Massively Parallel Localization of Pulsed Signal Transitions Using a GPU

Vinitha Khambadkar* Lee Barford†* Frederick C. Harris, Jr.*

*Department of Computer Science and Engineering
University of Nevada/0171
Reno, NV 89577-0171 USA

† Measurement Research Laboratory
Agilent Technologies
561 Keystone Ave. Unit 434
Reno, NV, 89503 USA

***Abstract*—Computer clock speeds which had been increasing tremendously over years is now slowing down and has reached its limit of saturation. In order to overcome this saturation of the clock speed, aggressively pursuing optimizations techniques are being developed to get more work done in each clock cycle in favor of parallel computing and concurrent programming. The GPUs massive parallel computing is now evolving as significantly faster processor than any other multi-core processors.The measurement analysis algorithm also has to be modified in order to make use of this parallelism. This paper presents one such measurement analysis, transition localization, which basically computes the high to low and vice versa transitions of a stream of signals in parallel. Measuring signals have dependencies on their previous signals when computed serially. However, now there is a parallel solution developed on the GPU which not only makes this algorithm efficient but also faster than any multi-core processors.**

Keywords: Parallel programming, Parallel algorithms, Signal analysis, Pulse measurements, Timing jitter

I. Introduction

Transition localization [1], the identification of the times at which a pulsed waveform changes levels, is an early step in a number of measurements of such waveforms, for example, total jitter, jitter histograms, and eye diagrams. The numbers of samples and pulses to be processed per waveform is increasing due to requirements for higher measurement precision and the need to find and measure malformed pulses that are ever rarer due to decreased bit error rates. Thus, it is of interest to increase the throughput (samples processed per unit time) of transition localization.

Here, we consider transition localization in a digital signal with two logic levels. The output of the analysis is the list of transition times, one for each edge in the digital signal, where the transition from a low state to a high state or from a high state to a low state occurred.

Figures 1 and 2 illustrate transitions. The voltage l is the upper state boundary of the low state, that is, the highest voltage where the signal is considered in the low state. Likewise, voltage h is the lower state boundary of the high state. When the signal is between l and h it is said to be in the intermediate state. The voltage m is usually midway between l and h. The voltages l, m, and h are determined before beginning transition localization. Usually, they are determined according to a process described in IEEE Standard 181 [1].

A crossing of m occurs whenever an adjacent pair of samples x_i and x_{i+1} are found that surround the signal's crossing of m, that is, $x_i < m \leq x_{i+1}$ or $x_{i+1} < m \leq x_i$. However, not all crossings of m constitute transitions. A transition occurs when the signal, having been less than l, rises above h. After rising above l but before attaining h, the signal crosses m an odd number of times. The time of the first of these crossings of m is the location of the transition.

This definition of transition location can be implemented easily on a single processor as follows. The meaured samples are processed in order. During the processing, the last logic level attained and the first pair of samples bracketing m after the signal enters the region are retained. When the state changes, that is, when the signal goes over h having been low or goes under l having been high, the transition time obtained from the most recent bracketing pair of samples is output. However, on multicore processors the dependency of the correct action at each sample on the previous history of the signal constitutes an obstacle to parallel implementation of transition localization. Nevertheless, there is a parallel algorithm suitable for multicore processors [2], [3].

The problem considered in this paper is transition localization using a rather different kind of parallel processor: the graphics processing unit (GPU). GPUs are massively parallel processors that have evolved to give much more faster performance on many computations when compared to multicore processors [4]. GPUs achieve this improved performance at the sacrifice of, among other things, an increased dependency on linear memory access patterns compared to multicore processors [4]. Thus, the transition localization algorithm presented in [2] and [3] is not well-suited to GPUs. Below we present a significant modification to the methods of [2] and [3] designed to produce a transition localization method with high throughput on a GPU. The correctness and performance of the proposed method is then investigated using actual measured digital waveforms.

II. Proposed Approach

We propose to use parallel scans (sometimes also called prefix scans or prefix sums) and parallel segmented scans [5] as primitive parallel operations from which to build our algorithm. A parallel scan takes as its inputs a binary associative operator $\oplus$ and an array X of n elements. Then $\texttt{scan}(X, \oplus) = [X_0, X_0 \oplus X_1, X_0 \oplus X_1 \oplus X_2, \ldots, X_0 \oplus \ldots \oplus X_n]$. For example, suppose $X = [1, 2, -5, 4]$. Then $\texttt{scan}(X, +) = [1, 3, -2, 2]$. Despite each output depending on every prior input, parallel scan can efficiently be parallelized. A good implementation of parallel scan makes efficient use of the available parallel hardware, including presenting balanced loads to the hardware [5], [6].

