

Towards Algorithm Transformation for Temporal Data Mining on GPU

Sean P. Ponce

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Yong Cao, Chair
Wuchun Feng
Naren Ramakrishnan

July 7, 2009
Blacksburg, Virginia

Keywords: temporal data mining, GPGPU, CUDA

Copyright 2009, Sean P. Ponce

Towards Algorithm Transformation for Temporal Data Mining on GPU

Sean P. Ponce

(ABSTRACT)

Data Mining allows one to analyze large amounts of data. With increasing amounts of data being collected, more computing power is needed to mine these larger and larger sums of data. The GPU is an excellent piece of hardware with a compelling price to performance ratio and has rapidly risen in popularity. However, this increase in speed comes at a cost. The GPU's architecture executes non-data parallel code with either marginal speedup or even slowdown. The type of data mining we examine, temporal data mining, uses a finite state machine (FSM), which is non-data parallel. We contribute the concept of algorithm transformation for increasing the data parallelism of an algorithm. We apply the algorithm transformation process to the problem of temporal data mining which solves the same problem as the FSM-based algorithm, but is data parallel. The new GPU implementation shows a 6x speedup over the best CPU implementation and 11x speedup over a previous GPU implementation.

Dedication

This written thesis is dedicated to my wife and best friend Chelsey, for her unending and constant support, through the easy and difficult times. It is also dedicated to my parents, for their advice and encouraging me to achieve my best.

Acknowledgments

I would like to thank my research partners throughout my graduate studies, all have been a pleasure to work with. This includes Debprakash Patnaik for contributions throughout the work described in this thesis, as well as Jeremy Archuleta and Tom Scogland for their work as described in sections 2.5, 3.2, and 3.3. Thanks also goes to Rick Battle, Lee Clagett, Taylor Eagy, Kunal Mudgal, and Chase Khoury for their contributions towards our crowd animation and simulation research. In addition, I would like to thank Seung In Park and Jing Huang for our first research venture into GPU-related work with the VCM GPU project. I would also like to acknowledge the efforts of other students that I have produced research-worthy class projects with, including AJ Alon, Bobby Beaton, Bob Edmison, Mara Silva, Regis Kopper, and Tejinder Judge. I am appreciative of the professors who helped shape my research, including Francis Quek, Wuchun Feng, Naren Ramakrishnan, and most importantly my advisor, Yong Cao, for his patience, wisdom, and guidance.

Contents

1	Introduction	1
2	Related Works	6
2.1	Data Mining	6
2.1.1	Temporal Data Mining	7
2.1.2	Level-set Mining	7
2.2	GPGPU	9
2.2.1	History	9
2.2.2	Architecture	10
2.3	Data Parallelism	12
2.4	Algorithm Transformation	12
2.5	Data mining on the GPU	13
3	Temporal Data Mining Algorithms	15
3.1	Finite State Machine	15
3.2	Per-thread per-episode (PTPE)	18

3.3	MapConcatenate	20
4	Algorithm Transformation	23
4.1	Finding Occurrences	24
4.2	Overlap Removal	26
4.3	Compaction	27
4.3.1	Lock-based atomic compaction	28
4.3.2	Lock-free prefix sum compaction	29
5	Results	32
5.1	Test datasets and algorithm implementations	33
5.2	Comparisons of performance	33
5.3	Analysis of the new algorithm	35
6	Conclusion	40

List of Figures

1.1	Micro-electrode array (MEA).	2
1.2	Electrical activity recorded from brain tissue, with spikes from several electrodes plotted on the timeline.	3
1.3	Patterns identified from spike data after a temporal data mining algorithm is applied. Each line represents an occurrence of a particular pattern, and different colors represent a different pattern.	3
2.1	Compute Unified Device Architecture (CUDA).	10
3.1	Sensor grid with a visualized episode. The numbers in parentheses describe the minimum and maximum time constraint between events.	16
3.2	Finite state machine used to count episodes in an event stream without temporal constraints.	17
3.3	An incorrect finite state machine used to count episodes in an event stream without temporal constraints.	17
3.4	Correct finite state machine with a fork used to count episodes in an event stream with temporal constraints.	18
3.5	Map step of the MapConcatentate algorithm.	21

3.6	Concatenate step of the MapConcatenate algorithm.	21
4.1	(a) Episode matching algorithm applied to an event stream. The first three lines all represent the same event stream, but each line illustrates a different event in the episode. (b) A listing of the intervals generated with a graph of the intervals.	26
4.2	Thread launches on a compacted and uncompact array, with threads represented by arrows and idle threads highlighted. (a) An uncompact array requiring 8 threads to be launched. (b) A compacted array only requiring 4 threads.	28
5.1	Performance of MapConcatenate compared with the CPU and best GPU implementation, counting 30 episodes in Datasets 1-8.	34
5.2	Performance comparison of the CPU and best GPU implementation, counting a single episode in Datasets 1 through 8.	35
5.3	Profile of time spent performing each subtask for AtomicCompact (a) and PrefixSum (b)	36
5.4	Performance of algorithms with varying episode length in Dataset 1.	37
5.5	Performance of algorithms with varying episode frequency in Dataset 1.	38

List of Algorithms

1	Level-set temporal data mining.	8
2	Find episode count in an event stream using FSM.	19
3	Find maximal count of non-overlapping occurrences of a given episode in an event stream.	24
4	Find all matches of a given episode in an event stream.	25
5	Return a set of non-conflicting intervals of maximal size, adapted from [8].	27

List of Tables

2.1	Number of episodes generated at each level for a small simulated dataset with 26 starting episodes.	9
4.1	Input and output of a prefix sum operation.	30
5.1	Hardware used for performance analysis	32
5.2	Details of the datasets used	33
5.3	CUDA Visual Profiler Results	34

Chapter 1

Introduction

Reverse-engineering the brain is listed as one of the top engineering challenges by the National Academy of Engineering [14]. Although today's computers are accurate and faster than the human brain in a number of tasks, the brain is able to perform tasks that computers cannot do. Relative to the computer, the brain excels at visual recognition, speech recognition, and other tasks. The human brain is a unique computational device, and understanding it could serve as inspiration to improve today's computing systems.

One step towards understanding how the brain works is by understanding how neuronal networks form. This can be done by recording electrical activity in brain tissue cultures, through the use of a micro-electrode array (MEA), as shown in figure 1. Sensors that detect electrical activity, called electrodes, are arranged in a grid-like fashion and placed on top on the living brain tissue culture. The brain tissue will produce electrical activity, spontaneously and/or with a stimulus. The activity is recorded at the electrode points, and the spikes of electrical activity are recorded.

The problem with this method of recording is the massive amount of data that is collected. Millions of spikes occur from only minutes of recording. The data contains a low signal-to-noise ratio, and is difficult to understand visually by plotting the data as shown in figure 1.

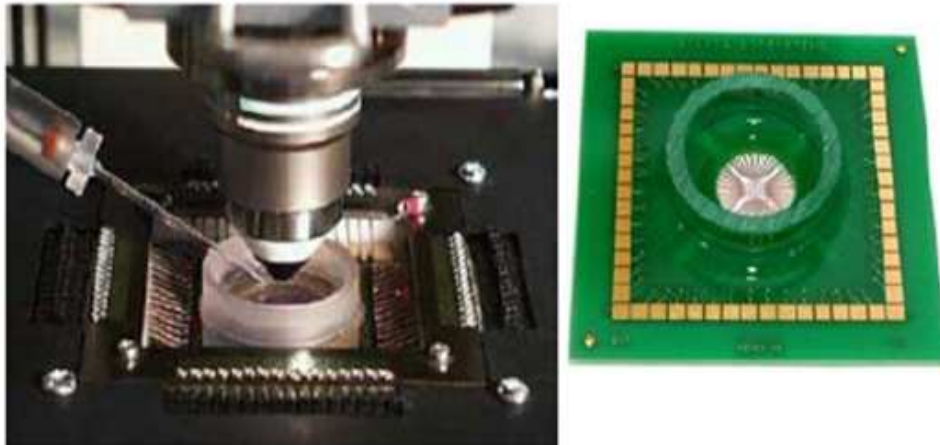


Figure 1.1: Micro-electrode array (MEA).

Without a statistical analysis of the data, the data does not have much meaning.

Data mining is the method of extracting correlations and patterns in data. Data mining algorithms are developed around statistical models, and aim to extract statistical information existing in a set of data. These algorithms allow a computer to mine data faster than humans could mine in a lifetime. One application is the use of associative rule mining to find associations between items purchased in a grocery store [1]. This is done by examining a series of transactions and determining which items are correlated. For example, one could determine that beer and diapers, an unusual combination, are often purchased together and one could make a business decision to make the two items co-located in order to boost sales.

Temporal data mining is the type of data mining we wish to examine, as it applies to the neuroscience problem presented. The electrical activity data collected from the brain tissue culture is a series of spikes, or events occurring along a timeline. One step along the way to reverse-engineering the brain is determining relationships among the spikes. After mining the data for frequent patterns, the relationships among the spikes can be asserted with statistical significance, as shown in figure 1.

