Accelerator-Oriented Algorithm Transformation for Temporal Data Mining

Debprakash Patnaik, Sean P. Ponce, Yong Cao, Naren Ramakrishnan
Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, USA
Email: {patnaik, ponce, yongcao, naren} @vt.edu

***Abstract*—Temporal data mining algorithms are becoming increasingly important in many application domains including computational neuroscience, especially the analysis of spike train data. While application scientists have been able to readily gather multi-neuronal datasets, analysis capabilities have lagged behind, due to both lack of powerful algorithms and inaccessibility to powerful hardware platforms. The advent of GPU architectures such as Nvidia's GTX 280 offers a cost-effective option to bring these capabilities to the neuroscientist's desktop. Rather than port existing algorithms onto this architecture, we advocate the need for *algorithm transformation*, i.e., rethinking the design of the algorithm in a way that need not necessarily mirror its serial implementation strictly. We present a novel implementation of a frequent episode discovery algorithm by revisiting "in-the-large" issues such as problem decomposition as well as "in-the-small" issues such as data layouts and memory access patterns. This is non-trivial because frequent episode discovery does not lend itself to GPU-friendly data-parallel mapping strategies. Applications to many datasets and comparisons to CPU as well as prior GPU implementations showcase the advantages of our approach.**

***Keywords*-GPGPU; Temporal data mining; Frequent episodes; Spike train analysis; Computational neuroscience; CUDA**

I. Introduction

Discovering frequently repeating patterns in event sequences is an important data mining problem that finds application in domains such as industrial plants/assembly lines, medical diagnostics, and computational neuroscience. Algorithms such as frequent episode discovery [1], [2], in particular, are adept at discovering patterns in neuronal spike train data using multi-electrode arrays (MEAs; shown in Fig. 1). They bring us one step closer to reverse-engineering the temporal connectivity map of neuronal circuits and yield insights into the network level activity of brain tissue. However, there is a combinatorial cost to exploring spike train datasets within the memory and processing power constraints of the single CPU.

In recent years, the peak-performance of a GPU has exceeded that of the CPU by several orders of magnitude. Intel's latest quad-core processors have a theoretical peak of 51.20 GFLOPS. Nvidia's latest single GPU card, the GeForce GTX 285, has a theoretical peak of 1062.72 GFLOPS. Application speedups up to 431x have been reported [3]. The demand for graphics applications like gaming have made GPUs widely available and inexpensive. In this paper we explore how this massively parallel

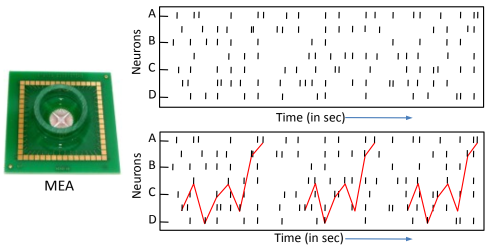

Figure 1. left: Illustration of a micro electrode array (MEA) used to record spiking activity of neurons in tissue cultures; top right: a raw spike train recorded by the MEA; and bottom right: a recurring pattern/cascade of neuronal activity.

computing platform can be used effectively to solve the challenges posed by temporal data mining. Although we focus on a specific algorithm, the issues encountered here are symptomatic of many temporal mining algorithms that must use state machines to monitor and process event streams. Hence the lessons learned from this effort will likely seed similar research efforts.

One major challenge in utilizing the GPU lies in transforming algorithms to operate efficiently on the GPU. Another challenge is understanding how to harness the GPU architecture to achieve superior performance. Data-parallel algorithms require relatively less work to map computations onto the GPU architecture. For algorithms with complex data dependencies, such as dynamic programming, it is difficult to achieve significant speedup. The nature of the temporal data mining problem addressed in this work limits the performance gain of standard hardware-oriented optimizations applied to direct ports of existing algorithms (due to their sequential dependencies implicit in processing event streams).

In this paper, we adopt the concept of Accelerator-Oriented Algorithm Transformation and introduce a new algorithm for counting occurrences of frequent episodes with temporal constraints (a key analysis task for event stream analysis) on the GPU. Here an episode is a sequential dependency of the form 'event A followed by event B followed by event C...' where there could be "don't care" or "junk" events interspersed between the pattern events. (The frequency of such episodes is defined as the maximum number of non-overlapped occurrences of the episodes in the event stream.)

The traditional approach to parallelizing existing algorithms is to start with the existing sequential algorithm, identify the data dependency patterns, and restructure the original algorithm to achieve maximum data parallelism. An alternate approach is to look at the problem being solved and formulate a decomposition using known parallel primitives. We follow the later approach and design a new algorithm by decomposing the original problem into two sub-problems: finding overlapped episode occurrences and resolving overlaps to obtain non-overlapped counts. We introduce a parallel local tracking algorithm to solve the first sub-problem which is computationally more demanding. On the other hand, the second sub-problem contributes only a very small overhead to the overall computation and is hence solved sequentially. The re-designed algorithm exhibits a higher degree of parallelism resulting in a performance gain over the sequential algorithm implemented on a single-core CPU and a previously optimized GPU implementation (*MapConcat*) [4] which achieves parallelism by mining segments of the data sequence.

