GPU Task Parallelism for Scalable Anomaly Detection

Koji Ueno
Tokyo Institute of Technology / JST CREST
ueno.k.ac@m.titech.ac.jp

Toyotaro Suzumura
Tokyo Institute of Technology
IBM Research – Tokyo / JST CREST
suzumura@cs.titech.ac.jp

ABSTRACT
Stream computing has emerged as a new processing paradigm that processes incoming data streams from large numbers of sensors in real time. At the same time, many recent efforts have shown the suitability of GPGPUs for batch-typed long-running applications. However, few studies conduct the applicability of GPGPU to stream computing and our first experiment shows that the performance does not scale as expected if one introduces the GPGPU to stream computing applications. This paper presents the workload characterization of GPGPU-based stream computing. We especially focus on computing SVD (Singular Value Decomposition) with GPGPUs for small-sized matrix since it can be widely applicable to real-time stream-based data mining applications such as real-time anomaly detection. In this paper, we not only show the workload characterization of GPGPU-based stream computing but also propose the optimization approach of SVD for stream computing applications called “GPU task Parallelism” to leverage the sufficient capability of GPGPUs. This optimization offers new levels of scalability for real-time operations with large numbers of sensors. Our experimental results show that GPU task parallelism provides roughly four-fold performance gains against a quad-core CPU for matrices of $300 \times 300$ to $500 \times 500$ data values. We also implemented a stream-based change-point and anomaly detection system based on the optimized SVD and stream computing system called System S. By porting the optimized version of SVD to the distributed stream computing system, it was easy to exploit multiple GPGUs on multiple nodes. The performance results showed performance around 7.6 times faster than CPUs. The scalability of our proposed system was tested up to 1,525 sensors, which were simultaneously handled for change-point detection every 5 seconds.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming

General Terms
Performance, Design, Measurement, Algorithm

Keywords
GPGPU, Task Parallelism, SVD, SST, Anomaly Detection, Stream Computing.

1. INTRODUCTION
We are entering an era in which the numbers of available sensors and their data are growing rapidly. Sensors can vary from physical sensors such as medical devices, image and video cameras, and RFID sensors to data generated by computer systems such as stock trading data or data from social media systems such as Twitter and SNS.

Stream computing (or data stream processing) is becoming a key computing paradigm for processing incoming data from such sources in real time. This computing paradigm processes a stream of data generated from sensors and stored in memory. By processing data in this way, real-time responses are possible, in contrast to traditional 3-phase batch processing where the incoming data is first received, then stored, and finally processed at some later time. Different applications have different performance requirements for latency and throughput, but recent industry application may require very low latency responses for the monitored stream data, as when performing latency-critical anomaly detection or algorithmic trading. The performance requirements of these applications could not be met by older computing paradigms such as stream processing. However, many kinds of middleware, such as IBM System S [20,21,22] and Yahoo S4 [26] have been developed for stream processing.

In addition, many recent research efforts have shown the huge performance benefits available from GPGPUs (General Purpose Graphics Processing Unit) and many of the top-ranked supercomputers in the most recent Top500 lists are using heterogeneous processors that combine CPUs and GPUs. Compared to CPUs, GPUs are usually high performance processors with large numbers of basic processing cores and lots of high-speed memory. Many prior reports [23] have shown how GPUs can accelerate those applications that can be implemented to effectively use the architectural characteristics of the GPUs. As a programming model for GPGPUs, NVIDIA’s CUDA [11] is widely used today.

In this paper we present the first approach that accelerates stream computing with GPGPUs by combining these two computing paradigms. We also show the workload characteristics and performance gains of the proposed optimization approach, which we call GPU Task Parallelism (GTP), in selected data stream applications. We integrated our GPU-based high performance data stream component into the data stream runtime of System S, allowing us to easily use multiple GPGPUs. Then we show the validity of our proposed method in two applications, SVD...
(Singular Value Decomposition) (which can be used for a wide variety of real-time data mining applications including anomaly detection) and SST (Singular Spectrum Transformation).

Here is the organization of our paper. In Section 2, we explain the need for more scalable stream computing that effectively exploits the capabilities of GPGPUs. In Section 3, we discuss possible methods for handling the multiple tasks required for stream computing applications while introducing GPU task parallelism. In Section 4, we describe how we implemented SST for the GPU task parallelism and assess the performance. In Section 5, we incorporate the task-parallel-ready SVD component into System S, a distributed stream computing system, and measure the performance with multiple GPUs in a cluster of nodes. We discuss our results in Section 6 and cover related work in Section 7. Concluding remarks and future directions appear in Section 8.