Although temporal data mining is able to accomplish the neuroscience problem, there is the performance issue. Not only is the number of spikes recorded large, but the number of

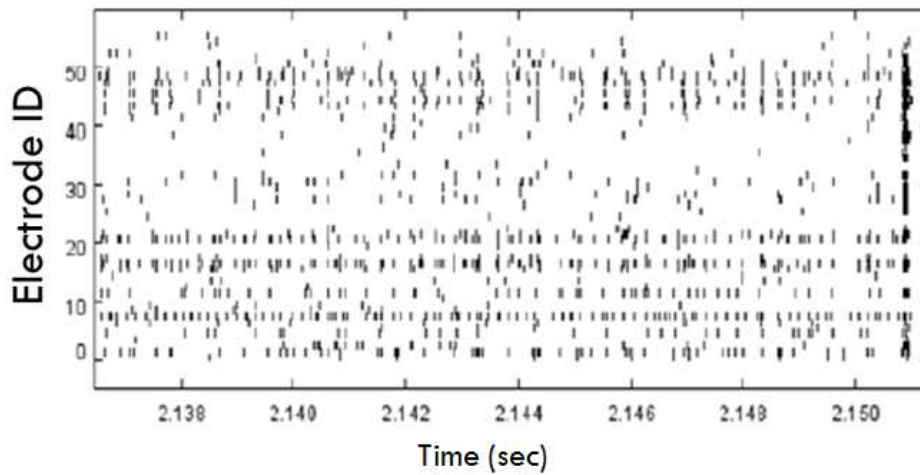


Figure 1.2: Electrical activity recorded from brain tissue, with spikes from several electrodes plotted on the timeline.

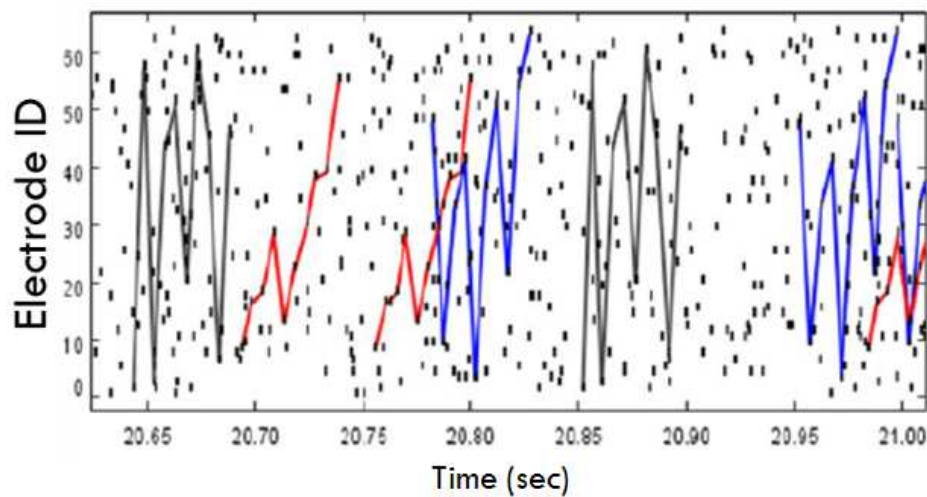


Figure 1.3: Patterns identified from spike data after a temporal data mining algorithm is applied. Each line represents an occurrence of a particular pattern, and different colors represent a different pattern.

patterns to search for is large. Frequent episode discovery, a statistical model for finding relevant patterns in data [10], can generate an exponentially bounded number of patterns to search for. The computational demand of the data mining approach requires a powerful computational device, which are reasons we turn to the Graphics Processing Unit (GPU) to provide that power.

GPUs are massively parallel computing devices with a low monetary cost per GFLOP and are widely available. The latest generation of cards from Nvidia contains up to 240 processing cores per chip. The theoretical maximum GFLOPS of Nvidia's GeForce GTX 295, containing two GPUs, is 1,788¹, compared to Intel's latest quad core calculated to be 140 GFLOPS. Although theoretical GFLOPS are not the only measure of computational performance, it provides a numerical comparison amongst different architectures. On some algorithms non-data parallel algorithms a top GPU can perform worse than a top CPU, but embarrassingly parallel algorithms can see speedups up to 431x [19].

The main attractiveness of GPUs over other high-performance devices is the cost and availability. The best performing GPU, containing 2x240 cores, will cost around 530 US dollars. CPU-based solutions can easily have more than 10x the cost for the same number of GFLOPS. For example, the Cray CX1, costing 66,939 dollars, can contain up to 16 Intel quad-core processors. Not counting networking overhead, this equates to 2240 GFLOPS, costing 30 dollars per GFLOP. The latest Nvidia card costs 30 cents per GFLOP. Nvidia can keep these costs down because of the volume at which these cards are produced. Nvidia has sold over 100 million CUDA-enabled devices to date, which shows how widely available these cards are, as well as the number of consumers which already own a CUDA-enabled device.

General-purpose programming on GPU (GPGPU) presents a challenge to developers. Not all algorithms are easily parallelizable. Some algorithms exhibit excellent performance when implemented on GPUs, often called embarrassingly parallel algorithms. This is where each

¹GFLOPS calculations are for single-precision floating point operations.

core can perform operations independent of what the other cores are doing. This reduces the communication overhead needed among cores. Many graphics applications are embarrassingly parallel because the operations performed are often per-pixel or per-vertex. It is not always the case that algorithms are parallelizable. For example, sorting requires some way of comparing a single value to all others and determining a rank. A GPU can achieve speedups up only 6x [22]. On smaller inputs, the GPU can perform worse than a single-core CPU executing a standard quicksort. There are other algorithms that are more serial in nature than sorting, and the computation will map poorly onto the GPU.

The problem faced is that there exists a computationally powerful, affordable, and widely available device, but can exhibit poor performance when a serial algorithm is executed. We introduce the concept of algorithm transformation for making a serial algorithm into a more parallel algorithm. By taking a serial algorithm, rearranging computation so that the level of parallelism is increased while the serial execution is minimized, an algorithm is created which exhibits great performance when executed on the GPU.

Applying algorithm transformation to a finite state machine based algorithm, we create an algorithm which exhibits high parallelism. We achieve speedups of up to 6x over our best CPU-based algorithm, and 11x over previous GPU-based attempts.

I contribute several ideas towards the parallelization of algorithms for the GPU. Specifically, I:

- Propose a type of algorithm transformation to increase parallelism
- Increase the amount of parallelism of a serial temporal data mining algorithm
- Present the resulting effect of algorithm transformation as implemented on the GPU
- Discuss and analyze methods to compact data in a fragmented array

Chapter 2

Related Works

This research analyzes a temporal data mining algorithm applied on the GPU. First, works related to data mining and temporal data mining will be examined. The history of GPGPU is provided to provide background for details about the current architecture of today's GPU. An explanation of data parallelism is given, as the GPU excels at data parallel algorithms. Finally, previous work on implementing temporal data mining algorithms is described, which are traditionally not data parallel.

2.1 Data Mining

Data Mining can describe a wide number of algorithms and applications. The focus in this paper is on temporal data mining, which is a subset of pattern mining. A popular example of pattern mining include grocery item associations [1]. Other examples include telecommunication alarm systems [12] and neuroscience [23].

2.1.1 Temporal Data Mining

Specifically, we care about temporal data mining, which is the mining of episodes within an event stream with temporal and ordering constraints. An **event** is a pair of values, one describing which event, and the other a timestamp describing when the event occurred. A sequence of events, called an **event stream** can be expressed as $((E_1, t_1), (E_2, t_2), \dots, (E_n, t_n))$. These events are ordered such that an event with an earlier index in the event stream cannot have a timestamp later than an even with a later index. The event stream is analyzed by searching for patterns called **episodes**, which is a sequence of events with time constraints between each event. For example, the episode $(E_1 \xrightarrow{(1,5]} E_2 \xrightarrow{(1,4]} E_3)$ specifies that E_1 is followed by E_2 within a time interval t such that $1 < t \leq 5$, and E_3 must follow E_2 within a time interval such that $1 < t \leq 4$. There can be any number of events between E_1 and E_2 or E_2 and E_3 , as long as the time constraints are met. A selection of events in an event stream that matches an episode is called an **occurrence**.

An event stream is mined for the frequency that an episode occurs. However, we want to count only non-overlapped occurrences. One reason non-overlapped occurrences are desired is because they have formally-defined statistical significance [10]. The other reason is because fast counting algorithms can be developed for it when the searching is constrained to non-overlapped occurrences [11] [16] [9]. A set of non-overlapping occurrences is such that for all occurrences O_i in the set R with start and end times (O_{i_s}, O_{i_e}) , no other occurrence in O has a start or end time t such that $O_{i_s} < t < O_{i_e}$.

2.1.2 Level-set Mining

It is likely that which episodes to mine for is not known before searching for occurrences. Level-set mining allows for automatic discovery of frequent episodes. This process is illustrated in algorithm 2.1.2 and consists of three steps: episode generation, episode mining, and infrequent episode elimination. Each level grows the size of the generated episode by

one. For example, the first level searches for occurrences of one-node episodes, the second level searches for two-node episodes, and so on. Level-set mining begins by generating all one-node episodes, which is simply the set of all types of events existing in the data. For example, if the event stream $((A, 1), (B, 2), (C, 3), (B, 4), (A, 5))$ is given, then the one-node episodes mined are $(A), (B), (C)$.

Algorithm 1 Level-set temporal data mining.

Given an event stream S and threshold t

Generate a list E of all one-node episodes

while E is not empty **do**

 Count occurrences of each episode in E in S

 Remove episodes from E which have fewer occurrences than t

 Generate episodes from remaining episodes in E , and store result back into E

end while

With these initial episodes, the occurrences of each episode are counted in the event stream to determine frequency. Then infrequent episodes are eliminated from further processing, as we only want to spend computing time on counting frequent episodes. For instance, if (A) is not frequent, then $(A \rightarrow B)$ cannot be frequent, and therefore time will not be spent counting $(A \rightarrow B)$. Then two-node episodes are generated by combining all pairs of frequent one-node episodes with varying time intervals. For the previous example, if episodes $(A), (B), (C)$ are all determined to be frequent, then the resulting two-node episodes are $(A \rightarrow B), (A \rightarrow C), (B \rightarrow A), (B \rightarrow C), (C \rightarrow A), (C \rightarrow B)$. With two given time intervals $(5, 10], (10, 15]$ and applying to the aforementioned two-node episodes, we have $(A \xrightarrow{(5,10]} B), (A \xrightarrow{(10,15]} B), (A \xrightarrow{(5,10]} C), (A \xrightarrow{(10,15]} C)$... and so on.

