GPU-BASED ACCELERATION OF SYMBOL TIMING RECOVERY

Scott C. Kim[1]*, William L. Plishker*[1]*, Shuvra S. Bhattacharyya*[1]*, and Joseph R. Cavallaro*[2]

[1]Dept. of Electrical & Computer Engineering, and Institute for Advanced Computer Studies
University of Maryland, College Park, USA, {sckim, plishker, ssb}@umd.edu
[2]Dept. of Electrical & Computer Engineering, Rice University, USA, cavallar@rice.edu

ABSTRACT

This paper presents a novel implementation of graphics processing unit (GPU) based symbol timing recovery using polyphase interpolators to detect symbol timing error. Symbol timing recovery is a compute intensive procedure that detects and corrects the timing error in a coherent receiver. We provide optimal sample-time timing recovery using a maximum likelihood (ML) estimator to minimize the timing error. This is an iterative and adaptive system that relies on feedback, therefore, we present an accelerated implementation design by using a GPU for timing error detection (TED), enabling fast error detection by exploiting the 2D filter structure found in the polyphase interpolator. We present this hybrid/heterogeneous CPU and GPU architecture by computing a low complexity and low noise matched filter (MF) while simultaneously performing TED. We then compare the performance of the CPU vs. GPU based timing recovery for different interpolation rates to minimize the error and improve the detection by up to a factor of 35. We further improve the process by utilizing GPU optimization and performing block processing to improve the throughput even more, all while maintaining the lowest possible sampling rate.

Index Terms— GPU, symbol timing recovery, synchronization, coherent receiver design, DSP accelerator.

1. INTRODUCTION

When data is transmitted over a wireless communication channel, it is corrupted due to various types of noise, such as fading, oscillator drift, frequency and phase offset, receiver thermal noise, etc. At the receiver, it is also immune to noise and symbol jitter in time domain because the transmitter and receiver clocks are not the same. Therefore, a timing recovery subsystem must be able to sample the data at a correct instant and detect its peak for correct symbol timing recovery (STR). Sampling just once at the receiver is ineffective due to noise — e.g., additive white Gaussian noise (AWGN). However, a matched filter (MF) can limit the noise at the receiver and provide a high signal to noise ratio (SNR) sampling point (due to correlation gain). The goal is to obtain best SNR while avoiding inter-symbol interference (ISI). To maximize SNR for detection, the demodulator must form inner products between the incoming signal and the reference signal. That means it must time-align the locally generated reference signal with the received signal. Since the inner product is formed in a convolving filter, the demodulator must determine the precise time position to sample the input and output of the filter.

Over the decades, engineers have tried to design and implement clever receivers that not only detect but correct the incoming signal. This was first introduced in the analog domain, however, with the availability of digital integrated circuits, the process was converted over to the digital domain using transformation methods, yet the overall concept and the process remains the same. This process employs a phase-locked loop (PLL), which has 3 major components: 1. a timing error detection (TED) circuit; 2. loop filter (LF) for phase and frequency offset detection; and 3. a controlled oscillator, such as a numerically controlled oscillator (NCO), to advance or retard the sample timing so that the peak of the incoming signal is matched with the reference signal. There are several widely used methods in TED: the Gardner method [1], Mueller and Muller (M&M) algorithm [2], early-late gate algorithm (ELGA) [3, 4, 5], and maximum likelihood (ML)-based TED [3, 6].

The goal is a TED that yields high SNR, resource efficient while maintaining the lowest possible sampling rate (ideally, 1 sample per symbol (spS)), and possibly exploits data independence by using parallelism to speed up the PLL. Therefore, we focus our design using ML-based TED using MF and derivative MF [3]. ML seeks the peak of correlation output using derivative MF (dMF). ELGA is the predecessor in that it essentially finds the derivative by approximation using early, current, and late samples. This provides a relatively low complexity structure for a high performance system, which is critical in terms of designing a resource efficient transceiver. However, it is compute-intensive: it requires 3 spS and often it also requires high order filters. M&M requires 1 spS but its carrier recovery must be performed before STR. Interpolation techniques for STR have been well discussed in the past (e.g., see [7]). Polyphase interpolator based ML TED was introduced in [3, 8]. This idea was taken further by moving MF into the interpolator, and the resulting structure onto FP-

GAs in [9]. Then the lowest error resolution was achieved by using an arbitrary resampler instead of a polyphase interpolator in [10] for FPGAs as well. The polyphase filterbank is a 2D matrix structure and its lattice decomposition of multirate filters has been introduced in [11]. While these implementations have made progress towards improved TED, the computational bottlenecks of the algorithm prohibit maximum SNR for low sample rates.