2. MOTIVATION – ACCELERATING STREAM COMPUTING APPLICATIONS WITH GPGPUS

In this section, we motivate our research by introducing a need for scalable stream processing platform, describes our idea of realizing such a scalable system with GPGPUs, and then points out some technical issues that need to be solved.

2.1 Motivating Scenario for Scalable System

Although stream computing paradigm can be used in a wide variety of fields, there are practical systems that require extremely high scalability and also real-time such as ITS (Intelligent Transportation System) [28], CDR (Call Detail Records) processing system for mobile phone systems, and so forth. In the case of ITS, there are millions of cars equipped with multiple sensors such as a GPS sensor, health monitoring sensors for various car parts, and so forth. Auto makers would provide more fine-grained support services to each customers (car drivers) by providing useful information such as certain anomaly behavior in the car engines. This kind of new services will appear and transform the traditional car support service in the near future since each driver needs to visit a car dealer when they notice unusual conditions at this moment. In order to realize such a new service, we need a scalable processing system that conducts the anomaly detection or change point detection in real time for those millions of sensors in a simultaneous fashion.

One possible architecture for realizing such a scalable system would be to divide millions of independent processing task into multiple compute nodes in a distributed system since each event stream can be treated as an independent task. This traditional approach is common and well studied. In our research we explore how GPGPUs can be employed to realize the above objective. Recently, GPGPUs is paid much attention to accelerate various applications from scientific simulations to data mining applications which mostly boils down to the problem of matrix computation for large-sized matrix. Since our target scenario needs to deal with more fine-grained and smaller sized computation, the way of how GPGPUs can be employed would be different from prior arts. As far as we investigated prior arts, no research effort has yet investigated how much the performance gain would be when compared to CPUs, and also how programmers need to implement such a system with GPGPUs.

2.2 Issues on Implementing Scalable Anomaly Detection with GPGPU

We initially attempted to accelerate one of the important stream computing applications, which is an anomaly detection algorithm called SST (Singular Spectrum Transformation) by using a GPU. The most dominant part of SST is a matrix computation, SVD (Singular Value Decomposition) that has $O(n^3)$ computational complexity, where $n$ is the matrix size needed for the window size of the target data stream. We achieved excellent SVD performance gains for large matrixes. We implemented our optimizations using the commercial CULA [10] library that provides an accelerated version of SVD for GPUs. The experimental results show the large performance gains for large matrixes compared to a CPU. With a GPU, the SST for a matrix of size 1,000 is computed in 4.44 seconds, which is 12.44 times faster than a CPU. This shows that an anomaly detection algorithm for a relatively large sliding time window is a viable option for an application the allows a response time around 4 seconds when looking for anomalies over a relatively long time period (of 1,000 time steps).

![Figure 1. Comparing a CPU and GPU for SST](image)

However as shown in Figure 1, with small-sized matrix less than 450, GPGPU does not provide performance advantage over CPU. Our first attempt showed that GPU-based methods are more beneficial for larger matrixes but relatively ineffective for smaller matrixes. From a different perspective, we can say that the initial experimental results lead to two important findings. One is that finer grained computations with smaller matrixes need to be optimized for a GPGPU, since there is not much difference between a CPU and GPU for a small matrix, as shown in Figure 1. The other finding involves scalability. Our initial method can handle only a “single” stream at one time, not the multiple streams that are continuously emitted from a large number of sensors. A useful system must handle large-scale multiple streams and process them in real time, which is not feasible for a naïve method as conducted in our first attempt. In our first approach, the performance degrades linearly with the number of data streams, since the standard GPU optimization method processes each data stream sequentially, one after another.
CUDA [10] assumes there is a single task and cannot fully utilize the GPU capabilities with a small matrix. The CULA documentation [10] shows the performance of QR decomposition using CULA. It shows the speed-up of CULA against Intel MKL (Math Kernel Library) with different matrix sizes. The experimental results in the documentation show two things: (1) the speedup of a GPU over a CPU increases with larger matrices, and (2) matrix sizes larger than 1,000 show better performances than a CPU.

In the results smaller matrices result in poor performance gains with the GPU. This is because global synchronization within the GPU is occurs frequently and there are too few threads, which leads to poor utilization of many of the available GPU cores.

In this paper, we propose the optimization approach GTP, GPU Task Parallelism. This approach simultaneously processes from 10 to 100 times more tasks on a GPU than the naïve approach with a single task, and provides good performance gains even with smaller matrices. With the proposed approach, high scalability and low response times can also be obtained in a scenario where there are large numbers of sensors and needs to do various kinds of processing such as anomaly detection.

