Multi-Dimensional Characterization of Temporal Data Mining on Graphics Processors*

Jeremy Archuleta Yong Cao Wu-chun Feng Tom Scogland
{jsarch, yongcao, feng, njustn}@cs.vt.edu
Department of Computer Science, Virginia Tech

Abstract

Through the algorithmic design patterns of data parallelism and task parallelism, the graphics processing unit (GPU) offers the potential to vastly accelerate discovery and innovation across a multitude of disciplines. For example, the exponential growth in data volume now presents an obstacle for high-throughput data mining in fields such as neuroinformatics and bioinformatics. As such, we present a characterization of a MapReduce-based data-mining application on a general-purpose GPU (GPGPU). Using neuroscience as the application vehicle, the results of our multi-dimensional performance evaluation show that a "one-size-fits-all" approach maps poorly across different GPGPU cards. Rather, a high-performance implementation on the GPGPU should factor in the 1) problem size, 2) type of GPU, 3) type of algorithm, and 4) data-access method when determining the type and level of parallelism. To guide the GPGPU programmer towards optimal performance within such a broad design space, we provide eight general performance characterizations of our data-mining application.

1 Introduction

There is a growing trend in scientific computing towards the use of accelerators to the reduce time to discovery. Unlike current general-purpose multicore CPU architectures, these accelerators combine hundreds of simplified processing cores executing in parallel to achieve the high-end performance demanded by scientists. Current examples of accelerators include the Cell Broadband Engine (Cell BE), field-programmable gate arrays (FPGAs), and general-purpose graphics processing units (GPGPUs) such as the NVIDIA CUDA-enabled GPU and AMD/ATI Brook+ GPU. Furthermore, upcoming technologies like AMD Fusion and Intel Larrabee point toward a future of accelerator-based computing platforms.

Current top-of-the-line GPGPUs possess hundreds of processing cores and memory bandwidth that is 10 times higher than conventional CPUs. The significant increase in parallelism within a GPGPU and the accompanying increase in performance has been successfully exploited for scientific, database, geometric, and imaging applications, which in many cases, has resulted in an order-of-magnitude performance improvement over top-of-the-line CPUs. GPGPUs also provide many other tangible benefits such as improved performance per dollar and performance per watt over conventional CPUs. Combining high performance, lower cost, and increasingly usable tool chains, GPGPUs are becoming increasingly more programmable and capable of solving a wider range of applications than simply three-dimensional triangle rasterization, and as such, is the target platform for our temporal data mining research.

Although temporal data mining is a relatively new area of data mining [15], this technique is becoming increasingly more important and being widely used in various application fields, such as telecommunication control [7], earth science [16], financial data prediction [11], medical data analysis [6], and neuroscience [19]. For example, neuroscientists would like to identify how the neurons in the brain are connected and related to each other. This usually involves stimulating one area of the brain and observing which other areas of the brain "light up." Recent technological advances

*This research was supported in part by an NVIDIA Professor Partnership Award.

in electrophysiology and imaging now allow neuroscientists to capture the timing of hundreds of neurons [14, 17]. However, the amount of data captured results in "data overload" and requires powerful computational resources. Current technology, like GMiner [18], is a step in the right direction, but limited to a single CPU running a Java virtual machine (VM), forcing output to be processed post-mortem.

Neuroscientists desire real-time, interactive visualization of the effects of their probes. This capability would open up an entirely new window of opportunities whereby patients can be tested, diagnosed, and operated upon in a single, *faster* procedure. We believe that GPGPUs can provide the performance necessary, and in this paper, characterize a temporal data-mining application in a multi-dimensional environment. Specifically, we evaluate its performance across the following five dimensions: 1) type of parallel algorithm, 2) data accesses, 3) problem size, 4) generation of GPGPU, and 5) the number of threads on the performance of the system.

Our results show that GPGPUs can provide the requisite performance, but that a "one-size-fits-all" approach is unsuitable for temporal data mining on graphics processors. Instead, the problem size and graphics processor determine which type of algorithm, data-access pattern, and number of threads should be used to achieve the desired performance. This result both corroborates and contrasts with previous similar MapReduce algorithms on graphics processors. However, while previous work only provided results in an optimal configuration, this paper presents general characterizations to help explain *how* a MapReduce-based, data-mining application should harness the parallelism in graphics processors.

To this end, we first present a background of current CUDA GPU technology and related MapReduce implementations in Section 2 followed by a detailed description of our temporal data-mining algorithm in Section 3. Section 4 lists relevant features of our testbed on which all of the tests were conducted with the results and characterizations presented in Section 5. Lastly, we offer some directions for future work and conclusions in Sections 6 and 7, respectively.

2 Background and Related Work

2.1 Graphics Processors

GPUs have been used for many years to accelerate the rendering of computer graphics. Fed by the increasing demand for improved 3-D graphics at higher framerates and larger resolutions, GPUs diverged from standard CPUs into exotic specialized architectures. In recent years, GPUs are transitioning away from being single-purpose devices into a more general-purpose architecture, capable of doing more than computing pixel values. This transition opens the doors for a range of applications to be accelerated. We describe here the architectural and programmatic features of state-of-the-art NVIDIA GPUs.

2.1.1 CUDA Architecture

While the internal organization of an NVIDIA GPU is a closely guarded secret, the state-of-the-art NVIDIA GPUs present to the programmer a Compute Unified Device Architecture (CUDA), as shown in Figure 1.

Core organization: The execution units of a CUDA-capable device are organized into multiprocessors, where each multiprocessor contains 8 scalar processor execution cores. The multiprocessor architecture is called SIMT (Single Instruction, Multiple Thread) and executes in a similar manner as a SIMD (Single Instruction, Multiple Data) architecture. While optimal performance is attained when groups of 32 threads, i.e. a "warp", follow the same execution path, individual threads can diverge along different thread paths while maintaining correctness. When divergence occurs within a warp, every instruction of every thread path is executed, with threads enabled or disabled depending on whether the thread is executing that particular thread path or not. A single instruction is completed by the entire warp in 4 cycles.