Graphics processing units (GPUs) represent an attractive class of computational resources for applications that can map to it. We recognize the independence among the filterbanks and multiplication between filter coefficients and input samples. We can exploit them using multiple forms of parallelism inside the GPU to speed up the overall filtering operation, which then speeds up the overall error detection because its output is directly responsible for the output and timing error [3]. Finally, by driving the LF and NCO (running at 1 spS as derived in [3]), it essentially aligns the reference symbol (matched filtered data) to the received symbol (same principles as other digital methods and as well as analog methods), and this method works well for this type of data-aided coherence receiver (i.e., phase modulated). With decreased detection time, we can increase the throughput of the system by performing faster locking.

To accommodate the iterative and adaptive nature of PLLs, we present in this work a specific decomposition and mapping of the application onto GPUs. With our careful implementation and the availability of many threads and cores in the GPU, we also perform simultaneous STR over multiple input samples. Instead of the sample-by-sample processing in traditional digital receivers, we enable block processing of multiple symbols simultaneously to improve the throughput even further, an attractive option for modern wireless communication systems. The rest of the paper is organized as follows: we discuss the details of ML-based TED; provide an overview of NVIDIA's CUDA programming language; present our mapping of TED onto GPUs; and present design and implementation details, followed by results analysis and summary comments. Fig. 1 shows a block diagram of our targeted communication system throughout the developments of this paper.

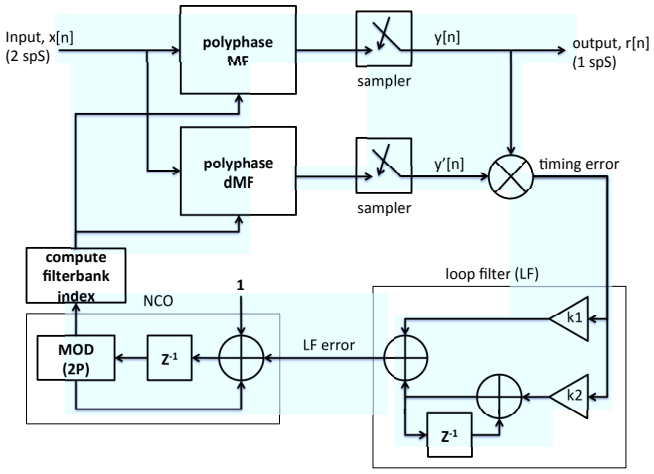

Fig. 1. Block diagram of ML-based symbol timing recovery.

2. BACKGROUND AND RELATED WORK

2.1. ML-based Timing Error Detection

The timing error in ML-based method is defined as: $t_{error}(n) = \dot{y}(n)y(n)$, where $y(n)$ is the output of the filter and $\dot{y}(n)$ is the output of the derivative filter. This equation is for low SNR. However, in practice, designers apply it for all SNR [6]. This approach can be implemented using 2 polyphase filters — a polyphase matched filter (MF) and a polyphase derivative matched filter (dMF). Polyphase filter implementation is well discussed in [3, 6]. This form of timing error detection has higher SNR than Gardner and faster locking average time [9]. Gardner is an approximation of $\dot{y}(n)y(n)$ using 2 spS by scaling zero-crossing of the eye diagram, therefore, it only uses 1 MF.

This polyphase filter based ML method offers an efficient option and is the most applicable to our purposes in this paper of efficient parallelization and low complexity implementation. The number of interpolation points corresponds to the number of filterbanks. Therefore, with increasing numbers of filterbanks, we can achieve higher interpolation points. Separate interpolation filter and MF structures result in extra processing delay. Hence, we combine the polyphase interpolator into an MF that uses 1 filter [9, 10].

This approach to ML-based TED is compute intensive, often requiring many multipliers for filtering, and filters that contain hundreds or thousands of coefficients depending on the interpolation rates and filter orders. Therefore, parallelizing and distributing the filtering tasks are attractive options and can be realized well in GPUs given their lightweight threads and data parallel processing structures. Our goal is thus to map the filtering operations across the GPU in such a way that GPU utilization is maximized, thereby offering reduced computation (from the polyphase structures employed) and higher throughput.

If the current timing estimate is too early, then the slope of the MF output is positive, so the timing phase should be advanced to an optimum sampling point. On the other hand, if the current timing estimate is too late, the slope of MF output is negative, and the timing phase should be retarded [8]. Harris and Rice presented a modern STR by integrating the MF and polyphase interpolator into a single structure as a polyphase MF [3].