### 3. CUDA PROGRAMMING MODEL FOR GPGPUS

CUDA [10] is a parallel computing architecture introduced by NVIDIA. CUDA gives developers access to the instructions and the memory of NVIDIA GPUs. In general, CUDA-based applications consist of host code which is running on CPU and “CUDA kernels” (abbreviated as kernels hereafter) which are functions running on GPU. The host code invokes the kernels. A kernel is executed by an array of threads and all threads run the same code. Each thread has an ID that it uses to compute memory addresses and make control decisions. A kernel launches a grid of so called thread blocks. A thread block is a batch of threads that can corporate with each other by sharing data through shared memory and synchronizing their execution. Threads in different blocks cannot cooperate. GPU hardware is free to schedule thread blocks on any processor. It also transfers data between the CPU memory and the GPU memory.

An NVIDIA GPU is made up of the Streaming Multiprocessors (SMs). The number of SMs on a GPU differs across GPUs. The SM manages, schedules and executes threads in groups of 32 parallel threads called warps. A warp is a groups of 32 threads within a block. One SM can schedule 24-32 warps. A warp executes one common instruction at a time.

### 4. OUR PROPOSED OPTIMIZATION APPROACH: GPU TASK PARALLELISM

A parallel programming model using GPGPUs is quite suitable for data parallel processing. For example, matrix computation libraries that use GPU computational capabilities, such as CUBLAS [12] or CULA [10], are implemented to divide each matrix into a set of blocks, which can then be computed independently in a data parallel fashion. Thus, for matrix computations with large matrices, high levels of parallelism can be achieved by using the GPU’s computational capabilities, even as a “single task”. A “task” is defined in this paper as the processing that is occurring at one time on a GPU. This notion of a task comes from the parallel computing field, where there are two paradigms for parallel processing, data parallelism and task parallelism. GPU computations are most often data parallel rather than task parallel. However when accelerating stream computing applications with GPUs, data parallelism is not sufficient. Task parallelism, where multiple tasks are computed simultaneously, must be used when a stream computing application requires relatively small computations for large amounts of streamed sensor data. For such applications, only task parallelism can execute multiple tasks simultaneously on many GPUs to efficiently use the GPUs’ computational capabilities. The abstract image for explaining the need for the GPU Task Parallelism is illustrated in Figure 2. For single task, the occupancy of GPU becomes quite low if the computational complexity of a task is not high. For such a purpose, we would like to make the occupancy high by running multiple tasks simultaneously.

![Figure 2. Overview of GPU Task Parallelism](image_url)

Hereafter we describe how we can achieve task parallelism with GPUs. As previously mentioned, there are two paradigms for parallel processing, data parallelism and task parallelism. Data parallelism is easily obtained with a GPU thanks for the fundamental nature of GPUs. For task parallelism on a GPU, multiple different instructions must be executed at the same time. The fundamental concept of the CUDA [11] programming model employs the SPMD (Single Process Multiple Data) model to execute a large number of GPU threads within a single kernel function (In CUDA each kernel function specifies the code to be executed by all of the GPU threads in a parallel phase.). The GPU normally executes the same code segment, but it is possible to switch to different code segments by using explicit conditional branches implemented in a kernel function by a developer.

An NVIDIA GPU has multiple SIMD cores to form a Streaming Multiprocessor (SM). Each core has an independent instruction issue unit. A single SM can execute a maximum of 24 to 32 warps simultaneously. One warp is defined as a group of 32 threads and is treated as an SIMD execution unit. Each warp has its own independent instruction pointer and multiple warps are executed in parallel. Therefore, if each warp executes a different instruction sequence by following a conditional branch, different instructions can be executed in parallel, providing task parallelism which fulfills the requirement for applying GPGPUs to stream computing.
To achieve the efficient task parallelism with GPGPUs, the synchronization is the important issue. CUDA supports two thread synchronization mechanisms, by using CUDA blocks or kernel functions. Each CUDA block is a collection of threads and each block can execute a maximum of 512 to 1,024 threads. The synchronization by using CUDA blocks is done by explicitly inserting synchronization instructions in the kernel function code. In contrast, the synchronization by using CUDA kernels requires terminations and restarts of kernel functions. This means different instruction codes can be executed in parallel by using CUDA warps for fine-grained tasks, but CUDA blocks or kernels can be used for coarse-grained tasks. We can therefore support task parallelism by (A) assigning different tasks to different blocks or warps, or (B) by assigning different tasks to different kernel functions to use the GPU hardware that can simultaneously execute multiple kernel functions.

