Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems | School of Computing and Information Systems

11-2011

Enabling GPU acceleration with messaging middleware

Randall E. DURAN
Singapore Management University, rduran@smu.edu.sg

Li ZHANG

Tom HAYHURST

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

 Part of the Databases and Information Systems Commons, and the Software Engineering Commons

Citation

DURAN, Randall E.; ZHANG, Li; and HAYHURST, Tom. Enabling GPU acceleration with messaging middleware. (2011). *Proceedings of the International Conference, ICIEIS 2011, Kuala Lumpur, Malaysia, November 14-16*. 410-423.
Available at: https://ink.library.smu.edu.sg/sis_research/6446

This Conference Proceeding Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Enabling GPU Acceleration with Messaging Middleware

Randall E. Duran[1,2], Li Zhang[1,2], and Tom Hayhurst[2]

[1] Singapore Management University, 80 Stamford Road, Singapore 178902
[2] Catena Technologies Pte Ltd, #11-04, 30 Robinson Road, Singapore 048546
{randallduran,lizhang}@smu.edu.sg, tom@catena-technologies.com

Abstract. Graphics processing units (GPUs) offer great potential for accelerating processing for a wide range of scientific and business applications. However, complexities associated with using GPU technology have limited its use in applications. This paper reviews earlier approaches improving GPU accessibility, and explores how integration with middleware messaging technologies can further improve the accessibility and usability of GPU-enabled platforms. The results of a proof-of-concept integration between an open-source messaging middleware platform and a general-purpose GPU platform using the CUDA framework are presented. Additional applications of this technique are identified and discussed as potential areas for further research.

Keywords: GPU, GPGPU, middleware, messaging, CUDA, k-means, clustering, ZeroMQ, stream processing.

1 Introduction

General-purpose graphics processing units (GPGPUs) offer great potential for accelerating processing for a wide range of scientific and business applications. The rate of advancement of GPU technology has been exceeding that of mainstream CPUs, and they can be utilized by a wide range of computationally intensive applications. For instance, GPUs have been found to be superior in terms of both performance and power utilization for N-body particle simulations [1]. For these computations, GPU-based solutions were found to be the simplest to implement and required the least tuning, as compared with multi-core CPU and Cell processor implementations. For all their benefits though, significant time is required to design and tune algorithms that make optimal use of GPUs' memory architectures.

GPU processing can be used for a variety of scientific and business applications. GPUs have been used to accelerate DNA analysis and sequencing [19], biomedical image analysis [8], seismic data interpretation [10], and more recently in the derivatives pricing and risk analytics domain of the financial services industry [3]. As a case in point, many of the algorithms used for seismic data analysis – Fourier transforms, calculation of finite differences, and image convolutions – are especially well suited for parallel implementation on GPUs. GPUs have been shown to perform 20-100 times faster than CPUs for these types of computations.

A. Abd Manaf et al. (Eds.): ICIEIS 2011, Part III, CCIS 253, pp. 410–423, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Nevertheless, GPU platforms have seen only limited use in scientific applications. While GPUs are commonly found in desktop computer systems, only a small proportion of consumer-grade systems readily support GPGPU programming and have the memory bandwidth and number of processing cores needed to perform faster than typical CPUs. High-end GPU cards or dedicated GPGPU computing processors are often used with server-class hardware configurations. Hence, in many cases providing a GPGPU-enabled workstation on every desk is impractical.

Another factor that has limited the use of GPUs is the need for application designers to understand the intricacies of GPU architectures and programming models. To take advantage of GPU acceleration, developers must determine how the computational aspects of an application should be mapped to advantage of the parallelism of the GPU platform. They must also understand the GPU's limitations and work around them. In particular, device-specific memory access constraints must be considered and catered for.

While some middleware has emerged that helps simplify GPU programming and provide remote access to GPUs, ease of use still creates a significant barrier to their widespread adoption. Device-specific programming considerations, such as the number of cores, size of memory and performance for floating point precision calculations, are still critical factors. Likewise, little attention has been given to using GPUs to accelerate real-time applications.

Outside the field of GPU processing, message-oriented middleware, referred to henceforth as messaging for brevity, has proliferated over the past two decades and has become a mainstay for distributed computing applications. Request-reply and publish-subscribe communications are commonly used for remote invocation of services and data stream processing. Furthermore, most messaging platforms promote and facilitate the development of common, reusable services. Access is commonly provided to such services through abstract interfaces that seek to minimize application-specific functionality and hide details of the underlying implementation.

