International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Parallel Mining of Neuronal Spike Streams on Graphics Processing Units

Yong Cao · Debprakash Patnaik · Sean Ponce · Jeremy Archuleta · Patrick Butler · Wu-chun Feng · Naren Ramakrishnan

Received: date / Accepted: date

Abstract Multi-electrode arrays (MEAs) provide dynamic and spatial perspectives into brain function by capturing the temporal behavior of spikes recorded from cultures and living tissue. Understanding the firing patterns of neurons implicit in these spike trains is crucial to gaining insight into cellular activity. We present a solution involving a massively parallel graphics processing unit (GPU) to mine spike train datasets. We focus on mining frequent episodes of firing patterns that capture coordinated events even in the presence of intervening background events. We present two algorithmic strategies—*hybrid mining* and *two-pass elimination*—to map the finite state machine-based counting algorithms onto GPUs. These strategies explore different computation-to-core mapping schemes and illustrate innovative parallel algorithm design patterns for temporal data mining. We also provide a multi-GPU mining framework, which exhibits additional performance enhancement. Together, these contributions move us towards a real-time solution to neuronal data mining.

Keywords Frequent episode mining · graphics processing unit (GPU) · spike train datasets · computational neuroscience · GPGPU

1 Introduction

Computational neuroscience is undergoing a data revolution similar to what biology experienced (and is still experiencing) beginning in the 1990s. Techniques such as electro-encephalography (EEG), functional magnetic resonance imaging (fMRI), and multi-electrode arrays (MEAs) provide spatial and temporal perspectives into brain function. It is now possible to measure the detailed electrophysiology of neural information flow networks through multiple modalities. However data analysis techniques to process these massive temporal streams and understand dynamic responses

Yong Cao
Virginia Polytechnic Institute and State University
Blacksburg, VA
E-mail: ycao@vt.edu

to stimuli and environmental factors are still in their infancy. Data mining algorithms are important in understanding the pathways that get triggered by sensory input and to expose the underlying anatomical connectivity of networks. Uncovering such networks can help understand the response of information pathways to the application of different kinds of stimuli.

A parallel revolution is happening in the related area of brain-computer interfaces [1]. Scientists are now able to not only analyze neuronal activity in living organisms, but also to understand the intent implicit in these signals and use this information as control directives to operate external devices. In a landmark study [14], Serruya et al. described how hands-free operation of a cursor can be achieved in real-time by monitoring the activities of just a few neurons in the motor cortex of a monkey. A brain-computer interface for controlling a humanoid robot using signals recorded from human scalp is described in [3]. Again, the real-time, responsive behavior of the interface is a remarkable feature that bodes well for its success. Even cognitive understanding can now be achieved algorithmically: Mitchell et al. [12] describe how they are able to map the semantics of words and nouns to regions of brain activity. There are now many technologies for modeling and recording neuronal activity including fMRI (functional magnetic resonance imaging), EEG (electroencephalography), and multi-electrode arrays (MEAs). In this paper, we focus exclusively on event streams gathered through multi-electrode array (MEA) chips for studying neuronal function although our algorithms and implementations are applicable to a wider variety of domains.

An MEA records spiking action potentials from an ensemble of neurons which, after various pre-processing steps, yields a spike train dataset providing real-time, dynamic, perspectives into brain function (see Figure 1). Identifying sequences (e.g., cascades) of firing neurons, determining their characteristic delays, and reconstructing the functional connectivity of neuronal circuits are key problems of interest. This provides critical insights into the cellular activity recorded in the neuronal tissue.

A spike train dataset can be modeled as a discrete symbol stream, where each symbol/event type corresponds to a specific neuron (or clump of neurons) and the dataset encodes occurrence times of these events. One class of patterns that is of interest is *frequent episodes* which are repetitive patterns of the form $A \to B \to C$, i.e., event A is followed by (not necessarily consecutively) B is followed by C. This is the primary class of patterns studied here.

With rapid developments in instrumentation and data acquisition technology, the size of event streams recorded by MEAs has concomitantly grown, leading us to exhaust the abilities of serial computation. For instance, just a few minutes of cortical recording from a 64-channel MEA can easily generate millions of spikes. Furthermore, it is not uncommon for a typical MEA experiment to span days or even months [15]. Hence it is imperative that algorithms support fast, real-time computation, data mining, and visualization of patterns.

In this paper, we address the challenge of finding a real-time solution for temporal data mining, which is under high demand by neuroscientists. Instead of providing a parallel data mining system on a supercomputer, we seek a much more economical solution on a desktop computer powered by many-core graphics processing units (GPUs). General-purpose computing on the GPU (GPGPU) gives us an alternative

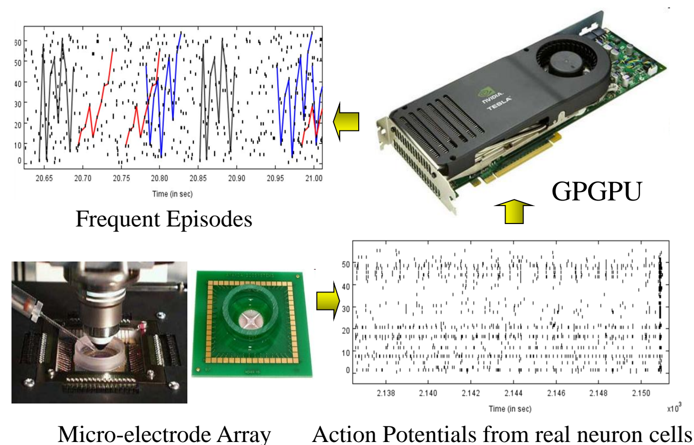


Fig. 1 Chip-on-Chip Neuroscience. Spike trains recorded from a multi-electrode array (MEA) are mined by a GPU to yield frequent episodes which can be summarized to reconstruct the underlying neuronal circuitry.

approach for solving data mining problems in a parallel computing fashion. As is well known, GPUs are designed to shine for a special type of applications—data parallel applications—for which it is easy to map independent computation units onto the processing cores of GPU. However, temporal data mining does not fall into this category of applications and makes it difficult to fully utilize the computational power of GPU. In this paper, we study the challenge of creating an intuitive and balanced computation-to-core mapping scheme for GPUs and propose real-time solutions with significant performance increases compared with traditional CPU implementation. More specifically, our technical contributions are:

1. A hybrid mining approach that leverages the advantages of two computation-to-core mapping schemes and automatically scales according to the number of input episodes.
2. A two-pass approach that first performs an episode elimination step with relaxed constraints, resulting in a large performance gain. The potential false positives introduced in the first pass are then eliminated in a second pass. In both, we employ computation-to-core mapping schemes suitable for frequent episode mining fully utilize the massively parallel computing architecture of GPUs.
3. A multi-GPU mining implementation that further enhances the performance of mining of massive datasets.

This paper extends our previous work [4] by providing a multi-GPU implementation of the episode mining algorithm and a detailed performance analysis comparing the multi-GPU implementation with our original approach.

2 Problem Statement

A spike train dataset can be modeled as an event stream, where each symbol/event type corresponds to a specific neuron (or clump of neurons) and the dataset encodes occurrence times of these events over the time course.

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.

One class of interesting patterns that we wish to discover are frequently occurring groups of events (i.e., firing cascades of neurons) with some constraints on ordering and timing of these events. This is captured by the notion of episodes, the original framework for which was proposed by Mannila *et al* [11].

Definition 2 An (serial) episode α is an ordered tuple of event types $V_\alpha \subseteq \xi$.

For example $(A \to B \to C \to D)$ is a 4-node serial episode, and it captures the pattern that neuron A fires, followed by neurons B, C and D in that order, but not necessarily without intervening 'junk' firings of neurons (even possibly neurons A, B, C, or D). This ability to intersperse noise or don't care states, of arbitrary length, between the event symbols in the definition of an episode is what makes these patterns non-trivial, useful, and comprehensible. Serial episodes can also have repeated event types, for example, $(A \to B \to A \to D)$ is an episode where the event A occurs twice, once before B and again after B and before D.

Frequency (or Support) of an episode: The notion of frequency of an episode can be defined in several ways. In [11], it is defined as the fraction of windows in which the episode occurs. Another measure of frequency is the non-overlapped count which is the size of the largest set of non-overlapped occurrences of an episode expressed as a fraction of the total number of events in the data. Two occurrences are non-overlapped if no event of one occurrence appears in between the events of the other. In the event stream shown in the following example (1), there are at most two non-overlapped occurrences of the episode $A \to B$, although there are 8 occurrences in total.

$$\langle (A, 1), (A, 2), (B, 5), (B, 8), (A, 10), (A, 13), (C, 15), (B, 18), (C, 20) \rangle \quad (1)$$

We use the non-overlapped occurrence count as the frequency or support measure of choice due to its strong theoretical properties under a generative model of events [9]. It has also been argued in [13] that, for the neuroscience application, counting non-overlapped occurrences is natural because episodes then correspond to causative, "syn-fire", chains that happen at different times again and again.

Temporal constraints: Besides the frequency threshold, a further level of constraint can be imposed on the definition of episodes. In multi-neuronal datasets, if

one would like to infer that neuron A's firings cause a neuron B to fire, then spikes from neuron B cannot occur immediately or spontaneously after A's spikes due to axonal conduction delays. These spikes cannot also occur too much later than A for the same reason. Such minimum and maximum inter-event delays are common in other application domains as well. Hence we place inter-event time constraints between consecutive pairs of events giving rise to episodes such as:

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

In a given occurrence of episode $A \to B \to C$, let t_A, t_B, and t_C denote the times of occurrence of corresponding event types. A valid occurrence of the serial episode satisfies

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

In general, an N-node serial episode is associated with $N-1$ inter-event constraints. In example (1) there is exactly one occurrence of the episode $A \overset{(5,10]}{\to} B \overset{(10,15]}{\to} C$ satisfying the desired inter-event constraints, i.e., $\langle (A,2),(B,8),(C,20)\rangle$.

