use thread coordinates (D, m') for block size $q \times M_\tau$. This would require block coordinate i for grid size N in global storage; there would only be one block in local storage. However, such a block size is very likely to exceed device limits because of resource constraints. Instead, for small q, we use a block size $q \times B_y^\gamma$, where B_y^γ is the smaller of M_τ or the largest allowed multiple of the warp size. In determining B_y^γ, we take into account device limits, register pressure and the shared memory required for a given thread count. This results in a grid size $N \times G_y^\gamma$ in the case of global storage or $1 \times G_y^\gamma$ for local storage, where $G_y^\gamma = \lceil M_\tau / B_y^\gamma \rceil$. We also limit B_y^γ so that the resulting grid size is not smaller than the number of streaming multiprocessors on the device, N_{SM}. Specifically, the constraint is $NG_y^\gamma \geq N_{SM}$ for global storage and $G_y^\gamma \geq (1/4)N_{SM}$ for local storage (where four such kernels are issued concurrently, potentially allowing greater occupancy). This ensures that we do not trade off parallel execution for an increased occupancy, since the former usually has a greater impact on speed.

A similar argument applies for the α and β computation kernels. For small q we use a block size $q \times B_y^{\alpha\beta}$, where $B_y^{\alpha\beta}$ is the smaller of M_τ or the largest allowed multiple of the warp size. This results in a grid size $G_x^{\alpha\beta} = \lceil M_\tau / B_y^{\alpha\beta} \rceil$. In addition to considerations for device limits, we limit $B_y^{\alpha\beta}$ so that $G_x^{\alpha\beta} \geq (1/2)N_{SM}$ for global storage (where α and β kernels are computed concurrently) and $G_x^{\alpha\beta} \geq N_{SM}$ for local storage.

We do not apply the same technique to the computation of L since the proportion of time spent in this kernel is not very significant on the GPU implementation. A summary of the kernels required for the complete parallel implementation, including the advanced considerations described above, for global and local storages, is given in Table 2. We also list in the table the complexity of computations for a single thread.

6 Performance analysis

We consider the GPU performance of the above implementation on two Fermi devices and one Kepler device. The GTX 480 and GTX 680 are the highest-performing single-GPU devices in the consumer-oriented GeForce range for the Fermi and Kepler architectures, respectively. The C2075 is the highest-performing processor in the computation-oriented Tesla range for the Fermi
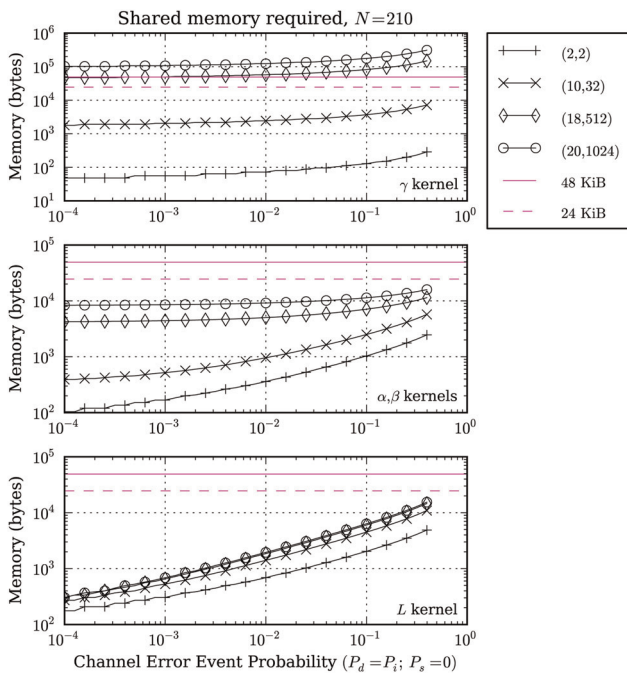
Table 2 Summary of the kernels required for the complete parallel implementation, including considerations for resource limits and multiprocessor occupancy, for global and local storage

Kernel	Storage mode	Grid size	Block size	Call	Shared memory[a]	Thread complexity
Compute γ	global	$N \times G_y^\gamma$	$B_x^\gamma \times B_y^\gamma$	1	$(n + m_n^+ + 1) \cdot B_x^\gamma \cdot B_y^\gamma$	$O(nM_n - (m_n^-(m_n^- - 1))/2)$
	local	$1 \times G_y^\gamma$	$B_x^\gamma \times B_y^\gamma$	N		
Initialise α, β	both	1	M_τ	1	—	$O(M_\tau)$
Compute α, β	both	$G_x^{\alpha\beta}$	$B_x^{\alpha\beta} \times B_y^{\alpha\beta}$	N	$(q + M_\tau) \cdot B_y^{\alpha\beta}$	$O(M_n)$
Normalise α, β	both	1	M_τ	$N+1$	—	$O(M_\tau)$
Compute L	global	$N \times G_y^L$	B_x^L	1	$2M_\tau$	$O(M_\tau M_n)$
	local	$N \times G_y^L$	B_x^L	N		
Compute Φ_T	both	1	M_τ	2	—	$O(M_\tau)$

[a]Shared memory is expressed here as the unit size of an array of real numbers; for the γ computation kernel these are single precision values, whereas for all other kernels these are double precision values.

architecture. CPU performance is considered for a serial reference implementation, on the same system as in [9]. Hardware specifications are summarised in Table 3.