The counting process is then repeated on these two-node episodes. Infrequent episodes are eliminated again, and three-node episodes are generated from the frequent two-node episodes. The rule is that episodes with one's starting event matching another's ending event are concatenated. For example, if both $(A \rightarrow B), (B \rightarrow C)$ are frequent, then the resulting three-node episode is $(A \rightarrow B \rightarrow C)$.

The level-set mining approach can generate massive numbers of episodes to count at each level due to the exponential increase of episodes. It depends on the number of unique events in the event stream, the number of time intervals desired, the frequency threshold for infrequent episode elimination, and the event stream size. We find that the number of episodes to count follows a diamond shape. That is, the number of episodes start small and grow to its largest size, then decline. Table 2.1 provides a listing of the number of episodes at each level for a small simulated dataset.

Table 2.1: Number of episodes generated at each level for a small simulated dataset with 26 starting episodes.

Level	1	2	3	4	5	6	7	8	9	10
Episodes	26	650	16225	6544	4466	2238	788	100	8	2

2.2 GPGPU

2.2.1 History

Graphics processing units (GPUs) were designed to offload a set of common raster graphics operations from the CPU. They did not contain any user-level programmable interface. In 2001, Nvidia introduced their first GPUs with programmable shaders. Access to these programmable shaders were available through graphics libraries such as OpenGL and DirectX. The GPU was used by Purcell et al. [18] for a non-raster method of rendering, called ray tracing, using these programmable shaders. It gained popularity for being able to accelerate tasks outside the realm of raster graphics.

One disadvantage of using these programmable shaders was the learning curve required. An understanding of computer graphics and rendering pipelines was required to understand how general purpose computation could be performed on the GPU. The GPU only accepts series of vertices passed to the graphics library and series of pixels stored as textures. The

programmer must understand how to encode data into textures and vertices, pass it to the GPU for processing, and decode the result from a framebuffer or texture back to the CPU. In addition, a shader-specific language is needed to program for GPUs. This introduces overhead to the development stage and execution itself.

Nvidia provided a solution to these problems with the introduction of its Compute Unified Device Architecture (CUDA) in 2007. It opens the GPU to general purpose programming without requiring knowledge of computer graphics. It introduces a framework that uses C code with a few CUDA-specific extensions. This has allowed a larger community to utilize the GPU. Non-graphical applications for GPUs such as cryptography [24] and virus detection [20] have been able to accelerate performance by using CUDA.

2.2.2 Architecture

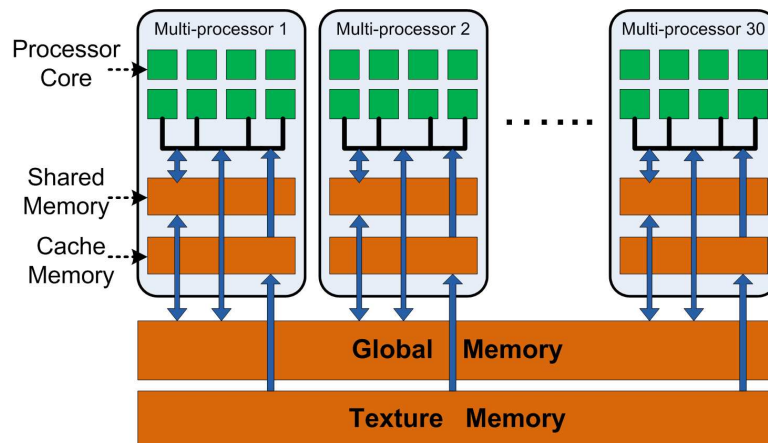


Figure 2.1: Compute Unified Device Architecture (CUDA).

A single chip is able to theoretically perform 933 GFLOPS, which is more than 10x more compared to an Intel Quad-Core CPU, which can theoretically perform 140 GFLOPS. Nvidia's latest chip, GT200 shown in figure 2.2.2, consists of 30 multiprocessors, which each contain 8 cores, 16K 32-bit registers, 16KB shared memory, and 16KB cache memory. Nvidia cards also have up to 4GB of off-chip device memory, which is high-bandwidth but high-latency

memory.

Due to the grouping of cores into multiprocessors, threads are grouped and scheduled differently than typical CPU threads. Threads are mapped to cores, and grouped into blocks. A single block consists of 32 to 512 threads and is mapped to a single multiprocessor. Within the block, threads are scheduled and executed in groups of 32, called a warp. These warps are scheduled in a round-robin fashion, with the possibility of being swapped with another warp within the block for various reasons such as waiting on memory requests. Blocks are scheduled one at a time and are not swapped out until all threads in the block are completed.

There are some issues about the architecture that require attention. Each multiprocessor issues a single instruction for all cores to execute. Because of this, if the threads inside of a warp are executing different instructions, then the threads will not execute concurrently. For instance, if execution reaches a conditional statement, and 16 branch into one section of code, and the other 16 branch into another, then it will require twice the time to execute than if all 32 threads executed the same code. Instead of 32 threads executing the same code, the first 16 threads must execute one section of code and then the second 16 threads will execute the other section of code. In the worst case, where all 32 threads within a warp execute different instructions, performance will decrease by a factor of 32.

Another issue is communication among threads. The only synchronization primitive officially provided by Nvidia's CUDA model is a thread barrier across all threads in an entire block. When there are data dependencies in a section of code, all threads in a block must wait until all threads have reached the barrier to continue. The only other form of inter-thread communication is through atomic operations. These atomic operations block other threads from operating on that memory until completed. In a massively threaded environment, these atomic operations can heavily impact performance due to the massive contention for a single memory address.

2.3 Data Parallelism

Data parallelism is the concept of performing a single operation simultaneously on multiple pieces of data. This type of computation is best suited for architectures with many processors, on the order of hundreds or even thousands of processors [7]. Data parallel algorithms allow each processor to operate on its own portion of data. Ideally, as the problem size increases, the number of processors can be increased and performance will not drop. However, inter-processor communication is the nemesis to high-scalability.

Data parallel algorithms have characteristics that allow performance to scale with the number of processors. One characteristic is that many pieces of data are operated on, and each piece of data can be operated on independently. That is, the result of an operation on one piece of input does not rely on other pieces. Another characteristic is that threads operating on each piece of input does not need to communicate with other threads to operate on a piece of data. Algorithms that exhibit these characteristics very well are called embarrassingly parallel algorithms. Examples of embarrassingly parallel algorithms include fractal image generation, where each pixels can be computed independently of another, and ray tracing, where rays can be processed independently of each other.

2.4 Algorithm Transformation

Algorithm transformation is a widely used term, but generally means to modify an existing algorithm for performance improvement. Neff [13] has used the term to describe the process of algorithm design, where one generates an initial algorithm and then performs successive refinements or transformations to reach a final algorithm. For the purpose of this paper, we use algorithm transformation as a general term for modifying an algorithm to express characteristics more suitable for a given piece of hardware. Algorithm transformation has been applied to digital signal processors, to improve performance [15], and decrease power

consumption [6].

The type of algorithm transformation we apply is the process of taking an algorithm which is not data parallel, and modifying it so that it is massively parallel. We do this in order to take full advantage of the GPU's massively parallel nature. However, it may not be possible to completely remove serial components of an algorithm. In the case of temporal data mining, an episode has dependencies, where one event must follow another chronologically. The goal is to minimize computation time spent on serial tasks, and maximize time spent on parallel tasks.

2.5 Data mining on the GPU

Data mining on the GPU is a new endeavor, with Fang et al. implementing k-means clustering and frequent pattern mining [5]. They introduce a bitmap-based approach for frequent pattern matching yielding up to a $12.1x$ speedup, but only achieve approximately $2x$ speedup with a GPU bitmap implementation over a CPU bitmap implementation.

Work by Archuleta et al. characterized temporal data mining performance on GPUs [2]. Some key findings is that thread parallelism increases performance only when the problem is sufficiently large. When there are many episodes to count, with one episode per thread, then simply having each thread count an individual episode is acceptable because performance scales with the problem set. However, when there are few episodes to count, the GPU is underutilized and performance suffers.

Cao et al. offer a solution to the non-data parallel nature of temporal data mining by introducing MapConcatenate [3], which is similar but distinct from the MapReduce mentioned by [4]. In the map step, the data stream is split amongst threads and the results are subsequently concatenated to develop a final count. The parallelism is increased, but the concatenate step incurs a large overhead. Although it is an improvement over a previous GPU implementation, this algorithm is still slower than the CPU on small problem sizes.

A recent effort by Patnaik et al., which is a part of this thesis, describes the process of algorithm transformation towards data parallel algorithms and corresponding improvements [17].

Chapter 3

Temporal Data Mining Algorithms

Fast counting algorithms for temporal data mining on CPU are based on finite state machines, which are not parallelizable. This chapter provides background on a CPU temporal data mining algorithm, and two simple approaches to parallelization of the algorithm. It will be shown that the level of parallelism in these two simple approaches is not enough, and when the number of episodes is smaller, the GPU does not perform as well as the CPU.

3.1 Finite State Machine

Temporal data mining has many applications, one of which is in neuroscience. A grid of sensors is placed onto a section of active brain cells. A diagram of this grid is shown in figure 3.1. These sensors measure electric potential. A spike, or rapid increase and subsequent decrease of electric potential, indicates that cells around the sensor has fired. The network of neurons can be studied by recording which sensor spikes and at which time it spikes, called an event.