Method (A) allows each task to be executed by a CUDA block or warp by inserting a conditional branch into a kernel function, so multiple tasks can be processed by a single kernel function. With this approach, if each task is sufficiently small, hundreds of tasks could be executed simultaneously. This approach can be also used together with the Method (B) that executes multiple kernel functions simultaneously. There are two inherent drawbacks of Method (A). First, when handling multiple tasks with different computational complexity, the next computation cannot start until all of the tasks are computed, even though some of the tasks may have already been finished. Second, since it requires developers explicitly insert conditional branches into the kernel function, the existing GPU libraries cannot be used.

The second approach (B) for simultaneously executing multiple kernel functions is to assign each kernel function invocation to one task. In this approach, even when different tasks have different computational complexities, the next computation can be started soon after each task is finished, because a kernel function is terminated after its computation is finished. Also, since a kernel function is implemented with the assumption that it is executed as a single task, it is not necessary to modify the existing kernel code. Therefore this approach easily supports parallel execution of tasks. However, the capability for multiple kernel function invocation is supported only by the relatively recent GPU devices built with the NVIDIA Fermi architecture. As of this writing, the latest GPU devices can execute a maximum of 16 kernel functions simultaneously, which limits the task parallelism that simultaneously handles more than thousands of sensors. Also a kernel function, implemented based on the assumption that it is executed as a single task, will have performance limitations for smaller problems. It can use the GPU cores, but there will be relatively expensive synchronizations of multiple kernel functions, leading to degraded performance.

Based on this analysis, we first decided to use Method (A) to assign tasks to blocks or warps even though Method (B) can eventually be used together with Method (A). There were two reasons for this design choice. First, it can be used on many existing GPUs, not just the latest GPU devices. Second it can cope with tasks of small computational complexity.

In the next section, we explain in more detail how the GPU task parallelism can applied to SVD (Singular Value Decomposition) as used in a wide variety of applications such as an anomaly detection algorithm (where it is called SST (Singular Spectrum Transformation)) [2], and then show the performance improvements and scalability for a large number of time-series data streams with smaller window corresponding to smaller matrix sizes.

5. SVD WITH GPU TASK PARALLELISM

In this section, we give a brief overview of SVD (Singlar Value Decomposition) and then explain how SVD is optimized with the GPU task parallelism for stream computing.

5.1 SVD for Stream Computing

SVD (Singular Value Decomposition) is one of the most important matrix computations in computational linear algebra, used in a wide variety of areas such as data mining, spectral clustering and image processing. Many data mining algorithms use SVD and when conducting them for large-scale sensors in a stream computing fashion, they require high-performance SVD for small-sized matrix. One of the applications for data mining is the anomaly detection (or change-point detection) algorithm called SST from Ide [2]. The essential computation of SST is based on SVD, which has high computational complexity, and thus is only viable for small matrices when it is used for stream computing.

Change-point detection or anomaly detection is used in many fields such as the management of sensors in a manufacturing factory or error detection in clusters of servers. SST [2] is characterized as a change-point detection algorithm that can cope with a wide variety of input data with only a small number of parameters. SST does feature extraction with SVD from a subsequence of the original time series data and one of the new time series data, and computes change-point scores by comparing the extracted feature vectors of the two subsequences. SVD is highly compute intensive task. Sufficient parallelism can be achieved by SVD for large matrices even with a single task. However it is not realistic to handle large matrices with real-time stream computing as shown in Figure 1, where roughly 4 seconds are needed for each result. Therefore our method only targets SVD computation for relatively small matrices of less than 500.

Next we explain how we optimize SVD with the GPU task parallelism by treating each matrix as one task and simultaneously computing multiple tasks with the GPU.

5.2 Implementation

The SVD for a matrix A decomposes A into Form (1), where U is an unitary matrix, is an diagonal matrix, and is an unitary matrix.

\[ A = U \Sigma V^T \] (1)

The diagonal entries of \( \Sigma \) are called the singular values of A. The matrix is called a left-singular vector and is called a right-singular vector. The most established algorithm for SVD has two phases, bidiagonalization and diagonalization. The bidiagonalization can be efficiently computed as a series of Householder transformations. For diagonalization, many algorithms have been proposed, such as the QR method [19], divide and conquer method [16], MR3 [18], and 1-SVD [27]. Our implementation is based on the QR method, because it is a well known algorithm used by LAPACK [17] and, more importantly for our purposes, it is easy to parallelize the computation for singular vectors. Figure 3 shows the image of our implementation.
by decomposing SVD into multiple stages. The stages is comprised of (1) a CPU sends the input matrix to GPU, (2) a GPU calculates the bidiagonalization (3) GPU sends the bidiagonal values to CPU, (4) CPU computes singular values, (5) CPU sends processed data to GPU, (6) GPU computes singular vectors, and finally (7) GPU sends singular vectors back to CPU.