Segmented scan [5] is similar to parallel scan except that it additionally performs logically separate parallel scans on arbitrary contiguous segments of the input array. Segmented scans provide as much parallelism as that of parallel scans, operating on data-dependent sections. Segmented scans automatically load balance, so that time is not wasted waiting for processing of long segments to complete. As a result, they are extremely helpful in mapping many irregular computations to parallel hardware. Segmented sequences are typically represented by a combination of (1) a sequence of values and (2) a segment descriptor that determines how the sequence is divided into segments. Often the segment descriptor is just a 0-1 sequence of the same length of as the input, where a 1 indicates the start of a new segment. The output of a segmented scan is the same as if the input array were broken into separate arrays at the segment boundaries, parallel scan were applied to each of these arrays, and then the outputs concatenated into a single array. For example, consider the input array $X = [1, 2, -5, 4, 7, 3, 10]$. A second array of the same length, S is used as the segment descriptor, where $S[i]$ is one when a segment begins at $X[i]$ and zero otherwise. In this example, $S = [0, 0, 0, 1, 0, 1, 0]$. That is, the segments of X are $[1, 2 - 5]$, $[4, 7]$ and $[3, 10]$. Then $\texttt{segmented\_scan}(X, S, +) = [1, 3, -2, 4, 11, 3, 13]$.

Those interested in how scan and segmented scan are efficiently implemented on parallel hardware are referred to [5] and [6].

The basic strategy of our approach is as follows. First, we use a parallel scan to determine segments of the waveform ending in fully established transitions, that is, that end where the waveform establishes itself as high having been low or *vice versa*. Then, a segmented scan is used to locate the first crossing of m within that segment. Finally, the transition time associated with each fully established transition is extracted from the result of the segmented scan, producing the output vector of transition locations. A fuller explanation of these steps follows.

Step 1: Determine logic level associated with each input sample For each sample $x[i]$, let $level[i]$ be 1 if $x[i] \leq l$ (low logic level), 2 if $h \leq x[i]$ (high logic level), or 0 otherwise (intermediate).

TABLE I
Definition of $\oplus_1$

$\oplus_1$	0	1	2
0	0	1	2
1	1	1	2
2	2	1	2

Step 2: Calculate crossing times If the waveform crosses m between samples i and $i+1$, let $time[i]$ be the crossing time obtained by linear interpolation. Otherwise, let $time[i]$ be a floating point infinity.

Step 3: For all intermediate level samples, determine whether previous state was low or high Let the binary operator $\oplus_1$ be given as in Table 1. Perform $\texttt{scan}(level, \oplus_1)$ and put the result into the vector $levelScan$. After the scan, according to the above definition of parallel scan, $levelScan[0] = level[0]$ and $levelScan[i] = levelScan[i-1] \oplus_1 level[i]$ for $i > 0$. The intuition behind the scan using $\oplus_1$ is that where $level[i]$ is intermediate, $levelScan[i]$ is low if the most recent non-intermediate $level$ was low and $levelScan[i]$ is high if the most recent non-intermediate $level$ was high. If an initial segment of $level$, say $level[0], \ldots, level[j]$, are all intermediate, then $levelScan[0], \ldots, levelScan[j]$ will also be intermediate. All other elements of $levelScan$ will be low or high. In this sense, the scan propagates a memory of the most recent non-intermediate $level$.

Step 4: Determine segments In the the present algorithm, a segment is a consecutive subset of the samples that ends with the sample where a new logic level is attained. A new logic level is attained at index $i > 0$ if $levelScan[i-1]$ is not intermediate and $levelScan[i-1] \neq level[i]$. In this step, a binary vector seg is created where $seg[i]$ is 1 at indices i where a new logic level is attained and 0 otherwise.

Step 5: Find first crossing within each segment Perform a segmented scan $segFirstCrossing = \texttt{segmented\_scan}(time, seg, \min)$ where $\min$ is the binary operator that returns the minimum of its two arguments.

Step 6: Extract transition locations Wherever $seg[i]$ is 1, the corresponding element $segFirstCrossing[i]$ contains the time of the first crossing of m in the segment that ends at sample i. So, gathering the values $segFirstCrossing[i]$ where $seg[i] = 1$ produces a vector containing the transition times. This vector is the desired result.