Relationships are discovered by searching for frequent episodes. An episode is simply a series of events. With temporal information, there are also constraints on the amount of time be-

tween each event in the episode. In figure 3.1, the episode $(B_2 \xrightarrow{(10,15]} C_3 \xrightarrow{(5,10]} C_4 \xrightarrow{(10,15]} D_5 \xrightarrow{(10,15]} C_6 \xrightarrow{(5,10]} C_7)$ is shown. This means that the mentioned sensors spiked in order, also following the listed time constraints as mentioned in section 2.1.

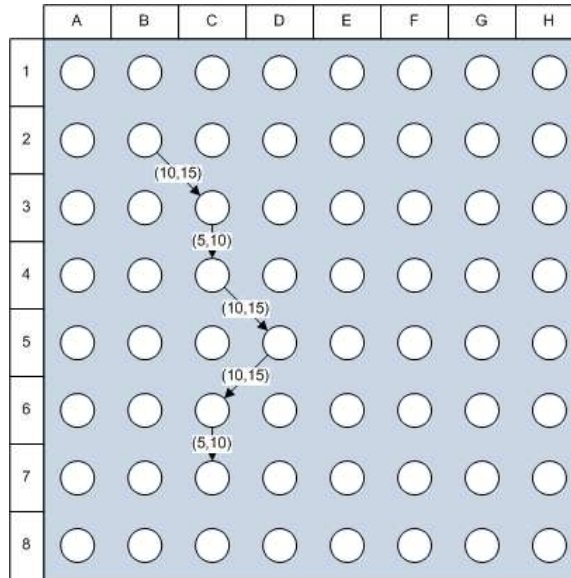


Figure 3.1: Sensor grid with a visualized episode. The numbers in parentheses describe the minimum and maximum time constraint between events.

The algorithm for discovering these episodes in an event stream is described in [12]. It utilizes a finite state machine (FSM) to track each instance of an episode. The state machine is complex, so first we remove temporal constraints from the problem and examine a state machine that would count occurrences without temporal constraints, shown in figure 3.1. This state machine tracks the episode $(A \rightarrow B \rightarrow C)$. It simply waits for an A in the event stream, until one is found and progresses to the next state, where it waits for a B . Again once a B is found, the next state is reached and a C is waited for. Finally, once the C is reached the occurrence count is incremented and the state machine transitions back to the starting state.

To include temporal constraints, we modify the previous state machine to check for temporal constraints shown in figure 3.1. We include a variable I which represents time that has passed

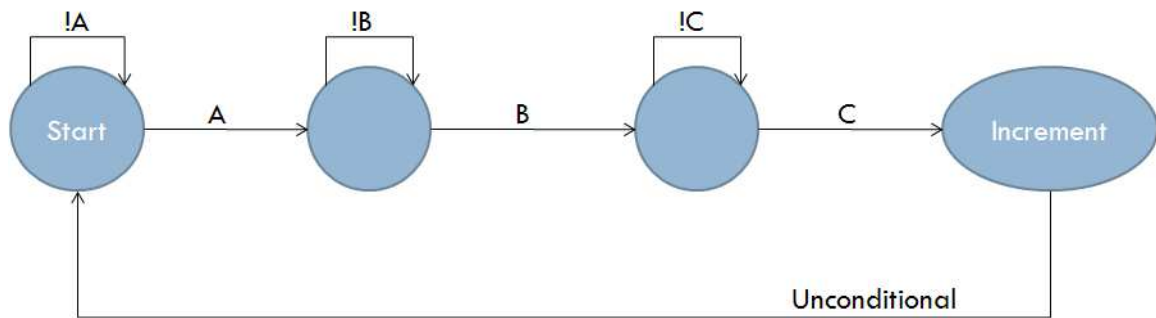


Figure 3.2: Finite state machine used to count episodes in an event stream without temporal constraints.

since transitioning into the current state. So we examine the state machine, and an A is found, and we transition into that state. Then 7 time units later a B is found. In this case $I = 7$ since there are 7 time units between the A and the B . Also note that additional transitions were added that lead back to start. This is in case the maximum temporal constraint is exceeded, there is no need to wait for the next event in the occurrence, because the temporal constraint can never be satisfied.

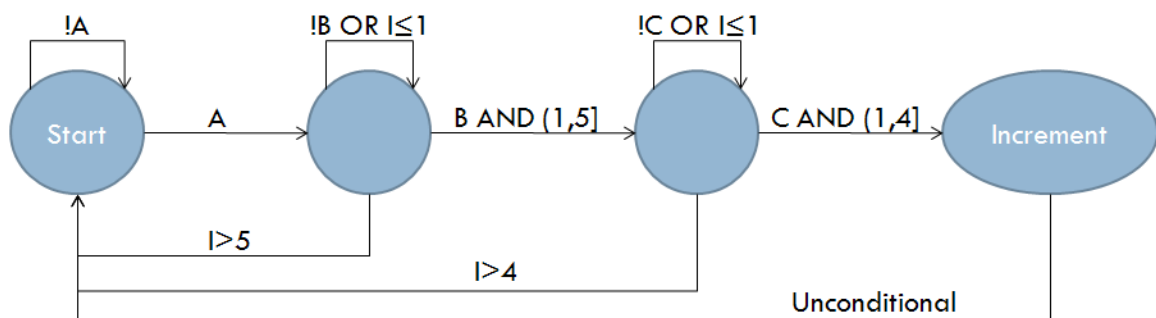


Figure 3.3: An incorrect finite state machine used to count episodes in an event stream without temporal constraints.

However, it is possible for this state machine to produce incorrect results. Consider the event stream $AXAXBXC$, where the X events are events that we do not care about, and the episode $(A \xrightarrow{(1,3]} B \xrightarrow{(1,3]} C)$. When the first A is found, it waits for a B within the constraints. However, when the B is reached, the interval I is 4 and the state machine transitions back

to the starting state. Therefore, this state machine does not capture this occurrence and is incorrect.

To correct this, we want to track all possible occurrences. So we add a fork to the state machine when each transition is made, so that one state machine will continue to wait for another event to transition the state, and another state machine will progress into the next state. This will find all occurrences existing in the event stream. However, we want only non-overlapping occurrences. This is enforced by simply clearing all state machines once one of the state machines reaches the final state. This clear will stop the other state machines from finding occurrences that overlap with the one that is found. Also note that this removes the need for a fork at the final transition, as the final transition will clear the forked state machine. This final, correct state machine is shown in figure 3.1. The algorithm that implements this state machine using a fast list-based approach as described in [16] is shown in figure 3.1.

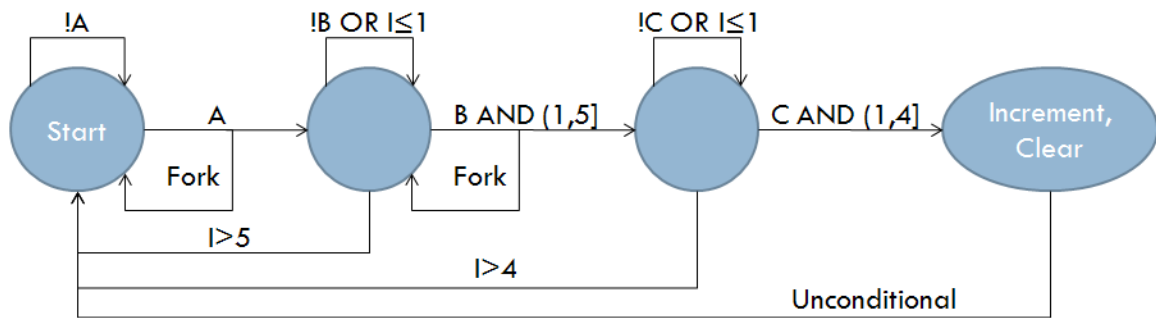


Figure 3.4: Correct finite state machine with a fork used to count episodes in an event stream with temporal constraints.

3.2 Per-thread per-episode (PTPE)

This per-thread per-episode algorithm uses a straightforward process to put the finite state machine (FSM) algorithm onto the GPU. We simply identify the data-parallelism of the

Algorithm 2 Find episode count in an event stream using FSM.

Given an event stream S , and an episode consisting of a sequence of events E of size N and a sequence of intervals I of size $N - 1$

Initialize $N - 1$ growable lists of timestamps, labeled L_0 to L_{N-2} , to empty

Initialize episode count $C = 0$

for all events and timestamps, e and t in S **do**

if $e = E_0$ **then**

 Add t to L_0

else

for i from 1 to $N - 1$ **do**

if $e == E_i$ **then**

for all timestamps s in L_{i-1} **do**

if $t - s > I_{i-1MIN}$ AND $t - s \leq I_{i-1MAX}$ **then**

if $i = N - 1$ **then**

 Increment C

 Clear all lists from L_0 to L_{N-2}

else

 Add t to L_i

end if

end if

end for

end if

end for

end if

end for

Return C

existing FSM algorithm, which is the data-independence amongst each episode. The occurrences of an episode can be counted by a single thread, so each thread processes the event stream modifying its state machine to generate a final count. There is a one-to-one mapping of threads to episodes.

The PTPE algorithm works rather well and achieves a speedup [3] over the CPU when there are a large number of episodes. However when there are few episodes, few threads are used, and the GPU is underutilized because it is given less threads than it is capable of. Counting many episodes in parallel is the simple approach to parallelization, however it only effective at levels with many episodes.

When simply mapping one thread per episode, the GPU becomes severely underutilized because it can support many thousands of threads. As table 2.1 mentioned in section 2.1.2 shows, there are enough episodes to fully utilize the GPU at the largest levels, but resources are wasted towards the beginning and end of the level-set approach. When the episodes are few, another approach must be taken to take full advantage of the GPU.