We employ BPSK for the simplicity and practicality of its implementation. For BPSK implementation, we follow the overall structure presented by Gardner [1] and Frerking [12]. The idea of interpolation of such digital modems is well cov-

ered in [13, 14]. We combine polyphase TED [6] and ML error detection in [3], and then we exploit the resulting algorithm structure using GPU technology to improve the performance of STR further. This method can be extended to M-PSK or M-QAM for complex signals as well, and pursuing such extensions is a useful direction for future work.

Once a corrected sample point has been selected at the output of the MF, we discard the remaining interpolated samples. For example, given an input data stream at 2 spS, after 1:32 interpolation, we have 64 samples to choose from. We sample once at the peak, and then discard the remaining 63 interpolated values or interpolants. With this sampled point, we calculate the error discussed earlier. This timing error is then fed to an LF structure where it is used to eliminate the phase and frequency offsets.

Because of its rapid variation, we cannot use the instantaneous error to correct our timing. Therefore, we must average over some time, a fundamental method in signal detection. However, using averaging methods such as moving average can take too much time to lock. If the error does not change, then the system remains locked, but unfortunately timing jitters due to noise may persist, and these errors must be detected and corrected (filtered) each time. An LF is useful to provide such correction. Following the LF, we need an NCO to correctly drive or adjust the timing sample and feed it back to TED for future adjustment as necessary. A control circuit is used to accomplish this task by selecting a corresponding filterbank index. An overall block diagram is shown in Fig. 1.

2.2. CUDA

GPUs have been used in the past for computer graphics, but with the advancement of GPU technology in recent years, they are being used for broader classes of applications, especially those that involve processing large arrays and matrices of data. In this work, we employ an NVIDIA GPU along with the GPU-oriented parallel programming language CUDA [15].

GPUs enable efficient heterogeneous computing. Modern GPU platforms comprise of one or more CPU cores and one or more GPUs, which have many powerful arithmetic engines capable of simultaneously running large numbers of lightweight threads. The smallest unit of parallelism, called a *warp*, on a CUDA-supported device has 32 threads. The NVIDIA GTX 260, which we employ in our experiments, has 216 processor cores, which collectively allow for more than 165,000 active threads. GPUs process active threads concurrently and to enhance the efficiency of such concurrent execution, no swapping or sharing among concurrent threads occurs. The threads are allocated separately and remain that way until they complete execution. To optimize our mapping, we utilize the memory hierarchy found in GPUs, especially with use of registers, shared memory (SM), and constant memory (CM), in addition to using the available multiprocessors (MPs).

To efficiently utilize a GPU platform, the programmer must structure the implementation such that GPU threads are kept as busy as possible. This means that opportunities for independent parallel execution must be identified, and spread across the GPU for effective resource utilization. In the platform that we employ, data transfers between the CPU and GPU occur over a PCI Express bus, and such transfers can be costly in terms of speed and energy consumption. Therefore, such transfers should be minimized with as much processing as possible performed on the GPU before results are transferred back to the CPU.

3. GPU-BASED TIMING ERROR DETECTION

Our design process began with the timing recovery system shown in Fig. 1, and we first identified the computational bottleneck in this system — i.e., the part that is most arithmetically demanding. The NCO is a sequential system that simply counts up at a certain rate and wraps around after it reaches its peak. We embedded a control circuit in the NCO to scale the output of the LF so that the NCO speeds up or down depending on the error value relative to the peak. The LF is also a sequential system that multiplies the detected timing error by LF gains to track the error over the time. Calculation of gains for TED and LF (K_i and K_p) are well covered in [16, 17], and BPSK timing recovery S-curve calculation is well covered in [4]. Our contribution in this work includes mapping such calculations efficiently into GPU implementation, and structuring parallelism within and across the different calculations to maximize performance.

The PLL operates in several modes. During its initial phase or the acquisition phase, it acquires the signal using a wide bandwidth, which in turn allows more noise to enter the loop, but reduces locking time. Once it locks onto a signal it stays in the tracking phase where the bandwidth can be narrowed as much as possible and the loop stays locked as long as the noise level remains stable.

It is important to note that in our system, we fixed our LF bandwidth to be as narrow as possible going into the loop. Normally, this would cause the loop to take a long time to lock onto the signal, however, with our GPU-based TED, we are able to lock quickly due to reduced error detection time. Maintaining this narrowest possible bandwidth in tracking mode is indeed an important novel feature of our GPU-based implementation. This feature provides rapid locking without leaving the system susceptible to larger noise levels across a larger bandwidth.

Switching or changing the bandwidth on the fly is a difficult task but we are able to apply a narrow bandwidth, as described above, and to acquire and lock onto the signal all at the same time. This not only simplifies the design but gives us a smooth tracking curve that is fine tuned over the symbols.However, the design and implementation of LF, NCO,

and PLL are beyond the scope of this paper. Therefore, the polyphase MF and dMF, which are based on matrix operations, are the obvious choices to implement on the GPU, and thereby reduce the overall processing time. In a heterogeneous processing fashion, we offload our filtering operations to the GPU and work with sequential subsystems (the LF and NCO) on the CPU.