II. Background

A. GPGPU Architecture

The initial purpose of specialized GPUs was to accelerate the display of 2D and 3D graphics, much in the same way that the FPU focused on accelerating floating-point instructions. However, the rapid technological advances of the GPU, coupled with extraordinary speed-ups of application "toy" benchmarks on the specialized GPUs, led GPU vendors to transform the GPU from a specialized processor to a general-purpose graphics processing unit (GPGPU), such as the NVIDIA GTX 280. To lessen the steep learning curve, GPU vendors have also introduced programming environments, such as the Compute Unified Device Architecture (CUDA).

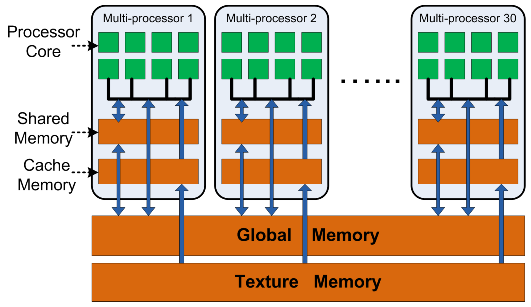

Figure 2. The NVIDIA GTX280 GPU architecture.

Processing Elements: The basic execution unit on the GTX 280 is a scalar processing **core**, of which 8 together form a **multiprocessor**. While the number of multiprocessors and processor clock frequency depends on the make and model of the GPU, every multiprocessor in CUDA executes in SIMT (Single Instruction, Multiple Thread) fashion, which is similar in nature to SIMD (Single Instruction, Multiple Data) execution. Specifically, a group of 32 threads form a **warp** and are scheduled to execute concurrently. However, when code paths within a warp diverge, the execution of all threads in a warp becomes serialized. This implies that optimal performance is attained when all 32 threads do *not* branch down different code paths.

Memory Hierarchy: The GTX 280 contains multiple forms of memory. The read-write **global memory** and read-only **texture memory** is located off-chip on the graphics card and provides the main source of storage for the GPU, as shown in Figure 2. Each multiprocessor on the GPU contains fast on-chip memory, which includes **cache memory** and **shared memory**. The texture cache is *read-only* memory providing fast access to immutable data. Shared memory, on the other hand, is user-controlled *read-write* space to provide each core with fast access to the shared address space within a multiprocessor.

Parallelism Abstractions: At the highest level, the CUDA programming model requires the programmer to offload functionality to the GPGPU as a compute **kernel**. This kernel is evaluated as a set of **thread blocks** logically arranged in a **grid** to facilitate organization. In turn, each block contains a set of **threads**, which will be executed on the same multiprocessor, with the threads scheduled in warps, as mentioned above.

B. Data Mining using GPGPUs

Many researchers have harnessed the capabilities of GPGPUs for data mining. The key to porting a data mining algorithm onto a GPGPU is to, in a sense, "dumb it down"; i.e., conditionals, program branches, and complex decision constraints are not easily parallelizable on a GPGPU and algorithms using these constraints will require significant reworking to fit this architecture (temporal episode mining unfortunately falls in this category). There are many emerging publications in this area but due to space restrictions, we survey only a few here. The SIGKDD tutorial by Guha et al. [5] provides a gentle introduction through classical problems such as k-means clustering. In [6], a bitmap technique is proposed to support counting and is used to design GPGPU variants of *Apriori* and k-means clustering. This work also proposes co-processing for itemset mining where the complicated tie data structure is kept and updated in the main memory of the CPU and only the itemset counting is executed in parallel on the GPU. A sorting algorithm on GPGPUs with applications to frequency counting and histogram construction is discussed in [7] which essentially recreates sorting networks on the GPU.

C. Temporal Data Mining

In event sequences, the notion of frequent episodes is used to express patterns of the form A $\rightarrow$ B $\rightarrow$ C, i.e., event A is followed (not necessarily consecutively) by B and B is, similarly, followed by C. It is also important to constrain the mining problem by imposing minimum and maximum delays between consecutive symbols in an episode, e.g., to look for episodes of the form $(A\xrightarrow{(2,5]}B\xrightarrow{(0,6]}C)$. This specifies that event A is to be followed by B within two to five milliseconds, and C follows B within at most six milliseconds. In either unconstrained or constrained episode mining, the occurrences of an episode allow "junk" or "don't care" events, of arbitrary length, between the event symbols of the episode. Many frequency measures [1], [8] for episodes have been defined that obey anti-monotonicity and hence search for such episodes can be structured level-wise, ala *Apriori*. The first measure to be proposed was the window based frequency measure [8]. Later the notion of *non-overlapped* occurrences count was shown to have properties that enable fast sequential counting algorithms [1].