3.3 MapConcatenate

The approach used by MapConcatenate is to split the input data into pieces, process each piece with the original algorithm, and combine the result with an appropriate scheme. Parallelism can be achieved by increasing the number of threads, and reducing the amount of work each thread does, as each thread has less data to process. This approach is similar to MapReduce [4], in which each thread receives a portion of data to process, and a single result is generated by reducing two results into one.

A MapReduce approach is difficult when splitting a data stream that a single state machine operates on. On a single piece of the data stream, it is unknown what state the FSM will be in when entering the piece. A solution to this problem is to process the data starting with multiple state machines configured to represent all possible states of the state machine. With

the temporal data mining example, when processing a k -node episode, k state machines are used to process a portion of data. This is implemented by seeking backwards in the event stream a certain amount of time depending on the intervals in the episode, shown in figure 3.3. The result is k counts, one for each state machine processed.

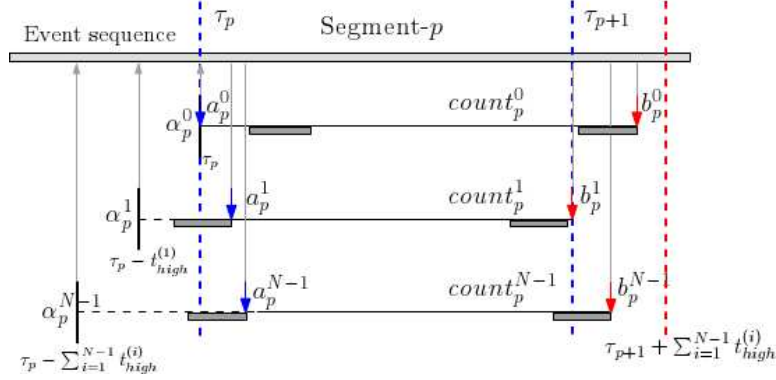


Figure 3.5: Map step of the MapConcatenate algorithm.

For our solution, the reduce step does not strictly follow the paradigm set by others in the parallel processing community. A reduce allows any two results to be combined into a single result. Our solution can only combine results whose data pieces are adjacent to each other in the data stream. Therefore, we call our solution MapConcatenate, shown in figure 3.3.

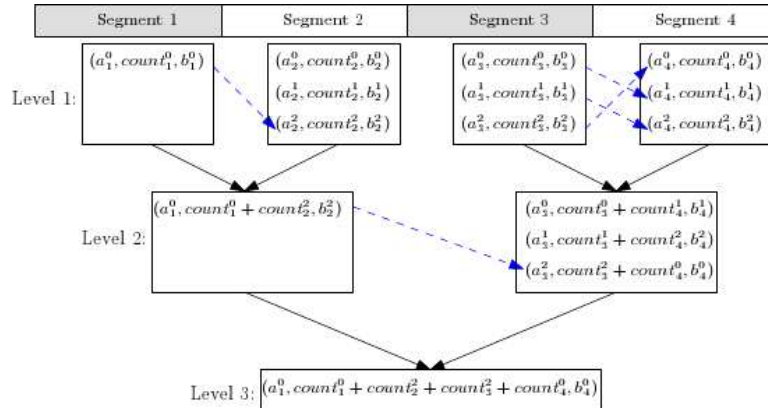


Figure 3.6: Concatenate step of the MapConcatenate algorithm.

MapConcatenate achieves better performance than PTPE when there are fewer episodes to count due to the increase of parallelism for counting a single episode. However, more work

is performed in both the map and the concatenate steps. The map step requires one thread per event in the episode for each piece of the event stream. The concatenate step requires computation that is not friendly to the GPU, because many threads become idle during the final phases. This is due to the number of data pieces to concatenate is reduced by a factor of two at each step, and the final steps only use a few threads out of many threads available.

The parallelism achieved by MapConcatenate incurs a cost by performing more work than both the FSM and PTPE algorithms. Although it outperforms the PTPE algorithm on small inputs, it does not perform better than the CPU. This work inefficiency must be addressed, and requires more than straightforward implementations of serial algorithms on the GPU. A new approach must be taken to solve the data mining problem in a highly parallel environment.

Chapter 4

Algorithm Transformation

Standard GPU optimization is not enough to achieve efficient use of the GPU. The PTPE approach has obvious disadvantages when applied to a small number of episodes. The lessons of MapConcatenate showed that splitting the data stream and applying a serial algorithm yields poor results for temporal data mining. The overhead introduced by MapConcatenate presented the need for a new algorithm. We set out to transform the algorithm by minimizing the serial aspects of the algorithm and increase the parallel work done. To do this, we returned to the problem statement and identified two key requirements:

1. Identify all occurrences of a given episode
2. Produce a maximal set of non-overlapping occurrences

From these two requirements, we devise a two-step process. First, all possible occurrences of given episode are identified. Second, a maximal set of non-overlapping occurrences are selected. The size of that set yields the count of non-overlapped occurrences for a particular episode. This two-step approach is shown in algorithm 4

Algorithm 3 Find maximal count of non-overlapping occurrences of a given episode in an event stream.

Given an event stream S , and an episode E

Find the set M of all possible occurrences of E in S

Produce a maximal set O of non-overlapping matches

Return size of O

4.1 Finding Occurrences

Finding occurrences of one-node episodes is a simple and highly parallel task. Each thread could take a section of the event stream, and simply count all events matching the single event of the episode. This is nothing more than an indexing task performed in parallel. With these occurrences indexed, finding occurrences for a two-node episode is simple, if the event of the one-node episode matches the first event in the two-node episode.

For example, we want to find the episode $(A \xrightarrow{(1,5]} B)$. First all events matching A are found and indexed. For each starting index of A in the event stream, we look for a matching B so that the distance t is $1 < t \leq 5$. This can be accomplished in parallel. Within a typical event stream, thousands of indices for A or any starting event are found, and therefore thousands of threads can be launched searching for a matching B within the time constraints.

This concept can be extended to an episode of any size. If now the episode $(A \xrightarrow{(1,5]} B \xrightarrow{(1,4]} C)$ is searched for, and the indices for all occurrences of $(A \xrightarrow{(1,5]} B)$ are known, then one thread per occurrence of $(A \xrightarrow{(1,5]} B)$ can be launched to find a corresponding C within the time constraints. That is, from an index of where the B is in an $(A \xrightarrow{(1,5]} B)$ pair, a matching C can be searched for in the event stream. This concept is presented in figure 4.1.

We want to record the index of where the occurrence starts, and the index of where the occurrence ends. We record both because the occurrences will be modeled as intervals, which is necessary for the second step of the two-step process.

This step is massively parallel. For each starting index, one thread is launched. On the input

Algorithm 4 Find all matches of a given episode in an event stream.

Given an event stream S of size L consisting of events e and timestamps t

Given an episode consisting of a sequence of events E of size N and a sequence of intervals I of size $N - 1$

Initialize a list of starting intervals X to empty, and a list of ending intervals Y to empty

for all i from 0 to $L - 1$ **do**

if $E_i = I_0$ **then**

 Add interval (i, i) to X

end if

end for

for all i from 1 to $N - 1$ **do**

for all j from 0 to $sizeof(X) - 1$ in parallel **do**

 Set s to $X_{j_{END}}$

while $t_s - X_{j_{END}} \leq I_{i_{END}}$ **do**

if $e_s = E_i$ AND $t_s - X_{j_{END}} > I_{i_{BEGIN}}$ **then**

 Add $(X_{j_{BEGIN}}, s)$ to Y

end if

end while

end for

 Swap X and Y

 Set Y to empty

end for

Return Y

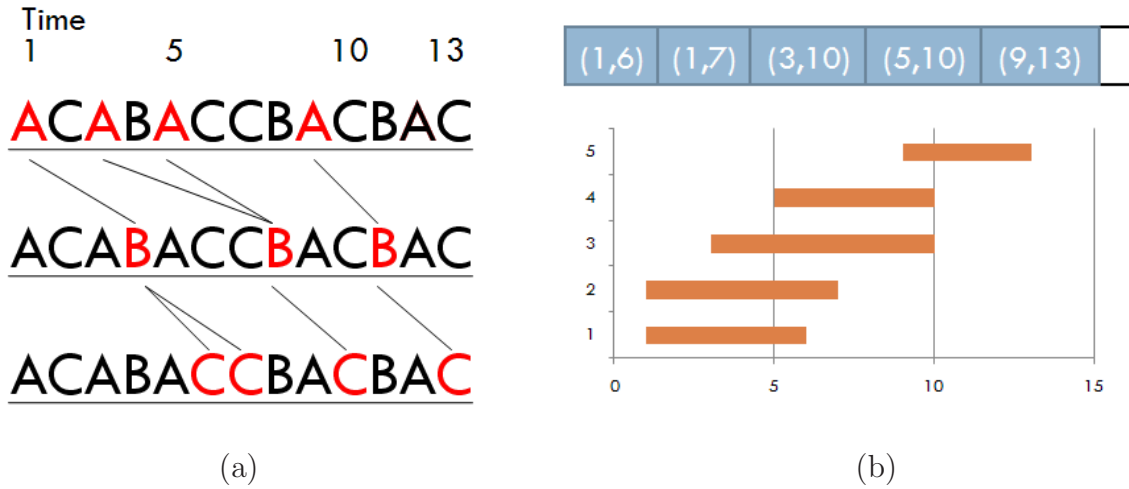


Figure 4.1: (a) Episode matching algorithm applied to an event stream. The first three lines all represent the same event stream, but each line illustrates a different event in the episode. (b) A listing of the intervals generated with a graph of the intervals.

sizes tested, there are typically thousands of starting indices, allowing for enough threads to keep the GPU fully utilized. Finding matches can be performed such that each thread performs computation independently of other threads.