6.1 Division of computation time

We consider first the time spent in each of the four metric computations as a proportion of the total time required to decode a frame. For global storage, this is shown in Fig. 5 for a moderate alphabet size $q = 32$ over a range of message sizes N, for the CPU and the C2075 device. Note that we show a combined timing for the computation of the α and β metrics as these are computed concurrently on the GPU. On the CPU the division of computation time is almost constant, with the γ metric taking over 75% of the time for moderate to large N. On the GPU we observe a number of distinct differences: (a) the computation of the γ metric takes a substantially smaller proportion of time, (b) the α and β metrics seem to make up for most of the difference for moderate to large N and (c) for smaller N a substantial proportion of time is unaccounted for. These differences may be explained as follows. Computation of γ is more easily parallelised than α and β, which also suffer from greater kernel call overhead. This makes the computation of γ considerably more efficient than α and β on the GPU. For smaller block sizes, the overhead in setting up and transferring data to and from the GPU is also a more significant contributor. Note that this timing depends on the processor, mainboard and memory speeds of the system where the GPU is fitted. Results for the other two GPU devices show a similar trend.

It is also instructive to repeat this experiment for a moderate message size $N = 210$ over a range of alphabet sizes q. Results for this are shown in Fig. 6. In this case, we can see that even on the CPU the time required to compute γ takes a more significant proportion of time as q increases. The time required by α, β and L decreases proportionally. This is expected, as while the computation of α, β and L scales with N, M_τ, M_n and q, the computation of γ also scales with n. For the codes considered in this experiment, $n = 2 \log_2 q$ to keep the same code rate. On the GPU, the change is more pronounced, with the γ computation again becoming dominant for large q. The main reason for this is that as q increases, the computation of α and β becomes considerably more efficient,
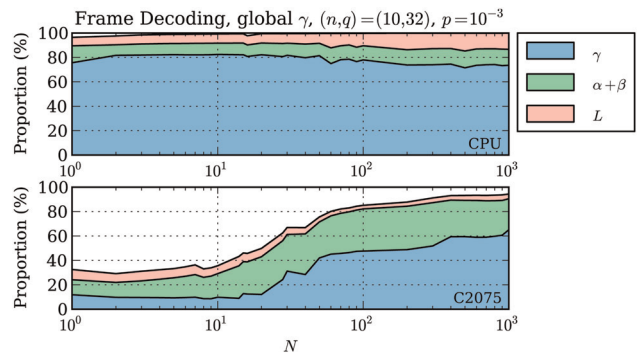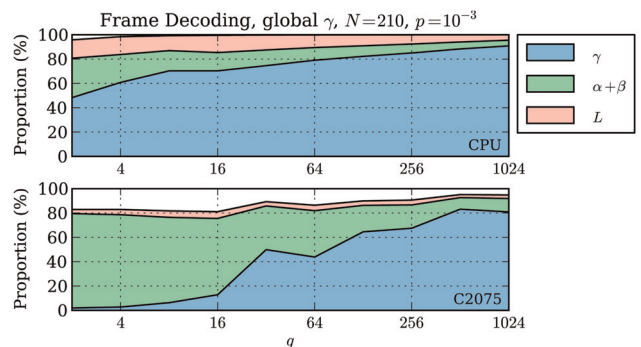

Fig. 5 *Time spent in each of the four metric computations as a proportion of the total time required to decode a frame, for global storage and a moderate alphabet size $q = 32$, over a range of message sizes N. Channel conditions are given by $p := P_i = P_d = 10^{-3}$; $P_s = 0$.*

each kernel call has more computation to perform and call overhead becomes less significant. Again, results for the other two GPU devices show a similar trend.

6.2 Computation efficiency

We consider next the efficiency of computing the γ, α and β metrics for the different architectures. For each metric, this can be visualised by plotting the time taken to compute the metric, normalised by the expected complexity of computation. This is shown in Fig. 7 for global storage and a moderate alphabet size $q = 32$ over a range of message sizes N. Starting with the γ metric, observe that the

Table 3 Hardware specifications for CPU and GPU systems

	Opteron 2431	GTX 480	C2075	GTX 680
Processors × cores	2×6	15×32	14×32	8×192
Core speed, GHz	2.412	1.401	1.147	1.0585
Memory, MiB	32 768	1536	5376	2048
GFLOPs (float)	2.412 (scalar)	672.48	513.856	1625.856
GFLOPs (double)	2.412 (scalar)	336.24	256.928	67.744

Fig. 6 *Time spent in each of the four metric computations as a proportion of the total time required to decode a frame, for global storage and a moderate message size $N = 210$, over a range of alphabet sizes q. Channel conditions are given by $p := P_i = P_d = 10^{-3}$; $P_s = 0$.*

normalised time is constant for the CPU; this is expected, and indicates that the expected complexity is an accurate estimate of actual complexity. This is most likely because of the tight optimisation of the implementation. For the GPU implementation, we can see that maximum efficiency is reached quickly, slightly beyond $N = 10$ on all devices. Efficiency fluctuates beyond this point, although is generally better for larger N. Comparing with the equivalent plot in [9, Fig. 1], observe that in this paper maximum efficiency is reached much earlier (for the GTX 480 this was achieved around $N = 100$ in [9]). This is because of the increased grid size (see Section 4.1) and the use of block aggregation to increase occupancy (see Section 5.2). Note that the normalised time units in this paper and those in [9] cannot be compared directly as the complexity is obtained for different algorithms.