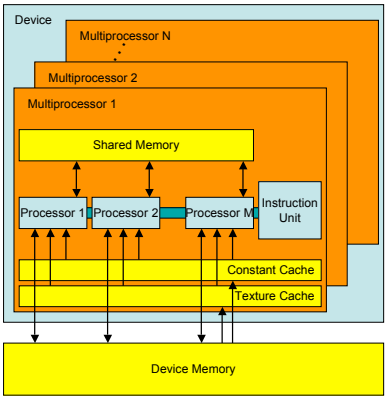


Figure 1: Overview of the NVIDIA Compute Unified Device Architecture (CUDA) [2]

Memory hierarchy: There are several types of memory accessible by an execution core on an NVIDIA GPU. Located on-chip, each multiprocessor contains a set of 32-bit registers along with a shared memory region which is quickly accessible by any core on the multiprocessor, but hidden from other multiprocessors. The exact number of registers available and size of the shared memory depend on the compute capability (i.e., "generation") of the GPU. In addition to shared memory, a multiprocessor contains two read-only caches, one for textures and another for constants, to improve memory access to texture and constant memory, respectively. Local and global memory are located in the device memory which is located off-chip and furthest from the execution cores (excluding the host machine's main memory). It may seem odd that local memory is not in fact local, but logically it serves the same purpose in that it is an overflow space for what will not fit in registers.

2.1.2 CUDA Programming Model

To enable the developer to harness the computing power of their GPUs, NVIDIA extended the C programming language to allow developers to easily offload computationally intensive kernels to the GPU for accelerated performance. This new programming model is commonly referred to as the CUDA programming model.

Parallelism: When a kernel executes, it is executed simultaneously by N threads in parallel. The programmer can logically arrange the N threads into one-, two-, or three-dimensional blocks. The index of a thread within this block and the ID of the thread have a one-to-one mapping to simplify identification. While threads within a thread block can share data within the same address space, each thread block has its own address space. This arrangement allows for synchronization between threads but not between thread blocks. To increase the amount of parallelism further, M "equally-shaped" thread blocks can be executed in parallel increasing the total amount of available parallelism to $M * N$.

Scheduling: As mentioned above, groups of 32 threads form a warp with multiple warps composing a thread block and multiple thread blocks forming a kernel. When a kernel is executed, the thread blocks are placed on different multiprocessors according to available execution capacity [2]. All of the threads (and by association, warps) of one thread block will execute on the same multiprocessor. A multiprocessor schedules warps with 0-cycle overhead and executes a single instruction of the warp in 4 cycles. Programmer-controlled placement and scheduling of the warps and thread blocks on the hardware is not currently available. As we will show in Section 5, this small, but rich feature could have a huge impact on the realizable performance of applications running on CUDA.

2.2 MapReduce

MapReduce is a programming model developed by Google to provide a convenient means for programmers to process large data sets on large parallel machines [8]. Moreover, the programmers that utilize a MapReduce framework do not need to have prior experience using parallel systems. While there has been debate on exactly how applicable this programming model is [9,10], the ability to process large data sets in parallel is an important requirement for real-time data-mining.

General Algorithm: The general MapReduce algorithm leverages two functional programming primitives, *map* and *reduce* in sequence. First, the *map* function is applied to a set of inputs consisting of a key/value pair to create a set of intermediate key/value pairs. Then, the *reduce* function is then applied to all intermediate key/value pairs containing the same intermediate key to produce a set of outputs. Due to the functional nature of both *map* and *reduce*, each phase can be executed in parallel in order to utilize the available resources in large data centers. A high-level view of the parallelism available in the algorithm is shown in Figure 2.

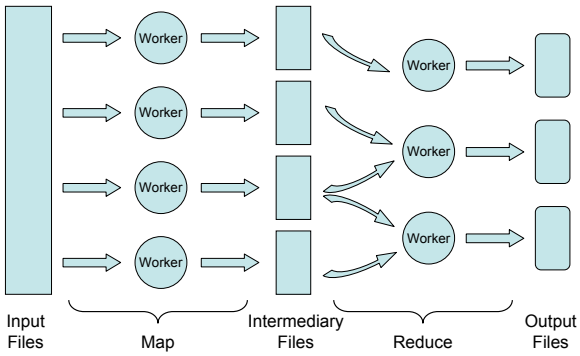


Figure 2: High-level view of the parallelism available in a MapReduce programming model

Implementations: The original implementation of MapReduce was built and optimized by Google to run on its private data centers. Providing the same functionality to the public, Hadoop is an open-source Java implementation that runs on everyday clusters and is under active development [1]. Specialized variations of the MapReduce framework also exist for multicore processors [5], the Cell processor [13], and graphics processors [4,12,20]. However, obtaining high performance within these frameworks is difficult ("... the cost of the Map and the Reduce function is unknown, it is difficult to find the optimal setting for the thread configuration." [12]), and often left to the user ("... performance can be strongly affected by the number of registers ... amount of local memory ... number of threads ... algorithm ... among other factors. ... We allow the programmer to expose these choices through template parameters... " [4]).

3 Temporal Data Mining

3.1 Overview

Association rule mining is a common data-mining technique used to discover how subsets of items relate to the presence of other subsets. Temporal data mining is a restricted variation of association rule mining where temporal relations between items will also be considered.

A prototypical example of temporal data mining can be found in "market-basket analysis", where a store might want to know how often a customer buys product B given that product A was purchased earlier. In other words, the store

wishes to know how often $\{peanut\ butter, bread\} \longrightarrow \{jelly\}$. We note that unlike association rule mining, temporal data mining differentiates $\{bread, peanut\ butter\} \longrightarrow \{jelly\}$ from $\{peanut\ butter, bread\} \longrightarrow \{jelly\}$.

In this paper, we focus on one specific area of temporal data mining, called frequent episode mining, where we want to find frequently appeared sequences of items (episodes) in a time ordered database [3]. We define frequent episode mining as follows.

Let $D = \{d_1, d_2, ..., d_n\}$ be an ordered database of items where d_i is a member of the alphabet $I = \{i_1, i_2, ..., i_m\}$. An episode, A_j, is a sequence of k items $< i_{j_1}, i_{j_2}, \ldots, i_{j_k} >$. There is an appearance of the episode A_j in database D, if there exists a sequence of indices $< r_1, r_2, \ldots, r_k >$ in increasing order that $i_{j_1} = d_{r_1}, i_{j_2} = d_{r_2}, \ldots, i_{j_k} = d_{r_k}$. The count of a episode, $count(A_j)$, is the total number of appearance of A_j in D. The task of frequent episode mining is to find all the episodes, A_j, for which the $count(A_j)/n$ is greater than a given threshold α.

The standard algorithm for frequent episode mining can be described as below.