The result of this step is the set of all possible occurrences of a given episode in an event stream. However, as shown in figure 4.1, the occurrences may overlap. Also, the occurrences may also share identical events, as in occurrence 1 uses the same *A* that occurrence 2 uses. Since this parallel approach may generate these undesirable cases, the second step of our two-step process aims to remove these cases to produce a maximal set of non-overlapping occurrences.

4.2 Overlap Removal

To produce a maximal set of non-overlapping occurrences, we turn towards interval scheduling algorithms. An interval is defined to be a pair of start and end times, (i_s, i_e) . The goal of

the interval scheduling problem is, given any set of intervals I , to produce a maximal set of intervals such that no two intervals in the set intersect, or conflict. Given two intervals (j_s, j_e) and (k_s, k_e) , intersection is defined as $k_s < j_s < k_e \vee k_s < j_e < k_e \vee j_s < k_s < j_e \vee j_s < k_e < j_e$.

This is solved using a greedy algorithm 4.2 given from [8]. Given a list of intervals, the algorithm selects one interval to be included in the maximal set. Then all intervals conflicting with the selected one are removed from the list. The rule for interval selection is to choose the interval with the smallest end time first and eliminate conflicting intervals. From a scheduling perspective, the idea is to choose the interval finishing earliest so that the resource can be freed as soon as possible.

This greedy algorithm is proven to produce a maximal set of non-conflicting intervals by [8]. When performed on a list of intervals sorted by end time, the performance is $O(n)$.

Algorithm 5 Return a set of non-conflicting intervals of maximal size, adapted from [8].

Initially let R be the set of all intervals, and let A be empty

while R is not yet empty **do**

Choose an interval $i \in R$ that has the smallest finishing time

Add request i to A

Delete all intervals from R that conflict with request i

end while

Return the set A as the set of accepted requests

The removal of overlaps is the second step in the two-step process. Once this algorithm is applied to the set of all matching episodes, the result is a set of non-overlapping matches of maximal size. The size of the resulting set is the frequency of an episode in the event stream.

4.3 Compaction

A difficulty in massively multi-threaded environments is dynamically growing lists shared amongst multiple threads. During the first step of the two-step process, algorithm 4.1 re-

quires it when adding intervals to global lists X and Y . The reason for doing this appears when performing a thread launch. For example, given episode $(A^{(1,5]}B^{(1,4]}C)$ and the algorithm has just found and written indices of all $(A^{(1,5]}B)$ to memory. Now a thread launch is performed to find C from previously found $(A^{(1,5]}B)$ occurrences. Each thread needs to be assigned an index, and this is done by calculating an offset determined by each thread's ID. With an uncompact array, illustrated in figure 4.3, some threads will not perform work. This can cripple performance, as in practice the uncompact arrays are sparse, leaving many threads idle. Since threads are executed in groups as warps, it will require the same amount of time to execute 32 active threads as it would 1 active thread and 31 idle threads. This will greatly reduce performance by under-utilizing the GPU.

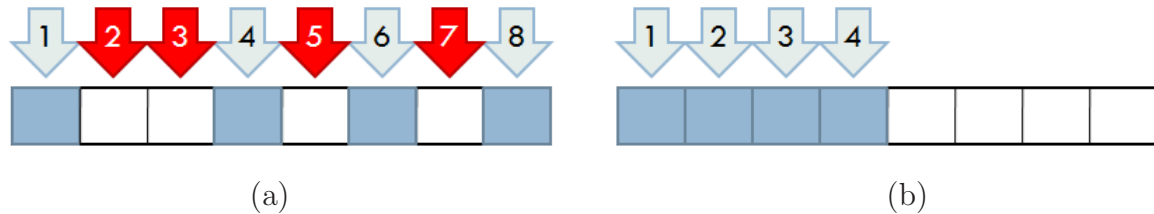


Figure 4.2: Thread launches on a compacted and uncompact array, with threads represented by arrows and idle threads highlighted. (a) An uncompact array requiring 8 threads to be launched. (b) A compacted array only requiring 4 threads.

Compaction can be accomplished by using either a lock-based method or a lock-free method. The lock-based method requires the acquisition of a lock before ever writing to global memory. The lock-free method instead allows each thread to write to a pre-allocated section of memory for that thread, and performs additional steps to produce a final compacted array.

4.3.1 Lock-based atomic compaction

Each thread needs to write all matches found to a global array. With lock-based compaction, each thread allocates space to write to by adding the number of matches found to a global counter. To prevent race conditions, the counter access and modification must be atomic.

A thread must acquire a lock, modify the counter, then release the lock. This functionality is available through the CUDA API with the *atomicAdd()* function.

Due to the GPU's massively parallel nature, data will be written in the order the threads allocate space in the global list. Since it is difficult to determine the order in which threads will finish finding matches, it is consequently difficult to determine the order in which space in the global list is allocated. Therefore it is unknown how occurrences will be written at the end of the first step of the two-step process. Since the second step requires a list of intervals sorted by end time, a $O(n \log n)$ sorting algorithm will be performed to sort the list.

4.3.2 Lock-free prefix sum compaction

The prefix sum method of compaction is a three step process. First, threads are launched to perform algorithm 4.1 except the threads write the only the count of matches found to an array. Then a prefix sum operation is applied to the count array, which yields the offset where the thread should write its matches. Then algorithm 4.1 is performed and the matches are written to the offsets computed during the prefix sum step.

Given an array of numerical values, an exclusive prefix sum will produce an array such that each element is the sum of all elements before it in the input array. As shown in table 4.1, the first value in the output array is 0, as there are no previous elements. The second value in the output array is simply the first value in the input array. The third output value in the output array is the sum of the first two input values, and the fourth output value is the sum of the first three input values. Given an input array I and output array O both of size n , this can be formally represented as:

$$O = [0, I_1, I_1 + I_2, I_1 + I_2 + I_3, \dots, I_1 + I_2 + \dots + I_n]$$

The output array calculated by the prefix sum operation acts as an array of offsets that each thread can write into. We can consider the input array of table 4.1 to be the number of

Table 4.1: Input and output of a prefix sum operation.

Input	7	9	3	12	5
Output	0	7	16	19	31

matches to be written, and the output array as the offsets of where to write to. The first thread will find 7 matches and write them to offset 0. The second thread will find 9 matches and write them to offset 7, which will produce no gaps as the first match of the second thread will be written at the next offset after the last match from the first thread. The result of this method is a compacted list of matches, without using any locking mechanism.

The CUDA Data Parallel Primitives (CUDPP) library contains a parallel work-efficient implementation of prefix sum, based on [21]. We use this for all algorithms requiring a prefix sum operation. Also included in the CUDPP library is a compact method, which compacts an array of fragmented data into a contiguous array. It compacts by accepting three key arguments:

- Fragmented array of elements
- Array of valid/invalid 32-bit values, one per element in fragmented array
- Size of fragmented array

The problem with the CUDPP implementation is that it cannot take advantage of the fact that all intervals written by a thread are contiguous. For a given thread t_i with n valid intervals, all n intervals will be written contiguously into memory. Using CUDPP's compact method, it requires a single 32-bit value per element in the fragmented array.

Because each thread writes intervals contiguously, we implement compaction similar to the CUDPP compact library, except that a count per thread is recorded rather than a valid/invalid 32-bit value per element. For example if each thread can find a maximum of 10 intervals, then this approach will reduce the amount of data to scan by 10x.

Unlike the the lock-based compaction, the matches written to the global list are ordered by thread. The first thread writes first, the second thread writes next, and so on. Since the list of starting events is sorted by start time, the threads will read the list in order. That is, the first thread’s starting event will have an earlier index than the second, and so on. Therefore, using prefix sum compaction, we can guarantee that the final list of occurrences is sorted by starting time.

However, the interval scheduling algorithm requires a list sorted by end time. The algorithm presented can be slightly altered in a number of different ways to produce a final list of occurrences sorted by end time. We implemented our algorithm so that the first step, occurrence searching, moves backwards through the event stream. Another modification is that we reverse the episodes given. For example, the episode $(A \xrightarrow{(1,5]} B \xrightarrow{(1,4]} C)$ would now become $(C \xrightarrow{(1,4]} B \xrightarrow{(1,5]} A)$. These modifications do not alter the final result other than the order in which they are written. The sorted list will eliminate to perform an $O(n \log n)$ sort on the final set of occurrences.

It is not obvious whether atomic compaction or prefix-sum compaction provides better performance. Atomic operations are blocking operations, meaning that any other thread attempting to access that resource while in use will be suspended. This has the potential to devastate performance on the GPU, because there are massive number of threads which could contend for this resource at the same time. This would serialize all threads on the GPU, effectively running one thread at a time, which is a severe under-utilization of the GPU. The third option is a non-blocking operation, but contains three steps in which all threads must complete before continuing. These three steps are interval counting, cumulative sum, then interval writing. A grid-wide barrier is not provided by CUDA, so three separate thread launches must occur, causing overhead.

Chapter 5

Results

In Section 5.2, we compare the performance of our new algorithm to *MapConcatenate* and a CPU implementation of the original algorithm described in Chapter 4. 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 *PrefixSum*, as shown in Section 5.3.

The hardware used for obtaining the performance results are given in Table 5.1:

Table 5.1: 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
Processor Clock (GHz)	2.33
Memory (MB)	4096

5.1 Test datasets and algorithm implementations

The datasets used here are generated from the non-homogeneous Poisson process model for inter-connected neurons described in [16]. This simulation model generates fairly realistic spike train data. For the datasets in this paper a networks 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 5.2 gives the duration and number of events in each dataset.

Table 5.2: 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

5.2 Comparisons of performance

We compare MapConcatenate performance to the best CPU and GPU versions by having each algorithm count 30 episodes. MapConcatenate 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.