The problem we address in this paper is defined as follows.

PROBLEM: Given an event stream $\{(E_i, t_i)\}$, $i \in \{1 \dots n\}$, a set of inter-event constraints $I = \{(t^k_{low}, t^k_{high}]\}$,$k \in \{1 \dots l\}$, find all serial episodes α of the form:

$$\alpha = \langle E_{(1)} \xrightarrow{(t^{(1)}_{low}, t^{(1)}_{high}]} E_{(2)} \dots \xrightarrow{(t^{(N-1)}_{low}, t^{(N-1)}_{high}]} E_{(N)} \rangle$$

such that the non-overlapped occurrence counts of each episode α is $\geq \theta$, a user-specified threshold. Here $E_{(.)}$'s are the event types in the episode α and $(t^{(.)}_{low}, t^{(.)}_{high}]$'s $\in I$ are the corresponding inter-event constraints.

Typically in neuroscience data the number of frequent episodes range from 100 to 1000 depending on the frequency (i.e. support) threshold. In real data frequent episodes of size larger than 10 (i.e. having more than 10 event types) are seldom seen.

3 Prior Work

We review prior work in two categories: mining frequent episodes and data mining using GPGPUs.

Mining Frequent Episodes: The overall mining procedure for frequent episodes is based on level-wise mining. Within this framework there are two broad classes of algorithms: window-based [11] and state machine-based [8,9], and they primarily differ in how they define the frequency of an episode. The window-based algorithms define frequency of an episode as the fraction of windows on the event sequence in which the episode occurs. The state machine-based algorithms are more efficient and define frequency as the size of largest set of non-overlapped occurrences of an episode. Within the class of state machine algorithms, serial episode discovery using non-overlapped counts was described in [9], and their extension to temporal

constraints is given in [13]. With the introduction of temporal constraints the state machine-based algorithms become more complicated. They must keep track of what part of an episode has been seen, which event is expected next and, when episodes interleave, they must make a decision of which events to be used in the formation of an episode.

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 one sense, "dumb it down" or simplify it. Specifically, the algorithms need to reduce the use of conditionals, branching, and complex decision constraints, which are not easily parallelizable on a GPU, and thus adversely impact performance. However, designing algorithms under such constraints require significant reworking to fit this architecture, and unfortunately, temporal episode mining 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. [7] provides a gentle introduction to the aspects of data mining on the GPU through problems like k-means clustering, reverse nearest neighbor (RNN), discrete wavelet transform (DWT), and sorting networks. In [5], 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 CPU and only the itemset counting is executed in parallel on the GPU. A sorting algorithm on the GPU with applications to frequency counting and histogram construction is discussed in [6] which essentially recreates sorting networks on the GPU. Li et al. [10] present a 'cut-and-stitch' algorithm for approximate learning of Kalman filters. Although this is not a GPGPU solution *per se*, we point out that our proposed approach shares with this work the difficulties of mining temporal behavior in a parallel context.

4 GPU Architecture

To understand the algorithmic details behind our approach, we provide a brief overview of the GPU and its 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 advances of GPU's many-core architecture, 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, as shown in Figure 2. GPUs provide a massively parallel computing architecture that can support concurrent execution of tens of thousands of threads and manage trillions of threads at the same time. To lessen the steep learning curve, GPU vendors also introduced programming environments, such as the NVIDIA's Compute Unified Device Architecture (CUDA).

Processing Elements: The basic execution unit on the GTX 280 is a scalar processing **core**, of which 8 together form a **multiprocessor**. The number of multiprocessors and processor clock frequency depends on the make and model of the GPU. For ex-

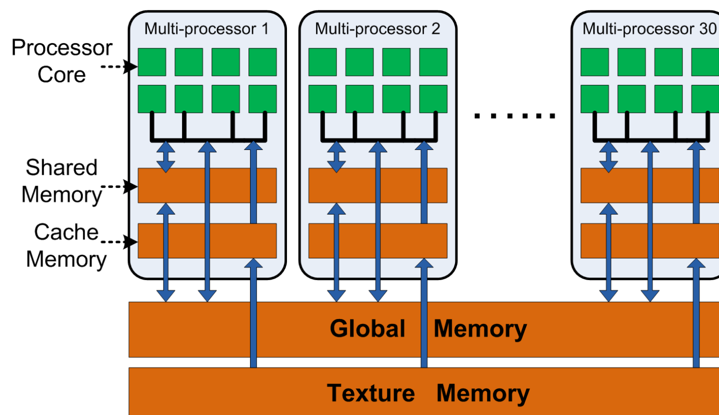


Fig. 2 Architecture of the NVIDIA GTX 280 GPU.

ample, GTX 280 has 30 multiprocessors and totally 240 cores, where each of the cores runs at a speed of 1.296 MHz. GPU multiprocessors execute in SIMT (Single Instruction, Multiple Thread) fashion, which is similar in nature to SIMD (Single Instruction, Multiple Data) execution. Each multiprocessor can manage the concurrent execution of a maximum 1024 threads, of which 32 forms a **warp**. Warp is the minimum threads set that is scheduled independently to run on multiprocessors in parallel. Therefore, GTX 280 can execute at least 960 threads in parallel at any given moment. Since each multiprocessor has only one instruction fetch unit, all threads in a warp must execute the same instruction in a GPU clock cycle. However, if a branch instruction causes the execution of diverged codepaths within a warp, all different codepaths have to be executed sequentially, which implies performance slowdown.

Memory Hierarchy: The GTX 280 contains multiple forms of memory. Beginning with the furthest from the GPU processing elements, the **device memory** is located off-chip on the graphics card and provides the main source of storage for the GPU while being accessible from the CPU and GPU. Each multiprocessor on the GPU contains three caches — a **texture cache**, **constant cache**, and **shared memory**. The texture cache and constant cache are both *read-only* memory providing fast access to immutable data. Shared memory, on the other hand, is *read-write* to provide each core with fast access to the shared address space of a thread block within a multiprocessor. Finally, on each core resides a plethora of registers such that there exists minimal reliance on local memory resident off-chip on the device memory.

Parallelism Abstractions: At the highest level, the CUDA programming model requires the programmer to offload functionality to the GPU 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. CUDA allows a two-dimensional grid organization. Each grid can have a maximum $65,535 \times 65,535$ blocks and each block can have a maximum of 512 threads. It

Algorithm 1 Serial Episode Mining

Input: Candidate N-node episode $\alpha = \langle E_{(1)} \xrightarrow{(t^{(1)}_{low}, t^{(1)}_{high}]} \ldots E_{(N)} \rangle$ and event sequence $S = \{(E_i, t_i) | i = 1 \ldots n\}$.
Output: Count of non-overlapped occurrences of α satisfying inter-event constraints

1: $count = 0$; $s = [[], \ldots, []]$ //List of $|\alpha|$ lists
2: **for all** $(E, t) \in S$ **do**
3:   **for** $i = |\alpha|$ down to 1 **do**
4:     $E_{(i)} = i^{th}$ event type of α
5:     **if** $E = E_{(i)}$ **then**
6:       $i_{prev} = i - 1$
7:       **if** $i > 1$ **then**
8:         $k = |s[i_{prev}]|$
9:         **while** $k > 0$ **do**
10:           $t_{prev} = s[i_{prev}, k]$
11:           **if** $t^{(i_{prev})}_{low} < t - t_{prev} \leq t^{(i_{prev})}_{high}$ **then**
12:             **if** $i = |\alpha| - 1$ **then**
13:               $count{+}{+}$; $s = [[], \ldots, []]$; **break** Line: 3
14:             **else**
15:               $s[i] = s[i] \cup t$
16:             **break** Line: 9
17:           $k = k - 1$
18:       **else**
19:         $s[i] = s[i] \cup t$
20: RETURN count

means CUDA-based applications can create and manage more than 2 trillion threads for massively parallel computing.

5 Approach

Our solution approach is based on a state machine algorithm with inter-event constraints [13]. There are two major phases of this algorithm: generating episode candidates and counting these episodes. We focus on the latter since it is the key performance bottleneck, typically by a few orders of magnitude. Therefore, our focus in this paper is on novel algorithm design for accelerating episode counting on GPUs. We leave the execution of candidate generation on CPU.

Let us first introduce the standard sequential counting algorithm for mining frequent episodes with inter-event constraints. In Algorithm 1, we outline the serial counting procedure for a single episode α. The algorithm maintains a data structure s which is a list of lists. Each list $s[k]$ in s corresponds to an event type $E_{(k)} \in \alpha$ and stores the times of occurrences of those events with event-type $E_{(k)}$ which satisfy the inter-event constraint $(t^{(k-1)}_{low}, t^{(k-1)}_{high}]$ with at least one entry $t_j \in s[k-1]$. This requirement is relaxed for $s[0]$, thus every time an event $E_{(0)}$ is seen in the data its occurrence time is pushed into $s[0]$.