The MF and dMF are the heart of our targeted design and critical to error detection. The smaller the error or closer to the peak, it is the better. Therefore, ideally we want to upsample as much as possible. For serial processors, upsampling heavily (e.g., 1:50 interpolation) is not desirable due to the required resource usage, and such interpolation can require long computation times, and even longer times required to lock onto the peak. An alternative to high interpolation TED is to use an arbitrary resampler, as presented in [6, 10]. However, such a method is complicated to implement. The arbitrary resampler takes interpolation filtering one step further by linearly interpolating between the available output samples of the P-path polyphase interpolator. It yields highly accurate TED without the need for a high P-path interpolator (in such an interpolator the filter is large and indexing through the filterbanks is slow, resulting in high overhead).

Therefore, a key trade-off is the complexity of the design vs. the resolution of the error.Our objective is to reduce the design complexity while achieving a high interpolation rate. By using a polyphase interpolator to interpolate at a very high rate to achieve arbitrary resampler like performance, and by carefully mapping the filter operations into efficient parallel realizations on the GPU, we achieve this objective through our new approach.

This type of implementation is commonly used in embedded devices such as FPGAs. However, in FPGAs, we commonly share a single multiplier to perform the filtering by time-sharing, otherwise we end up using one multiplier per tap, which can easily amount to hundreds of multipliers for a high-performance filter, such as the interpolating filter we are using. Multipliers are costly in hardware area and can increase power consumption as well. However, in GPUs, we can perform large numbers of parallel multiplications using special arithmetic units that are available to all of the threads. In addition, we have used floating point operations to implement our design, which has eased implementation issues. Our use of floating point operations has also simplified our testing and verification processes since conversion or scaling between data types was not needed.

To map the TED onto the targeted GPU architecture, we use warps, shared memories (SMs), and groups (blocks) of multiprocessors (MPs) to optimize utilization of the NVIDIA GTX device. The filter equation has two parts, one for multiply-and-accumulate (MAC) operations to perform the inner product between two vectors — the input array and filter coefficients, and the other for indexing through the filterbanks. A typical polyphase interpolator implementation can be described as shown in Algorithm 1.

Algorithm 1 Iterative MAC operation

for $jj = 0$ to $P - 1$ **do**
  for $ii = 0$ to $M - 1$ **do**
    $prod = h[ii \times P + jj] \times r[ii]$
    $accum = accum + prod$
  end for
end for

Here, h is the filter array, r is an array of input samples, P is the interpolation rate, and M is the length of a subfilter. Thus, the original filter length is $N = P \times M$. We simply rearrange or reshape this $1 \times N$ filter vector into a $P \times M$ polyphase filter matrix. Due to its 2-dimensional structure, we use double `for`-loops to accomplish this filtering task, which serially indexes through the filter taps and input samples. We utilize multiple forms of parallelism in this structure. Specifically, we parallelize: (1) across the filterbanks (outer loop, `jj` index); (2) across the filter (inner loop, `ii` index); and (3) at a higher level, across the filter and the filterbanks.

When we parallelize across the filterbanks, we exploit the independence of accumulation across the filterbanks. The modified computation structure can be described as shown in Algorithm 2.

Algorithm 2 Parallel MAC operation

for $ii = 0$ to $M - 1$ **do**
  $prod = h[ii \times P + iy] \times r[ii]$
  $accum[iy] = accum[iy] + prod$
end for

Here, we replace `jj` with `iy`, the polyphase filterbank index, and place one filterbank per block in the GPU. Thus, each bank produces one interpolated value or an interpolant. Similarly, when we parallelize across the filter (`ii` index) itself, we simply assign one multiply operation to one thread in a block. So we simply replace `ii` with `ix`, the thread index of the block. The value of M is chosen to match the warp size or 32 threads. We eliminate `for`-loops in the GPU implementation as long as there are no data dependencies, and we can calculate the iterations independently.

Based on this approach, we combine multiple levels of parallelism to parallelize across the entire polyphase filter matrix. The resulting computational structure is shown in Algorithm 3.

Algorithm 3 Fully parallel MAC operation

$prod[ix] = h[ix \times P + iy] \times r[ix]$
$SYNC$
for $kk = 0$ to $M - 1$ **do**
  $accum = accum + prod[kk]$
end for

In this version, the filter is accessed via thread index `ix` and bank index, `iy`. We use the "sync thread" function in CUDA to synchronize our threads, and ensure that all of the products are available before they are summed. Since we are summing across a relatively small number of threads (i.e. less than 32 threads), it is not necessary to perform further reduction of the accumulator part. Therefore, we sum the products over the threads using a simple `for`-loop, as shown in Algorithm 3.