Definition 1: An *Event Stream* is denoted by a sequence of events $\langle(E_1,t_1),(E_2,t_2),\ldots(E_n,t_n)\rangle$, where n is the total number of events. Each event (E_i,t_i) is characterized by an event type E_i and a time of occurrence t_i. The sequence is ordered by time i.e. $t_i \leq t_{i+1} \forall i = 1,\ldots,n-1$ and E_i's are drawn from a finite set ξ.

In neuroscience, a spike train is a series of discrete action potentials from several neurons generated spontaneously or as a response to some external stimulus. This data neatly fits into the frequent episodes framework of analyzing event streams.

Definition 2: An (serial) episode α is an ordered tuple of event types $V_\alpha \subseteq \xi$.
For example $(A \rightarrow B \rightarrow C)$ is a 3-node serial episode, and it captures the pattern that neuron A fires, followed by neurons B, and C in that order, but not necessarily consecutive.

Frequency of episodes: A frequent episode is one whose frequency exceeds a user specified threshold. The notion of frequency of an episode is intended to capture the repeating nature of an episode in an event sequence. In this work we shall focus on the measure of frequency defined as the size of the largest set of *non-overlapped* occurrences of an episode [1].

Temporal constraints: Episodes can be further specialized by specifying constraints on the timing of the events in episode occurrences [2]. Placing inter-event time constraints giving rise to episodes of the form:

$$(A\xrightarrow{(t^1_{low},t^1_{high}]}B\xrightarrow{(t^2_{low},t^2_{high}]}C)$$

That is, in an occurrence of episode $A \rightarrow B \rightarrow C$ let t_A, t_B, and t_C denote the time of occurrence of corresponding event types. A valid occurrence of the serial episode satisfies

$$t^1_{low} < (t_B - t_A) \leq t^1_{high}, t^2_{low} < (t_C - t_B) \leq t^2_{high}.$$

(In general, an N-node serial episode is associated with $N-1$ inter-event constraints.)

Level-wise discovery procedure for frequent episodes finds the complete set of episodes with frequency or count greater than a user-defined threshold. At each level, N-size candidates are generated from $(N-1)$-size frequent episodes and their count is determined by making a pass over the event sequence. Only those candidates whose count is greater than the threshold are retained. The event sequences typically runs very long and hence the counting step is computationally the most expensive step. For initial passes of the algorithm where we have several short episodes to count the counting task can be embarrassingly parallel by making each thread of execution count occurrences of only one episode. But in later stages there are relatively fewer but longer episodes leading to severe under-utilization. We focus our work here on the later situation and attempt to solve this counting problem.

III. Algorithm

A. Episode Counting Algorithm

The algorithm presented in [2] is based on finite state machines. This sequential algorithm for mining frequent episodes with inter-event constraints works by maintaining a data-structure (shown in Figure 3) as the read head moves down the event sequence. In this example we are counting occurrences of episode $A \overset{(5,10]}{\rightarrow} B \overset{(10,15]}{\rightarrow} C$.

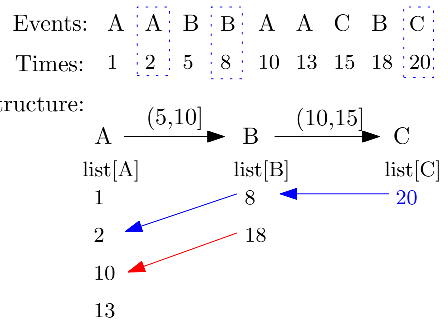

Figure 3. Illustration of the algorithm for counting non-overlapped occurrences of $A \overset{(5,10]}{\rightarrow} B \overset{(10,15]}{\rightarrow} C$. This proceeds from left-to-right of the event steam keeping track of sufficient information required to obtain the count of non-overlapped occurrences under the given inter-event time constraints.

The general approach, on finding an event that belongs to the episode, is to look up the list of occurrences of the previous symbol (or event-type). If there exists an event of the previous event type which together with the new event satisfies the inter-event constraint for the pair of event-types, then the new event is added to the data-structure under its corresponding symbol. An occurrence is said to be complete when we can add an event for the last symbol in the episode. Then the count is incremented and the data-structure is cleared. For example when $(B,18)$ arrives, it is found that the pair $(A,10)$ and $(B,18)$ satisfy the inter-event constraint $(5,10]$ (i.e. $18-10=8$ and $5<8\leq10$). And therefore the $t = 18$ is recorded in $list[B]$. On arrival of $(C, 20)$ we are able to complete one occurrence of the episode under consideration.

B. *MapConcat*

The initial idea for parallelization was to allocate one thread to count occurrences of one episode. However, when the number of candidate episodes is less than the number of cores, a one-thread per-episode model suffers from under utilization. Our first attempt to achieve a higher level of parallelism within the counting of a single episode was to segment the input event stream into several sub-streams and use one thread block to count one episode [4].