Algorithm 1 *Frequent Episode Mining(D,α).*

Input: supporting threshold α, sequential database $D = \{d_1, d_2, ..., d_n\}$
Output: frequent episode set $A = A_1, A_2, \ldots A_m$

1: $k \leftarrow 1$, $S \leftarrow \emptyset$
2: level $k \leftarrow 1$, $A'_k \leftarrow \{\{i_1\}, \{i_2\}, \ldots \{i_m\}\}$, (generate candidate episode for level 1)
3: **while** $A' \neq \emptyset$ **do**
4: Calculate $count(A'_{k_j})$ for all episodes A'_{k_j} in A'_k (**counting step**)
5: Eliminate all non-frequent episodes, $count(A'_j)/n \leq \alpha$, from A'_k (**elimination step**)
6: $S_A \leftarrow S_A \cup A'_k$
7: $A \leftarrow A + A'$; $k \leftarrow k + 1$
8: Generate candidate episode set A'_k from A'_{k-1} (**generation step**)
9: **end while**
10: return S_A

Algorithm 1 generates candidate episode set for each level (the length of an episode), counts the number of appearances for all the candidate episodes, eliminates non-frequent ones, and generate the candidate episode set for the next level.

While the elimination and generation steps takes care to only include useful subsets, the computational demands of the counting step can *potentially* grow exponentially as a function of the size of a subset A_j and the alphabet I, as shown in Table 1. One method to decreasing the runtime is to limit the length of A_j from n to q at the cost of losing information for all A_j with length greater than q.

Episode Length	1	2	3	4	...	L
Episodes	N	$N(N-1)$	$N(N-1)(N-2)$	$N(N-1)(N-2)(N-3)$	...	$\frac{L!}{(N-L)!L}$

Table 1: Potential number of episodes with length L from an alphabet I of size N

Alternatively, the runtime can be decreased through advanced algorithms and/or distributing the computational demand among multiple processors [21]. With the recent advances of graphics processors to more general computing platforms with large I/O bandwidth, parallel association rule mining on these platforms appears to be a natural fit. While several data mining algorithms exist on the GPU, to the best of our knowledge, this paper presents the first temporal data mining algorithm ported to a GPU.

3.2 Core Algorithm

The core algorithm to discover the presence of an episode $A_j =< a_1, a_2, ..., a_L >$ can simply be implemented as a finite state machine as shown in Figure 3. Each item a_i in the episode is a state with transitions to a_{i+1}, a_1, and to itself

depending on the value of the next character c. Because we are counting *all* episodes present in the database, when the *final* state is reached, a counter keeping track of the total number of episodes found is incremented and the finite state machine is reset back to *start* where the process repeats until all of the characters in the database are compared.

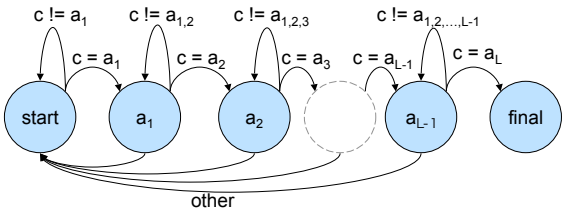


Figure 3: Finite state machine to discover the presence of an episode

3.3 Parallelism on CUDA

Due to the wide range of parallelism available and the inability of a programmer to explicitly control the scheduling and placement of thread blocks on the multiprocessors in an NVIDIA GPU, we implemented four algorithms using the CUDA programming model. These four algorithms can be generally classified as the cartesian product of (1) thread-level parallelism or block-level parallelism, and (2) with or without local buffering following a MapReduce programming model. The algorithms are shown graphically in Figure 4.

3.3.1 MapReduce

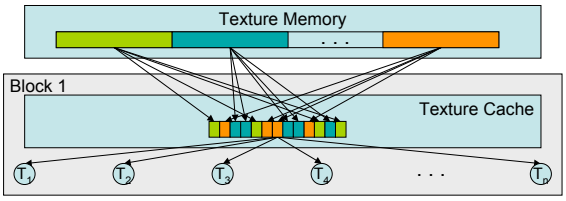
At a high-level, our algorithms follow a MapReduce programming model to achieve efficient parallelism. Using the observation that counting for the number of appearance of A_k is independent from counting for A_l, the *map* function returns the number of appearance for A_j for a given A_j in the database D. The *reduce* function is dependent on the whether thread or block parallelism is used as described below.

3.3.2 Thread-Level Parallelism

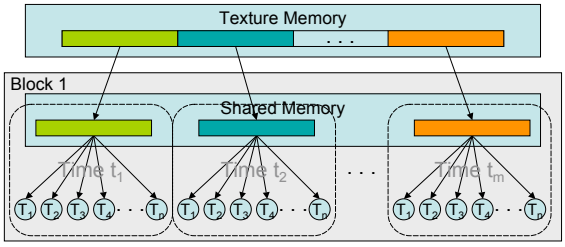
Our first two algorithms implement strict thread-level parallelism to assign one thread to search for one episode, A_j, in the database D. Using one thread to search for one episode causes the *reduce* function to be an identity function simply outputting the value produced by the *map* function.

Algorithm 1: without Buffering Since each thread will scan the entire database, our first algorithm places this database in the read-only texture memory such that each thread will be able to utilize the high bandwidth available on the graphics processor. With each thread starting from the same point in the database, the spatial and temporal locality of the data-access pattern should be able to be exploited by the texture cache on each multiprocessor. Furthermore, threads are assigned to thread blocks in order until the maximum number of threads per thread block is reached. For example, if the maximum number of threads per thread block is 512, then threads 1-512 are assigned to the thread block 1, threads 513-1024 to the second thread block, and so on, until no threads are left.

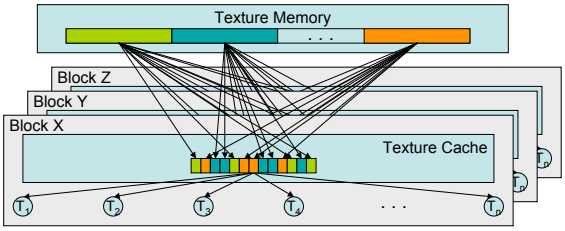
Algorithm 2: with Buffering Our second algorithm also uses thread-level parallelism, but instead of using texture memory, this algorithm buffers portions of the database in shared memory in order to minimize the thread contention for the memory bus. Following this approach, a thread copies a block of data to a shared memory buffer, processes the data in the buffer, then copies the next block of data to the memory buffer, and so on, until the entire database is processed. The scheduling of threads to thread blocks follows the same approach in Algorithm 1.