A block method [5] is effective with the Householder transformation for bidiagonalization. A block method is also a well-known optimization technique for LAPACK [13]. A block method does not decrease the computational complexity by itself, but leads to better performance since the computation is simplified and the locality of memory accesses are improved. To investigate the effectiveness of the block method, we implemented two versions, a non-block version and a block version. The performance results show that the block method was twice as fast as the non-block method. Although the code for the block method is slightly more complicated, the performance improvements for bidiagonalization extend down to matrixes as small as 10s to 100s.

### 5.4 Diagonalization

For the diagonalization phase, the bidiagonalized matrix B is decomposed by using the QR method to Form (3), where $A$ is a diagonal matrix and $X$ and $Y$ are orthogonal matrices.

$$B = X\Sigma Y^T$$

In Formula (1), the matrices $U$ and $V$ are computed as $U = QX$ and $V = (PY)^T$, but the matrices $X$ and $Y$ are not needed for SVD. The QR method obtains the matrixes $U$ and $V$ by transforming $Q$ and $P$ instead of transforming the matrixes $X$ and $Y$.

The computations for obtaining the singular vectors in the QR method can easily be parallelized. However, it is not straightforward to parallelize the calculations for the singular values. Therefore, our implementation computes the singular values on the CPU while computing the singular vectors on the GPU. This technique is also used in [4].

For a single task, it is straightforward to execute the computations with the GPU and CPU in parallel. To parallelize the computation, the GPU kernel function can be invoked as the transformations of singular vectors are required during the computations of the singular values by the CPU. However, the GPU must execute the computations for all of the tasks in parallel when calling the kernel function each time. Therefore, to parallelize the tasks by extending the same approach as a single task, one approach is to parallelize the tasks from the CPU computations. There are several methods to parallelize the CPU tasks.

One method is to invoke one CPU thread for each task. Each CPU thread does its computations up to the GPU processing step. Then all of the CPU threads are synchronized and the GPU computations to be done for each CPU thread are merged into one GPU kernel function. After this kernel function is invoked, each CPU thread is restarted. However with this method, since CPU must spawn a CPU thread for each of many tasks, there are large resource demands for the many threads and expensive global synchronization of all of the threads is needed.

Another method is for the CPU to create a function that handles all of the computations until the GPU computations are required, and then restart the computation on the CPU side immediately after the GPU computation is done. Thus, a CPU function explicitly manages the state of each task such as running, stopping, or suspending. This method avoids spawning excessive threads as in the first method, which leads to a more high efficient use of resources and lower thread synchronization costs. However, the problem of this method is that the resulting function is quite complicated since it needs to explicitly manage the state of each task, while the thread-based approach manages the tasks implicitly.
After considering the trade-offs of the two methods, we used this approach. The computation of the singular values by the CPU runs to its completion without interruption for any GPU processing. For the required GPU computations, all of the necessary processing data is first stored in the CPU memory. After all of the tasks compute the singular values on the CPU, the data stored in the CPU memory is transferred to GPU, and the GPU tasks are processed in parallel. This approach has several benefits. First it minimizes the frequency of the costly global synchronizations for the GPU since all of the required data for the GPU processing is transferred at one time. Also, this approach can efficiently use the CPU cache, since the singular values are computed on the CPU without interruption. The main drawback of this approach it is that it requires more memory to store all of the data for transfer to the GPU. Our experiment found that this approach more doubled the memory needed for the SVD computation of an $n \times n$ matrix.

6. PERFORMANCE EVALUATION

In this section we compare the SVD using the proposed GPU parallel task method described in Section 4 against a CPU-based implementation. The CPU-based SVD uses ATLAS 3.8.3 and LAPACK 3.3.0. Our experimental environment consisted of an AMD Phenom X4 9850 (4 cores, 2.5 GHz) as a CPU with 8 GB of RAM, a Tesla C1060 and GeForce 8800 GTS 512 as GPUs, and CentOS 5.4 as the operating system. The specification for two GPU devices, Tesla C1060 and GeForce 8800 GTS 512 are 1.3 and 1.1 as compute capability, 4GB and 512MB as memory size, 30 and 16 as the number of multiprocessors, 240 and 128 as CUDA cores, 16K and 8K as the number of registers per multiprocessor respectively.