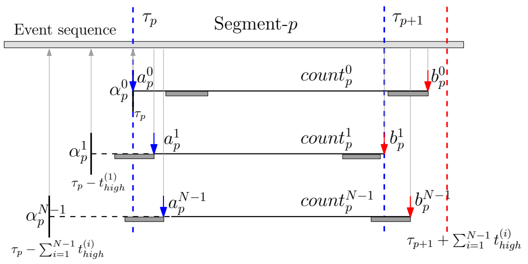

Figure 4. Illustration of a *Map* step. Multiple state machines are needed to track occurrences of N-size episode α in the p^{th}-Segment of the data. Each state machine starts at a different offset into the previous segment and continues over into the next segment to complete the last occurrence. The end-time of the first a and last occurrence b seen by a state machine are recorded beside the *count* for the reduce/concat step.

When we divide the input stream into segments, there are chances that some occurrences of an episode span across the boundaries of consecutive segments. It turns out that in order to obtain the correct count we are required to run multiple state machines within the same data segment anticipating all possible end states of the state machine in the previous segment. The final count is obtained by a reduce step where state machines for consecutive segments with matching start and end states are concatenated. The counting step (or the Map-step) is illustrated in Fig. 4 and the reduce step (or Concat-step) is shown in Fig. 5.

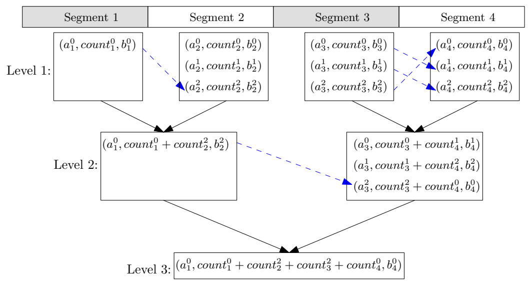
Segment 1
Segment 2
Segment 3
Segment 4
Level 1:
$(a_1^0, count_1^0, b_1^0)$
$(a_2^0, count_2^0, b_2^0)$
$(a_2^1, count_2^1, b_2^1)$
$(a_2^2, count_2^2, b_2^2)$
$(a_3^0, count_3^0, b_3^0)$
$(a_3^1, count_3^1, b_3^1)$
$(a_3^2, count_3^2, b_3^2)$
$(a_4^0, count_4^0, b_4^0)$
$(a_4^1, count_4^1, b_4^1)$
$(a_4^2, count_4^2, b_4^2)$
Level 2:
$(a_1^0, count_1^0 + count_2^2, b_2^2)$
$(a_3^0, count_3^0 + count_4^1, b_4^1)$
$(a_3^1, count_3^1 + count_4^2, b_4^2)$
$(a_3^2, count_3^2 + count_4^0, b_4^0)$
Level 3:
$(a_1^0, count_1^0 + count_2^2 + count_3^2 + count_4^0, b_4^0)$

Figure 5. Illustration of a *Concatenate* step. As noted in Fig 4, the first and last occurrence of each state-machine is used to stitch together the total count for an episode. The blue arrow in this figure indicates that the last occurrence seen by the state-machine on the left matches with the first occurrence seen by the state-machine on the right. These state-machines can then be merged together into the next level.

IV. Redesigned Algorithm

The original state-machine based algorithm has a lot of data dependency. It is difficult to increase the degree of parallelism by optimizing this algorithm. In order to transform the algorithm to map well onto the GPGPU architecture, we revisit the problem and formulate a decomposition using known parallel primitives.

A. *Problem decomposition*

We adopt a two phase approach. To identify the desired non-overlapped occurrences we first find a super-set of them that could potentially be overlapping. This can be solved with a high degree of parallelism as we demonstrate below. Next, we organize the episodes that discovered in the form of a task/job assignment problem. Each occurrence of an episode can be viewed as a job with a start time and an end time. The problem of finding the largest set of non-overlapped occurrences then becomes akin to a job scheduling problem, where the goal is to maximize the number of jobs. This can be solved with a greedy $O(n)$ algorithm.

We first pre-process the entire event stream noting the positions of events of each event-type. Then for a given episode, beginning with the list of occurrences of the start event-type in the episode, we find occurrences satisfying the temporal constraints in parallel. Finally we collect and remove overlapped occurrences in one pass. The greedy algorithm for removing overlaps requires the occurrences to be sorted by end time and the algorithm proceeds as shown in Algorithm 1. Here, for every set of consecutive occurrences, if the start time is after the end time of the last selected occurrence then we select this occurrence, otherwise we skip it and go to the next occurrence.