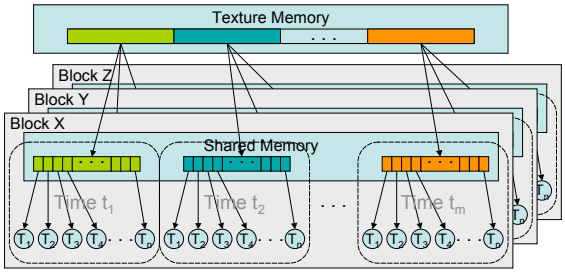
(a) Algorithm 1: each thread searches for a unique episode using data stored in texture memory



(b) Algorithm 2: each thread searches for a unique episode but uses shared memory to buffer the data first



(c) Algorithm 3: threads within a block search for the same episode with different blocks searching for different episodes using data in texture memory



(d) Algorithm 4: threads within a block search for the same episode with different blocks searching for different episodes but uses shared memory to buffer the data first

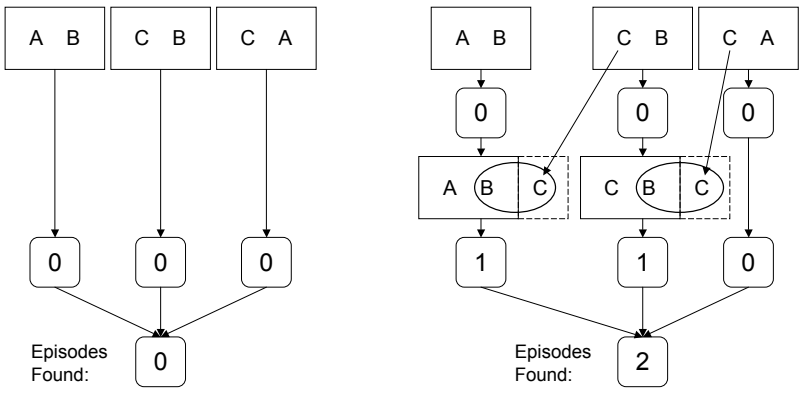
Figure 4: Available Parallelism for each of the four algorithms

3.3.3 Block-Level Parallelism

At a higher level of parallel abstraction, the CUDA programming model enables parallelism at the block level. At this level, our two block-level algorithms assign one block to search for one episode. Within a block, the threads collaborate to perform the search by having each thread search a portion of the database. With multiple threads searching for the same episode, the *reduce* function cannot be the identity function. Instead, the *reduce* function sums the counts from each thread. However, since an episode might span across threads, an intermediate step to check for this possibility occurs between the *map* and *reduce* functions. An example of an episode spanning threads is shown in Figure 5.

Algorithm 3: without Buffering Similarly to Algorithm 1, we implement this version of our block level parallel algorithm without buffering of the database. Instead, the threads within each block access the data through texture memory. However, unlike Algorithm 1, each of the t threads within a block start at a different offset in the database while threads with the same ID in *different* blocks start at the same offset. The total number of threads available using Algorithm 3 is $t * e$ where e is the number of episodes to be searched for.

Algorithm 4: with Buffering The final algorithm we discuss in this paper uses block level parallelism with buffering of the database to shared memory. The starting offset for each thread in Algorithm 4 is relative to the buffer size and *not* the database size as in Algorithm 3. Therefore, thread T_i will always access the exact same block of shared memory addresses for the entire search – the data at those addresses will change as the buffer is updated. The total number of available threads is identical to Algorithm 3.



(a) Searching for $B \Longrightarrow C$ without taking spanning into account

(b) Searching for $B \Longrightarrow C$ while taking spanning into account

Figure 5: Example scenario where searching for $B \Longrightarrow C$ results in a different number of episodes found depending on whether spanning is taken into account or not

Graphics Card	GeForce 8800 GTS 512	GeForce 9800 GX2	GeForce GTX 280
GPU	G92	2xG92	GT280
Memory (MB)	512	2x512	1024
Memory Bandwidth (GBps)	57.6	2x64	141.7
Multiprocessors	16	16	30
Cores	128	128	240
Processor Clock (MHz)	1625	1500	1296
Compute Capability	1.1	1.1	1.3
Registers per Multiprocessor	8196	8196	16384
Threads per Block (Max)	512	512	512
Active Threads per Multiprocessor (Max)	768	768	1024
Active Blocks per Multiprocessor (Max)	8	8	8
Active Warps per Multiprocessor (Max)	24	24	32

Table 2: Summary of the architectural features of GeForce 8800 GTS 512, GeForce 9800 GX2, and GeForce GTX 280 NVIDIA graphics cards.

4 Experimental Testbed

To analyze the performance characteristics of MapReduce on graphics processors, we performed the same series of tests on three variations of NVIDIA graphics cards representing recent and current technology. An overview of the architectural features are shown in Table 2. We present here a more detailed description of each card along with the host machine in which the tests were performed. A complete list of the specifications for the different compute capability levels of the GPUs found in these cards can be found in Appendix A of [2].

4.1 Host

The host machine used for these tests consists of an E4500 Intel Core2 Duo operating at 2.2 GHz with 4 GB (2x2GB) of 200 MHz DDR2 SDRAM (DDR2-800). The operating system in use is the 64-bit version of the Ubuntu GNU/Linux 7.04 (Feisty Fawn) distribution running the 2.6.20-16-generic Linux kernel as distributed through the package management system. Programming and access to the GPUs used the CUDA 2.0 toolkit and SDK with the NVIDIA driver

version 177.67. Furthermore, all processes related to the graphical user interface (GUI) were disabled to limit external traffic to the GPU.

4.2 Cards

4.2.1 NVIDIA GeForce 8800 GTS 512 with NVIDIA G92 GPU

To evaluate a recent generation of the NVIDIA CUDA technology, we ran our tests on an GeForce 8800 GTS 512 graphics card with NVIDIA G92 GPU and 512 MB of onboard GDDR3 RAM. NVIDIA lists the G92 GPU as having compute capability 1.1, where compute capability determines the features and specifications of the hardware. Informally speaking, different compute capabilities signify different hardware generations. The GeForce 8800 GTS 512 contains sixteen multiprocessors with each multiprocessor containing eight 1625 MHz execution cores, 8196 registers, and 16 KB of shared memory. The warp size is 32 threads with warps scheduled in intervals of four cycles. There can be at most 512 threads per block with 768 active threads per multiprocessor implying that two blocks of 512 threads can *not* be active simultaneously on the same multiprocessor. Furthermore, there can be at most eight active blocks and twenty-four active warps per multiprocessor. The texture cache working set is between six and eight KB per multiprocessor.

Beginning with compute capability 1.1, the GPU supports atomic operations between threads on 32-bit words in shared or global memory allowing programmers to write thread-safe programs. It is worth recalling, however, that this improvement does not allow for threads in different blocks to synchronize as each block is independent of other blocks.