The performance results are the average elapsed times for 5 SVD computations. To make the comparison fair, the performance results for the GPU include the times for the data transfers. Each matrix was generated with a random number generator and each matrix entry was a single-precision floating-point number. The CPU implementation used the SGESVD function in LAPACK, which uses the same QR method as our GPU implementation. Each task computes the SVD for one matrix in this experiment.

Figure 4 and Figure 5 shows the elapsed computational times for SVD. The number of tasks is 64. To evaluate the performance of the CPU implementation, we measured the performance for a single thread shown in Figure 4 and for 4 threads shown in Figure 5, since the CPU has 4 cores. We conducted these two experiments to understand the performance difference with GPU when using either 1 thread or 4 threads on the quad-core CPU processor. For the GPU implementation, the diagonalization computation executed on the CPU can be also handled by either a single thread or 4 threads, so the experimental results also includes the elapsed time for the computation at the CPU side.

The elapsed times increased for larger matrixes, as expected, but the results of the CPU implementation show that the elapsed times do not steadily increase with larger matrixes, such as 448 and 480. This unexpected result might be because the CPU implementation does not work well when the matrix size is larger than 256 and a multiple of 64. The reason why we could not obtain the result for more than 448 matrix size with GeForce 8800 comes from the limitation of its register size.

Figure 6 and Figure 7 shows the acceleration of the GPU over the CPU. We measured two cases, a single thread (Figure 6) or 4 threads (Figure 7), and also 64 or 256 tasks for a total of four cases. However for fair comparisons, we excluded the cases when the matrix size was a multiple of 64, since these sizes caused problems for the CPU implementation. Again, the reason why we could not obtain the result for more than 228 matrix size and 256 tasks with GeForce 8800 comes from the limitation of its memory size.

As shown in Figure 7, these results were obtained with 4 CPU cores and 256 tasks. Overall, the speed-up ratio of the GPU over the CPU increases with larger matrixes. For matrixes larger than 96, the GPU implementation has a performance advantage over the CPU implementation from 1.43 to 4.73 times with Tesla C1060, and 1.3 to 2.5 times with GeForce 8800 when 4 CPU cores are used in the CPU implementation. The speed-up ratio of Tesla is 4.14 with a matrix size of 416 and 256 tasks, and the one of GeForce is 2.53 with a matrix size of 416 and 64 (due to the limitation of memory size).
Figure 8 and Figure 9 shows the breakdown of elapsed times for the GPU implementation using 256 parallel GPU tasks with Tesla C1060 and GeForce 8800 GT S 512 respectively. The breakdown has 4 phases, bidiagonalization on the GPU, the part of the diagonalization that ran on the CPU, the remaining part of the diagonalization on the GPU, and then the data transfer between CPU and GPU.

In the breakdown of the execution time, the bidiagonalization only dominates 20% to 30% of the total time with both GPU devices. The portion for diagonalization on CPU decreases for larger matrices. The current GPU implementation has two sequential phases, a first phase for diagonalization on the CPU and a second phase of diagonalization on the GPU. If these two phases could be parallelized, we could improve the performance. The graph also shows the data transfer between CPU and GPU, which consists of stage 1, 3, 5, 7 as shown in Figure 3. The data transfer of Tesla dominates from 11.9% (matrix size:32) to 3.73% (matrix size: 512) and its ratio decreases as the matrix size is increased since other computational time dominates more CPU time relatively. Around 80% of the data transfer is consumed by stage 5 which transfers the data from CPU to GPU, which is required for computing singular vectors in stage 6. GeForce shows the same performance characteristics as Tesla.

Figure 10 shows the acceleration of the GPU (Tesla C1060) over the CPU when changing the number of tasks with a fixed matrix size of 288.
7. SCALABLE STREAM-BASED CHANGE-POINT DETECTION WITH MULTIPLE GPUS

In this section we apply SVD with our proposed GTP (GPU Task Parallelism) to an anomaly detection algorithm called SST and evaluate the performance.

7.1 SST: Anomaly Detection Algorithm

Here is the algorithm for SST-SVD:

1. Given the time series data \( \tau = \{x_t \mid t = \text{real numbers} \} \), a window size \( w = \{x_t \mid t = \text{a subsequence of length} \} \), and a column vector \( s(t) = (x_{t-w+1}, \ldots, x_{t-1}, x_t)^T \).