For the α and β metrics, we can see in Fig. 7 that CPU efficiency is almost constant, decreasing slightly as N increases. Although unexpected, this is not surprising, as the complexity expression considers only the floating-point arithmetic and ignores overheads of loop handling and memory access. On the other hand, GPU efficiency continues to improve as N increases. This is most likely because an increase in N causes an increase in M_τ under the same channel conditions; this increases the computation done in each kernel call, reducing the significance of kernel call overhead. Furthermore, the efficiency of each kernel call also improves because of the increased grid size and the use of block aggregation.

The experiment is repeated for a moderate message size $N = 210$, over a range of alphabet sizes q. Results are shown in Fig. 8. In this case, observe that the efficiency of computing the γ metric remains approximately constant throughout the range, with a decrease in efficiency only for the smallest alphabet sizes. We can achieve good efficiency throughout the range because of the use of block aggregation, which increases occupancy and allows good device utilitisation even for small q. For the α and β metrics, again efficiency continues to improve as q increases; furthermore, this improvement happens more quickly than when N is increased (see Fig. 7). This is because an increase in q has two effects: it directly increases the block size and indirectly increases the grid size (since M_τ depends on n, which depends on q if the code rate is fixed). A similar experiment with larger N shifts the curves to the left, so that maximum efficiency is reached at a lower q.
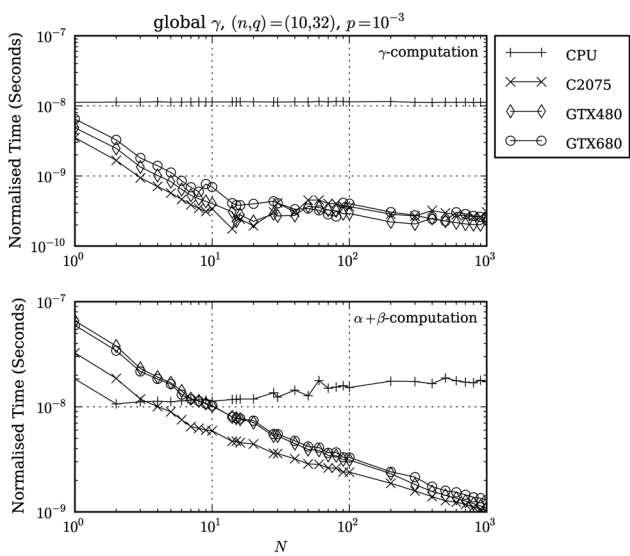

Fig. 7 *Time to compute γ, normalised by a factor $NqM_\tau(nM_n - m_n^-(m_n^- - 1)/2)$, and α and β, normalised by a factor $NqM_\tau M_n$, for global storage and a moderate alphabet size $q = 32$, over a range of message sizes N. Channel conditions are given by p: $= P_i = P_d = 10^{-3}$; $P_s = 0$.*
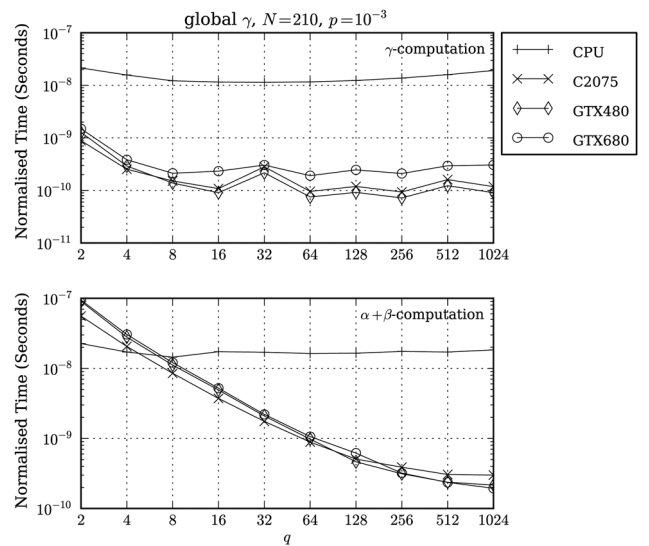

Fig. 8 *Time to compute γ, normalised by a factor $NqM_\tau(nM_n - m_n^-(m_n^- - 1)/2)$, and α and β, normalised by a factor $NqM_\tau M_n$, for global storage and a moderate message size $N = 210$, over a range of alphabet sizes q. Channel conditions are given by p: $= P_i = P_d = 10^{-3}$; $P_s = 0$.*

6.3 Multiprocessor occupancy and usage

To understand the effect of changes in code parameters on computation efficiency, it is instructive to look at metrics that directly measure the effectiveness of parallelisation. At multiprocessor level, it is important to keep the hardware busy by having a sufficiently high thread count per block. Since instructions are issued at warp level, the block size is ideally a multiple of the warp size, ensuring that each thread in each warp is doing something useful. Furthermore, the hardware hides latencies in instruction issue and memory access by executing warps that are ready. It is therefore useful to maximise the number of active warps per multiprocessor, which can be achieved by increasing the thread count per block or by ensuring that multiple blocks can be resident. The latter is often limited by register pressure, so that the former is often the most practical approach. Occupancy, the proportion of active warps to the maximum supported by the hardware, is a useful measure to determine how well latency is hidden.

At device level, effectiveness of parallelisation depends on how many of the available multiprocessors are executing a kernel. We refer to this proportion as the device usage. Clearly, if only a single kernel is being executed, usage depends on the grid size, which is ideally a multiple of the number of multiprocessors N_{SM}. However, this presents a trade-off between the grid size and block size. Usage can also be improved by running multiple kernels concurrently. A good strategy is to maximise block size, but only so far as to allow a grid size which keeps all multiprocessors busy.