4.2.2 NVIDIA GeForce 9800 GX2 with NVIDIA G92 GPU

We also evaluated an NVIDIA GeForce 9800GX2 which contains *two* NVIDIA G92 GPUs and *two* units of 512MB of GDDR3 RAM. Essentially, the 9800GX2 is two 8800 GTS 512 cards merged onto a single graphics card with the execution cores running at 1500 MHz instead of 1500 MHz as in the 8800 GTS 512. Additionally, the 9800 GX2 has a modest 10% increase in memory bandwidth over the 8800 GTS 512 (64 GBps versus 57.6 GBps).

4.2.3 NVIDIA GeForce GTX 280 with NVIDIA GT200 GPU

The most recent generation of CUDA technology has compute capability 1.3. For our tests we used an GTX 280 graphics card with GT200 GPU. With 1024 MBs of GDDR3 RAM and 30 multiprocessors, this card has the largest amount of device memory, number of processing cores (240), and memory bandwidth (141.7 GBps) of the cards tested. Furthermore, this GPU has 100% more registers per multiprocessor (16384), 33% more active warps (32), and 25% more active threads (1024) than the G92 GPUs in addition to supporting double-precision floating-point numbers.

5 Results

We present here several performance characterizations of our algorithms running on different graphics cards at different episode levels with varying numbers of threads per block. At episode level L, an algorithm is searching for an episode of length L where $\{I_1, I_2, ..., I_{L-1} \Longrightarrow I_L\}$. In the results presented, $L \in \{1, 2, 3\}$, I_l a member of the set of upper-case letters in the English alphabet (i.e., $I_l \in \{A, B, ..., Z\}$), and the database contains $393,019$ letters. In our experiments, level 1 contained 26 episodes, level 2 contained 650 episodes, and level 3 contained 15,600 episodes.

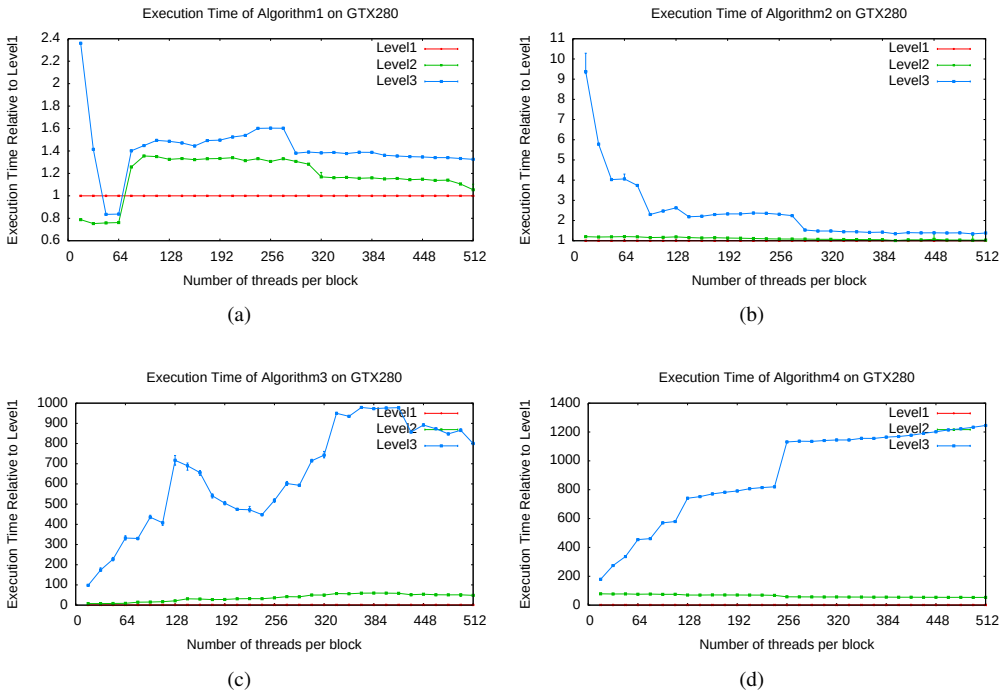


Figure 6: Impact of Problem Size on the GTX280 on Different Algorithms

A single test consists of selecting a single episode level, algorithm, card, and block size with the execution time counted as the amount of time between the moment the kernel is invoked, to the moment that it returns. Although we restricted access to the GPU by disabling all non-vital graphical services, to minimize the effect of errant GPU calls, each test was performed ten times with the average used to represent the test time. However, the minimum and maximum execution times are also displayed to show the range of times that the execution can take, which in some cases is quite large.

While the complete results from our tests are shown in Appendix A, we detail below some characterizations from these tests with respect to three higher-level criteria – level, algorithm, and card – and their impact on execution time. We also note that because some of the low-level architectural information of the NVIDIA GPUs is unavailable to the public, that the characterizations presented are *general* in nature; we discuss plans to uncover some of these low-level architectural features in Section 6.

5.1 Impact of Level on Execution Time

To understand the impact of the problem size on execution time, we performed a series of tests where the hardware and algorithm remained constant, but the level, L, varied. Because the number of episodes to search for increases exponentially as a function of L as shown earlier in Table 1, the scalability of an algorithm with respect to problem size is important.

5.1.1 Characterization 1: Thread Parallel Algorithm has O(C) Time Complexity Per Episode

Algorithm 1 and 2 are constant time algorithms per episode. Whether performing 26, 1560, or 25,320 searches, the amount of time to complete each individual search is essentially the same. Since a search for each episode is com-

pletely independent of other searches, and each search is assigned to one thread, there is no added complexity needed during the reduce phase to identify episodes that span thread boundaries. In addition, the search is based on a simple state machine the complexity of searching for a single episode in a single dataset stays constant regardless of the level. Therefore, the entire execution time can be spent performing the *map* function across the entire database. We explain in Section 5.2 whether Algorithm 1 or Algorithm 2 has an overall faster execution time for various levels.

Since these algorithms are constant time per search, we actually find that they are effectively constant time for up to a rather large number of searches when executed on the GPU as can be seen in Figures 6(a) and 6(b). By this we mean that 26, 650, or even several thousand searches complete in the same amount of time on the GPU, reasons for this will become more clear in Characterizations 4 and 7.

5.1.2 Characterization 2: Buffering Penalty in Thread Parallel Can be Amortized

Algorithm 2 uses buffering to combine the memory bandwidth of all threads in a block and reduce contention on texture memory. This does not however, come without a cost. The initial load time is high, and since only one block may be resident on a multiprocessor during this time, no computation is done during the load. As more threads are added to a block Algorithm 2 exponentially decreases in execution time as shown in Figure 6(b). This characteristic shows that Algorithm 2 is able to make use of the processing power of a greater number of threads, largely thanks to the fact that the load time is either equal to or lower than the time taken for lower numbers of threads. Since that cost stays constant or drops slightly, and all threads can access the resulting shared memory block without additional cost, the more threads one has, the faster results will be calculated up to the point where scheduling overhead on the multiprocessor overwhelms the computation time.