Algorithm 1 Obtaining the largest set of non-overlapped occurrences

```
Input: List $\mathcal{C}$ of occurrences with start and end times $(s_i, e_i)$
    sorted by end time, $e_i$.
Output: Size of the largest set of non-overlapped occurrences
    Initialize $count = 0$
    $prev_e = 0$
    for all $(s_i, e_i) \in \mathcal{C}$ do
        if $prev_e < s_i$ then
            $prev_e = e_i$; $count = count + 1$
    return $count$
```

B. *Finding occurrences in parallel*

The aim here is to find a super-set of non-overlapped occurrences in parallel. The basic idea is to start with all events of the first event-type in parallel for a given episode and find occurrences of the episode starting at each of these events. There can be several different ways in which this can be done. We shall present two approaches that gave us the most performance improvement. We shall use the episode $A \xrightarrow{(5-10]} B \xrightarrow{(5-10]} C$ as our running example and explain each of the counting strategies using this example. This example episode specifies event occurrences where an event A is to be followed by an event B within 5-10 ms and event B is to be followed by an event C within 5-10 ms delay. Note again that the delays have both a lower and an upper bound.

C. Parallel Local Tracking

In the pre-processing step, we have noted the locations of each of the event-types in the data. In the counting step, we launch as many threads as there are events in the event stream of the start event-type (of the episode). In our running example these are all events of type A. Each thread searches the event stream starting at one of these events of type A and looks for an event of type B that satisfies the inter-event time constraint $(5-10]$ i.e., $5 < t_{B_j} - t_{A_i} \leq 10$ where i, j are the indices of the events of type A and B. One thread can find multiple B's for the same A. These are recorded in a preallocated array assigned to each thread. Once all the events of type B (with an A before them) have been collected by the threads (in parallel), we need to compact these newfound events into a contiguous array/list. This is necessary as in the next kernel launch we will find all the events of type C that satisfy the inter-event constraints with this set of B's. This is illustrated in Figure 6.

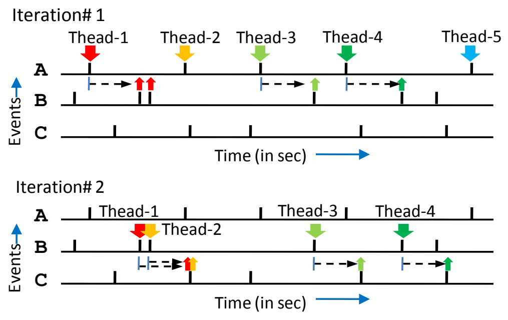

Figure 6. Illustration of Parallel local tracking algorithm (See Algorithm 2), showing 2 iterations for the episode $A \rightarrow B \rightarrow C$ with implicit inter-event constraints. Note that each thread can find multiple next-events. Further, a thread stops scanning the event sequence when event-times go past the upper bound of the inter-event constraint.

Algorithm 2 presents the overall work done in each kernel launch. In order to obtain the complete set of occurrences of an episode, we need to launch the kernel $N-1$ times where N is the size of an episode. The list of qualifying events found in the i^{th} iteration is passed as input to the next iteration. Some amount of book-keeping is also done to keep track of the start and end times of an occurrence. After this phase of parallel local tracking is completed the non-overlapped count is obtained using Algorithm 1. The

Algorithm 2 Kernel for Parallel Local Tracking

```
Input: Iteration number $i$, Episode $\alpha$, $\alpha[i]$: $i^{th}$ event-type in $\alpha$, Index list $I_{\alpha[i]}$, Data sequence $S$.
Output: $I_{\alpha[i+1]}$: Indices of events of type $\alpha[i+1]$.
  for all threads with distinct identifiers $tid$ do
    Scan $S$ starting at event $I_{\alpha[i]}[tid]$ for event-type $\alpha[i+1]$ satisfying inter-event constraint $(t^{(i)}_{low}, t^{(i)}_{high}]$.
    Record all such events of type $\alpha[i+1]$.
  Compact all found events into the list $I_{\alpha[i+1]}$.
  return $I_{\alpha[i+1]}$
```

compaction step in Algorithm 2 presents a challenge as it requires concurrent updates into a global array.

D. Lock-based compaction

Nvidia graphics cards with CUDA compute capability 1.3 support atomic operations on shared and global memory. Here we use atomic operations to perform compaction of the output array into the global memory. After the counting step each thread has a list of next-events. Subsequently each thread adds the size of its next-events list to the block-level counter using an atomic add operation and in return obtains a local offset (which is the previous value of the block-level counter). After all threads in a block have updated the block-level counter, one thread from a block updates the global-counter by adding the value of the block-level counter to it and, as before, obtains the offset into global memory. Now all threads in the block can collaboratively write into the correct position in the global memory (resulting in overall compaction). A schematic for this operation is shown for 2-blocks in Figure 7. In the results section, we refer to this method as *AtomicCompact*.