Given this context, this paper explores how the integration of messaging technologies can help improve the accessibility and usability of GPU platforms. Messaging can help to hide the complexities of the underlying GPU platform and programming environment and also facilitate remote access to GPU-enabled hardware platforms. The paper is organized as follows. First, a brief survey of background and related research is presented. Second, the design of a messaging-accessible, GPU-enabled platform is described. Third, practical applications of accessing GPU-enabled services over messaging are identified. Fourth, the results of a proof-of-concept integration between an open-source messaging platform and a general-purpose GPU platform built on the CUDA framework are provided. Finally, further areas of potential research are discussed.

2 Background and Related Work

Three different aspects of background and previous research are reviewed in this section. The types of calculations that are well suited to GPU acceleration are considered in relation to which functions might be best implemented in conjunction with messaging. Other middleware approaches that have been used to simplify local

and remote access to GPUs are also examined. The use of messaging to support GPU acceleration is also reviewed.

A number of different types of calculations have been mapped to GPUs to achieve improved performance for both scientific and financial applications. Monte Carlo simulations – which are used in computational physics, engineering, and computational biology – are one type of calculation that has been shown to benefit from execution on GPUs [16]. Hidden Markov models and other Bayesian algorithms – which are used in bioinformatics and real-time image processing – can also benefit significantly from GPU acceleration [6][13]. Likewise, k-means clustering – used in computational biology, image analysis, and pattern recognition – has been a common target for GPU acceleration [12][20].

The ease by which these different types of calculations can be implemented as common, abstracted services that can be executed remotely using messaging varies. On one hand, Monte Carlo simulations are not well suited for remote invocation because they require application-specific algorithms to be loaded and run on the server that hosts the GPU. There is no simple device-independent means of bundling up application-specific logic for remote execution on remote servers. On the other hand, k-means clustering calculations can be easily separated from application logic, parameterized, and presented as remote services that can be accessed through request-reply messaging functions. Accordingly, the research described in this paper focuses on the k-means algorithm for the proof-of-concept implementation.

There have also been a number of industry and research efforts focused on developing middleware that helps simplify GPU programming, including CUDA, HiCUDA, and OpenCL. CUDA (Compute Unified Device Architecture) is a framework that provides C language programming extensions that can be used to access the GPU hardware through a general purpose hardware interface opposed to a graphics processing-oriented API [15]. HiCUDA [7] goes further, providing higher-level programming abstractions that hide the complexity of the underlying GPU architecture. Alternatively, OpenCL [14] provides a device-level API similar to CUDA that is portable across different types of processors and GPUs, whereas CUDA is only supported on GPUs made by NVIDA. While these middleware implementations have improved the usability of GPU devices, they only support access to GPUs cards installed locally on the server that are controlled by the local operating system.

vCUDA and rCUDA help improve GPU accessibility by enabling applications to access GPUs indirectly. vCUDA [17] provides guest operating systems that run inside virtual machines with CUDA-based access to the host server's GPU. CUDA API calls are intercepted by a proxy client library installed in the guest OS that routes the calls to a vCUDA server component running in the host OS. The vCUDA server component then passes the API request to the GPU and returns the response to the proxy, which in turn delivers it to the application running in the guest OS. rCUDA [5] takes a similar approach to provide remote access to GPUs over the network. A client proxy library intercepts applications' CUDA API calls and forwards them using TCP/IP sockets to an rCUDA server component running on the remote host where the GPU physically resides. These two approaches help address the problem of needing local access to the GPU, but they still require the application developer to be aware of

and design around device-specific considerations of the remote systems' hardware configurations.

While a wealth of information on messaging middleware has been developed over the past two decades, little research to date has focused on how GPUs can be combined with messaging. To this effect, King et al [11] demonstrated how convertible bond pricing calculations could be accelerated by GPUs and accessed remotely via a request-reply style web service interface. They reported performance gains of 60x of the GPU-based hardware cluster over CPU-based hardware cluster configuration. Furthermore, the GPU-based cluster was substantially less expensive and consumed half the power of the CPU-based cluster. This example demonstrated the benefits of and potential for middleware-based GPU services; however, its use is limited. The calculation implemented on the GPU was domain specific and could not be easily leveraged for other purposes.

The design and prototype presented in this paper continues in the direction that King et al began, but seeks to provide a more generic and widely applicable model. As a proof of concept, a broadly applicable algorithm, k-means clustering, was implemented on the GPU and remotely accessed using an off-the-shelf messaging platform, ZeroMQ [9]. Access to the GPU was implemented in request-reply mode, as with King et al, and also in publish-subscribe mode, to help assess the practicality of using the GPU to accelerate network-based, real-time data stream processing applications. ZeroMQ was used as the messaging middleware because it is a readily available open source platform that provides a lightweight messaging implementation with low end-to-end latency. No unique features of ZeroMQ were used, though, and similar results would be expected were other messaging platforms to be used instead.