In order to optimize the GPU implementation for our experiments, we choose the number of threads per block (tpb) to be a multiple of the warp size to avoid wasting bandwidth, facilitate coalescing, and grouped memory access. Each thread is assigned a lightweight operation such as multiplication for both filters. The interpolation rate is chosen so that all MPs are uniformly loaded — i.e., the same number of blocks is launched on each MP, and also the amount of work of interest per block is the same and provides more consistent results from run to run, which allows a high interpolation rate and higher utilization of blocks in the GPU. The speedup in our implementation is achieved from invoking more GPU blocks, since there are many GPU blocks compared to threads, assuming the threads are kept busy enough (at least 64 tpb).

The output of the STR comes from the MF directly. Therefore, the higher the value of P (the number of polyphase paths), the more accurate the output will be. Furthermore, since the results are based on the actual value rather than the sign, as in [1], it is critical that we align the sample to the peak as close as possible to yield a high SNR. However, increasing P does not always yield a better result, as there is a limit on how far we can interpolate and potentially it can slow down the locking time because of the large number of interpolants that must be processed.

In the following section, we experiment with different values of M and P to find where the GPU performs best. We strive for 50% occupancy, which amounts to 256 tpb in the targeted GPU. Our design is structured to utilize SM as much as possible and use constant memory (CM) for read only data, such as filter coefficients, for faster cached access. We minimize register spills to local memory by minimizing local variable declarations and keeping the local array (e.g., product vectors) in the SM as much as possible for grouped access, and avoiding bank conflicts within SM.

4. DESIGN AND IMPLEMENTATION OF GPU ACCELERATED SYMBOL TIMING RECOVERY

We model and simulate our entire design shown in 1 using MATLAB, and then develop optimized implementations in C and CUDA targeted to the CPU and GPU, respectively. For our experiments, a BPSK signal is generated and pulse shaped at 2 spS using a root-raised-cosine (RRC) filter (with a roll-off factor of 0.5). To emulate a burst transmission or a data packet, we choose our data to be 2000 symbols or 4000 samples after pulse shaping. Typically, the system requires hundreds symbols to lock, so it is reasonable to validate the STR operation with 2000 symbols. AWGN is then added to the transmitted data to emulate the timing jitter in the receiver. It is assumed that the data has been properly modulated then demodulated to the baseband and downsampled to 2 spS immediately going into the timing recovery loop. We also assume that there is no carrier phase or frequency offset. The matched filter is also an RRC filter (with a roll-off factor of 0.5), and chosen to be of 864 taps, which reshapes it to give a $P \times M$ matrix with size 27×32. Therefore, the number of filterbanks or interpolation rate is 27 and each filterbank has 32 taps. The interpolation rate is varied in order to profile the performance of our system, and help tune the system for maximum performance. The CPU used in our experiments is a dual core Intel Xeon 3.0 GHz CPU, and the GPU is an NVIDIA GTX 260.

In our design of the STR, we have improved the NCO such that the zone test shown in [3] is not required. The zone test is used in [3] because given 2 spS, even and odd samples are continuously arriving, and the system needs to determine where the peak is. Therefore, tests are performed to determine the decision sample for each input sample. In our modified approach, we streamlined the NCO to simply count up to the total number of samples per symbol. For example, given $P = 27$ and input data at 2 spS, we simply count up to 54 samples per symbol every time. This method eliminates the need for the zone test and significantly reduces the design complexity.

From these 54 samples, only one sample is selected to be used as an error and output of the system. Therefore, running at the lowest possible sample rate is important. The scale factor is given by

$$K_v = \frac{2\pi}{S \times P},$$

where S is the pulse shape rate, and (as defined earlier), P is the interpolation rate.

When the LF error is scaled with this value, the control loop will update the corresponding filterbank index, and the updated index will be used to select the peak on the next symbol. A sample plot of the timing error and the corresponding polyphase bank index is shown in Figure 2.