III. Results

The proposed parallel algorithm for transition localization was implemented on NVIDIA Tesla C2050, GTX 275 and GTX 480 GPU cards, each used as a co-processor to an Intel x86 processor. The computers ran Red Hat Linux with NVIDIA driver version 260.19.21. CUDA Toolkit version 3.1, Thrust [7] version 1.3.0 (a library that provided the parallel scan and segmented scans), and GCC version 4.1.2.

First, to check the correctness of the proposed algorithm, a small synthetic waveform was used to check if the transitions obtained were correct. The synthetic data were chosen so that a number of cases, including unusual ones, were exercised. The synthetic signal and the resulting transition times is shown in Figure 1. The sample indices between 10-15 and 35-40 are noteworthy because they have oscillations around m, i.e., regions where the signal crosses m multiple times before changing to high or low state. The proposed algorithm generates correct results for these cases.

Second, the proposed method was tested on an actual measured waveform, a pseudo-random binary sequence, with 7,000,000 samples. This is the same actual measured waveform used in [2] and [3], which also contain a description of the measurement system used to generate and capture the waveform. Figure 2 shows the results of applying the proposed algorithm to the first 5000 samples of that waveform.

A serial version of transition localization was written into the same program as the GPU implementation. The outputs of the serial implementation running on the CPU and parallel implementation on the GPU were compared and verified to give identical results. That is, there was no additional measurement error from using the proposed algorithm instead of the serial algorithm.

The overall execution times of the serial implementation and the parallel implementation on each GPU cards were measured. These are shown in Table II and Table III. The parallel implementation test was run on various GPU cards using different data sizes. The waveforms shorter than 7,000,000 samples are the initial portions of the actual measured waveform of Figure 2. The execution times were measured on these different data sizes.

Figure 3 shows the throughput using the various GPUs tested. The results show that GTX 480 performs much faster than CPU. It is also interesting when GTX 480's performance is much better when compared with that of the Tesla C2050 and the GTX 275. This is likely because the GTX 480 has a clock rate of 700 MHz and has 480 SMs, both of which figures are much higher than those for the other GPUs.

The throughput results of obtained on the GPUs can be compared with those of the throughput graphs from [3], where the throughput of transition localization on an 18 core processor are given. The measured signals used in that study are the same as those used presently, so the results are directly comparable. The GPU performs much faster compared to a serial microprocessor code as well as a multicore processor with 18 cores. The GTX 275 GPU, which performs slowest of the three GPUs studied, gives at least 2 times the throughput of the 18 core processor. Moreover, the algorithm implemented in GTX 480 provided approximately 3.8 times the throughput of the 18 core processor and was approximately 11 times faster than the serial implementation. The best results were obtained with GTX 480 processor with approximately 5 times the throughput of the 18 core processor and approximately 15 times the throughput of the serial implementation.

IV. Conclusion

The novel contribution of this paper is to show how transition localization on large numbers of samples can be performed with high throughput using a GPU. Making use of efficient scan operations provided by the Thrust [7] library led to a short (under 200 lines) and easily-written code.

The proposed parallel GPU method was tested on a lengthy, actual measured waveform and found to produce identical results as a serial implementation. This is strong evidence that the proposed method adds no additional measurement error compared to prior approaches. Nevertheless, this new transition localization method running on several different GPUs had many times the throughput than parallel transition localization running on a multicore CPU with more cores than are typically available today on most desktop PCs.

TABLE II
Execution times in milliseconds for the different signal lengths as a result of test runs on CPU and Tesla C2050 card

Signal length	CPU (serial)	Tesla C2050
875,000	11.70 ms	3.88 ms
1,750,000	23.79 ms	6.82 ms
3,500,000	48.49 ms	12.99 ms
7,000,000	108.55 ms	25.27 ms

TABLE III
Execution times in milliseconds for the different signal lengths as a result of test runs on CPU and GTX cards

Signal length	CPU (serial)	GTX 275	GTX 480
875,000	11.70 ms	6.63 ms	2.91 ms
1,750,000	23.79 ms	10.69 ms	6.30 ms
3,500,000	48.49 ms	18.98 ms	8.53 ms
7,000,000	108.55 ms	34.88 ms	15.99 ms