The aim of this effort was to answer several questions and help set the stage for further research. Specifically, the goal was to determine whether:

- From an architectural standpoint, it is possible to package certain types of computations so that they can be executed remotely on GPUs to achieve faster processing
- The CPU interacting simultaneously with the network interface and GPU would cause any conflicts or performance degradation
- It is practical to use GPUs to act as pipeline components processing real-time data streams using publish-subscribe messaging facilities

The main of the focus of this paper is on the request-reply oriented GPU-enabled services. Preliminary results are also briefly described regarding the feasibility and potential gains that might be achieved for publish-subscribe-oriented GPU applications.

3 Architecture

Traditional local-access GPGPU computing architectures are structured as shown in Fig. 1. Each client application runs on a separate GPU server that hosts the GPU processor. Each application has its own computation-intensive algorithms that have been designed to run on the GPU, and each application is implemented independently. Hence, two applications may use the same basic algorithm, but have different source

code and have been tuned for different GPU processor configurations. Moreover, the GPU processor on each server may remain idle when the application is not run, and the GPU capabilities of on one server cannot be shared by an application running on another server.

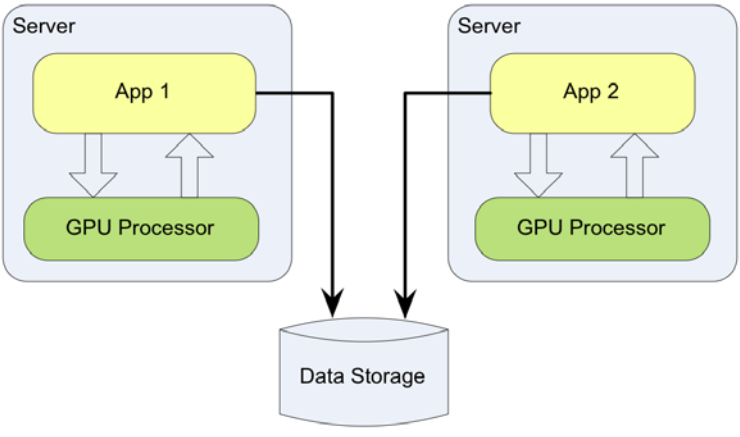

Fig. 1. A traditional local-access GPGPU architecture

Alternatively, by integrating middleware-messaging technologies with GPU platforms, it is possible to improve the accessibility, usability, and hardware utilization rates of GPU processors for general purpose computing. Fig. 2 shows a service-oriented architecture view of the middleware-accessible GPU-enabled platform. As compared to the traditional local-access model, client applications can run on remote workstations. Computation-intensive algorithms are invoked by passing data using request-reply messaging. These algorithms can run on servers that host high-performance GPU processors that serve multiple remote client applications. The algorithm implementation can be tuned specifically for each host platform's hardware configuration, without requiring each client application to address this concern. Likewise, the GPU-based algorithm implementation can be updated when the GPU hardware is upgraded without requiring changes to the client applications, assuming that the API remains constant.

In summary, the benefits of this architecture are to:

- Enable remote access to GPU processors
- Hide the complexity of underlying GPU platform and programming environment
- Provide abstract interfaces to common services
- Allow the GPU resources to be shared more efficiently across multiple applications
- Simplify maintenance of GPU-accelerated applications

The algorithm's input data may be passed directly across the messaging layer, or it may be more practical to include a reference to a remote data location – such as a

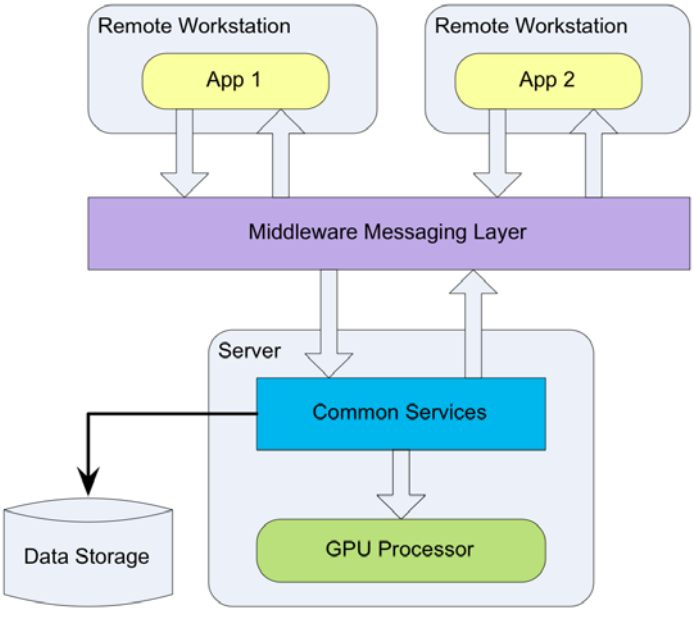

Fig. 2. The architecture of a middleware-accessible GPU-enabled platform

database stored procedure or URL – as part of the service request message. The service can then directly retrieve the data for processing.