For this analysis, we limit our results to the GTX 480 and GTX 680 devices. Multiprocessor occupancy and usage for the C2075 are similar to those of the GTX 480, as the two devices have multiprocessors of the same architecture and only differ in the number of multiprocessors available (by one).

6.3.1 Global storage: Starting first with global storage, we determine occupancy for the γ, α, β and L computation kernels, based on the actual block size used and assuming no hardware overhead. Note that this is the theoretically achievable occupancy with the chosen parameters; when measured with the profiler the true value will usually be marginally lower. This is shown in Fig. 9 for a moderate alphabet size $q = 32$ over a range of message sizes
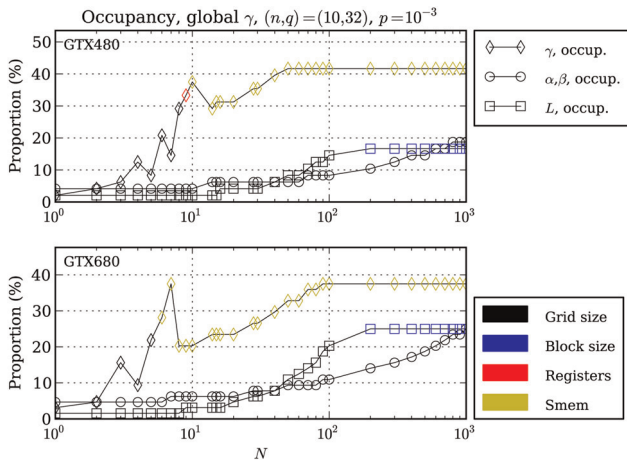
Fig. 9 *Multiprocessor occupancy for the γ, α, β and L computation kernels for global storage and a moderate alphabet size $q = 32$, over a range of message sizes N. Channel conditions are given by p: $= P_i = P_d = 10^{-3}$; $P_s = 0$.*

N. We annotate in colour the limiting factor that stops us from achieving higher occupancy. For each of the computation kernels, the nominal block size (equal to *q*) is the same as the warp size. If this was the chosen block size, occupancy would be limited by the number of resident blocks. For Fermi devices this would be at most eight, for a maximum occupancy of 16%. Increasing the block size as explained in Section 5.2, however, allows us to achieve an occupancy of 40% for the γ kernel on Fermi (and very close to that on Kepler). Observe that occupancy is limited at the higher end by the shared memory requirement for the γ kernel. For the α and β kernels, occupancy is limited by the grid size; this depends on M_τ and there is nothing we can do about it. Comparing these results with Fig. 7, it is worth realising that peak efficiency for γ is reached at the same point when peak occupancy is reached. Similarly, efficiency for the α and β kernels increases with the kernel occupancy as *N* is increased.

The corresponding device usage for each kernel is shown in Fig. 10. As expected, device usage is optimal for the γ and *L* kernels whenever *N* is a multiple of N_{SM} for the given device. Note that for the α and β kernels, device usage peaks at 50% as we know that these kernels will be running concurrently. This allows us to reach peak usage earlier for these kernels.
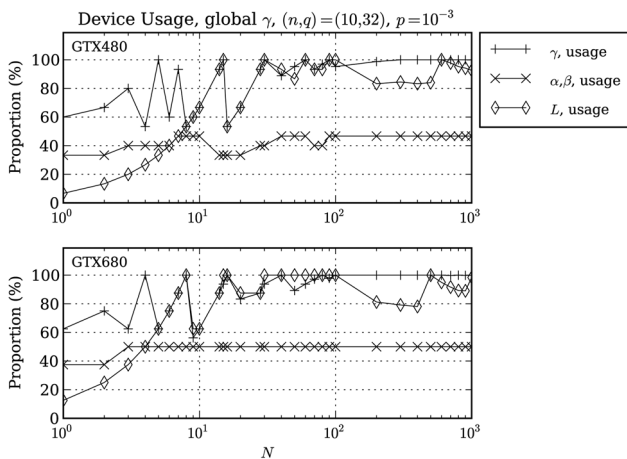
Comparing these results with Fig. 7, observe that the fluctuations in device usage for γ explain the corresponding fluctuations in efficiency.

The experiment is repeated for a moderate message size $N = 210$, over a range of alphabet sizes *q*. Multiprocessor occupancy and device usage are shown, respectively, in Figs. 11 and 12. Consider first the Fermi device. For the γ kernel, observe that occupancy remains fairly constant, limited at the lower end by register pressure and otherwise by shared memory requirements. Together with the constantly high device usage for this kernel, this explains the flat efficiency in Fig. 8. For the α and β kernels, however, occupancy increases until it reaches a peak limited by register pressure. This is consistent with the point where maximum efficiency is reached in Fig. 8.