MapConcatenate is clearly a poor performer compared to the CPU, with up to a 4x slowdown. Compared to our best GPU method, *MapConcatenate* is up to 11x slower. This is due to the

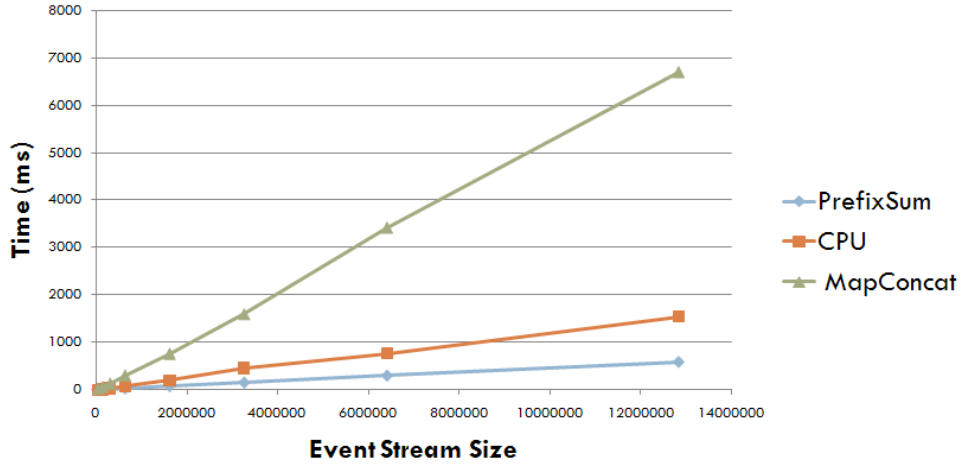


Figure 5.1: Performance of MapConcatenate compared with the CPU and best GPU implementation, counting 30 episodes in Datasets 1-8.

overhead induced by the merge step of *MapConcatenate*. 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 *MapConcatenate* and one of our redesigned algorithms, *PrefixSum*. Dataset 2 was used for profiling each implementation. Due to its complexity, *MapConcatenate* exhibited poor features such as large amounts of divergent branching and a large total number of instructions executed, as shown in Table 5.3. Comparatively, the *PrefixSum* implementation only exhibits divergent branching.

Table 5.3: CUDA Visual Profiler Results

	MapConcatenate	PrefixSum
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

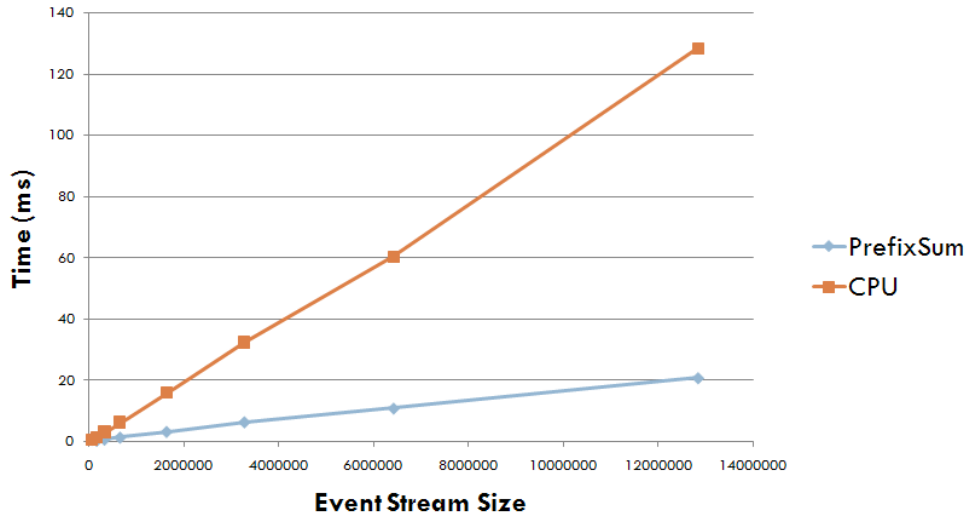


Figure 5.2: Performance comparison of the CPU and best GPU implementation, counting a single episode in Datasets 1 through 8.

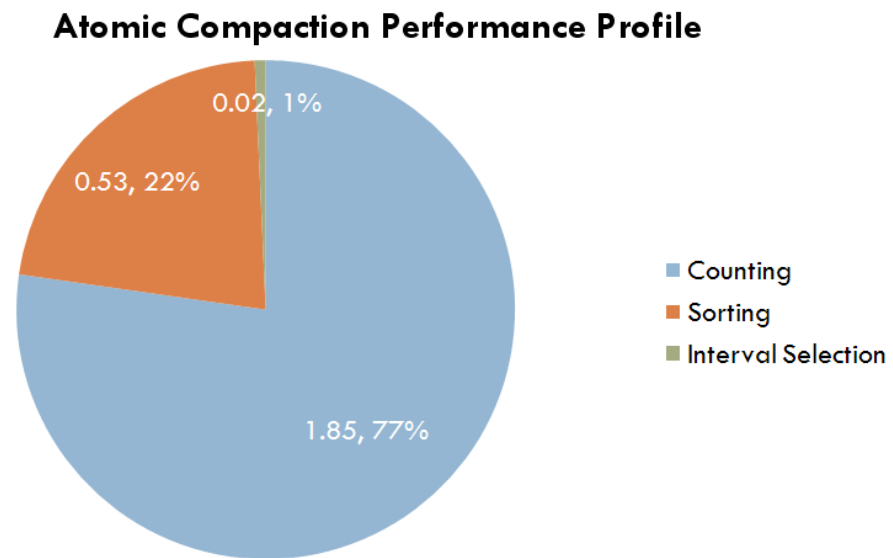
when the episodes are few.

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 5.2.

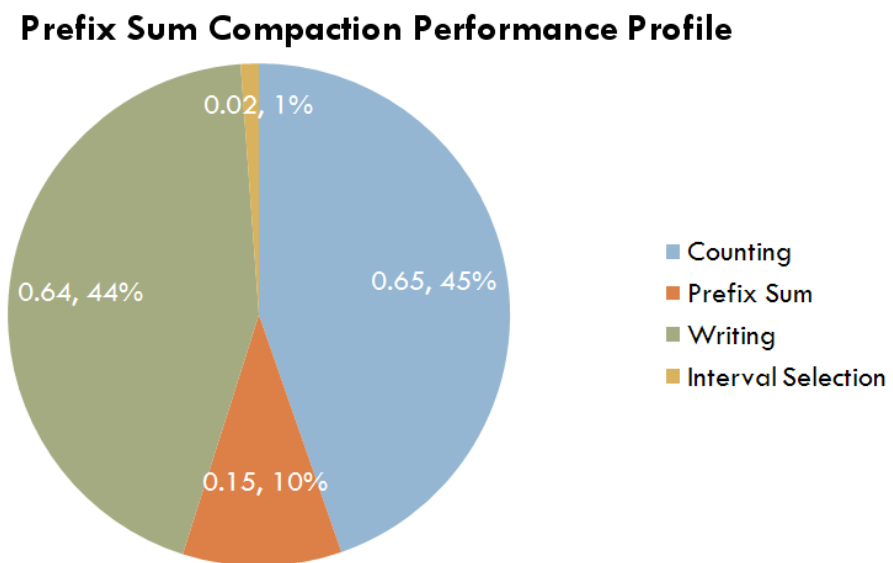
5.3 Analysis of the new algorithm

To better illustrate the performance differences between *AtomicCompact* and *PrefixSum*, we examine the amount of time spent in each step of execution for each compaction method. For this experiment, we counted occurrences of a four-node episode. For *AtomicCompact*, we measure the combined counting time and compaction time (due to both steps being in the same thread launch), interval sorting, and interval selection time. For *PrefixSum*, we measure the counting time, prefix sum operation time, writing time, and interval selection time.

AtomicCompact spends a majority of time during the counting step, due to the atomic



(a)



(b)

Figure 5.3: Profile of time spent performing each subtask for AtomicCompact (a) and PrefixSum (b)

compaction performed. Sorting takes a smaller yet significant portion of time, and interval selection is a very tiny part of the overall time. *PrefixSum* compaction spends equal portions in counting and writing because both perform the same computation except for the data written to memory. The prefix sum operation takes a small part of the overall time, and interval selection again uses a tiny slice of time.

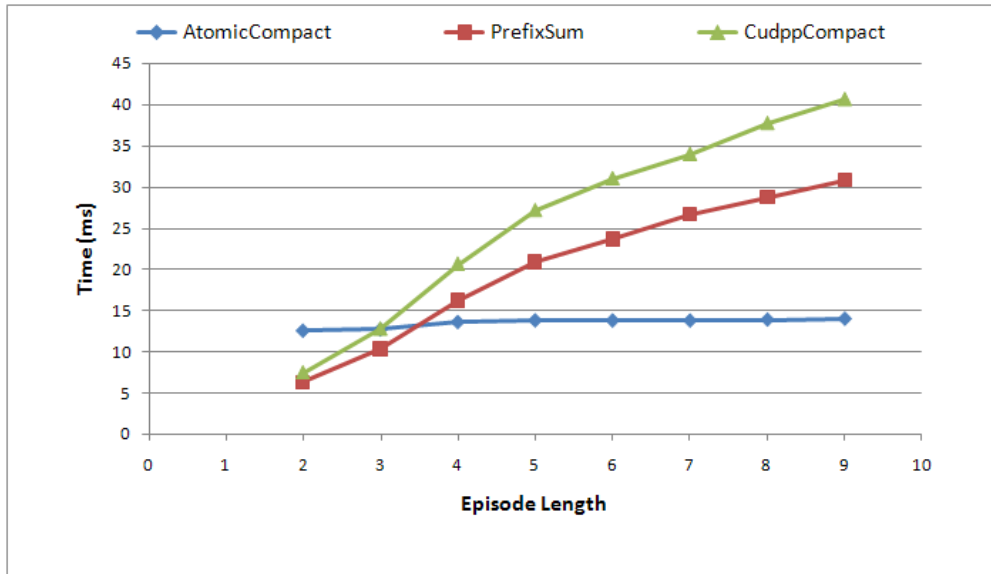


Figure 5.4: Performance of algorithms with varying episode length in Dataset 1.