4.1. Sequential Symbol Timing Recovery

In Section 3, we discuss how we exploit multiple forms of parallelism found in our TED. We first parallelize across filterbanks, followed by parallelization within individual filters. For the initial implementation, we use $P = 27$ and $M = 32$. Each filterbank is assigned to a block in the GPU, where each block has 32 threads assigned to a 32-tap filtering operation. The interpolated values of the input sample are obtained as the matrix-vector product

$$\overline{p} = \overline{\overline{H}} \times \overline{r}, \tag{1}$$

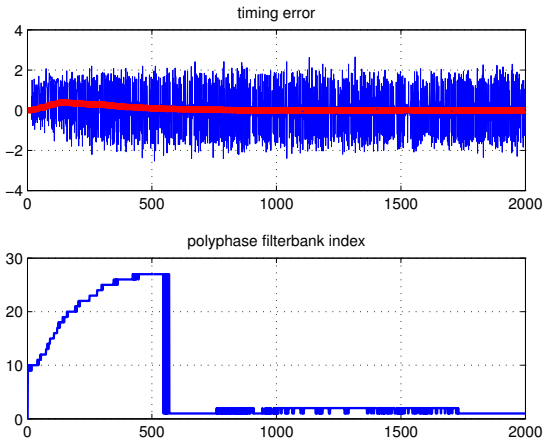

Fig. 2. An example plot of timing error and corresponding polyphase filterbank index for SNR of 10 dB.

where $\overline{\overline{H}}$ is the polyphase filter matrix (dimensions of $P \times M$), $\overline{r}$ is the input array (dimensions of $M \times 1$), and $\overline{p}$ (dimensions of $P \times 1$) gives the interpolated values of the input sample. This process is then repeated for both the MF and dMF to give us filtered results in real time.

Polyphase filtering already gives us reduced multiplications due to its use of filterbanks — given our filter size of $27 \times 32 = 864$ filter taps, we only have to perform 32 multiplications, giving us a workload savings of 96.3%. In addition, we parallelize across the polyphase filtering operations, providing significant savings in terms of computation time.

We vary the interpolation rate P to be multiples of MPs (i.e., multiples of the number of multiprocessors per core). In particular, we employ $P = 27, 54, 81$. This type of high interpolation is not desirable in typical FPGA or CPU devices, due to the large number of multipliers, and the large amount of time and memory required. However, it maps efficiently into GPU implementation, and therefore demonstrates an important kind of processing in which GPUs are especially well suited to communication system development.

We transfer data back to the CPU from the GPU every time the TED block is called. Due to the sequential and recursive nature of the PLL and NCO, they are not well suited for GPU acceleration, and thus we incur the data transfer overhead required to perform the PLL and NCO computations on the CPU.

4.2. Simultaneous Multi-Symbol Timing Recovery

So far in the paper, we have presented a one-to-one mapping of a sequential filter indexing into matrix operations by unrolling the loops across the polyphase filter matrix. However, this is still a sequential and iterative system that needs to be updated on a sample-by-sample basis. To utilize more banks and threads per kernel launch on the targeted GPU, we recognize that the input samples do not have to be processed sequentially to produce interpolated outputs. Instead, input samples can be interpolated independently on the GPU using the same kernel, while the PLL is updated sequentially as usual on the CPU. Therefore, we apply block processing on the input samples, which significantly improves throughput and minimizes data transfer overhead.

With this new grouping of input data samples, based on a block processing configuration, we introduce the notion of sub-blocks and sub-thread indexing within a single block. Each sub-block is responsible for a single set of $M = 32$ input words. Therefore, a total of $(M + K - 1)$ samples are stored in SM. Here, K is the number of input samples we wish to process and $(M - 1)$ gives the number of previous words to process. In our case, $K = 8$, so there are 8 independent processing subsystems spanning 32 input samples each, which results in 256 active threads per block, an optimum occupancy level for the targeted GPU.

Since we are processing 8 input samples or 4 symbols per kernel launch, our throughput also increases by a factor of 4. Furthermore, we also achieve a reduction in memory transfer bandwidth by a factor of 8. This is achieved because we do not have to transfer the interpolated data back to the CPU for every sample, and our rate for initiating such transfers is reduced by a factor of 8. In addition, the filter coefficients are stored in constant memory (CM), which is optimized for broadcast and for constants, and product vectors and accumulation registers are stored in shared memory (SM) for fast read-and-write operations. This also ensures coalesced or grouped data access at the block-level. Our approach to sub-grouping (sub-block organization) is shown in Fig. 3.

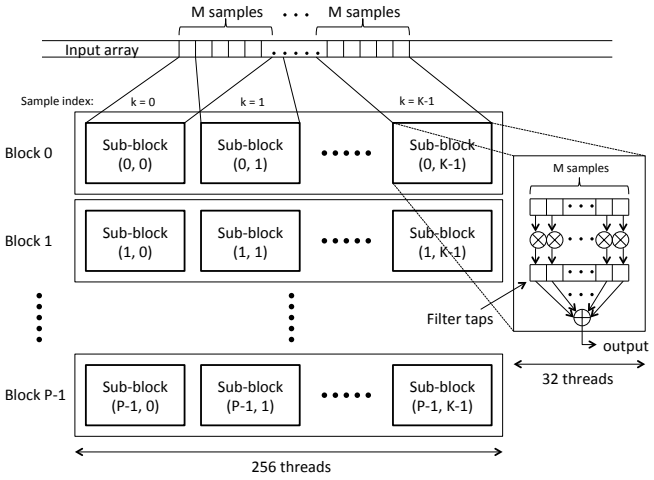

Fig. 3. Organization of 256 threads to handle eight 32-input words at a time for filtering inside the GPU. $(M + K - 1$ samples are loaded onto the shared memory, where $M = 32$ (a warp) and $K = 8$ (the number of input samples for block processing).