A few points are worth noting with respect to the Kepler device. This has a higher peak single-precision performance, so one would expect faster performance for the γ kernel. However, it is much harder to reach the necessary efficiency, for two reasons: (a) each multiprocessor has six times as many single-precision cores, so requires more resident warps to hide latency and (b) the available shared memory per multiprocessor remains the same, limiting the occupancy that can be reached.
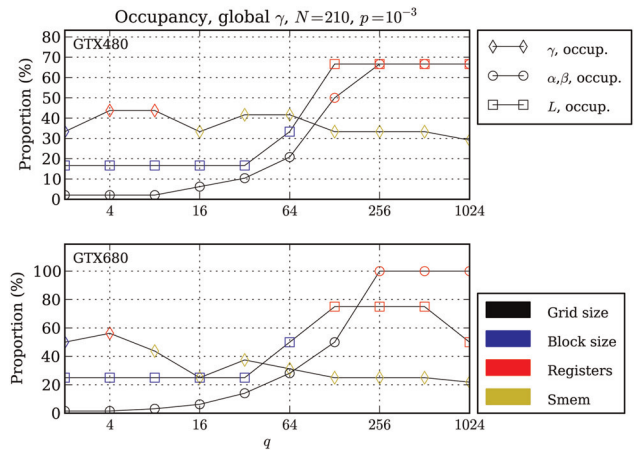

Fig. 11 *Multiprocessor occupancy for the γ, α, β and L computation kernels for global storage and a moderate message size $N = 210$, over a range of alphabet sizes q. Channel conditions are given by p: $= P_i = P_d = 10^{-3}$; $P_s = 0$.*
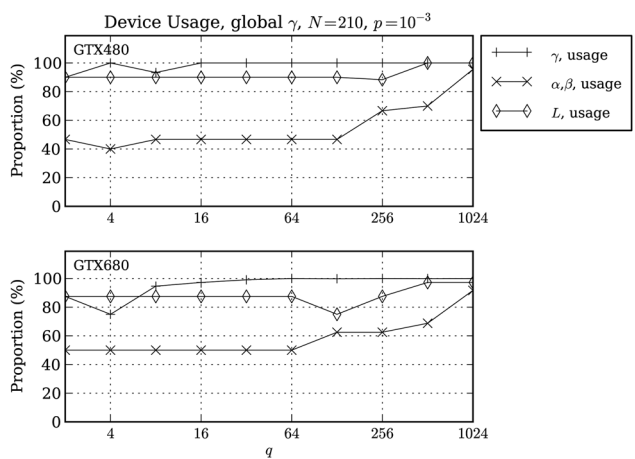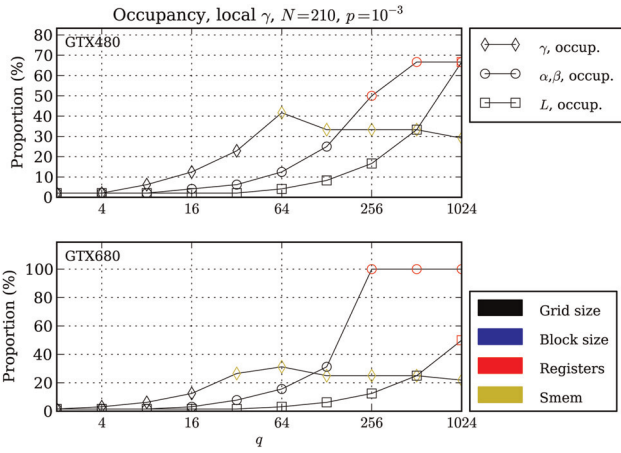

Fig. 10 *Device usage for the γ, α, β and L computation kernels for global storage and a moderate alphabet size $q = 32$, over a range of message sizes N. Channel conditions are given by p: $= P_i = P_d = 10^{-3}$; $P_s = 0$.*

Fig. 12 *Device usage for the γ, α, β and L computation kernels for global storage and a moderate message size $N = 210$, over a range of alphabet sizes q. Channel conditions are given by p: $= P_i = P_d = 10^{-3}$; $P_s = 0$.*

Fig. 13 *Multiprocessor occupancy for the γ, α, β and L computation kernels for local storage and a moderate message size N = 210, over a range of alphabet sizes q. Channel conditions are given by p: = P_i = P_d = 10^{-3}; P_s = 0. Shared memory per multiprocessor remains the same, limiting the occupancy that can be reached.*

6.3.2 Local storage: We repeat the analysis for local storage with a moderate message size $N = 210$, over a range of alphabet sizes q. Multiprocessor occupancy and device usage are shown, respectively, in Figs. 13 and 14. For local storage, each γ kernel computes the metric for a single index i; this reduces the grid size by a factor of N in comparison with global storage. This places a considerable constraint on the achievable occupancy, especially if we want to avoid a drop in multiprocessor usage. For this reason, peak occupancy is only reached for larger q. There is no direct change to the α and β kernels with local storage, except that now we do not run these concurrently. Therefore we now try to keep all multiprocessors busy with these kernels (rather than only half). This limits the block size by a factor of two, so that we now reach the same occupancy when q is doubled.

A similar issue occurs with the L metric, which is also computed for a single index i, reducing the grid size by a factor of N. In this case, we make no attempt to increase the grid size, relying instead on the concurrent execution of this kernel with the γ and β computation kernels. Given the above observations, we expect the local storage implementation to be less efficient than global storage, except for large q.
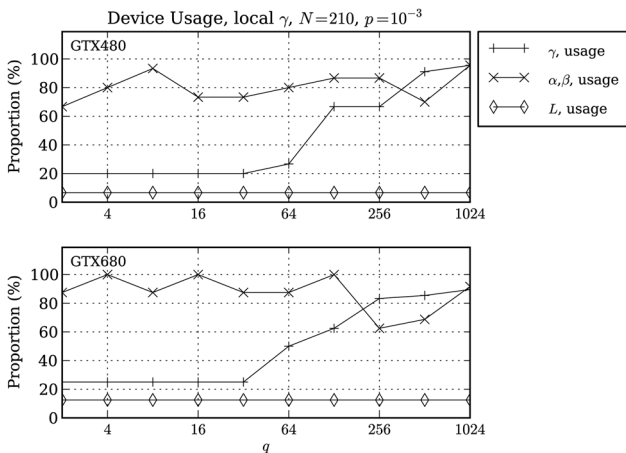