4 Application

Messaging middleware could provide scientific applications with access to a range of computationally intensive algorithms that run on high performance GPU processors. To demonstrate the feasibility of this idea, two proof-of-concept applications were implemented and tested. The first application implemented a k-means clustering algorithm on the GPU and exposed it as a shared service that was made accessible to applications running on different servers through ZeroMQ's request-reply messaging interface. In a real-world scenario, remote computer vision, data mining, and computational biology applications could similarly make use of a GPU-accelerated clustering algorithm to partition and find common sets in collections of data.

The second application implemented filtering algorithms on the GPU and was made accessible via ZeroMQ's publish-subscribe interface. In this real-time scenario, raw streams of seismic data could be published using ZeroMQ, and then processed by GPU-accelerated filter algorithms, with the filtered results then being published back on the messaging middleware so that they can be accessed by downstream applications running on different servers.

4.1 GPU-Based K-Means Clustering Accessed Using Request-Reply Messaging

Cluster analysis has been used in the fields of computer vision, data mining, and machine learning, amongst others, to divide data objects into groups based on their features and patterns. K-means is a widely used partitional clustering algorithm. It randomly chooses a number of points to form the centers of each cluster, and then assigns each datum to the cluster whose centre is nearest it, in a process called labeling. The center of each cluster is then recalculated from the points assigned to it, and these new cluster centers are used to label the points again. This process is repeated until the recalculated cluster centers stop moving between successive iterations. Fig. 3 shows an example of clustering output using the k-means algorithm with 3 clusters.

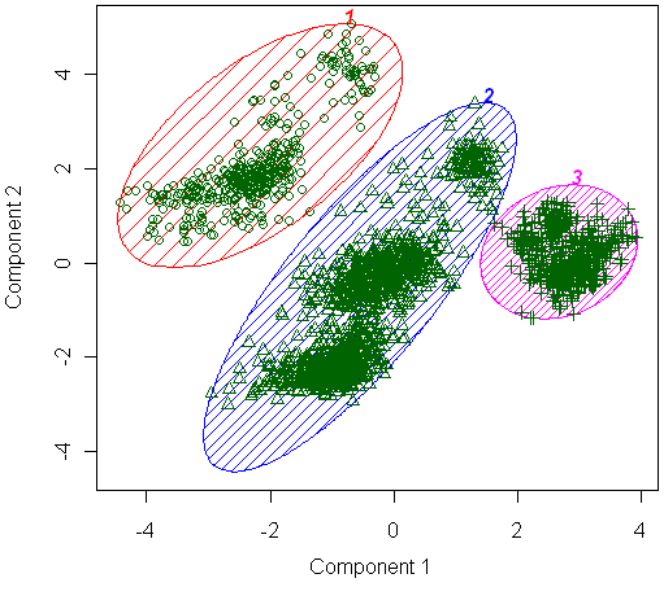

Fig. 3. Example of k-means clustering results with three clusters

The traditional k-means clustering algorithm has a known problem whereby non-optimal solutions may be found, depending on the initial random selection of cluster centers. To address this, the algorithm is normally repeated multiple times, to increase the chance of finding a near-optimal solution. Hence, k-means is a computationally intensive algorithm, especially for large data sets.

When executing the k-means algorithm, the time-consuming process of labeling can be transferred to the GPU for parallel execution to increase performance [2]. In the proof-of-concept implementation, the k-means algorithm was implemented using CUDA. Distance calculations were performed in parallel on the GPU while the CPU sequentially updated cluster centroids according to the results of the distance calculations [12]. Fig. 4 illustrates the processing flow of the GPU-based k-means algorithm. A CPU thread begins by reading the data points from the source location

and randomly initializing K cluster centers. To save data transfer cost between CPU and GPU, the data points are copied to the GPU's global memory only once. The GPU processor then labels the data points and transfers the new labels back to the CPU. Again, transferring only the new labels helps to save on data transfer between the GPU and CPU. The CPU thread calculates new centroids based on the updated labels and determines whether to invoke the GPU again to label the data points. Once the algorithm terminates, the final data labels can be downloaded to the CPU and stored.