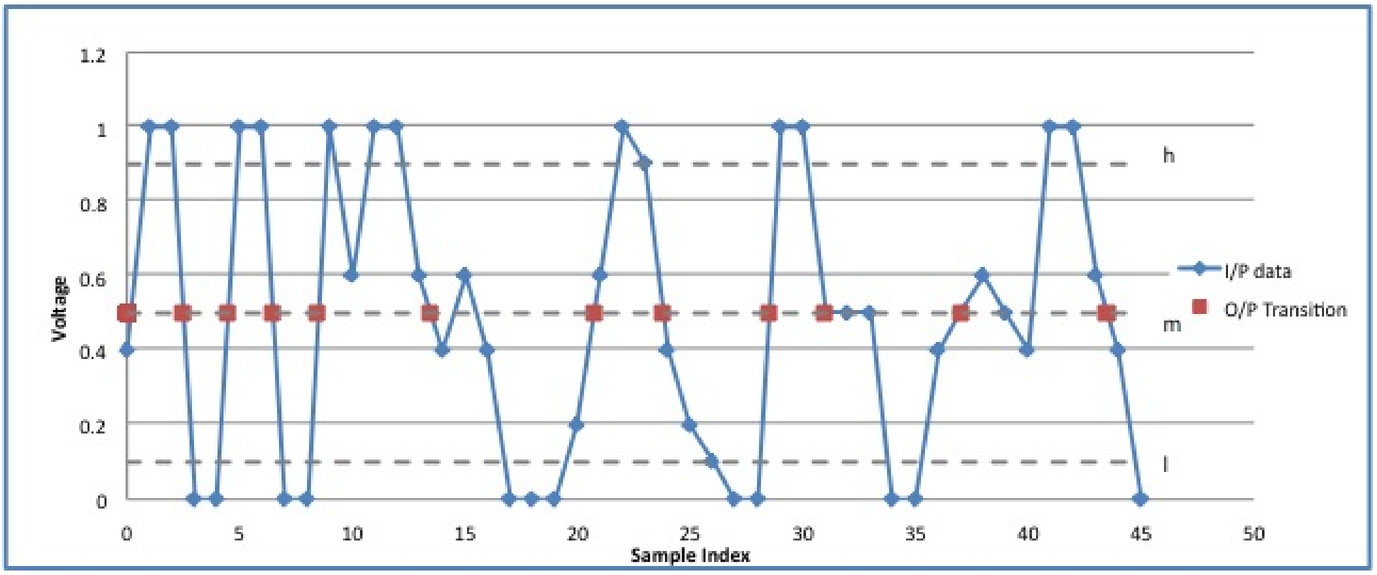

Fig. 1. Proposed parallel transition localization operating on a synthetic data set