In this adaptive communication system design, it is not obvious how to process multiple input samples simultaneously and still perform iterative updates. However, with our TED operating in the GPU, we can interpolate many input samples all at once, while the CPU updates the sequential loop as usual using the LF and NCO. As the NCO traverses from symbol to symbol, new errors are detected and updated accordingly in the CPU. Regardless of whether input samples are processed one at a time or in groups, the architecture developed in this section provides a significant advantage in that it allows for highly optimized block processing of the data. This results in enhanced real time communication system performance, as we demonstrate experimentally in Section 5.

5. RESULTS AND ANALYSIS

In this paper, we have presented 5 different TED implementations using CPU and GPU devices. In the CPU, we used double `for`-loops to sequentially index through the filter matrices, whereas in the GPU, we exploited multiple levels of parallelism. These levels of parallelism and their associated implementations are denoted as: (P1) across the filterbank (y-direction); (P2) across the filter (x-direction); (P3) across the entire filterbank matrix (both x and y-directions); and (P4) simultaneous filtering using block processing of the input. Figure 4 compares the performance of these different TED designs for different values of the interpolation rate P. Different trade-offs between the interpolation rate and execution time are shown for the CPU-only implementation ("CPU"), and implementations P1-P4, as defined above. We used single precision floating point for all of our data, which led to more efficient memory utilization (compared to double precision), and as described earlier, simpler validation processes (compared to fixed point data).

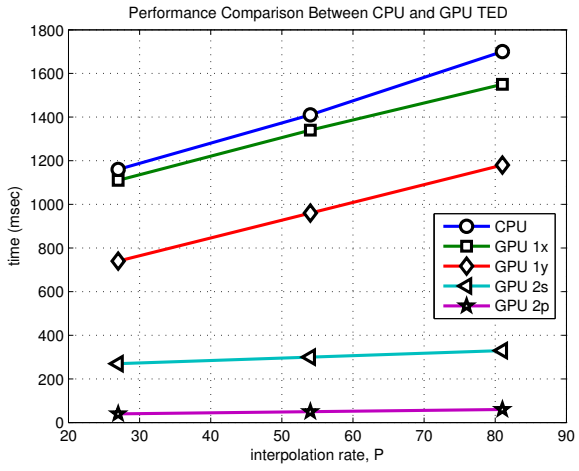

Fig. 4. Comparison of different TED designs.

The speedup times (GPU vs. CPU) are summarized in Table 1 and 2. In this experiment, we used a single GPU version of TED, which was executed in the block processing mode, and with the following additional implementation characteristics: 256 tpb, register ratio of 50% ($8192/16384$ or 7 registers per thread), SM ratio of 62.5% ($10240/16384$ or 2300 bytes per block), active blocks per SM of 4:8, and active threads per SM of 1024:1024. Furthermore, none of the interpolation rates (P values) that we experimented with exhibited occupancy as a limiting factor. Only the grid sizes (i.e., the numbers of blocks) or P values were changed in these experiments on overall achieved acceleration.

	P = 27	P = 54	P = 81
Achieved occupancy (%)	25%	50%	75%
GPU kernel time (us)	21	36	53
GPU memory transfer (us)	3.7	3.9	4.3
Global memory overall throughput (GB/s)	2.86	3.35	3.42
CPU TED time (ms)	1160	1400	1690
GPU TED time (ms)	40	40	50
TED speedup	29x	35x	33.8x

Table 1. Achieved speedup for GPU-based implementation of TED.

	P = 27	P = 54	P = 81
CPU-based (sec)	1.63	2.22	2.85
GPU-based (sec)	0.52	0.85	1.19
Overall speedup	3.13x	2.61x	2.39x

Table 2. Comparison of the overall speedup for the STR loop between CPU- and GPU-based implementations.

As expected, higher occupancy does not necessarily mean higher performance, and our experiments helped to quantify at what point this kind of saturation occurs for our GPU-based TED implementation. Our largest performance gain was achieved with $P = 54$, and an occupancy level of 50%.