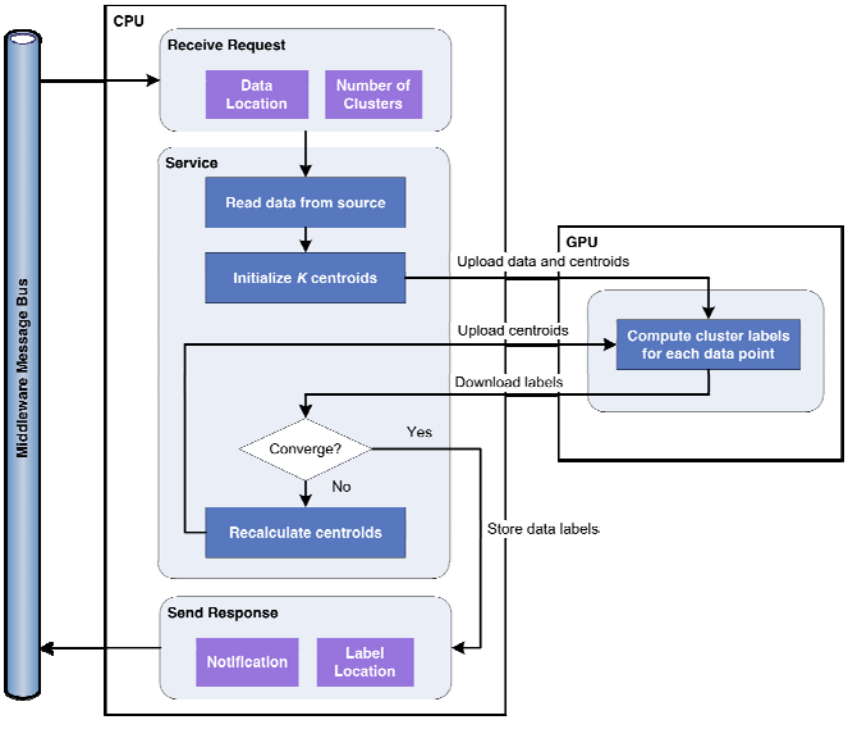

Fig. 4. Process flow of the GPU-based k-means clustering algorithm

The GPU-based k-means algorithm was then exposed as a shared service, accessed by remote client applications using a request-reply message interface as described in section 3. A client sends a request message specifying the location of the input data points (in a database table or file) and the number of clusters to be generated. After receiving the request, the service retrieves the input data and invokes the clustering algorithm. Once the algorithm has completed, the server will notify the client, which can then retrieve the clustered data set from the same location as the input data.

4.2 GPU-Based Filtering Using Publish-Subscribe Messaging

Sensor networks are commonly used to collect data for seismic data analysis generating large volumes of real-time data. Query-based filtering mechanisms, which

compare sensor readings against a large number of predicates, help identify patterns that correspond to or may predict specific types of events [4]. The process of comparing the current reading with a set of predicates is a time-consuming process that can be parallelized. Therefore, performance benefits may be achieved, especially for large data streams, by migrating the comparison function to a GPU [18]. Messaging middleware can provide a convenient and efficient mechanism for collecting and distributing data from remote sensors for analysis. Sensors typically publish their measurements asynchronously, filters consume these data and republish information when patterns match, and interested downstream applications will subscribe to the relevant information from the filters.

A proof-of-concept GPU-based filtering service was implemented and made available via a publish-subscribe messaging interface. The filtering service uses a binary "greater than" operator to compare current data readings with a historical data set read from a data source stored either as a file on disk or in a database table. For example, this filtering service could be used to monitor an ongoing stream of real-time sensor readings. If the filter detected that a current reading was greater than 90% of previous movement readings, another event would be published on the messaging middleware. While quite simple in its current form, this filtering mechanism could be easily extended to compare data with more complex predicates. Likewise, a large number of different predicates could be compared simultaneously by the GPU, taking advantage of its massively parallel architecture.

To test the prototype, an application that simulates a remote sensor publishes a stream of measurement readings. The filtering component running on the GPU server subscribes to this data stream compares each data reading to the filtering criteria. When a match occurs, filtering service publishes a message on a new topic, creating a derivative event stream.