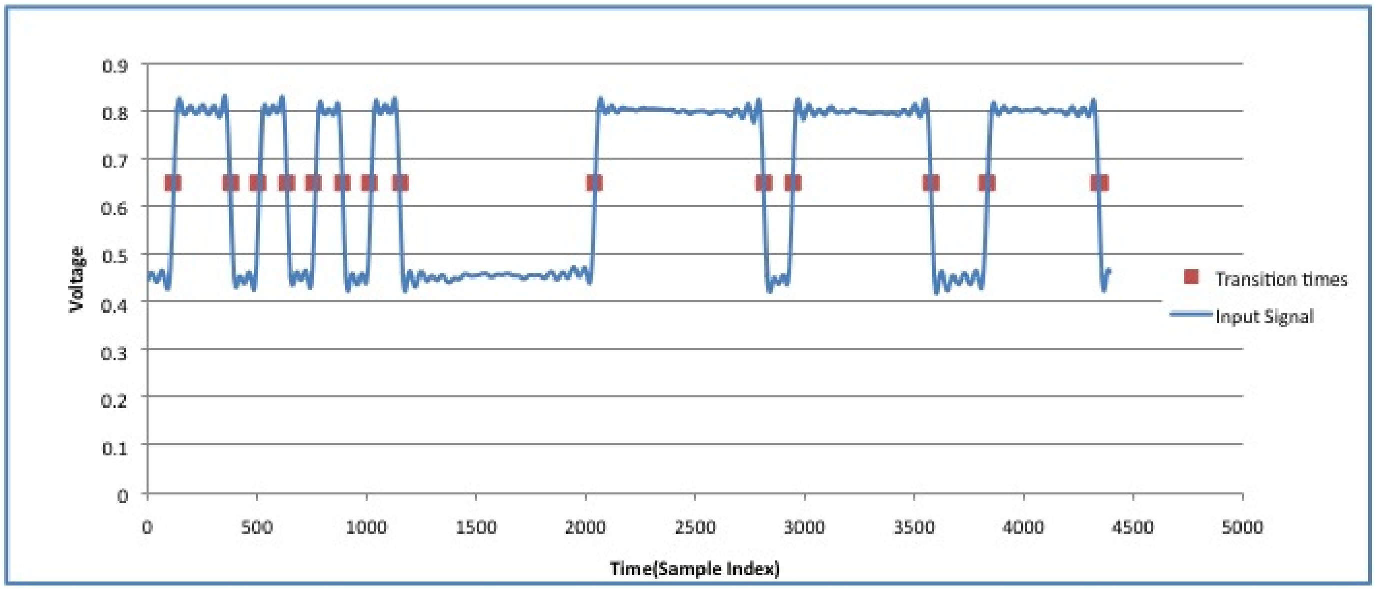

Fig. 2. Proposed parallel transition localization algorithm operating on an actual measured signal. This figure shows the first 5000 samples of the 7,000,000 sample signal used in the performance study shown in Figure 3.

V. Future Work

For a more effective analysis of parallel signal transition, we plan on automating the selection of the l, h, and m voltages. The novelty of this work will be that it is completely done on the GPU. This can be calculated with the use of histograms of the input voltage data. Histograms can be implemented to automatically calculate cut-off voltages for dynamic voltage signals. In this proposed work, histograms can be effectively used to calculate the voltage l, the upper state boundary of the low state, voltage h, the lower state boundary of the high state and the voltage m which lies midway between l and h [1]. This will allow for a fully automated system running on the GPU which will be useful under many situations.

References

[1] "IEEE standard 181-2003: Transitions, pulses, and related waveforms," IEEE, Piscataway, NJ, 2003.

[2] L. Barford, "Parallel transition localization," in *Proc. IEEE Int'l. Measurement and Instrumentation Technology Conf.*, May 2010, pp. 176–180.

[3] ——, "Speeding localization of pulsed signal transitions using multicore processors," *IEEE Trans. Instrumentation and Measurement*, vol. 60, no. 5, pp. 1588–1593, May 2011.

[4] D. B. Kirk, W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Elsevier, 2010.

[5] G. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, pp. 1526–1538, 1987.

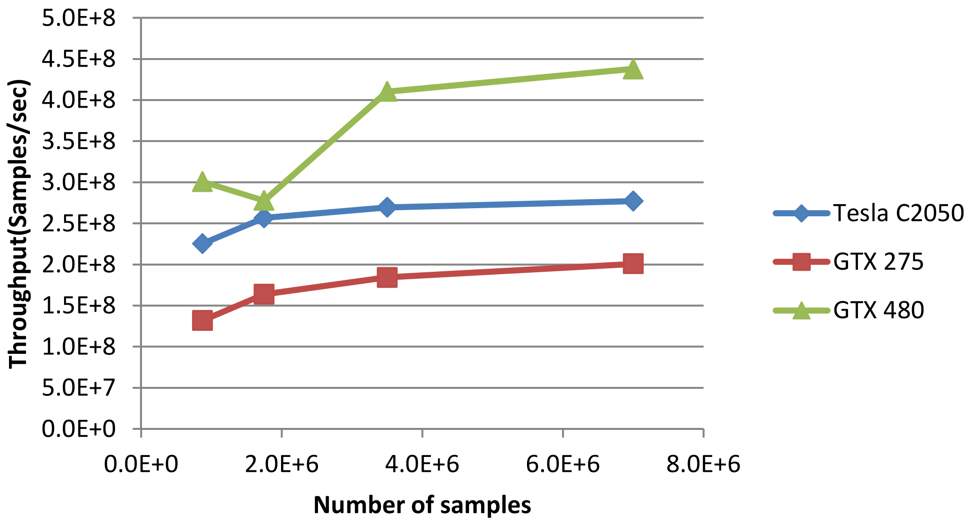

Fig. 3. Throughput of the proposed parallel transition locater for various waveform lengths and various GPUs

[6] S. Sengupta, M. Harris, M. Garland, and J. D. Owens, "Efficient parallel scan algorithms for many-core GPUs," in *Scientific Computing with Multicore and Accelerators*, in J. Kurzak, D. A. Bader, and J. Dongarra, eds. Taylor & Francis, Jan. 2011, ch. 19, pp. 413–442.

[7] J. Hoberock and N. Bell, "Thrust: A Productivity-Oriented Library for CUDA", in W. W. Hwu, ed., *GPU Computing Gems: Jade Edition*, Morgan Kauffman, 2011, pp. 359–373.