5.1.3 Characterization 3: Block Parallel Does Not Scale with Block Size

Unlike Algorithm 2, Algorithms 3 and 4 actually lose performance per episode as the number of threads per block and level increase. Figures 6(c) and 6(d) show a general trend of larger execution times as the number of threads increase with Algorithm 4 at an almost constant slope when solving the problem size at Level 3. Furthermore, the change in execution time between Level 1 and Level 2 and between Level 2 and Level 3 is also increasing. These two trends are due to the extra complexity of finding episodes that span thread boundaries and the cost of loading more blocks than can be active on the card simultaneously.

As the number of threads increases, the number of boundaries increases. As the level increases, the likelihood that an episode spans the boundary between threads increases. With the number of boundaries increasing and the probability that an episode will span a boundary increasing as well, the necessary computation needed to be performed after the *map* function and before the *reduce* function and therefore longer overall execution times increase to match.

5.2 Impact of Algorithm on Execution Time

While the scalability of an algorithm with respect to problem size is important, it is often the case that a user wishes to examine a problem of specific size and only has access to one type of hardware. That is, a user wants to solve the same problem on the same card and can only vary the algorithm and number of threads to use for that algorithm. In this case, the user would want to use the fastest algorithm for the specific problem. Our characterizations below are limited to the GTX280 as it has the highest compute capability of the cards tested.

5.2.1 Characterization 4: Thread Level Alone not Sufficient for Small Problem Sizes ($L = 1$)

When evaluating small problem sizes, low levels, there are not enough episodes to generate enough threads to utilize the resources of a card. Add parallelism through blocks and then throw in threads on top. See Figure 7(a). Since

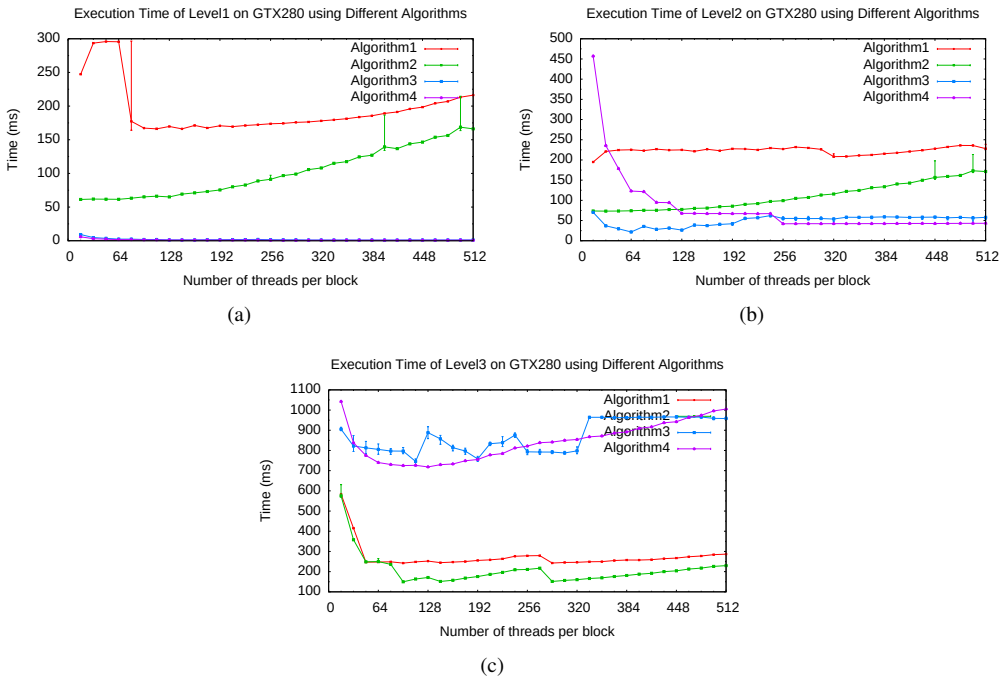


Figure 7: Impact of Algorithm on the GTX280 at different problem sizes

episodes are fixed and there is 1 thread per episode, adding more than 26 threads to Algorithm 1 or Algorithm 2, contributes nothing but contention which is why Algorithms 1 and 2 have an uptrend as threads increase. Algorithms 3 and 4, on the other hand, are orders of magnitude faster as they create 26 blocks and add threads to help search for same episode. So they display a decreasing trend but plateau because there is still not enough work.

Algorithm 4 on the GTX280 is sub-millisecond making true real-time data mining possible.

5.2.2 Characterization 5: Block Level Depends on Block Size for Medium Problem Sizes ($L = 2$)

Figure 7(b). With a medium problem size, there is enough parallelism at thread level to outperform block level at low numbers of threads (16 on GTS512) and therefore a reasonable number of blocks. That is Algorithms 1 and 2 decrease the number of blocks as the number of threads per block increases because there are a fixed number of episodes, and thus a fixed number of threads, so the number of blocks changes with the number of threads per block. At level 2, the blocks will vary as a function of threads per block starting with 650/16 and decreasing to 650/512, unlike at level 1 where a value of threads per block greater than 16 will only result in 1 block.

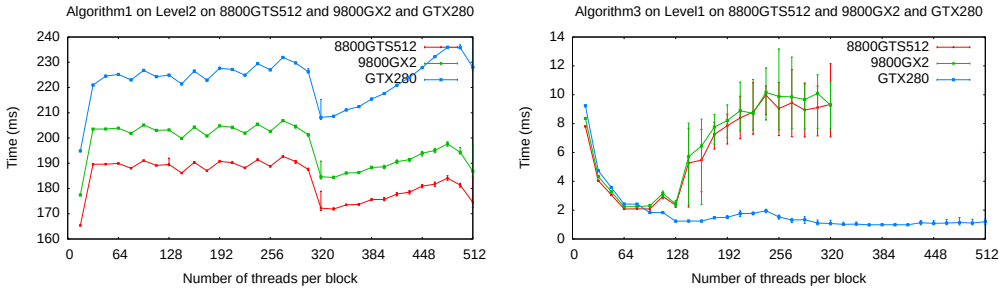
Eventually, Algorithm 4 outperforms Algorithm 3 at a specific thread level (240), but never achieves the *best* execution time which is Algorithm 3 at 64 threads. An explanation of this is hard to pinpoint exactly as the internal workings and scheduling of the GTX280 are not publicly available. However, we believe that Algorithm 3, using texture memory and heavy caching will see optimal bandwidth reached at low numbers of threads and contention since the contention will only increase as more threads are added. The buffering to local memory is a one time penalty which, as we mentioned in Characterization 2, is amortized over the large number of threads accessing the shared memory in read-only fashion.