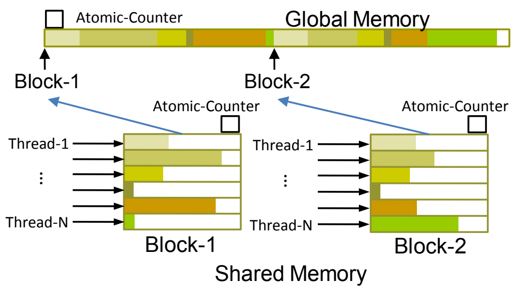

Figure 7. Illustration of output compaction using AtomicAdd operations. Note that we use atomic operations at both block and global level. These operations return the correct offset into global memory for each thread to write its next-event list into.

Since there is no guarantee for the order of atomic operations, this procedure requires sorting. The complete occurrences need to be sorted by end time for Algorithm 1 to produce the correct result.

E. Lock-free compaction

Prefix-scan is known to be a general-purpose data-parallel primitive that is a useful building block for algorithms in a broad range of applications. Given a vector of data elements $[x_0, x_1, x_2, \ldots]$, an associative binary function $\oplus$ and an identity element i, exclusive prefix-scan returns $[i, x_0, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, \ldots]$. Although the problem is seemingly sequential the first parallel prefix-scan algorithm was proposed in 1962 [9]. With recently increasing interest is GPGPU, several implementations of scan have been proposed for GPU, the most recent ones being [10] and [11]. This later implementation is available as the *CUDPP: CUDA Data Parallel Primitives Library* and forms part of the CUDA SDK distribution.

Our lock-free compaction is based on prefix-sum and we reuse the implementation from CUDPP library. Since the scan based operation guarantees ordering we modify our counting procedure to count occurrences backwards starting from the last event. This results in the final set of occurrences to be automatically ordered by end-time and therefore completely eliminates the need for sorting (as required by atomic operations based approach).

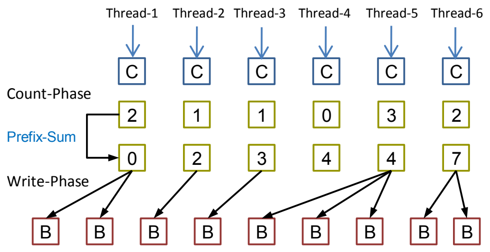

Figure 8. Illustration of output compaction using Scan primitive. Each iteration is broken into 3 kernel calls: Counting the number of next events, Using scan to compute offset into global memory, finally launching count-procedure again but this time allowing write operations to the global memory

The CUDPP library provides a compact function which takes an array d_{in}, an array of $1/0$ flags and returns a compacted array d_{out} of corresponding only the "valid" values from d_{in} (it internally uses `cudppScan`). In order to use this, our counting kernel is now split into three kernel calls. Each thread is allocated a fixed portion of a larger array in global memory for its next-events list. In the first kernel, each thread finds its events and fills up its next-events list in global memory. The `cudppCompact` function (implemented as two GPU kernel calls) compacts the large array to obtain the global list of next-events. A difficulty of this approach is that the array on which `cudppCompact` operates is very large resulting in a scattered memory access pattern. We refer to this method as *CudppCompact*.

In order to further improve performance, we adopt a counter-intuitive approach. We again divide the counting process into three parts. First, each thread looks up the event sequence for suitable next-events but instead of recording the events found, it merely counts and writes the count to global memory. Then an exclusive scan is performed on the recorded counts. This gives the offset into the global memory where each thread can write its next-events list. The actual writing is done as the third step. Although each thread looks up the event sequence twice (first to count, and second to write) we show that we nevertheless obtain better performance. This entire procedure is illustrated in Figure 8. We refer to this method of compaction as *CountScanWrite* in the ensuing results section.

Note that prefix-scan is essentially a sequential operator applied from left-to-right to an array. Hence the memory write operations into memory locations generated by prefix scan preserve order. The sorting step (i.e. sorting occurrences by end time,) required in the lock-based compaction can be completely avoided by counting occurrences backwards starting from the last event-type in the episode.

V. Results

The hardware used for obtaining the performance results are given in Table I:

Table I
HARDWARE USED FOR PERFORMANCE ANALYSIS

GPU	Nvidia GTX 280
Memory (MB)	1024
Memory Bandwidth (GBps)	141.7
Multiprocessors, Cores	30, 240
Processor Clock (GHz)	1.3
CPU	Intel Core 2 Quad Q8200
Processors	4 (only 1 used)
Processor Clock (GHz)	2.33
Memory (MB)	4096

A. Test datasets and algorithm implementations

The datasets used here are generated from the non-homogeneous Poisson process model for inter-connected neurons described in [2]. This simulation model generates fairly realistic spike train data. For the datasets in this paper, a network of 64 artificial neurons was used. The random firing rate of each neuron was set at 20 spikes/sec to generate sufficient noise in the data. Four 9-node episodes were embedded into the network by suitably increasing the connection strengths for pairs of neurons. Spike train data was generated by running the simulation model for different durations of time. Table II gives the duration and number of events in each dataset.