2. Define the following two matrices from the \( n \) subsequences where \( \gamma \) is a positive integer number:

   \[ M = \begin{bmatrix} \gamma \cdot s(1) & \gamma \cdot s(2) & \cdots & \gamma \cdot s(n) \end{bmatrix} \]

3. Compute the left singular vectors of \( H(t) \), select \( r \) (< \( w \), \( n \)) vectors from the largest left singular values to the smallest, and then define those vectors as \( u_t^{(1)}, u_t^{(2)}, \ldots, u_t^{(r)} \). These are the feature vectors for the past sequences.

4. Let the maximum left singular vector of \( G(t) \) be \( \mu_t \). The vector \( \mu_t \) is the current feature vector.

The change-point score \( z(t) \) is computed as

\[ z(t) = 1 - \sum_{i=1}^{r} \left( \mu_t^T u^{(i)} \right)^2 \quad (4) \]

Since the maximum left-singular vector of \( G(t) \) is the maximum eigenvalue vector of \( G(t)^T G(t) \), \( \mu_t \) can be efficiently computed with a power method. However, the objective of our research was to investigate the performance characteristics of the SVD implementation based on the proposed GPU task parallelism, so \( \mu_t \) is computed from the SVD of \( G(t) \). Therefore, for the change-point scores, SVDs for both \( H(t) \) and \( G(t) \) are needed.

7.2 Scalable SVD-SST on Distributed Stream Computing Platform

We implemented SVD-SST (the SVD-based SST of Section 6.1) using a stream computing runtime, IBM System S from IBM Research. Several papers [6, 20-22] provide detailed information about this runtime and its programming language, SPADE [21]. SPADE is a data flow language designed for stream computing with a set of operators for streams. Each operator receives incoming data from one or more streams and does some operation such as filtering or transforming the data. In addition to its built-in operators, developers can incorporate their own customized operators into the runtime. Such a customized operator is called a UDOP (User-Defined Operator).

The SVD-SST is implemented as a UDOP. System S allows a SPADE program to be executed on a distributed environment with multiple compute nodes. We used this capability to run SVD-SST on multiple GPUs and multiple nodes. This integration with a stream computing system was also important, since it is difficult to implement a distributed system that uses multiple GPUs and nodes.

We used up to 4 nodes and 16 GPUs for our performance evaluations. Each node has the same specifications described in Section 4, but in this experiment we used 4 GPUs (GeForce 8800 GTS 512) in each node. For the SST-SST parameter, the window size was 320. For the input data stream, we used wave data from an unsteady sine wave.

![Fig. 11. Throughput by increasing # of GPUs and CPUs](image)

Figure 11 shows the performance results. In the figure, “GPU” indicates the SVD computations used GPU task parallelism, and “CPU” indicates computations based on LAPACK and ATLAS. The horizontal axis is the number of GPUs or the number of CPU cores. For example if the horizontal axis is \( N \), there were \( N \) GPUs and \( N \) CPU cores. With \( N \) is from 1 to 4, the experiment ran on a single node. The figure shows that the GPU implementation (our approach) achieved linear acceleration up to 16 GPUs. The throughput with 16 GPUs was 305 scores per second. This performance result corresponds to handling 1,525 sensors simultaneously and in real-time if the computation of change-point scores for the sensor stream data is done every 5 seconds.

8. DISCUSSION

Block-based task parallelism, which assigns each task to each block, has several advantages. First the relatively expensive synchronizations (such as at the kernel function level) are not required, since there is no communication between blocks. Synchronization at the kernel function level requires terminating and restarting kernel functions, which degrades the performance. Frequent synchronization within each block is required, but the costs are limited thanks to the high-speed synchronization mechanisms of the GPU hardware. Also, a programming and implementing this approach is less difficult, since each block can be executed independently without coordinating with other blocks. Also, it is not necessary to terminate kernel functions for
synchronization, which supports larger kernel function. Larger kernel function can more efficiently use the shared memory and registers, resulting in better performance. However, if a kernel function is too large and complicated, attention must be paid to how it uses the available registers. In particular, the generations of GPUs before the NVIDIA Fermi architecture tended to use larger numbers of registers, since each function invocation within a GPU kernel function was inline-expanded. When many registers are used, the efficiency is reduced since the number of simultaneous threads per SM decreases as each SM runs out of registers. Developers should try to limit the number of registers in use.