Fig. 14 *Device usage for the γ, α, β and L computation kernels for local storage and a moderate message size N = 210, over a range of alphabet sizes q. Channel conditions are given by p: = P_i = P_d = 10^{-3}; P_s = 0.*

7 Results

In this section, we compare the overall performance of the proposed implementation on all three GPU systems of Section 6 with the CPU implementation of the same algorithm and with the earlier GPU implementation of [9]. Following this, we consider the limitations imposed by our implementation and the overall performance achieved under a range of conditions.

7.1 Overall speedup

We first consider the speedup for complete frame decoding on the different GPU devices as compared with the CPU implementation using the same storage methodology. This is shown in Fig. 15 for both global and local storage. When we set a moderate alphabet size $q = 32$, the speedup improves as the message size N is increased. Observe that for small N, any improvement is very poor; indeed for the smallest values of N, the CPU implementation is faster than the GPU implementations. This is because of the increased impact of GPU overhead when N is small, as we have seen in Section 6.1. For this value of q, observe also that the speedup for global storage is better than for local storage, for moderate to large N. This is because of the decreased occupancy and multiprocessor usage for small q and low channel error rate, which affects local storage computation in a more significant way (see Section 6.3).
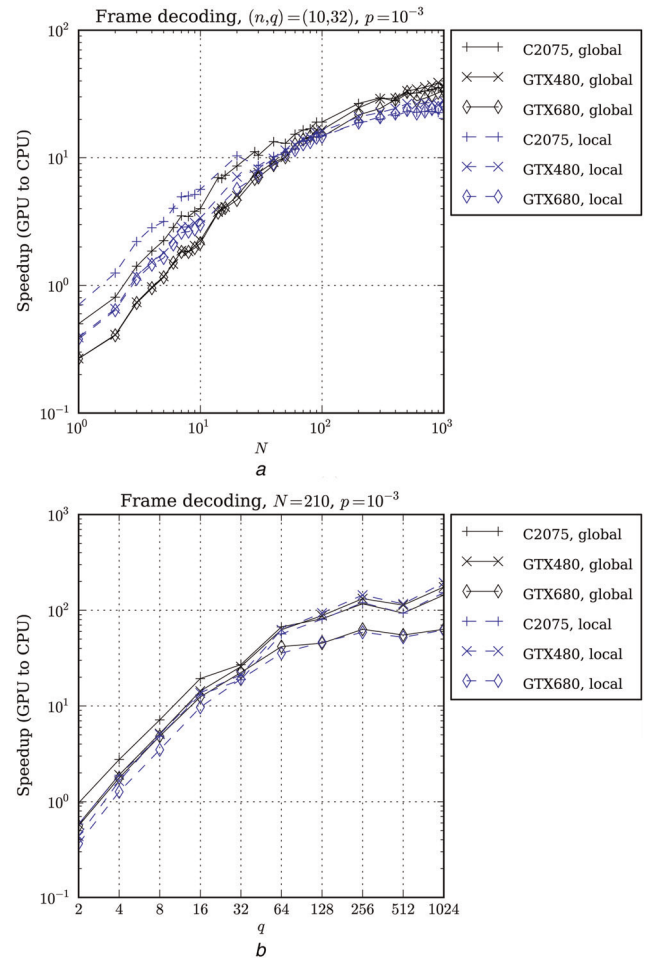

Fig. 15 *Ratio of complete frame decoding timings on CPU as compared with C2075, GTX 480 and GTX 680 GPU devices, for global and local storage. Channel conditions are given by p: = P_i = P_d = 10^{-3}; P_s = 0. Comparisons are for:*
a Moderate alphabet size q = 32, over a range of message sizes N
b Moderate message size N = 210, over a range of alphabet sizes q.

Setting a moderate message size $N = 210$, we find that speedup improves as the alphabet size q is increased. Observe that for small q, the speedup achieved for local storage is slightly less than that for global storage. For larger q, the speedup achieved for local storage is practically the same as that for global storage. This indicates the increased efficiency of local storage computation for larger q, as expected.

7.2 Speedup over previous parallel implementation

Next, we repeat the simulations of [9] using the improved GPU implementation. Results comparing the overall decoding speed, for the same codes and on the same GTX 480 system, are shown in Fig. 16. The speedup obtained for the improved implementation compares favourably with the expected speedup because of algorithmic changes, shown in Fig. 1. For larger values of N and q, this speedup approaches or exceeds the speedup achieved on the CPU implementation, indicating that the improved GPU implementation achieves at least the same efficiency. This is significant, as a significant fraction of time is spent in computing the α and β metrics, which are much harder to parallelise efficiently. As expected, speedup improves for poorer channel conditions.