Table II
DETAILS OF THE DATASETS USED

Data -Set	Length (in sec)	# Events	Data -Set	Length (in sec)	# Events
1	4000	12,840,684	5	200	655,133
2	2000	6,422,449	6	100	328,067
3	1000	3,277,130	7	50	163,849
4	500	1,636,463	8	20	65,428

In Section V-B, we compare the performance of our new algorithm to *MapConcat* and a CPU implementation of the original algorithm described in Section II-C. The CPU version is sequential as it is clear for our discussion that parallelizing the the episode mining task is non-trivial and hence runs on only one core of the CPU. In order to analyze the lock-based and lock-free compaction strategies, we present the performance of a lock-based method, *AtomicCompact*, and two lock-free methods, *CudppCompact* and *CountScanWrite*, as shown in Section V-C.

B. Comparisons of performance

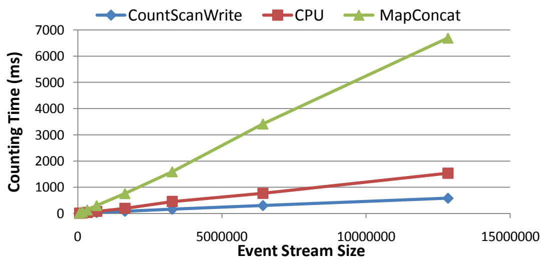

Figure 9. Performance of MapConcat compared with the CPU and best GPU implementation, counting 30 episodes in Datasets 1-8.

We compare MapConcat performance to the best CPU and GPU versions by having each algorithm count 30 episodes. MapConcat counts one episode per multiprocessor, and the GTX 280 contains 30 multiprocessors, so we count 30 episodes to fully utilize the GPU to make a fair comparison. The CPU counts the episodes in parallel by using one thread per core, and distributing the count of 30 episodes amongst the threads.

MapConcat is clearly a poor performer compared to the CPU, with up to a 4x slowdown. Compared to our best GPU method, *MapConcat* is up to 11x slower. This is due to the overhead induced by the merge step of *MapConcat*. Although at the multiprocessor level each episode is counted in parallel, the logic required to obtain the correct count is complex.

We run the CUDA Visual Profiler on *MapConcat* and one of our redesigned algorithms, *CountScanWrite*. Dataset 2 was used for profiling each implementation. Due to its complexity, *MapConcat* exhibited poor features such as large amounts of divergent branching and a large total number of instructions executed, as shown in Table III. Comparatively, the *CountScanWrite* implementation only exhibits divergent branching.

Table III
CUDA VISUAL PROFILER RESULTS

	MapConcat	CountScanWrite
Instructions	93,974,100	8,939,786
Branching	27,883,000	2,154,806
Divergent Branching	1,301,840	518,521

The best GPU implementation is compared to the CPU by counting a single episode. This is the case where the GPU was weakest in previous attempts, due to the lack of parallelization when the episodes are few. In terms of the performance of our best GPU method, we achieve a 6x speedup over the CPU implementation on the largest dataset, as shown in Figure 10.

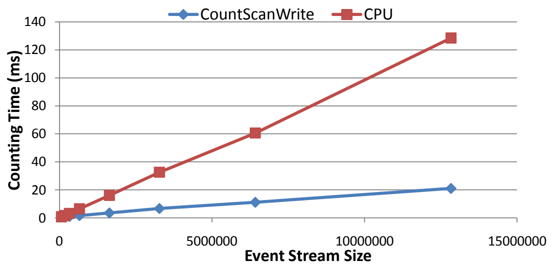

Figure 10. Performance comparison of the CPU and best GPU implementation, counting a single episode in Datasets 1 through 8.

C. Analysis of the new algorithm

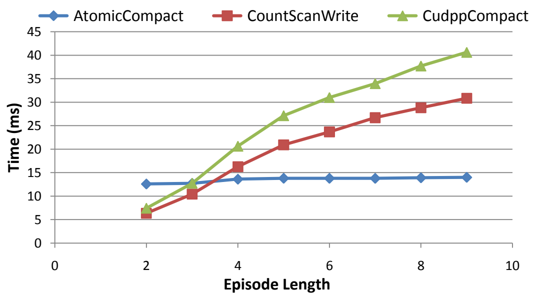

Figure 11. Performance of algorithms with varying episode length in Dataset 1.

Figure 11 presents timing information for the three compaction methods as a function of episode length. Compaction using CUDPP is the slowest of the GPU implementations, as expected. It requires each data element to be either in or out of the final compaction, and does not allow for compaction of groups of elements. For small episode lengths, the *CountScanWrite* approach is best because sorting can be completely avoided. However, at longer episode lengths, compaction using lock-based operators shows the best performance. This method of compaction avoids the need to perform a scan and a write at each iteration, at the cost of sorting the elements at the end. The execution time of the *AtomicCompact* is nearly unaffected by episode length, which seems counter-intuitive because each level requires a kernel launch. However, each iteration also decreases the total number of episodes to sort and schedule at the end of the algorithm. Therefore, the cost of extra kernel invocations is offset by the final number of potential episodes to sort and schedule.