When an event of type $E_{(k)}, 2 \leq k \leq N$ at time t is seen, we look for an entry $t_j \in s[k-1]$ such that $t - t_j \in (t^{(k-1)}_{low}, t^{(k-1)}_{high}]$. Therefore, if we are able to add the event to the list $s[k]$, it implies that there exists at least one previous event with

event-type $E_{(k-1)}$ in the data stream for the current event which satisfies the inter-event constraint between $E_{(k-1)}$ and $E_{(k)}$. Applying this argument recursively, if we are able to add an event with event-type $E_{(|\alpha|)}$ to its corresponding list in s, then there exists a sequence of events corresponding to each event type in α satisfying the respective inter-event constraints. Such an event marks the end of an occurrence after which the *count* for α is incremented and the data structure s is reinitialized. Figure 3 illustrates the data structure s for counting $A \overset{(5,10]}{\rightarrow} B \overset{(10,15]}{\rightarrow} C$.

The maximality of the left-most and inner-most occurrence counts for a general serial episode has been shown in [9]. Similar arguments hold for the case of serial episodes with inter-event constraints and are not shown here for lack of space.

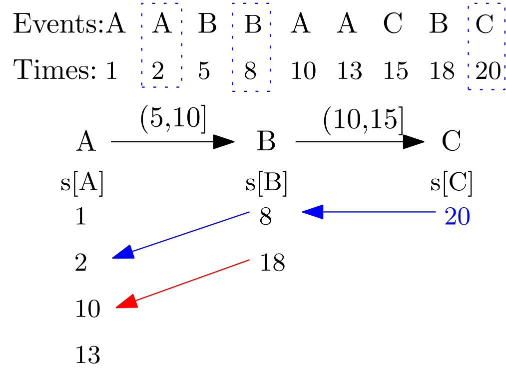


Fig. 3 Illustration of the data structure s for counting $A \overset{(5,10]}{\rightarrow} B \overset{(10,15]}{\rightarrow} C$

In the next two sections, we first present a hybrid approach for counting M episodes on the massively parallel computing architecture of GPUs. The approach leverages the advantages of two different computation-to-core mapping schemes, where the highest level of parallelism is achieved. We then propose a two-pass counting approach, where the first counting pass eliminates most of unsupported episodes and the second counting pass completes the counting tasks for the remaining episodes. Since the first counting pass uses a less complex algorithm the execution time saved at this step contributes to an overall performance gain even we though go through two-pass counting.

5.1 A Hybrid Approach

To parallelize the sequential counting algorithm on GPU, we need to segment the overall computation into independent units that can be mapped onto GPU cores and executed in parallel to fully utilize GPU resources. Different computation-to-core mapping schemes can result in different levels of parallelism, which are suitable for different inputs for the counting algorithm. We design two mapping schemes for counting M episodes: 1) one thread for counting each episode; and 2) multiple

threads per episode. Each mapping scheme has its own advantages and disadvantages when counting different numbers of episodes. We propose a hybrid counting approach that can sense the input condition, so that the optimized level of parallelism can be chosen to achieve the best performance. Let us present the detail of each mapping scheme and how our hybrid approach optimizes the counting by selecting between different mapping schemes according to the number of input episodes M.

Per Thread Per Episode (PTPE). One heuristic for segmenting computation into parallelizable units is to enforce the maximum independence between these units, so that minimum effort is needed to combine the results from different units. An intuitive mapping scheme is to ask each GPU thread to count support for one episode. Since there is no computational dependency between the counting of different episodes, all counting threads can be executed in parallel with no result/data synchronization at the end of the execution. In our implementation, we simply implement Algorithm 1 as the CUDA kernel for each GPU thread, and create M threads to cover the counting of all input episodes.

This mapping scheme, PTPE, is very intuitive and simple. It fits perfectly to GPU's massive data-parallel architecture. However, there is one major disadvantage of this mapping scheme for episode counting: if the number of episodes M is smaller than a certain threshold, the GPU resource is underutilized. As we mentioned in Section 4, the GTX 280 GPU is capable of executing 960 threads in parallel. Since we generate one thread for counting one episode, if the number of episodes M is less than 960, saying $M = 1$ in the extreme case, the most of GPU cores are left idle, and GPU resource is heavily underutilized.

Multiple Threads Per Episode (MTPE). Due to inefficiencies in resource usage when M is small, we seek to achieve a higher level of parallelism within the counting of a single episode, so that multiple threads are created for counting one episode. The basic idea is to segment the input event stream into segments of sub-streams and map the counting in one segment onto one thread. Therefore, in this new computation-to-core mapping scheme, MTPE, we increase the level of parallelism by introducing parallel counting of multiple segments of the input stream. If the number of the segments is R (which is controllable by the counting algorithm), the total number of threads we generate is $M \times R$. We need to ensure that $M \times R$ is larger enough to fully utilize GPU processing cores.

The disadvantage of the mapping scheme MTPE is the extra step needed to merge the sub-counts of all segments for the final count. In this step, we can not simply add all the sub-counts together for the final count, because there are cases that an occurrence of an episode might be divided by the segmentation of the input stream. Additional work is needed to concatenate the divided occurrence between neighboring segments. The bigger the number of segments R is, the more computation is introduced.

Let us discuss the detail about how this two-step, *Counting* and *Merging*, mapping scheme is designed. When we divide the input stream into segments, there are chances that some occurrences of an episode span across the boundaries of consecutive segments. As an example, see Fig. 4 which depicts a data sequence divided into two segments. The shaded rectangles on the top mark the non-overlapped occurrences $h_1 \ldots h_4$ of an episode $(A \rightarrow B \rightarrow C)$ (assume for now that inter-event constraints

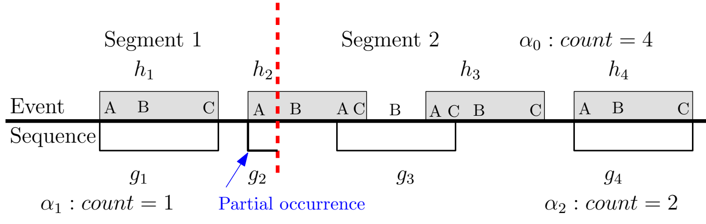


Fig. 4 Illustration of splitting a data sequence into segments and counting within each segment.

are always satisfied), as seen by a state machine α_0 on the unified event sequence. α_0 is thus the reference (serial) state machine. Let α_1 and α_2 be the state machines counting occurrences in segment 1 and segment 2 respectively. During the *Counting* step, α_1 and α_2 are executed in parallel, and each state machine can see a local view of the episode occurrences, which are shown by empty rectangles below the event sequence. α_1 sees the occurrence g_1 and a partial occurrence g_2. For α_2, it will first see the occurrence g_3 and therefore miss h_3 before moving onto g_4.

We propose *Counting* and *Merging* steps that use *multiple state machines* in each segment, so that the counting of a segment is able to anticipate partial occurrences near boundaries. Let us explain in detail why multiple state machines are necessary, and how we design the *Counting* step and *Merging* step to maintain the correctness of counting.

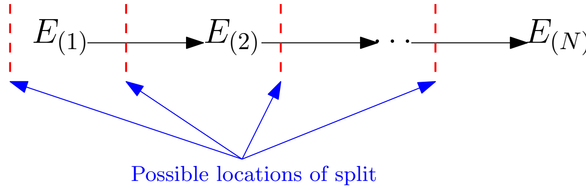


Fig. 5 Illustration of different possibilities of an occurrence splitting across two adjacent data segments.

Assume that we are counting an episode $\alpha = \langle E_{(1)} \xrightarrow{(t^{(1)}_{low}, t^{(1)}_{high}]} \ldots E_{(N)} \rangle$, the data sequence is divided into P segments, and events in the pth data segment are in the range $(\tau_p, \tau_{p+1}]$. An occurrence of episode α can be split across two adjacent segments in at least N ways as shown in Figure 5. For each possible split, we need one state machine, $\alpha^k_p, 0 \leq k \leq N-1$, to count the second segment, starting at $t = \tau_p - \sum_{i=1}^{k} t^{(i)}_{high}$. So we have N different state machines all counting occurrences of episode α using Algorithm 1, handling all possible cases of split between current segment and previous segment.

For each segment p, the *Counting* step is designed as follows, and illustrated in Figure 6.

1. Each state machine α^k_p maintains its own count $= count^k_p$.
2. α^k_p does not increment count for occurrences ending at time $t \leq \tau_p$.

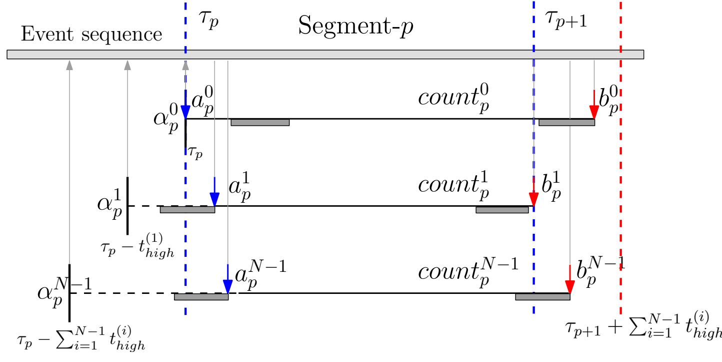


Fig. 6 Illustration of a *Counting* step.

3. α_p^k stores the end time of the first occurrence that completes at time t, $\tau_p < t < \tau_p + \sum_{i=1}^{N-1} t_{high}^{(i)}$. Let this be a_p^k. If there is no such occurrence a_p^k is set to τ_p.
4. α_p^k on reaching end of the segment, crosses over into the next segment to completes the current partial occurrence and continues until $t < \tau_{p+1} + \sum_{i=1}^{N-1} t_{high}^{(i)}$. Let the end time of this occurrence be b_p^k. Note that count is not incremented for this occurrence. In case the occurrence cannot be completed b_p^k is set to τ_{p+1}.