Figure 5.4 contains the timing information of three compaction methods of our redesigned GPU algorithm with varying episode length. Compaction using CUDPP is the slowest of the GPU implementations, due to its method of compaction. 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 *PrefixSum* 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.

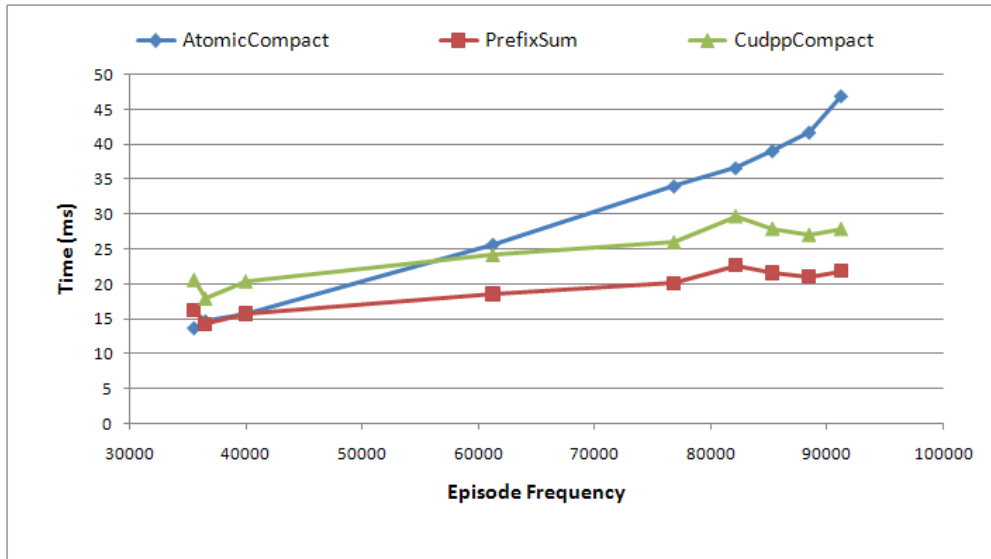


Figure 5.5: Performance of algorithms with varying episode frequency in Dataset 1.

We find that counting time is related to episode frequency as shown in Figure 5.5. There is a linear trend, with episodes of higher frequency require 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 5.5 that requires explanation is the bump where the episode frequency is slightly greater than 80,000. This is because it is not the final non-overlapped count that affects the running time, it is the total number of overlapped episodes found before the scheduling algorithm is applied to remove overlaps. The x-axis is displaying 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 *PrefixSum* 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 methods of the redesigned algorithm.

Chapter 6

Conclusion

The task of reverse engineering the brain is a monumental task. One part of this process is understanding how neural networks are formed through the lifetime of brain cells. This is one of many applications that temporal data mining can be applied to in order to achieve scientific understanding of large and complex datasets. The computational demand for processing these datasets can be met through utilizing the GPU. It excels at data parallel tasks, achieving impressive speedups for such a low cost device.

One caveat of the GPU is the poor performance exhibited by running non-data parallel algorithms on the device. The state machine-based temporal data mining algorithm is an example of a non-data parallel algorithm that had not yet seen performance improvement on the GPU. Standard optimizations for applying algorithms onto the GPU do not perform well for some non-data parallel algorithms. Two straightforward approaches, PTPE and MapConcatenate, show that fast implementations cannot be achieved through standard approaches.

The concept of algorithm transformation to increase data parallelism is introduced. The application of algorithm transformation increased the level of parallelism and reduced the amount of time performing serial execution. Parallelism was increased by searching through

the event stream in parallel with little communication amongst threads required. Serial execution is still required, but as shown by the performance profiles very little time is spent performing serial execution. Through this application of algorithm transformation, we achieve a 6x speedup over our best CPU-based algorithm and 11x speedup over previous GPU-based attempts.

The increase of parallelism does come at a price. The serial algorithm finds only non-overlapping occurrences, and the new parallel version finds all occurrences and applies a post-processing step to remove overlap. More work is performed than the serial algorithm, but the performance is better due to the computational power of the GPU. A reduction in work while maintaining the level of parallelism would further improve the performance.

Future work would include the reduction of serial work performed. Although the new algorithm exhibits high parallelism, there is still serial execution being performed. For an episode consisting of n events, $n - 1$ steps must be performed one after the other. An idea for further parallelization can be explored to find large episodes in parallel. For example, to find all occurrences of an episode $A \rightarrow B \rightarrow C \rightarrow D$, both the $A \rightarrow B$ matchings and the $C \rightarrow D$ matchings can be found in parallel. Then the $A \rightarrow B$ and $C \rightarrow D$ instances would have to be stitched together to obtain a set of occurrences. This could be one direction in which the existing work for parallelizing temporal data mining algorithms can be improved.

Other possibilities for the future include implementations on different hardware environments. What kind of issues are encountered when scaling up to two GPUs and beyond? Also, the GPU is one of many tools for solving this temporal data mining problem. Hybrid approaches involving both the CPU and GPU can be applied through the use of pipelining execution.

In addition to other approaches to solving this problem, future work would include approaches to solving similar yet distinct problems. This work focuses on the non-overlapped counting of episodes, but there are other criteria for choosing which occurrences to include as part of the frequency measurement. The non-overlapped requirement is solved through

existing interval scheduling algorithms, but other criteria may not have such simple greedy algorithms to use as to process the set of all occurrences as found by the first step of the two-step process.

Bibliography

- [1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 207–216. ACM Press, 1993.
- [2] Jeremy Archuleta et al. Multi-dimensional characterization of temporal data mining on graphics processors. Technical Report 1058, Virginia Tech Department of Computer Science, Jan 2009.
- [3] Yong Cao, Debprakash Patnaik, Sean P. Ponce, Wuchun Feng, and Naren Ramakrishnan. Towards chip-on-chip neuroscience: Fast mining of frequent episodes using graphics processors. Technical Report 0905.2200v1, Virginia Tech Department of Computer Science, May 2009.
- [4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. OSDI'04*, San Francisco, CA, December 2004.
- [5] W. Fang, K.K. Lau, M. Lu, X. Xiao, C.K. Lam, P.Y. Yang, B. He, Q. Luo, P.V. Sander, and K. Yang. Parallel data mining on graphics processors. Technical Report HKUST-CS08-07, Hong Kong University of Science and Technology, Oct 2008.
- [6] Manish Goel and Naresh R. Shanbhag. Dynamic algorithm transformation (dat) for low-power adaptive signal processing. In *ISLPED '97: Proceedings of the 1997 international*

- symposium on Low power electronics and design*, pages 161–166, New York, NY, USA, 1997. ACM.
- [7] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.
- [8] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Pearson Education, Inc., Boston, Massachusetts, 2005.
- [9] S. Laxman, P.S. Sastry, and K.P. Unnikrishnan. Fast algorithms for frequent episode discovery in event sequences. In *Proceedings of the 3rd Workshop on Mining Temporal and Sequential Data, SIGKDD'04*, 2004.
- [10] S. Laxman, P.S. Sastry, and K.P. Unnikrishnan. Discovering frequent episodes and learning hidden markov models: A formal connection. *IEEE Transactions on Knowledge and Data Engineering*, Vol 17(11):pages 1505–1517, Nov 2005.
- [11] S. Laxman, P.S. Sastry, and K.P. Unnikrishnan. A fast algorithm for finding frequent episodes in event streams. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'07)*, pages 410–419, 2007.
- [12] H. Mannila, H. Toivonen, and A.I. Verkamo. Discovery of frequent episodes in event sequences. *DMKD*, Vol. 1(3):pages 259–289, Nov 1997.
- [13] Norman Neff. Algorithm design by successive transformation. In *CCSC '01: Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges*, pages 270–275, , USA, 2001. Consortium for Computing Sciences in Colleges.
- [14] National Academy of Engineering. Grand challenges for engineering, 2009.
- [15] Keshab K. Parhi. High-level algorithm and architecture transformations for dsp synthesis. *J. VLSI Signal Process. Syst.*, 9(1-2):121–143, 1995.

- [16] D. Patnaik, P.S. Sastry, and K.P. Unnikrishnan. Inferring Neuronal Network Connectivity from Spike Data: A Temporal Data Mining Approach. *Scientific Programming*, 16(1):49–77, 2008.
- [17] Debprakash Patnaik, Sean P. Ponce, Yong Cao, Wuchun Feng, and Naren Ramakrishnan. Accelerator-oriented algorithm transformation for temporal data mining. In *6th IFIP International Conference on Network and Parallel Computing*, 2009.
- [18] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, 2002.
- [19] Shane Ryoo, Christopher I. Rodrigues, Sara S. Bagsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.
- [20] Elizabeth Seamans and Thomas Alexander. Fast virus signature matching on the gpu. *GPU Gems 3*, pages 771–783, 2007.
- [21] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.
- [22] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, 2008.
- [23] D. A. Wagenaar, Jerome Pine, and Steve M. Potter. An extremely rich repertoire of bursting patterns during the development of cortical cultures. *BMC Neuroscience*, 2006.
- [24] Takeshi Yamanouchi. Aes encryption and decryption on the gpu. *GPU Gems 3*, pages 785–803, 2007.