Finally, we compared the overall speedup of the STR loop between CPU-based and GPU-based implementations. In this experiment, we used the block processing version of the GPU TED to maximize the speedup. The results are summarized in Table 2.

As P increases, the GPU spends more time in the kernel due to the increased matrix size, but the memory transfer time remains low even though more interpolants are transferred. However, this increased number of interpolants causes a reduction in performance, since the LF and NCO must make a sequential update, which is left to the CPU to process. This is unavoidable due to the nature of the PLL, which must adjust sample timing iteratively. This is a trade-off between high interpolation and LF/NCO update using this type of GPU-based STR. The implementation spends significant amounts of time updating sequential loops in the CPU, not performing GPU computation or memory transfers. Optimizing this trade-off

remains as a future research area for GPU-based STR.

6. CONCLUSION

In this paper, we use a coherent synchronization technique to explore ways to improve the performance of symbol timing recovery (STR) for a digital receiver. Our targeted STR system is a sequential, adaptive feedback system that must accurately time incoming digital communication symbols under stringent real time constraints. Our goal is to approach the optimal sampling peak as closely as possible while minimizing the error without using filtering that is excessively complex. We use maximum likelihood (ML) based timing error detection (TED) to interpolate the data at a high resolution, and minimize the timing error or detect the symbol peak.

Although we use already streamlined polyphase filterbanks to perform interpolation, the filterbanks create large computational loads due to the high orders of the filters involved. Therefore, we use a graphics processing unit (GPU) to accelerate the operation of TED. Our GPU-based TED enables instant error detection, narrow loop filter (LF) bandwidth (i.e., low input noise) with faster lock, low complexity, and high signal to noise ratio (SNR) with increased throughput. Our experimental results demonstrate that our design methods for real-time STR map efficiently into GPU-based implementation, and we provide analysis to quantify some of the key trade-offs involved in this kind of implementation. Building on our proposed STR implementation techniques to develop and optimize complete GPU-based transceiver systems is a useful area for future work.

7. ACKNOWLEDGMENTS

This work was supported in part by the Laboratory for Telecommunications Sciences, and the US National Science Foundation (grants ECCS-1232274 and EECS-0925942).

8. REFERENCES

[1] F. Gardner, "A BPSK/QPSK timing-error detector for sampled receivers," *IEEE Transactions on Communications*, vol. 34, no. 5, pp. 423–429, May 1986.

[2] K. Mueller and M. Muller, "Timing recovery in digital synchronous data receivers," *IEEE Transactions on Communications*, vol. 24, no. 5, pp. 516–531, May 1976.

[3] F. J. Harris and M. Rice, "Multirate digital filters for symbol timing synchronization in software defined radios," *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 12, pp. 2346–2357, 2001.

[4] U. Mengali and A. N. D'Andrea, *Synchronization Techniques for Digital Receivers*, Springer, 1997.

[5] B. Sklar, *Digital Communications: Fundamentals and Applications*, Prentice Hall, 2001.

[6] F. J. Harris, *Multirate Signal Processing for Communication Systems*, Prentice Hall, 2004.

[7] J. Vesma, M. Renfors, and J. Rinne, "Comparison of efficient interpolation techniques for symbol timing recovery," in *Proceedings of the IEEE Global Telecommunications Conference*, 1996, pp. 953–957.

[8] C. Dick, F. Harris, and M. Rice, "Synchronization in software radios-carrier and timing recovery using fpgas," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 2000.

[9] C. Dick, B. Egg, and F. Harris, "Architecture and simulation of timing synchronization circuits for the FPGA implementation of narrowband waveforms," in *Proceedings of the SDR Technical Conference and Product Exposition*, 2006.

[10] M.-u.-R. Awan and P. Koch, "Combined matched filter and arbitrary interpolator for symbol timing synchronization in SDR receivers," in *Proceedings of the International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2010, pp. 153–156.

[11] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall, 1993.

[12] M. Frerking, *Digital Signal Processing In Communications Systems*, Springer, 2010.

[13] F. M. Gardner, "Interpolation in digital modems. I. fundamentals," *IEEE Transactions on Communications*, vol. 41, no. 3, pp. 501–507, 1993.

[14] L. Erup, F. M. Gardner, and R. A. Harris, "Interpolation in digital modems. II. implementation and performance," *IEEE Transactions on Communications*, vol. 41, no. 6, pp. 998–1008, 1993.

[15] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide, Version 1.0*, June 2007.

[16] F. M. Gardner, *Phaselock Techniques*, Wiley-Interscience, third edition, 2005.

[17] H. Meyr, M. Moeneclaey, and S. A. Fechtel, *Digital Communication Receivers, Synchronization, Channel Estimation, and Signal Processing*, Wiley-Interscience, 1997.