The result of the *Counting* step for each segment p is tuples of $(a_p^k, count_p^k, b_p^k)$. Based on these results, we design our *Merging* step for pairs of consecutive segments as follows, and illustrated in Figure 7.

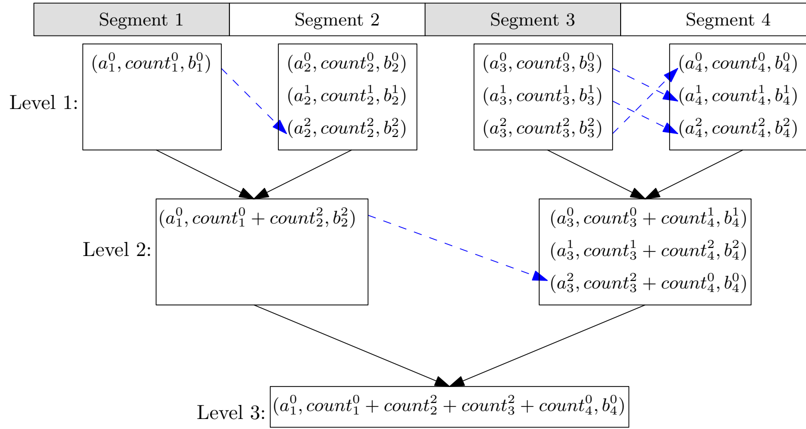


Fig. 7 Illustration of a *Merging* step.

1. Start *Merging* step at level 1.
2. For level i, concatenate the tuples of segment $(j-1)2^i + 1$ with the tuples of segment $(j-1)2^i+1+2^{j-1}$ for all possible j. The procedure for concatenating the

Algorithm 2 A Hybrid GPU Mining Algorithm

1: **if** $S > MP \times B_{MP} \times T_B \times f(N)$ **then**
2: Call PTPE Algorithm
3: **else**
4: Call MTPE Algorithm

tuples of sth segment and t^{th}segment is: find all pairs of tuples $(a_s^k, count_s^k, b_s^k)$ and $(a_t^l, count_t^l, b_t^k)$ such that $b_s^k = a_t^l$, and then concatenate these pairs to obtain the next level $(i+1)$ tuples $(a_s^k, count_s^k + count_t^l, b_t^l)$ for s^{th}segment.

3. After all adjacent segment pairs are concatenated for level i, increase the level to $i+1$ and repeat the previous step until there is only one segment left for this level.

It is worth mentioning that, at level i of *Merging* step, segment $(j-1)2^i + 1$ and segment $(j-1)2^i + 1 + 2^{j-1}$ are considered as adjacent segments. We also choose segment number p to be a power of 2, say 2^q, so that the *Merging* step takes exactly $q+1$ levels and $2^{q+1} - 1$ concatenate operations to finish.

A Hybrid Algorithm. The MTPE mapping scheme can greatly outperform the PTPE in cases when the number of episodes, M, is small and some GPU cores are idle for the PTPE algorithm. For other cases, the PTPE algorithm would run much faster than MTPE. So, as a practical approach, we would like to use their selective superiorities to automatically decide the right algorithm to execute. The logic for making these choices can be very simple: if the GPU can be fully utilized with the PTPE algorithm, then we choose it, else we choose MTPE. In NVIDIA's CUDA framework, the GPU is fully utilized when:

$$S > MP \times B_{MP} \times T_B \quad (2)$$

where S is the number of episodes to be counted, T_B is the number of threads per block as defined by the algorithm, MP is the number of multiprocessors on the GPGPU, and B_{MP} is the number of blocks that the compiler determines can be fit into one multi-processor.

Another key factor that determines the selective superiority is the size/length (N) of the episode being counted. For example, MTPE will run slower on larger N, since the *Counting* step needs N state machines to count each event segment, forcing the *Merging* step to take more time to concatenate the state machines together. The performance of the PTPE algorithm is also dependent on N, but will not change as much as MTPE, since it only uses one state machine to count each episode. Therefore, we need to consider the effect of N when deciding which algorithm to use:

$$S > MP \times B_{MP} \times T_B \times f(N) \quad (3)$$

where $f(N)$ is a performance penalty factor dependent on the episode length/level of the mining algorithm. We use the term **crossover point** to designate the point (i.e., number of episodes) at which the PTPE algorithm will outperform MTPE.

By considering both GPU utilization and episode size, we propose the *Hybrid* algorithm for mining temporal episodes on the GPU as shown in algorithm 2.

5.2 A Two-Pass Elimination Approach

After a thorough analysis of GPU performance of our hybrid mining algorithm, we find that the performance is largely limited by the requirement of large amount of shared memory and large number of GPU registers for each GPU thread. For example, if the episode size is 5, each thread requires 220 bytes of shared memory and 97 bytes of register file. This means that only 32 threads can be allocated on a GPU multi-processor, which has 16K bytes of shared memory and register file. When each thread requires more resources, only fewer threads can run on GPU at the same time, resulting in more execution time for each thread.

The only way to address this problem is to reduce the complexity of the algorithm without losing correctness. In this section, we introduce a two-pass elimination approach that more efficiently searches larger numbers of episodes, further improving the overall performance. The idea is to use a far less complex algorithm, called *PreElim*, to eliminate most of non-supported episodes, and only use the complex hybrid algorithm to determine if the rest of the episodes are supported or not. In order to introduce algorithm *PreElim*, we consider the solution to a slight relaxed problem, which plays an important role in our two-pass elimination approach.

Less-Constrained Mining: Algorithm *PreElim*. Let us consider a constrained version of out data mining problem. Instead of enforcing both lower limits and upper limits on inter-event constraints, we design a counting solution that enforces only upper limits.

Let α' be an episode with the same event types as in α, where α uses the original episode definition from Problem 1. The lower bounds on the inter-event constraints in α are relaxed for α' as shown below.

$$\alpha' = \langle E_{(1)} \xrightarrow{(0,t^{(1)}_{high}]} E_{(2)} \ldots \xrightarrow{(0,t^{(N-1)}_{high}]} E_{(N)} \rangle$$

Observation 1 *In Algorithm 1, if lower bounds of inter-event constraints in episode α are relaxed as α', the list size of $s[k], 1 \le k \le N$ can be reduced to 1.*

Proof In Algorithm 1, when an event of type $E_{(k)}$ is seen at time t while going down the event sequence, $s[E_{(k-1)}]$ is looked up for at least one t_i^{k-1}, such that $t - t_i^{k-1} \in (0, t_{high}^{(k-1)}]$. Note that t_i^{k-1} represents the ith entry of $s[E_{(k-1)}]$ corresponding the $(k-1)^{th}$ event-type in α.

Let $s[E_{(k-1)}] = \{t_1^{k-1} \ldots t_m^{k-1}\}$ and t_i^{k-1} be the first entry which satisfies the inter-event constraint $(0, t_{high}^{(k-1)}]$, i.e.,

$$0 < t - t_i^{k-1} \le t_{high}^{(k-1)} \tag{4}$$

Also Equation 5 below follows from the fact that t_i^{k-1} is the first entry in $s[E_{(k-1)}]$ matching the time constraint.

$$t_i^{k-1} < t_j^{k-1} \le t, \forall j \in \{i+1 \ldots m\} \tag{5}$$

From Equation 4 and 5, Equation 6 follows.

$$0 < t - t_j^{k-1} \le t_{high}^{(k-1)}, \forall j \in \{i+1 \ldots m\} \tag{6}$$

Algorithm 3 Less-Constrained Mining: *PreElim*

Input: Candidate episode $\alpha = \langle E_{(1)} \xrightarrow{(0,t_{high}^{(1)}]} \ldots E_{(N)} \rangle$ is a N-node episode, event sequence $S = \{(E_i, t_i)\}, i \in \{1 \ldots n\}$.
Output: Count of non-overlapped occurrences of α

1: $count = 0$; $s = []$ //List of $|\alpha|$ time stamps
2: **for all** $(E, t) \in S$ **do**
3:   **for** $i = |\alpha|$ to 1 **do**
4:     $E_{(i)} = i^{th}$ event type $\in \alpha$
5:     **if** $E = E_{(i)}$ **then**
6:       $i_{prev} = i - 1$
7:       **if** $i > 1$ **then**
8:         **if** $t - s[i_{prev}] \leq t_{high}^{(i_{prev})}$ **then**
9:           **if** $i = |\alpha|$ **then**
10:             $count++$; $s = []$; **break** Line: 3
11:           **else**
12:             $s[i] = t$
13:       **else**
14:         $s[i] = t$
15: **Output**: count

This shows that every entry in $s[E_{(k-1)}]$ following t_i^{k-1} also satisfies the inter-event constraint. This follows from the relaxation of the lower-bound. Therefore it is sufficient to keep only the latest time stamp t_m^{k-1} only in $s[E_{(k-1)}]$ since it can serve the purpose for itself and all entries above/before it, thus reducing $s[E_{(k-1)}]$ to a single time stamp rather than a list (as in Algorithm 1).

Based on Observation 1, we develop a new Algorithm $A2$ for 'less-constrained' mining. In comparison to Algorithm $A1$, $A2$ reduces the time complexity of the inter-event constraint check from $O(|s[E_{(k-1)}]|)$ to $O(1)$, and also gives a static memory bound for s, which greatly decreases the execution time and runtime memory requirement.

Combined Algorithm: Two-Pass Elimination. Now, we can return to the original mining problem (with both upper and lower bounds). By combining Algorithm *PreElim* with our hybrid algorithm, we can develop a two-pass elimination approach that can deal with the cases on which the hybrid algorithm cannot be executed. The Two-Pass Elimination algorithm is as follows.