5 Experimental Results

To compare the performance of a CPU-based k-means algorithm and the GPU-based version that were invoked through messaging, a set of tests were performed on a quad-core Intel i5-760 2.80 GHz machine with 8 GB of RAM, and a NVIDIA GeForce 450GTS graphics card [15]. For comparison purpose, the GPU-based algorithm was run on both a GeForce 450GTS graphics card with 192 cores and a Tesla C1060 graphics card with 240 cores, respectively. Each test used a different number of 3-dimensional data points and ran the k-means algorithm repeatedly 10000 times with 10 iterations per run to cluster the data points into different number of clusters. For the GPU-based algorithm, the number of threads per block was fixed at 32 for the GeForce 450GTS card, and 1024 for the Tesla C1060 card, and the number of blocks was computed based on the number of data points and the number of threads. Table 1 shows the average total time taken to generate five clusters using the CPU-based algorithm and the GPU-based version, respectively. The GPU total time is the total service processing time as shown by the Service component in Fig. 4, which includes the data upload and download time. The results obtained from the Tesla C1060 card surpass those from the GeForce 450GTS card significantly. This also shows that fine-tuning the number of threads per block as appropriate to the data size has a large effect on the GPU performance.

Table 1. CPU-based and GPU-based k-means service processing times

Processing Time	No. of Data Points				
	100	1K	10K	100K	1M
Avg single CPU total time (ms)	0.06	0.85	8.99	81.8	817
Avg GPU total time (ms) GeForce 450GTS	0.38	0.84	2.83	24.3	232
Avg GPU total time (ms) Tesla C1060	0.20	0.37	0.47	1.2	6

Fig. 5 shows the average total time taken for running the k-means clustering service when invoked remotely over messaging both on a single CPU, an estimated quad-core CPU, a GeForce 450GTS GPU, and a Tesla C1060 GPU, respectively. For simplicity, the time for the quad-core CPU is estimated based on a 3.4 speedup factor from the single CPU results. The results show that the processing time of the single CPU-based algorithm is very low for a small number of points, but it increases exponentially when the number of points increases. In contrast, the processing time of the GPU-based algorithm for a small number of points is relatively high, but it increases less as the number of points increases. The GPU shows superior performance at around 1000 points.

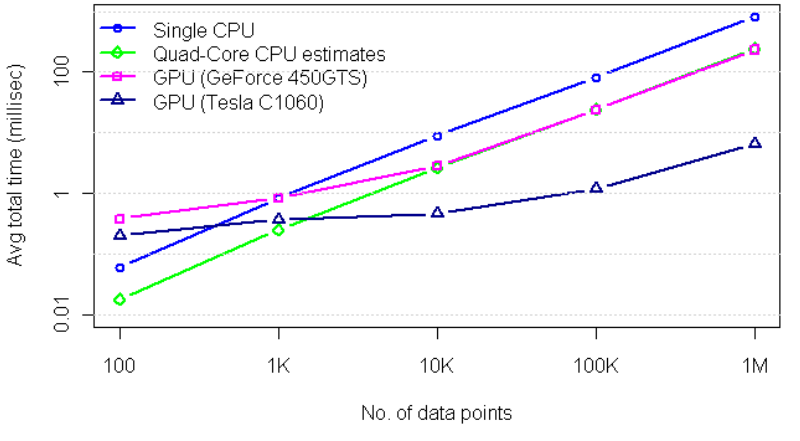

Fig. 5. Average total processing time for different number of data points using CPU-based and GPU-based k-means algorithm

Fig. 6 shows the average total time taken for invoking CPU and GPU versions of k-means algorithm using request-reply messaging to process one million points with different number of clusters. It demonstrates that the time taken for the single CPU-based algorithm as well as the quad-core estimate increases more rapidly than the GPU-based version. The GPU's performance benefits increase as the number of points and their dimension increase.

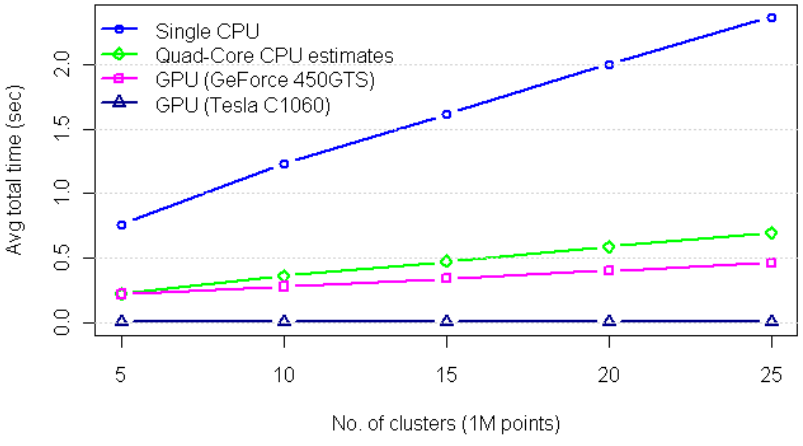

Fig. 6. Average total processing time for different number of clusters using CPU-based and GPU-based k-means algorithm