7.3 Limitations and overall performance

The main limits imposed by our implementation are discussed below. The largest supported state space M_τ is limited by the block size for the α, β initialisation and normalisation kernels and the Φ_T computation kernel. Since these kernels are very simple, this limit corresponds to the maximum number of threads per block, which on current hardware is 1024. Now M_τ increases with τ and the channel error probabilities, and depends on the allowed probability for an actual drift to exceed these limits. Even for a poor channel with $P_i = P_d = 0.2$ and an exclusion probability $P_r = 10^{-10}$, this allows us to support $\tau \lesssim 12\,000$. For a very poor channel having $P_i = P_d = 0.4$, this limit drops to $\tau \lesssim 4000$; note that this is beyond the decoding ability of any known codes.
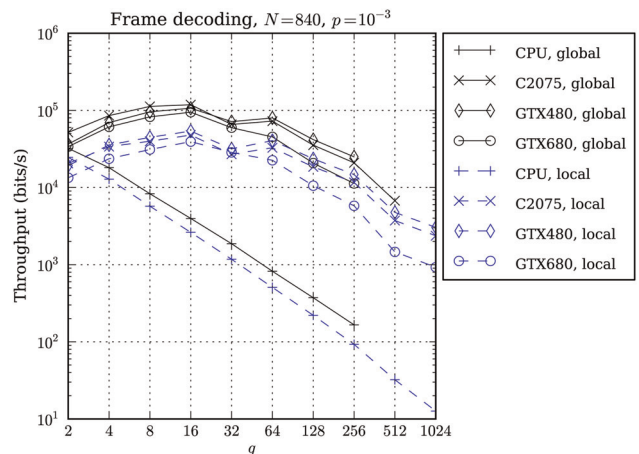
Fig. 17 *Decoder throughput on CPU and C2075, GTX 480 and GTX 680 GPU devices, for global and local storage, and a large message size $N = 840$, over a range of alphabet sizes q, under moderate to good channel conditions. Channel conditions are given by $p: = P_i = P_d = 10^{-3}$; $P_s = 0$.*

Larger message lengths are possible when the target channel error rates are lower.

Another limitation depends on available device memory. The use of local storage goes a long way to increase the limits of supported code parameters. By way of example, a code with $N = 840$, $n = 20$ and $q = 1024$ uses a little over 1 GiB of device memory at $P_i = P_d = 0.1$ using local storage. In general, this tends to be an issue only with larger alphabet sizes.

Although the implementation considered is intended for use in a simulator, it is interesting to consider whether the achieved performance is suitable for real-time applications. We show in Fig. 17 the achieved throughput for our CPU and GPU implementations for global and local storage, a large message size $N = 840$ and a range of alphabet sizes q, under moderate to good channel conditions. Even with the algorithm improvements of [11], the CPU implementation is clearly too slow for real-time use except in low throughput applications. The GPU implementation, however, reaches 100 kbit/s even with this relatively large message length. Note that the missing simulation results for global storage with large q indicate conditions where available memory was exceeded.

8 Conclusions

In this paper, we have presented an optimised parallel implementation of the MAP decoder of [11] with algorithmic improvements over the equivalent decoder of [7]. This implementation achieves a speedup of more than $100 \times$ over the CPU implementation of the same algorithm and more than $10 \times$ over the previous GPU implementation of [9], on the same hardware. This increases our ability to analyse larger codes and poorer channel conditions and makes practical use of such codes more feasible.

We also present a reduced memory implementation where some intermediate metrics are computed twice: once for the forward pass and once again for the backward pass and final results. This variant trades off some decoding speed for significantly reduced-memory requirements. This allows the decoder to work with longer message sizes and poorer channel conditions than would otherwise be possible.

The speed improvements of this implementation are made possible by a number of specific optimisations. We use shared memory judiciously to reduce global memory transfers and to improve memory access patterns. We also introduce a dynamic strategy for choosing kernel block sizes, ensuring efficient use of device resources under a wide range of code and channel
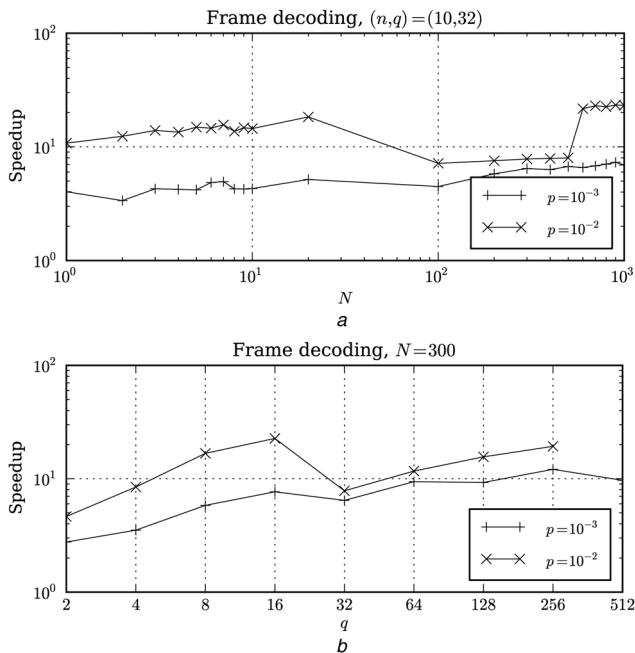
Fig. 16 *Decoding speedup for the GPU implementation of this paper (new) over the GPU implementation of [9] (old), at different channel conditions, given by $p: = P_i = P_d$; $P_s = 0$, for a range of*
a Message size N
b Alphabet size q

parameters. Specifically, we determine settings that maximise occupancy while avoiding idle time on multiprocessors. Taking these decisions at run-time also has the advantage that this automatically caters for different devices. We hope that other researchers working with CUDA will also find these techniques relevant to their work.