5.2.3 Characterization 6: Thread Level Parallelism is Sufficient for Large Problem Sizes ($L = 3$)

The GTX280 has 30 multiprocessors with a maximum of 1024 active threads per multiprocessor for a total of $30,720$ active threads available. When $L = 3$, there are $25,230$ episodes to be searched. As seen in Figure 7(c), the thread level parallel algorithms (Algorithm 1 and 2) are significantly faster than the block level algorithms (Algorithms 3 and 4). This performance difference can be attributed to the fact that Algorithms 1 and 2 can have more episodes being searched simultaneously than Algorithms 3 and 4 for a given number of threads per block. Algorithms 3 and 4 are limited to 240 episodes being searched due to the limitation of 8 active blocks on each of the 30 multiprocessors in the GTX280 and each block searching for a single episode. Algorithms 1 and 2 on the other hand, can have up to $240 * t$ active episodes as each *thread* within a block will search for a unique episode. The actual number of active episodes for Algorithms 1 and 2 are determined by the resources that each thread consumes and the available resources on each multiprocessor.

5.3 Impact of Card on Execution Time

The other major decision which can affect performance is the hardware itself on which the algorithm will run. Some users may have a variety of hardware and wish to know which will return results the fastest, or still others may wish to determine the optimal card for their problem when considering a new purchase. The characterizations below showcase two of the major factors which come into play in determining the right card for the job.



(a) (b)

Figure 8: Impact of Card

5.3.1 Characterization 7: Thread Level Dependent on Shader Frequency for Small to Medium Problems

Algorithms 1 and 2 are greatly dependent on the Shader Frequency for small and medium problem sizes, and scale essentially linearly by that measure as one can see in Figure 8(a). The frequencies of the three cards tested are 1625 MHz , 1500 MHz, and 1296 Mhz the frequencies matching the range of times very closely. This trend is due to the fact that levels 1 and 2 are incapable of filling enough multiprocessors for the number of processors or contention to become a determining factor in performance. In the appendix one can see the same test at level 3 produces a very different picture, where the 30 core 280 GTX outperforms the 16 cored 9800GX2 and the 8800GTS512 for nearly all thread counts.

5.3.2 Characterization 8: Block Level Algorithms Affected by Memory Bandwidth

Algorithm 3 can be greatly effected by Memory Contention when scaled to large problem sets. Total threads accessing memory is "active episodes (or active blocks)" times threads per block. Given the massive number of blocks, the

algorithm holds a very high thread count per multiprocessor over a long period, and produces a great amount of memory contention. The two cards which perform slower, actually have less processors to contend for memory, but have memory bandwidth in the range of 62 to 64 GBps, whereas the GTX 280 has a far greater 141GBps producing the results seen in Figure 8(b).

6 Future Work

Although we offer eight characterizations, there is still some work to be done as real-time, data-mining analysis is the overall goal. To this end, we plan to study the effects of larger episodes (e.g., $L >> 3$) and its effect on the constant-time, thread-level algorithms as well as the effect of pipelining multiple phases of the overall algorithm together as searching for candidates of episode length 3 can proceed while episode lengths of 2 and 4 are also computed.

We are also looking at the effect of feature changes on the algorithm execution time. One feature is episode expiration where $A \Rightarrow B$ iff $B.time() - A.time() < Threshold$. Currently, there is no expiration on the episodes which makes spanning boundaries likely. With episode expiration, we expect the *reduce* phase in Algorithms 3 and 4 will be decreased as less episodes will span boundaries.

In addition, though we characterized the performance of an algorithm solving a specific problem on a certain card, it is difficult to conclude exactly how many threads per thread block should be used. That is, while being able to identify general performance, we cannot identify optimal performance which is important to neuroscientists hoping for the possibility of real-time data mining. We plan to pursue a series of micro-benchmarks to discover the underlying hardware and architectural features such as scheduling, caching, and memory allocation. We are aware of the CUDA Occupancy Calculator, but it is insufficient in identifying optimal performance as it only shows the utilization of a given multiprocessor. Unfortunately, 30 multiprocessors of occupancy 66% might perform better than 15 multiprocessors at 100% occupancy.

7 Conclusion

The ability to mine data in real-time will enable scientists to perform research in ways that have only been dreamed about. However, as the sheer volume of data grows, the algorithms used to mine this data need to keep pace or the utility of the data is lost. This is unacceptable in the field of neuroscience, where the inability to understand the data in real-time can have fatal consequences.

One approach to this problem of "data overload" is to create new parallel algorithms capable of extracting the computational performance available on accelerators. In this paper, we have developed and characterized the performance of a parallel temporal data mining application on NVIDIA CUDA graphics processors.

As one might expect, the best execution time for large problem sizes always occurs on the newest generation of the hardware, the NVIDIA GeForce GTX 280 graphics card. What is surprising however, is that the oldest card we tested was consistently the fastest for small problem sizes. Beyond that, the extent to which the type and level of parallelism, episode length, and algorithm impact the performance of frequent episode mining on graphics processors is surprising as well. Furthermore, as our results showed, a MapReduce-based implementation must *dynamically* adapt the type and level of parallelism in order to obtain the best performance. For example, when searching for episodes of length 1, an algorithm using blocks with 256 threads and buffering to shared memory achieves the best performance. However, episodes of length 2 require block sizes of 64 *without* buffering, and episodes of length 3 should use 96 threads per block with every thread searching for a unique episode.

Lastly, due to the ever-increasing volume of data and demand for high performance in neuroscience and bioinformatics, we have provided 8 performance characterizations as a guide for programming temporal data mining applications on GPGPUs.

References

[1] Hadoop: Open source implementation of MapReduce, 2008.

[2] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide, Version 2.0*, 2008.

[3] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993.

[4] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A Map Reduce Framework for Programming Graphics Processors. In *Third Workshop on Software Tools for MultiCore Systems (STMCS 2008)*, 2008.

[5] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In *Advances in Neural Information Processing Systems (NIPS) 19*, 2006.

[6] Rajai El Dajani, Maryvonne Miquel, Marie-Claire Forlini, and Paul Rubel. Modeling of Ventricular Repolarisation Time Series by Multi-layer Perceptrons. In *AIME '01: Proceedings of the 8th Conference on AI in Medicine in Europe*, pages 152–155, London, UK, 2001. Springer-Verlag.