Preliminary experiments were also conducted on a publish-subscribe-based GPU service, as described in Section 4, to determine the latency involved when using messaging and GPUs to process real-time data streams. Fig. 7 shows the latencies that were measured. The input message latency measures the time taken for the input data message to reach the remote GPU service. The GPU processing time corresponds to the time taken by the GPU to process the input data received from the messaging layer. The output latency measures the time taken for a remote subscriber to receive event notifications once the GPU identified a criteria match.

As part of the tests, a client application published a stream of floating point data readings. After receiving each data reading, the server ran the GPU-based filtering algorithm to check whether the current data reading is greater than a set of historical readings read from the same data source. If the criteria matched, the corresponding cell of the bit map will be updated to 1, otherwise 0. The CPU thread on the server then determined, based on the bit map, whether current reading should be republished with an alert flag attached.

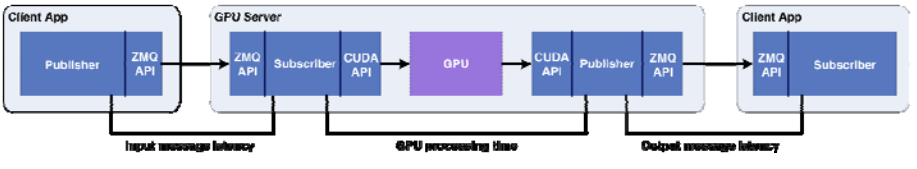

Fig. 7. Latency definitions for the publisher-subscriber model

The streaming data tests were carried out on the same hardware as for the k-means test described above. Table 2 shows the CPU and GPU processing time of the filtering algorithm and the two pub-sub latencies described above for 1,000 data readings each

comparing with 10K, 100K, and 1 million historical readings. The results show that for this simple filtering algorithm, running on GPU did not provide performance benefits as compared to running it on a single CPU due to the higher memory allocation and data transfer cost. However, if more complex filtering algorithms, such as complex predicate comparisons or Fourier transforms, were applied, it is more likely that the GPU-based implementation would outperform the CPU-based version. The input message latency is measured with a message publishing rate of 5000 messages per second. It is observed that as the publishing rate decreases, the input message latency is reduced as well but to a certain extent. When the publishing rate becomes too low, the input message latency increases. The same pattern is observed for the output message latency. The output message latency shown in Table 2 is comparable with the input message latency because the publishing rate is about the same as well. These preliminary results demonstrate the feasibility of the proposed architecture. Further work is necessary to fine-tune the performance of GPU-based filtering algorithm for real-time data stream processing.

Table 2. CPU-based and GPU-based filtering service processing times and input/output message latency

Median Processing Time	No. of Comparisons		
	10K	100K	1M
CPU processing time (μs)	28	260	1,820
GPU processing time (μs) GeForce 450GTS	156	717	5,065
GPU processing time (μs) Tesla C1060	205	721	4,189
Input message latency (μs)	46	54	55
Output message latency (μs)	36	51	56

6 Conclusion and Future Work

The aim of this paper was to answer several questions and help set the stage for further research. First, it was determined that from architectural standpoint, it is possible to encapsulate k-means computations so that they can be executed remotely on GPUs to achieve faster processing than locally on CPUs. Second, no significant conflicts arose when integrating messaging with the GPU processing. It is beneficial, however, to multithread services so that receipt of inbound messages and invoking local processing on the GPU are managed independently. Third, while using GPU-based services to process real-time data streams, borne by messaging middleware, was shown to be feasible, further work is required to demonstrate this approach can outperform similar CPU-based configurations.

Moreover, an important and easily measured benefit of messaging-enabled GPU architectures is cost and environment savings from reduced power consumption. Making GPUs more easily accessible can offload traditional CPU server implementations and reduce the number of GPU cards that are required to support

some types of computationally intensive algorithms and processing of high throughput real-time data flows. In this regard, Duato et al [5] estimated that halving the number of GPUs used in a high performance server cluster – which could be easily achieved through more efficient sharing of GPU resources – could reduce the cluster's overall power consumption by 12%.