Algorithm 4 Two-Pass Elimination Algorithm

1: (First pass) For each episode α, run *PreElim* on its less-constrained counterpart, α'.
2: Eliminate every episode α, if $count(\alpha') < CTh$, where CTh is the support count threshold.
3: (Second Pass) Run the hybrid algorithm on each remaining episode, α, with both inter-event constraints enforced.

The two-pass elimination algorithm yields the correct solution for Problem 1. Although the set of episodes mined under the less constrained version are not a superset of those mined under the original problem definition, we can show the following result:

Theorem 1 *$count(\alpha') \geq count(\alpha)$, i.e., the count obtained from Algorithm* PreElim *is an upper-bound on the count obtained from the hybrid algorithm.*

Proof Let h be an occurrence of α. Note that h is a map from event types in α to events in the data sequence S. Let the time stamps for each event type in h be $\{t^{(1)} \ldots t^{(k)}\}$. Since h is an occurrence of α, it follows that

$$t^i_{low} < t^{(i)} - t^{(i-1)} \leq t^i_{high}, \forall i \in \{1 \ldots k-1\} \tag{7}$$

Note that $t^i_{low} > 0$. The inequality in Equation 7 still holds after we replace t^i_{low} with 0 to get Eqn.8.

$$0 < t^{(i)} - t^{(i-1)} \leq t^i_{high}, \forall i \in \{1 \ldots k-1\} \tag{8}$$

The above corresponds to the relaxed inter-event constraint in α'. Therefore every occurrence of α is also an occurrence of α' but the opposite may not be true. Hence we have that $count(\alpha') \geq count(\alpha)$.

In our two-pass elimination approach, algorithm *PreElim* is less complex and runs faster than the hybrid algorithm, because it reduces the time complexity of the inter-event constraint check from $O(|s[E_{(k-1)}]|)$ to $O(1)$. Therefore, the performance of two-pass elimination algorithm is significantly better than the hybrid algorithm when the number of episodes is very large and the number of episodes culled in the first pass is also large as shown by our experimental results described next.

6 Multi-GPU Mining

With multi-GPU becoming a standard setting in high performance computing research, we implement a multi-GPU mining solution for our two-pass elimination algorithm. In this section, we first describe how multi-GPU computing is supported by NVIDIA's CUDA programming framework. We then illustrate the multi-GPU mining solution for the two-pass elimination algorithm.

6.1 Multi-GPU Computing and Synchronization Support in CUDA

The CUDA programming framework provides utility functions to launch kernels across multiple GPUs in parallel. In this framework, kernel execution on each GPU is managed by a corresponding thread on the CPU (we shall call this a device thread). Multi-threading on the CPU is required for multi-GPU applications as each GPU kernel-launch effectively blocks the calling CPU thread.

In order to manage device memory and other device specific resources from within a CPU (or device) thread, CUDA framework provides the notion of a *context*. A context is created for each GPU device and is specific to the CPU thread controlling it. It is also available only so long as the CPU thread is alive and is maintained in a way transparent to the application programmer.

The following are some of the general considerations for multi-GPU computing:

1. Data communication between GPUs has to go through CPU and host memory.

2. Each GPU has to be controlled by a separate CPU thread.
3. Synchronization between GPUs has to be achieved by the synchronization between CPU threads.
4. Device memory pointers are only retained within a GPU *context*.

6.2 Multi-GPU Mining for Two-Pass Elimination Algorithm

Multi-GPU architecture provides us another layer of parallel computing hierarchy. It means additional levels of parallelism can be achieved by distributing tasks between different GPUs. In our multi-GPU mining framework, we improve the level of parallelism by dividing the total episode candidates into several groups and assigning the counting of each candidate group to a different GPU. Inside each GPU, the PTPE algorithm is used where each GPU thread will count one episode candidate from the candidate group. Given N GPUs, we evenly divide the episode candidates into N groups, and each group contains $1/N^{th}$ of the total episodes. Therefore, our main CPU thread will create another N CPU threads (we call them *device threads*), each of which controls one GPU device and synchronizes with other CPU threads. The ith device thread will be responsible for the following tasks:

1. Copy the entire event sequence into the ith GPU's device memory.
2. Copy the assigned $\frac{1}{N}$ of episode candidates to the ith GPU's device memory.
3. Launch the GPU kernel for counting episodes.
4. Copy the support or frequency count of each episode back to the CPU

The tasks 2 and 3 are interleaved with candidate generation on the CPU main thread.

It might seem natural to have different sets of threads control each of the tasks independently. That is, first N threads are launched to copy the event sequence into the devices while the main thread waits for them to terminate (using a barrier or thread-join). After completion, the main thread generates the candidate episodes and another set of N threads are launched which in turn copy the candidates to each device, launch the counting kernel and copy back the supports and so on. The problem with this approach is the following. The CUDA context which is responsible for maintaining the device memory pointers and other resources is created locally for each CPU device thread. The second set of threads which launch the GPU kernel do not have the same context as the first set of threads. The device pointers and other resources are context-bound and will be lost as soon as the first set of threads terminate. This is because a completely new context would have been created by now for each thread.

To resolve this GPU context issue, we keep the device threads alive until the entire level-wise mining is completed. We use mutexes and conditional variables (*pthread_cond_t*) to synchronize the execution between all device threads and the main CPU thread. While mutexes implement synchronization by controlling thread access to data, condition variables allow signaling threads to wait or block and subsequently notify causing them to unblock. Figure 8 shows the synchronization points between the device threads and the main thread. for the PTPE algorithm. The main CPU thread after completing candidate generation issues a broadcast wake-up signal to all the N device threads and the main thread itself blocks on a conditional

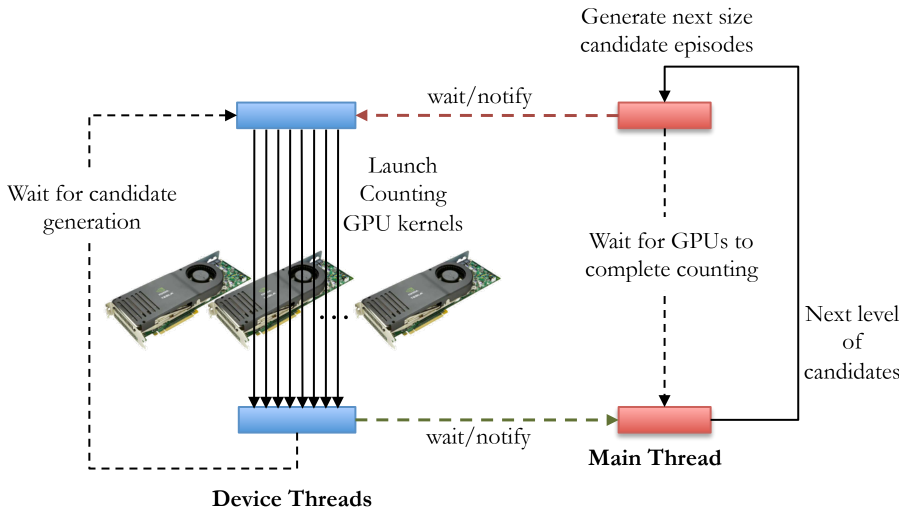


Fig. 8 Synchronization between GPU kernel execution and CPU main thread in a multi-GPU setting.

variable. Subsequently, the device threads launch the GPU counting kernel on corresponding devices. At completion of the kernel launch each device thread indicates that its counting task is complete and wakes up the main thread. The main thread keeps a counter and does not resume until all the device threads have reported their task-completion status. Once all device threads have finished their counting task, the main thread determined the frequent episodes from the counts and continues with candidate generation. This implementation has the least overhead in terms of both initialization of thread contexts (and CUDA device contexts) and also waiting times for the different CPU threads (compared to polling based synchronization). In total we use two mutexes, two conditional variables, a counter for tracking the status of the N threads GPU controlling threads, and a logical variable for the status of main thread.

For our two-pass elimination approach, besides the set of tasks described above, there is another set of data transfer for episode candidates across the device and CPU memory, and an extra kernel-launch for *PreElim* algorithm.

7 Experimental Results

7.1 Datasets and Testbed

Our datasets are drawn from both mathematical models of spiking neurons as well as real datasets gathered by Wagenar et al. [15] in their analysis of cortical cultures. Both these sources of data are described in detail in [13]. The mathematical model involves 26 neurons (event types) whose activity is modeled via inhomogeneous Poisson processes. Each neuron has a basal firing rate of 20 Hz and two causal chains of connections—one short and one long—are embedded in the data. This dataset

(*Sym26*) involves 60 seconds with 50,000 events. The real datasets (*2-1-33, 2-1-34, 2-1-35*) observe dissociated cultures on days 33, 34, and 35 from over five weeks of development. The original goal of this study was to characterize bursty behavior of neurons during development.

We evaluated the performance of our GPU algorithms on a machine equipped with Intel Core 2 Quad 2.33 GHz and 4GB system memory. We used a NVIDIA GTX280 GPU, which has 240 processor cores with 1.3 GHz clock for each core, and 1GB of device memory.

There are two CUDA runtime parameters we need to determine for each execution on the GPU: number of threads per block, T, and the total number of blocks. The second parameter is always calculated based on T so that all required computation can be finished with T threads per block and within one CUDA kernel function call.