We find that counting time is related to episode frequency as shown in Figure 12. There is a linear trend, with episodes

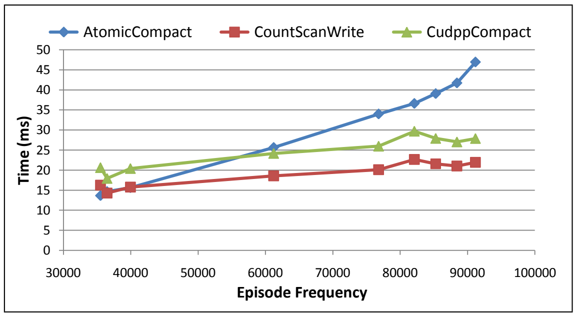

Figure 12. Performance of algorithms with varying episode frequency in Dataset 1.

of higher frequency requiring more counting time. The lock-free compaction methods follow an expected trend of slowly increasing running time because there are more potential episodes to track. The method that exhibits an odd trend is the lock-based compaction, *AtomicCompact*. As the frequency of the episode increases, there are more potential episodes to sort and schedule. The running time of the method becomes dominated by the sorting time as the episode frequency increases.

Another feature of Figure 12 that requires explanation is the bump where the episode frequency is slightly greater than 80,000. This is because the running time is affected, not by the final non-overlapped count, but by the total number of overlapped episodes found before the scheduling algorithm is applied (to remove overlaps). The x-axis displays non-overlapped episode frequency, where the run-time is actually affected more by the overlapped episode frequency.

We used the CUDA Visual Profiler on the other GPU methods. They had similar profiler results as the *CountScan-Write* method. The reason is that the only bad behavior exhibited by the method is divergent branching, which comes from the tracking step. This tracking step is common to all of the GPU method of the redesigned algorithm.

VI. Conclusion

Just as algorithms for secondary storage are quite distinct from algorithms for main memory (e.g., two-phase merge sort is preferable over quick-sort), we have shown similarly that approaches for temporal data mining on a GPU must adopt fundamentally different strategies than that on a CPU. Through this work, we aim to have conveyed that even with an application such as frequent episode mining that is unequivocally ‘sequential’ in nature, it is possible to obtain reasonable speedup by an order of magnitude using careful redesign and algorithm-oriented transformation. A key lesson from our efforts is the importance of investigating both “in-the-large” and “in-the-small” issues. We have demonstrated the effectiveness of our approach to handling large scale event stream datasets modeled by inhomogeneous Poisson processes.

VII. Future Work

Similar to our motivations stemming from computational neuroscience, there are a large class of data mining tasks in bioinformatics, linguistics, and event stream analysis that require analysis of sequential data. Our work opens up the interesting issue of the extent to which finite state-machine based algorithms for these tasks can be accelerated using GPU platforms. Are there fundamental limitations to porting such algorithms on GPUs? We believe there are and hope to develop a theoretical framework to investigate GPU-transformation issues for these algorithms. Second, we aim to study the development of streaming versions of these algorithms where approximate results are acceptable but near real-time responsiveness is important.

References

[1] S. Laxman *et al.*, “Discovering frequent episodes and learning hidden markov models: A formal connection,” *IEEE TKDE*, vol. Vol 17, no. 11, pp. pages 1505–1517, Nov 2005.

[2] D. Patnaik *et al.*, “Inferring neuronal network connectivity from spike data: A temporal data mining approach,” *Scientific Programming*, vol. 16(1), pp. 49–77, 2008.

[3] S. Ryoo *et al.*, “Optimization principles and application performance evaluation of a multithreaded gpu using cuda,” in *Proc. PPoPP*, 2008, pp. 73–82.

[4] Y. Cao *et al.*, “Towards chip-on-chip neuroscience: Fast mining of frequent episodes using graphics processors,” *arXiv.org*, 2009.

[5] S. Guha *et al.*, “Data visualization and mining using the gpu,” Tutorial at ACM SIGKDD’05, 2005.

[6] W. Fang *et al.*, “Parallel data mining on graphics processors,” Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS08-07, Oct 2008.

[7] N. Govindaraju *et al.*, “Fast and approximate stream mining of quantiles and frequencies using graphics processors,” in *Proc. SIGMOD’05*, 2005, pp. 611–622.

[8] H. Mannila *et al.*, “Discovery of frequent episodes in event sequences,” *DMKD*, vol. 1, no. 3, pp. 259–289, 1997.

[9] K. E. Iverson, *A Programming Language*. Wiley, New York, 1962.

[10] M. Harris *et al.*, *GPU Gems 3*. Addison Wesley, 2007, ch. Parallel prefix sum (scan) with CUDA.

[11] S. Sengupta *et al.*, “Scan primitives for gpu computing,” in *Graphics Hardware 2007*. ACM, Aug. 2007, pp. 97–106.