Several areas of further research would be beneficial. One area is exploring how more advanced features of messaging middleware, such as one-of-N delivery semantics, could be used to support load balancing across different servers in GPU-enabled server clusters. Another area of interest is whether other computation types, such as hidden Markov models and Bayesian algorithms, are suitable for abstraction, parameterization, and remote invocation as a similar manner as was demonstrated for k-means clustering. Finally, further investigation of the potential for GPUs to support the analysis and filtering of high-throughput, real-time data flows would be beneficial.

References

1. Arora, N., Shringarpure, A., Vuduc, R.W.: Direct N-body Kernels for Multicore Platforms. In: 2009 International Conference on Parallel Processing, pp. 379–387 (2009)
2. Bai, H.T., He, L.L., Ouyang, D.T., Li, Z.T., Li, H.: K-Means on Commodity GPUs with CUDA. In: World Congress Computer Science and Information Engineering, pp. 651–655 (2009)
3. Clive, D.: Speed is the key - Balancing the benefits and costs of GPUs (2010), http://www.risk.net/risk-magazine/feature/1741590/balancing-benefits-costs-gpus
4. Daniel, J.A., Samuel, M., Wolfgang, L.: REED: Robust, Efficient Filtering and Event Detection in Sensor Networks. In: 31st VLDB Conference, pp. 769–780 (2005)
5. Duato, J., Peña, A.J., Silla, F., Mayo, R., Quintana-Orti, E.S.: rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In: 2010 International Conference on High Performance Computing and Simulation (HPCS), pp. 224–231 (2010)
6. Ferreira, J.F., Lobo, J., Dias, J.: Bayesian Real-Time Perception Algorithms on GPU - Real-Time Implementation of Bayesian Models for Multimodal Perception Using CUDA. Journal of Real-Time Image Processing (published online February 26, 2010)
7. Han, T.D., Abdelrahman, T.S.: hiCUDA: High-Level GPGPU Programming. IEEE Transactions on Parallel and Distributed Systems 22(1) (2011)
8. Hartley, T.D.R., Catalyurek, U., Ruiz, A., Igual, F., Mayo, R., Ujaldon, M.: Biomedical image analysis on a cooperative cluster of GPUs and multicores. In: 22nd Annual International Conference on Supercomputing ICS 2008, pp. 15–25 (2008)
9. Hintjens, P.: ØMQ - The Guide, http://zguide.zeromq.org/ (accessed April 2011)
10. Kadlec, B.J., Dorn, G.A.: Leveraging graphics processing units (GPUs) for real-time seismic interpretation. The Leading Edge (2010)
11. King, G.H., Cai, Z.Y., Lu, Y.Y., Wu, J.J., Shih, H.P., Chang, C.R.: A High-Performance Multi-user Service System for Financial Analytics Based on Web Service and GPU Computation. In: International Symposium on Parallel and Distributed Processing with Applications (ISPA 2010), pp. 327–333 (2010)
12. Li, Y., Zhao, K., Chu, X., Liu, J.: Speeding up K-Means Algorithm by GPUs. In: 2010 IEEE 10th International Conference on Computer and Information Technology (CIT), pp. 115–122 (2010)

13. Ling, C., Benkrid, K., Hamada, T.: A parameterisable and scalable Smith-Waterman algorithm implementation on CUDA-compatible GPUs. In: 2009 IEEE 7th Symposium on Application Specific Processors, pp. 94–100 (2009)
14. Munshi, A.: OpenCL Specification Version 1.0. In: The Khronos Group (2008), http://www.khronos.org/registry/cl
15. NVIDIA Corporation. NVIDIA® CUDATM Architecture. Version 1.1 (April 2009)
16. Preisa, T., Virnaua, P., Paula, W., Schneidera, J.J.: GPU accelerated Monte Carlo simulation of the 2D and 3D Ising modelstar, open. Journal of Computational Physics 228(12), 4468–4477 (2009)
17. Shi, L., Chen, H., Sun, J.: vCUDA: GPU Accelerated High Performance Computing in Virtual Machines. In: 2009 IEEE International Symposium on Parallel & Distributed Processing (2009)
18. Tsakalozos, K., Tsangaris, M., Delis, A.: Using the Graphics Processor Unit to realize data streaming operations. In: 6th Middleware Doctoral Symposium, pp. 274–291 (2009)
19. Tumeo, A., Villa, O.: Accelerating DNA analysis applications on GPU clusters. In: 2010 IEEE 8th Symposium on Application Specific Processors (SASP), pp. 71–76 (2010)
20. Zechner, M., Granitzer, M.: Accelerating K-Means on the Graphics Processor via CUDA. In: The First International Conference on Intensive Applications and Services, INTENSIVE 2009, pp. 7–15 (2009)