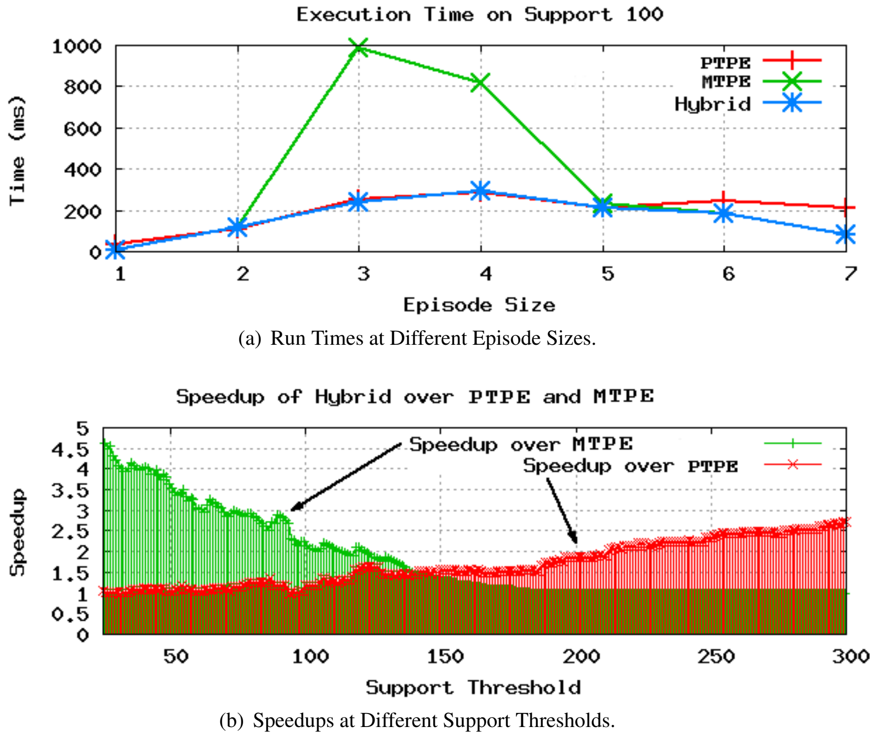
Parameter T is determined by the algorithm and the size of the episode (N). For the PTPE algorithm, we calculate the maximum number of threads per block at each N. The larger N is, more shared memory is needed per thread. When $N = 1$, we use 128 threads, and as N increases, the maximum number of threads per block decrease due to the shared memory limit. When $N = 6$, we cannot have more than 32 threads per block. For MTPE, the event stream is segmented into a number (R) of sub-streams, as mentioned in Section 5.1. Recall that we need to create multiple threads to count all sub-streams independently and run multiple state machines, as shown in Figure 6. The number of threads for each block T can be calculated as $T = R \times N$, since there are R sub-streams and N state machines. Again, we must limit the number of sub-streams to reduce the number of threads due to the shared memory limit affected by N. For the *PreElim* algorithm, we generate as many threads as possible per block until shared memory usage reaches the hardware limit (16KB). In this case, T is normally much larger than 32, since we do not have a strict memory requirement for the GPU for *PreElim* algorithm.

7.2 Performance of the Hybrid Algorithm

In order to evaluate the performance of the hybrid algorithm, we shall first provide the performance comparison between PTPE and MTPE algorithms. As seen in Figure 9(a), it is clear that blindly choosing to execute the PTPE or MTPE approach for *all* levels is not the best solution. For episode sizes of 1, 2, and 5 both approaches complete in roughly the same amount of time. However, PTPE significantly outperforms MTPE at episode sizes of 3 and 4 by 3.96X and 2.84X, respectively. On the other hand, PTPE performs slower than MTPE for episodes of size 6 (1.32X) and 7 (2.63X). In Figure 9(b), we also provide the evaluation results for our hybrid algorithm. Since the hybrid algorithm leverage the advantages of both PTPE and MTPE algorithms, it demonstrates a better performance than PTPE and MTPE.

The crossover points exist in all of our tests for this dataset. Table 1 shows the crossover points determined experimentally for the Sym26 dataset.

Recall Equation 3 in Section 5.1 where optimal execution occurs when the GPU is fully utilized and a small factor of episode size is taken into account. Using the



(a) Run Times at Different Episode Sizes.

(b) Speedups at Different Support Thresholds.

Fig. 9 Comparison of PTPE, MTPE, and Hybrid Algorithm on *Sym26* dataset.

Table 1 Crossover Points on number of episodes below which MTPE should be run (for the fewer episodes case). For other episode sizes—1, 2, and >8—MTPE should be chosen.

Level	3	4	5	6	7	8
Crossover	415	190	200	100	100	60

table above, with $M = 30$, $T_B = 32$, and $B_M = 1$, we find that $f(N)$ of the form $\frac{a}{N} + b$ is a better fit than $a \times N + b$ as seen in Figure 10.

With these crossover points determined, the hybrid approach was evaluated on the same support thresholds and the speedup for this approach over both PTPE and MTPE is visible in Figure 9(b). The range of improvement over PTPE is almost 3X and over 4.5X for MTPE. When the number of episodes is large (i.e., low support threshold) there are enough episodes to fully utilize the GPU and as such the hybrid algorithm shows little improvement over PTPE. Conversely, the hybrid algorithm shows little improvement over MTPE when the support threshold is high with very few episodes.

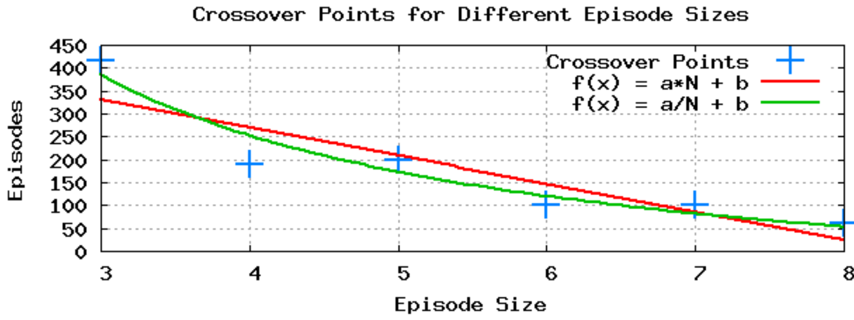
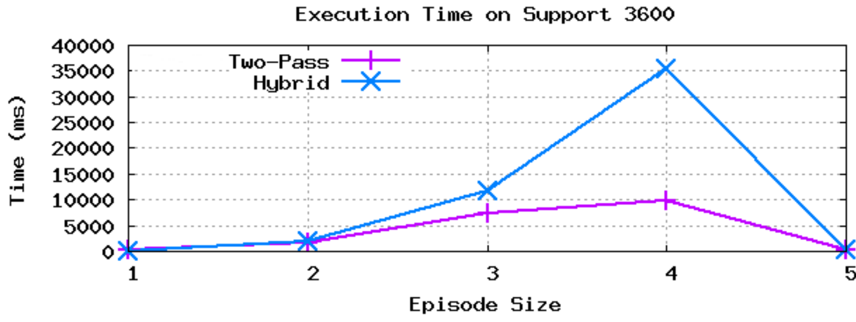


Fig. 10 Crossover points fitted to $\frac{a}{N} + b$ and $a \times N + b$

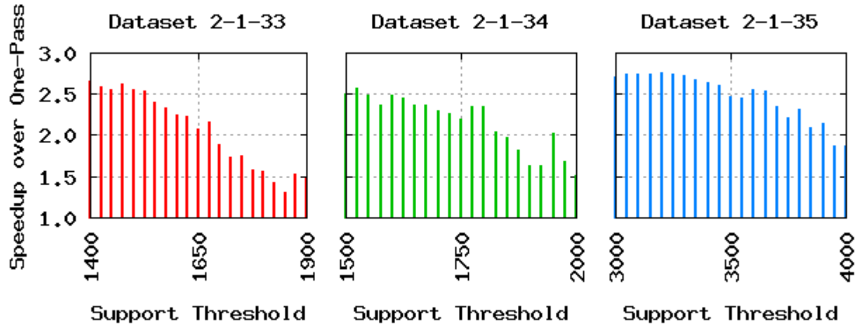
7.3 Performance of the Two-Pass Elimination Algorithm

As stated in Section 5.2, the performance of the hybrid algorithm suffers from the requirement of large shared memory and large register file, especially when the episode size is big. So we introduce algorithm *PreElim* that can eliminate most of the non-supported episodes and requires much less shared memory and register file, then the complex hybrid algorithm can be executed on much fewer number of episodes, resulting in performance gains. The amount of elimination that *PreElim* conducts can greatly affect the execution time at different episode sizes. In Figure 11(a), the *PreElim* algorithm eliminates over 99.9% (43634 out of 43656) of the episodes of size four. The end result is a speedup of 3.6X over the hybrid algorithm for this episode size and an overall speedup for this support threshold of 2.53X. Speedups for three different datasets at different support thresholds are shown in Figure 11(b) where in every case, the two-pass elimination algorithm outperforms the hybrid algorithm with speedups ranging from 1.2X to 2.8X.

We also use *CUDA Visual Profiler* to analyze the execution of the hybrid algorithm and *PreElim* algorithm to give a quantitative measurement of how *PreElim* outperforms the hybrid algorithm on the GPU. We have analyzed various GPU performance factors, such as GPU occupancy, coalesced global memory access, shared memory bank conflict, divergent branching, and local memory loads and stores. We find the last two factors are primarily attributed to the performance difference between th e hybrid algorithm and *PreElim*, which are shown in Figure 12. The hybrid algorithm requires 17 registers and 80 bytes of local memory for each counting thread, while *PreElim* algorithm only requires 13 registers and no local memory. Since local memory is used as supplement for registers and mapped onto global memory space, it is accessed very frequently and has the same high memory latency as global memory. In Figure 12 (a), the total amount of local memory access of both two-pass elimination algorithm and the hybrid algorithm comes from the hybrid algorithm. Since the *PreElim* algorithm eliminates most of the non-supported episodes and requires no local memory access, the local memory access of two-pass approach is much less than one-pass approach when the size of episode increases. At the size of 4, the *PreElim* algorithm eliminates all episode candidates, thus there is no execu-



(a) Execution time of Two-Pass Elimination and Hybrid algorithms for Support=3600 on Dataset *2-1-35* at different episode sizes.