[7] Gautam Das, King ip Lin, Heikki Mannila, Gopal Renganathan, and Padhraic Smyth. Rule Discovery from Time Series. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, pages 16–22. AAAI Press, 1998.

[8] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04*, 2004.

[9] David J DeWitt and Michael Stonebraker. MapReduce: A major step backwards, 2008.

[10] David J DeWitt and Michael Stonebraker. MapReduce II, 2008.

[11] Martin Gavrilov, Dragomir Anguelov, Piotr Indyk, and Rajeev Motwani. Mining the stock market (extended abstract): which measure is best? In *KDD '00: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 487–496, New York, NY, USA, 2000. ACM.

[12] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT08: IEEE International Conference on Parallel Architecture and Compilation Techniques 2008*, 2008.

[13] Marc De Kruijf. Mapreduce on the cell broadband engine architecture. Technical report, 2007.

[14] Debprakash Patnaik, P. S. Sastry, and K. P. Unnikrishnan. Inferring Neuronal Network Connectivity using Time-constrained Episodes. *CoRR*, abs/0709.0218, 2007.

[15] John F. Roddick and Myra Spiliopoulou. A Survey of Temporal Knowledge Discovery Paradigms and Methods. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):750–767, 2002.

[16] Michael Steinbach, Steven Klooster, Pang ning Tan, Christopher Potter, and Vipin Kumar. Temporal Data Mining for the Discovery and Analysis of Ocean Climate Indices. In *Proceedings of the KDD Temporal Data Mining Workshop*, pages 2–13, 2002.

[17] K. P. Unnikrishnan, Debprakash Patnaik, and P. S. Sastry. Discovering Patterns in Multi-neuronal Spike Trains using the Frequent Episode Method. *CoRR*, abs/0709.0566, 2007.

[18] K.P. Unnikrishnan, P.S. Sastry, and Naren Ramakrishnan. GMiner, http://neural-code.cs.vt.edu/gminer.html.

[19] Pablo Valenti, Enrique Cazamajou, Marcelo Scarpettini, Ariel Aizemberg, Walter Silva, and Silvia Kochen. Automatic detection of interictal spikes using data mining models. *Journal of Neuroscience Methods*, 150(1):105–110, 2006.

[20] Jackson H.C. Yeung, C.C. Tsang, K.H. Tsoi, Bill S.H. Kwan, Chris C.C. Cheung, Anthony P.C. Chan, and Philip H.W. Leong. Map-reduce as a Programming Model for Custom Computing Machines. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008.

[21] Mohammed J. Zaki. Parallel and Distributed Association Mining: A Survey. *IEEE Concurrency*, 7(4):14–25, 1999.

Appendix

A Complete Results

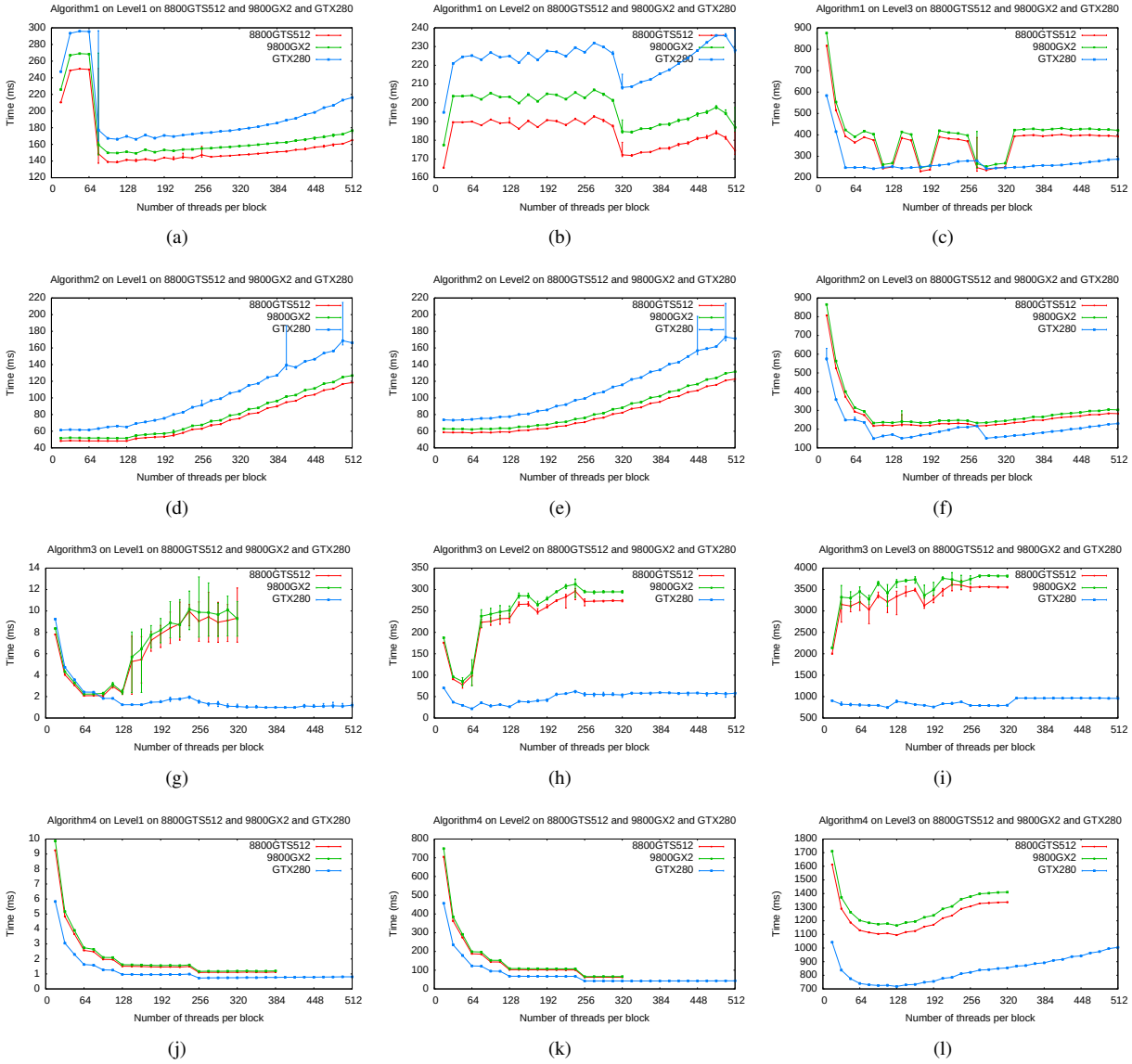


Figure 9: Overview of all of the tests