Although this implementation represents a considerable improvement on earlier implementations, some aspects bear further analysis. In particular, efficiency of the parallel implementation for small alphabets is still significantly lower than for larger alphabets. Aggregation of work previously done on multiple blocks into a single block has helped by improving occupancy. However, not all kernels can be improved equally, and for smaller alphabets a greater proportion of decoding time is spent in the α and β metric computations. This effect is more pronounced under better channel conditions, when the required state space is much smaller. Another aspect that may be improved is the optimisation of multiprocessor usage. Although our dynamic strategy for choosing kernel block size ensures that the usage is never low, we do not seek to optimise the value, prioritising instead a higher occupancy. Of course, improving usage is further complicated by the concurrent execution of kernels of different sizes and durations. However, it may be possible to increase overall efficiency by adapting our dynamic strategy to take into account empirical kernel timings, which may, for example, be obtained during system initialisation. Clearly, neither of the above is a trivial proposition; both are the subject of further work.

9 Acknowledgments

10 References

[1] Mercier H., Bhargava V., Tarokh V.: 'A survey of error-correcting codes for channels with symbol synchronization errors', *IEEE Commun. Surv. Tutor.*, 2010, **12**, (1), pp. 87–96, First Quarter

[2] Hu J., Duman T., Erden M., Kavcic A.: 'Achievable information rates for channels with insertions, deletions, and intersymbol interference with i.i.d. inputs', *IEEE Trans. Commun.*, 2010, **58**, (4), pp. 1102–1111

[3] Hu J., Duman T., Kurtas E., Erden M.: 'Bit-patterned media with written-in errors: modeling, detection, and theoretical limits', *IEEE Trans. Magn.*, 2007, **43**, (8), pp. 3517–3524

[4] Coumou D.J., Sharma G.: 'Insertion, deletion codes with feature-based embedding: a new paradigm for watermark synchronization with applications to speech watermarking', *IEEE Trans. Inf. Forensics Sec.*, 2008, **3**, (2), pp. 153–165

[5] Bardyn D., Briffa J.A., Dooms A., Schelkens P.: 'Forensic data hiding optimized for JPEG 2000'. Proc. IEEE Int. Symp. Circuits and Systems, Rio de Janeiro, Brazil, 15–18 May 2011

[6] Davey M.C., MacKay D.J.C.: 'Reliable communication over channels with insertions, deletions, and substitutions', *IEEE Trans. Inf. Theory*, 2001, **47**, (2), pp. 687–698

[7] Briffa J.A., Schaathun H.G., Wesemeyer S.: 'An improved decoding algorithm for the Davey–MacKay construction'. Proc. IEEE Int. Conf. Communications, Cape Town, South Africa, 23–27 May 2010

[8] Buttigieg V., Briffa J.A.: 'Codebook and marker sequence design for synchronization-correcting codes'. Proc. IEEE Int. Symp. Information Theory, St. Petersburg, Russia, 31 July–5 August 2011

[9] Briffa J.A.: 'A GPU implementation of a MAP decoder for synchronization error correcting codes', *IEEE Commun. Lett.*, 2013, **17**, (5), pp. 996–999

[10] NVIDIA CUDA C Programming Guide, NVIDIA Corporation, October 2012, version 5.0

[11] Briffa J.A., Buttigieg V., Wesemeyer S.: 'Time-varying block codes for synchronization errors: MAP decoder and practical issues', *J. Eng.*, 2014, doi: 10.1049/joe.2014.0062

[12] Lee D., Wolf M., Kim H.: 'Design space exploration of the turbo decoding algorithm on GPUs'. Proc. Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems. ACM, 2010, pp. 217–226

[13] Wu M., Sun Y., Wang G., Cavallaro J.: 'Implementation of a high throughput 3GPP turbo decoder on GPU', *J. Signal Process. Syst.*, 2011, **65**, (2), pp. 171–183

[14] Xianjun J., Canfeng C., Jaaskelainen P., Guzma V., Berg H.: 'A 122Mb/s turbo decoder using a mid-range GPU'. 2013 Ninth Int. Wireless Communications and Mobile Computing Conf. (IWCMC), 2013, pp. 1090–1094

[15] Bahl L.R., Jelinek F.: 'Decoding for channels with insertions, deletions, and substitutions with applications to speech recognition', *IEEE Trans. Inf. Theory*, 1975, **21**, (4), pp. 404–411

[16] Ratzer E.A.: 'Marker codes for channels with insertions and deletions', *Ann. Telecommun.*, 2005, **60**, pp. 29–44

[17] Briffa J.A., Schaathun H.G.: 'Improvement of the Davey–MacKay construction'. Proc. IEEE Int. Symp. Information Theory and its Applications, Auckland, New Zealand, 7–10 December 2008, pp. 235–238

[18] Granlund T.: 'GNU MP: the GNU multiple precision arithmetic library', Free Software Foundation, December 2012, edition 5.1.0. [Online]. Available at http://www.gmplib.org/gmp-man-5.1.0.pdf

[19] Rennich S.: 'CUDA C/C++ streams and concurrency'. GPU Technology Conf. NVIDIA, 2011

[20] Eckel B.: 'Thinking in C++' (Pearson Education, 2000, 2nd edn.), vol. 1

[21] Eckel B., Allison C.: 'Thinking in C++' (Pearson Education, 2003, 2nd edn.), vol. 2

[22] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, NVIDIA Corporation, 2012, version 1.0

[23] NVIDIA CUDA C Best Practices Guide, NVIDIA Corporation, October 2012, version 5.0