(b) Speedup of Two-Pass Elimination over Hybrid Algorithm for multiple support thresholds on multiple datasets.

Fig. 11 Execution time and speedup comparison of the Hybrid algorithm versus Two-Pass Elimination algorithm.

tion for the hybrid algorithm and no local memory access, resulting a large performance gain for two-pass elimination algorithm over the hybrid algorithm. As shown in Figure 12 (b), the amount of divergent branching also affects the GPU performance difference between the two-pass elimination algorithm and the hybrid algorithm.

7.4 Performance of Multi-GPU Mining

We test our multi-GPU implementation of the episode mining algorithm on a workstation with eight GPUs (i.e., four GTX 295 graphics cards, each with two GPUs). The set of candidate episodes are distributed among the different GPUs and the counting kernel is launched in parallel. This approach incurs additional performance overhead due to the data transfer between main memory and the GPU device memory. Since the approach is task parallel, the data needs to be copied to each device separately. In addition, there is synchronization overhead for thread barriers and communication between threads. Despite the performance overhead, we observe significant speedup at *low support thresholds*, when comparing the eight-GPU implementation to the

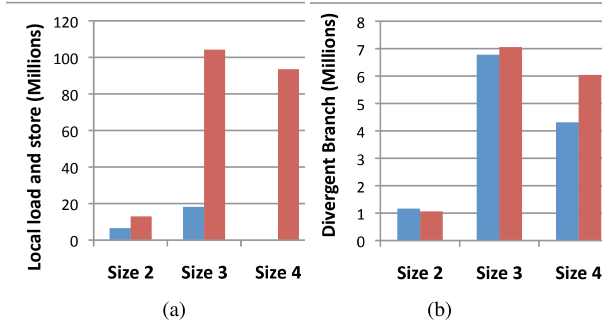
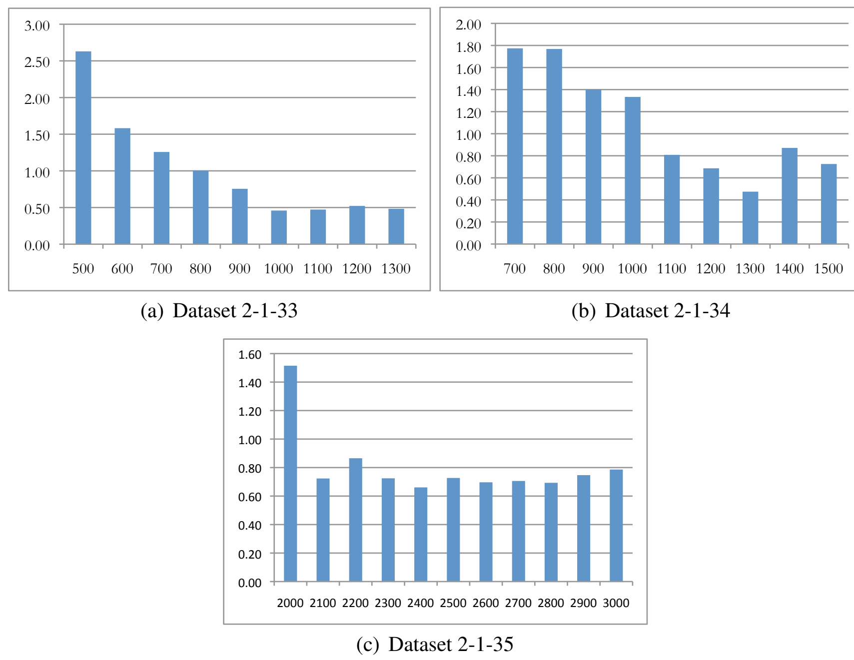


Fig. 12 Comparison between the Hybrid algorithm and Two-pass Elimination algorithm for support threshold 1650 on dataset *2-1-33*. (a) Total number of loads and stores of local memory. (b) Total number of divergent branches.

single-GPU implementation, as shown in Figure 13. For dataset *2-1-33*, we achieve more than a 2.5-fold speedup when using a threshold of 500. For a threshold larger than 800, the performance overhead of the eight-GPU implementation is so high that the overall performance is worse than the single GPU one. Similar trends can be seen for the other datasets with different thresholds. As such, the advantages of a multi-GPU implementation emerge only at low support thresholds, i.e., when there are an extremely large number of episode candidates.

We also compare the performance between the PTPE and the two-pass elimination algorithms with multi-GPU implementations. Figure 14 shows the overall performance speedup of the two-pass elimination algorithm over the PTPE algorithm. There is not a large performance gain using this approach in the multi-GPU implementation. For example, for dataset *2-1-34* with a support threshold of 1,500, the speedup of the two-pass elimination algorithm over the PTPE algorithm is 1.1 using the eight-GPU implementation. When using the single-GPU implementation, the speedup is 2.5. Because there are fewer candidates to count in each GPU for the eight-GPU implementation, the performance gain obtained from the early elimination of candidates by *PreElim* algorithm is not very significant.

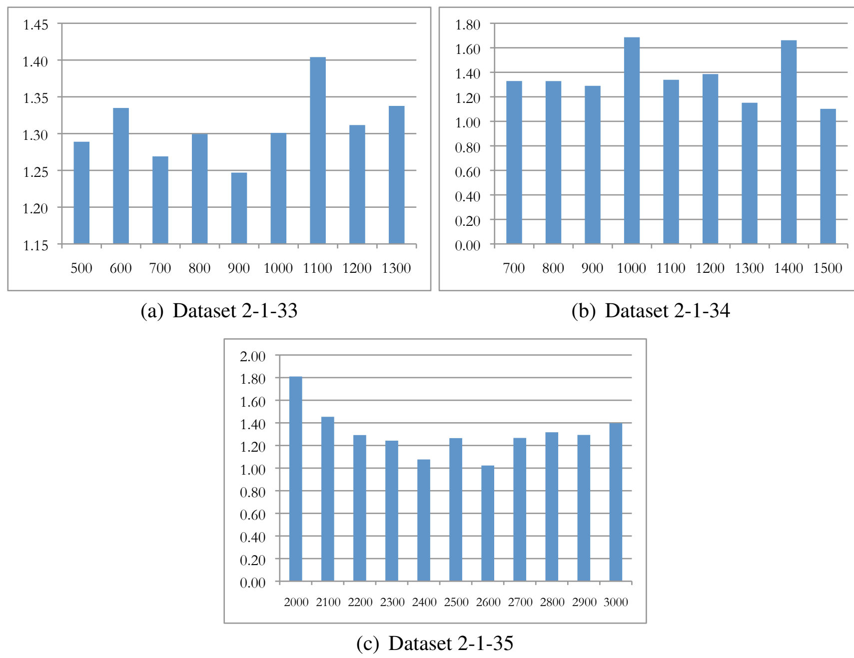
To quantify the performance overhead involved in the multi-GPU implementation of the two-pass elimination mining algorithm, we profile the execution time for different steps of the single GPU implementation and the eight-GPU implementation, as shown in Figure 15. When using the Dataset 2-1-33, Figure 15(a) shows the execution time profile with a threshold of 500, and Figure 15(b) shows the execution time profile with 1300 as the threshold. From the figure, we see that the data transfer time for the eight-GPU implementation is eight times that of the single GPU implementation. This is due to the fact that the data transfer from CPU main memory to each GPU has to share the same system bus (PCI-E), resulting in serialized copies from the CPU to all GPUs. While the data transfer time is kept the same for both counting thresholds, the counting time is quite different. For the threshold of 1300, there are much fewer episode candidates for each GPU to count, as shown in Table 2, resulting



(a) Dataset 2-1-33 (b) Dataset 2-1-34

(c) Dataset 2-1-35

Fig. 13 Speedup of multi-GPU over single GPU implementation.



(a) Dataset 2-1-33 (b) Dataset 2-1-34

(c) Dataset 2-1-35

Fig. 14 Speedup of two-pass elimination algorithm over PTPE algorithm with eight-GPU implementation.

in GPU under-utilization. For example, only 1 episode of size of 3 is assigned for a GPU to count, whereas at least 960 episodes (32 threads per MP $\times$ 30 MPs) are needed to fully utilize the GPU.

Table 2 Number of candidate episodes counted by each GPU in the eight-GPU implementation at frequency thresholds of 500 and 1300.