In contrast, synchronization is not required if each task is assigned to one warp (though we have not yet studied this approach). If 32 threads correspond to one warp, then all of the threads can be handled synchronously by the SIMD capacity of one GPU. The problem must be small if an entire task is assigned to one warp, but the proposed GPU task parallelism can run the computations efficiently even for such small problems.

When assigning each task to a block or warp, it is better to align the computations with the complexity of each task. If there are imbalances in the running times among the tasks and the target GPU supports simultaneous invocation of multiple kernel functions, then it is possible to divide the tasks by their different computational complexities to different kernel invocations. This method provides higher efficiency since some kernel can be invoked soon after each computation is finished.

In addition, when a task with high computational complexity has a high block ID, its waiting time may become large. The CUDA runtime assigns a block to an SM in increasing order of the block IDs. Therefore, if a block with a high block ID uses expensive operations that run for a long time, the system may have to wait for the termination of its kernel function even though all of the other blocks have finished. To avoid this, we would prefer to sort all of the tasks in decreasing order of their computational complexities if their execution times can be predicted before their kernel invocations.

When parallelizing an application that alternates its computations between the CPU and GPU while using the proposed GPU task parallelism, the CPU’s task parallelism may become a problem. When the GPU computation is represented as a kernel function invocation, the GPU computations are handled as subroutines of the CPU code. This design offers high programming productivity, but the CPU computations should also be parallelized for our proposed GPU task parallelism.

In our research, we used the approach of storing all of the required data in memory without parallelizing the CPU parts, but this approach can use too much memory for some applications. To avoid this limitation and parallelize the computations on the CPU, we might be able to use a co-routine approach that reduces the resource consumption and synchronization costs.

Our proposed approach is ineffective for application that cannot run their computations with data parallelism. Each warp of 32 threads is computed using SIMD, so if an application assigns multiple tasks to one warp, each task will attempt to execute a different sequence of operations, which leads to inefficient usage of the SIMD compute unit. However, if the application can run multiple tasks in data parallel mode, the GPU performance will be relatively high.

9. RELATED WORK

There has been a lot of recent research on GPGPUs, such as [23,25], but none of the research is directly about using GPUs for stream computing. For task parallelism on a GPU, Guevara [1] proposed a method suitable for general-purpose computing. Their approach supports task parallelism that needs no modifications to the existing GPU kernel function. They developed an intervening runtime library between the user-level and CUDA runtime level, and the intervening runtime has a kernel invocation queue. When a GPU kernel function is invoked, it is added to the queue and then an optimizer automatically merges multiple kernel functions residing in the queue into a single kernel function. Each kernel within a merged large kernel function can be identified by a distinguished ID, and each kernel function can be accessed with a conditional branch. The functional argument for each kernel function is added to the arguments of the merged kernel function. However, the number of conditional branches and function arguments will be increased in this design, and also a task with low computational complexity may need to wait until all of the tasks are finished when different tasks have different computational complexities. Therefore, this design cannot handle large numbers of simultaneous tasks as in our proposed approach.

The optimization of SVD with a GPU was done by [4]. They only targeted individual tasks and showed solid performance gains for very large matrices, but not for the small matrices we handled. Many of the efforts to optimize matrix computations with GPUs have primarily targeted large matrices, and there has been little work on parallel processing for small matrices.

10. CONCLUSIONS AND FUTURE WORK

In this paper we provide the workload characterization of applying GPGPUs to real-time and scalable stream computing applications. We showed the limitation of the straightforward approach with GPGPUs and proposed the optimization method called GPU Task Parallelism for stream computing. The proposed approach brings high performance and scalability to stream computing with GPUs. We validated our approach by applying it to SVD, which is used in a wide variety of applications, and also implemented an anomaly detection algorithm called SVD-SST based on the optimized SVD. The experimental results showed the expected scalability. Up to 1,054 sensors can be handled simultaneously in real time with 16 multiple GPUs on 4 nodes, whereas the CPU-based approach could only handle 1/7 as many sensors.

In the future, we plan to use the optimized SVD for other applications besides SVD-SST. Also, some advanced algorithms for the diagonalization step of SVD have been proposed [27]. I-SVD [27] may be better than the QR method used in the work reported here. We will investigate how to implement I-SVD with GPU task parallelism and also study the performance. The implementation in this paper does not yet utilize the GPU functions to invoke multiple kernels simultaneously. Combining this with our method described should allow us to provide even more flexible and efficient GPU task parallelism.

11. REFERENCES


[10] CULA. CULA Programmers Guild R10-1