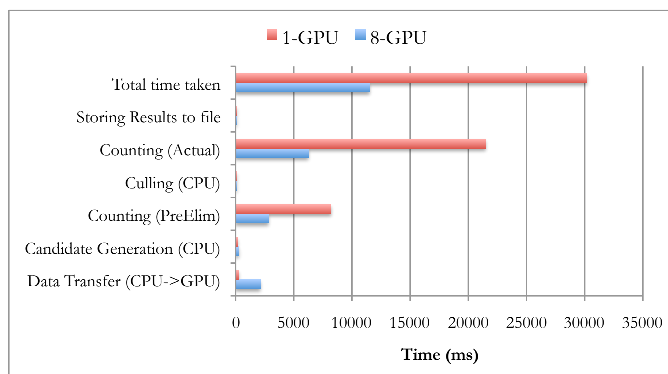
Threshold 500			**Threshold 1300**		
Episode size	#Episodes (PreElim)	#Episodes (Second Pass)	Episode size	#Episodes (PreElim)	#Episodes (Second Pass)
1	8	7	1	8	6
2	358	266	2	259	145
3	6028	6023	3	1	1
4	2870	2879	4	N/A	N/A

7.5 Performance Gain over CPU

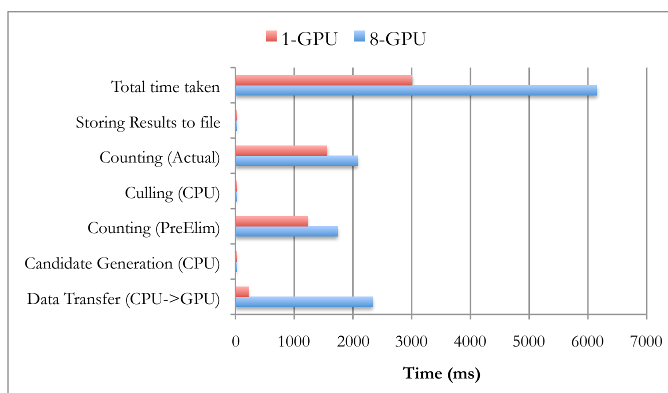
To demonstrate the performance gain of our GPU approach, we developed a C++ implementation of Algorithm 1 on a quad-core CPU with four threads. Each thread counts one-fourth of the total episodes and accesses each event in the input stream only once to ensure the best cache performance. As illustrated in Algorithm 1, we use a state machine to count each episode. At the beginning, each state machine is initialized to a wait state corresponding to the first event in the episode. As the thread processes each event in the stream, the algorithm looks through the list of episodes waiting for that particular event and updates the state machines so that they are now also waiting for the next event in the episodes. Furthermore, each state machine is annotated with the time of the event so that intervals may be accounted for. This complex state machine implementation is required to correctly account for intervals between two subsequent events in a candidate episode. When the thread finds the last event in an episode, it then increments that count for the corresponding episode and resets the state machine to the initial state of waiting for the first event of the episode.

In order to have a fair comparison with the GPU implementation, we also implemented the two-pass elimination algorithm on the CPU. Counting using the two-pass algorithm allows a simplified and faster state machine to quickly eliminate possible candidates before using the slower but more complete state machine described in the preceding paragraph.

Compared with the performance of the CPU implementation, our GPU algorithms exhibit a significant speedup, as shown in Figure 16. For the 2-1-33 dataset, the speedup is approximately 15-fold for a support threshold of 500 when using a single GPU. With the eight-GPU implementation, the speedup is about 35-fold. Note that due to additional overhead involved in the multi-GPU solution, the speedup is *not* eight times that of the single GPU case. Similar speedups are seen for the other datasets at low support thresholds. For higher thresholds, the advantage of using multiple GPUs erods as the number of episode candidates becomes smaller and the GPUs



(a) Frequency threshold = 500



(b) Frequency threshold = 1300

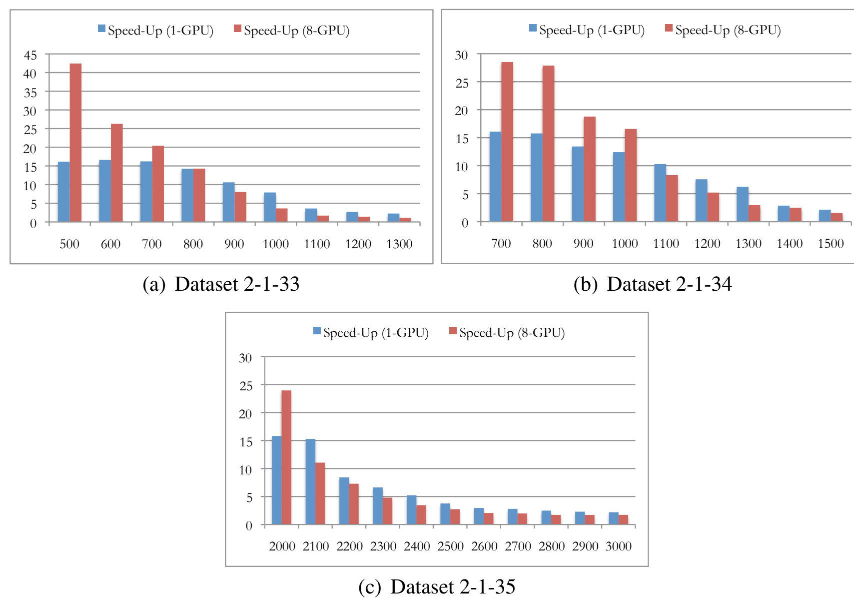
Fig. 15 Comparison of time-taken by different subroutines in the frequent episode mining algorithm (2-Pass) for the Dataset 2-1-33 at different frequency thresholds.

tend to get under-utilized. This is what causes the speedup of the multi-GPU implementation to fall behind the single GPU implementation, as shown in all three datasets.

8 Discussion

We have presented a powerful and non-trivial framework for conducting frequent episode mining on GPUs and shown its capabilities for mining neuronal circuits in spike train datasets. For the first time, neuroscientists can enjoy the benefits of data mining algorithms without needing access to costly and specialized clusters of workstations to track evolving cultures to reveal the progression of neural development in real-time.

Our future work is in four categories. First, our experiences with the neuroscience application have opened up the interesting topic of mapping finite state machine



(a) Dataset 2-1-33

(b) Dataset 2-1-34

(c) Dataset 2-1-35

Fig. 16 Speedup of GPU implementations (single and multiple GPUs) over the CPU implementation.

based algorithms onto the GPU. A general framework to map any finite state machine algorithm for counting will be extremely powerful not just for neuroscience but for many other areas such as (massive) sequence analysis in bioinformatics and linguistics. Second, the development of the hybrid algorithm highlights the importance of developing new, additional, programming abstractions specifically geared toward data mining on GPUs. Third, we found that the two-pass approach performs significantly better than running the complex counting algorithm over the entire input. The first pass generates an upper bound that helps reduce the input size for the complex second pass, speeding up the entire process. We seek to develop better bounds that incorporate more domain-specific information about neuronal firing rates and connectivities. Finally, we wish to integrate more aspects of the application context into our algorithmic pipeline, such as candidate generation, streaming analysis, and rapid "fast-forward" and "slow-play" facilities for visualizing the development of neuronal circuits.

References

1. Adee, S.: Mastering the Brain-Computer Interface. IEEE Spectrum (2008)
2. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94, pp. 487–499. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1994). URL http://portal.acm.org/citation.cfm?id=645920.672836
3. Bell, C., Shenoy, P., Chalodhorn, R., Rao, R.: Control of a humanoid robot by a noninvasive brain-computer interface in humans. J. Neural Eng. **Vol. 5**, 214–220 (2008)

4. Cao, Y., Patnaik, D., Ponce, S., Archuleta, J., Butler, P., chun Feng, W., Ramakrishnan, N.: Towards chip-on-chip neuroscience: Fast mining of neuronal spike streams using graphics hardware. In: CF '10: Proceedings of the 7th ACM international conference on Computing frontiers, 978-1-4503-0044-5, pp. 1–10. ACM, Bertinoro, Italy (2010)
5. Fang, W., Lau, K., Lu, M., Xiao, X., Lam, C., Yang, P., He, B., Luo, Q., Sander, P., Yang, K.: Parallel data mining on graphics processors. Tech. Rep. HKUST-CS08-07, Hong Kong University of Science and Technology (2008)
6. Govindaraju, N., Raghuvanshi, N., Manocha, D.: Fast and approximate stream mining of quantiles and frequencies using graphics processors. In: Proc. SIGMOD'05, pp. 611–622 (2005)
7. Guha, S., Krishnan, S., Venkatasubramanian, S.: Data Visualization and Mining using the GPU. Tutorial at ACM SIGKDD'05 (2005)
8. Hingston, P.: Using finite state automata for sequence mining. Australian Computer Science Communications **Vol. 24**(1), 105–110 (2002)
9. Laxman, S., Sastry, P., Unnikrishnan, K.: A fast algorithm for finding frequent episodes in event streams. In: Proc. KDD'07, pp. 410–419 (2007)
10. Li, L., Fu, W., Guo, F., Mowry, T., Faloutsos, C.: Cut-and-stitch: efficient parallel learning of linear dynamical systems on SMPs. In: Proc. KDD'08 (2008)
11. Mannila, H., Toivonen, H., Verkamo, A.: Discovery of frequent episodes in event sequences. DMKD **Vol. 1**(3), pages 259–289 (1997)
12. Mitchell, T., Shinkareva, S., Carlson, A., Chang, K., Malave, V., Mason, R., Just, M.: Predicting Human Brain Activity Associated with the Meanings of Nouns. Science **Vol. 320**(1191) (2008)
13. Patnaik, D., Sastry, P., Unnikrishnan, K.: Inferring Neuronal Network Connectivity from Spike Data: A Temporal Data Mining Approach. Scientific Programming **16(1)**, 49–77 (2008). DOI 10.3233/SPR-2008-0242
14. Serruya, M., Hatsopoulos, N., Paninski, L., Fellows, M., Donoghue, J.: Brain-machine interface: Instant neural control of a movement signal. Nature **Vol. 416**, 141–142 (2002)
15. Wagenaar, D.A., Pine, J., Potter, S.M.: An extremely rich repertoire of bursting patterns during the development of cortical cultures. BMC Neuroscience (2006)