Chapter 12
A Middleware Framework for Programmable Multi-GPU-Based Big Data Applications

Ettikan K. Karuppiah, Yong Keh Kok, and Keeratpal Singh

Abstract Current application of GPU processors for parallel computing tasks shows excellent results in terms of speedups compared to CPU processors. However, there is no existing middleware framework that enables automatic distribution of data and processing across heterogeneous computing resources for structured and unstructured Big Data applications. Thus, we propose a middleware framework for "Big Data" analytics that provides mechanisms for automatic data segmentation, distribution, execution, information retrieval across multiple cards (CPU and GPU) and machines, a modular design for easy addition of new GPU kernels at both analytic and processing layer, and information presentation. The architecture and components of the framework such as multi-card data distribution and execution, data structures for efficient memory access, algorithms for parallel GPU computation, and results for various test configurations are shown. Our results show proposed middleware framework, providing alternative and cheaper HPC solution to users. Data cleansing algorithms on GPU show a speedup of over two orders of magnitude compared to the same operation done in MySQL on a multi-core machine. Our framework is also capable of processing more than 120 million of health data within 11 s.

Keywords GPGPU • CUDA • GPU • Architecture • Big Data • High-performance computing • Middleware framework

12.1 Introduction

NVIDIA CUDA-enabled GPGPU (general purpose graphic processing unit) has made its name by being part of world super computers to enable high-performance computation. Thus, GPGPUs are widely accepted and becoming common for many high-performance computing applications. GPGPUs are used for both specific and general purpose applications either running in large-scale system or desktop PCs.

E.K. Karuppiah (✉) • Y.K. Kok (✉) • K. Singh (✉)
MIMOS Berhad, Kuala Lumpur, Malaysia
e-mail: ettikan.karuppiah@mimos.my; kk.yong@mimos.my; keeratpal.singh@mimos.my

Y. Cai, S. See (eds.), *GPU Computing and Applications*,
DOI 10.1007/978-981-287-134-3_12

The design of PC pluggable GPGPU cards provides new programmable computing paradigm as the cost-effective solution. High-performance parallel applications and algorithms can be designed and developed, utilizing both CPU and GPU processors capabilities. However, the environment including middleware, framework, applications, and supporting tools must be capable of supporting parallel computing and execution, otherwise serial performance will be the outcome.

Big Data processing certainly has become imminent for enterprises that wish to process large amount of data which mainly comes from the social network, semantic web, sensor networks, geo-based service information, patient information, and employee or transaction-based applications. These areas observe quick growth of large data which needs either timely analytics or batched processing. Thus, the challenge is to analyze and mine these big data in order to effectively exploit the information to improve efficiency and quality of service for consumers and producers alike. However, the computing capabilities of current multi-core microprocessors are unable to meet the data mining requirements to effectively mine the data on time, thus needing parallel acceleration hardware such as GPUs [1] to accelerate the data mining. Even though high-performance computing solutions are available today for the above processing usage, the cost is still relatively high for general deployment and usage. For example, Netezza-, Teradata-, and Vertica-based systems are computationally fast and cater for terabytes of data processing in milliseconds but not affordable for small and medium enterprises. On the other hand, MapReduce framework-based applications such as Apache Hadoop and Drill which are free and stable are suitable for large-scale data processing. GPU-based Big Data processing system complements the above MapReduce-based solution. In our proposed Big Data and BI (business intelligence) solution framework, we have positioned GPU in two different layers, namely, analytics and processing, as illustrated in Fig. 12.1. These positioning provides flexibility to application-specific analytics algorithm coupled with data processing algorithms. For example, edit distance algorithms (analytics component) which are written in CUDA/GPU are tightly coupled with other generic data processing (processing component) component such as sorting, searching, etc., providing high-performance solutions. Thus, we believe GPU-based solution will coexist with other MapReduce systems as a complementing solution. The implementation section will demonstrate an example of this combination.

GPUs are massively parallel multi-threaded multi-core processors that allow large amounts of data to be processed in parallel to speed up computation time. The single instruction multiple threads (SIMT) architecture of the NVIDIA GPU places it between the single instruction multiple data (SIMD) architecture for vector processing and the simultaneous multithreading (SMT) architecture for hardware multithreading in terms of flexibility and efficiency. Current benchmarking shows that GPUs can execute up to a few orders of magnitude faster than CPUs for certain types of algorithms [2] and a large set of work has been done in order to leverage on the GPU computing capabilities [3, 4].

Since GPUs are treated as a coprocessor with its own architecture, applications must be designed to reflect the two-processor nature of the system. As such, data

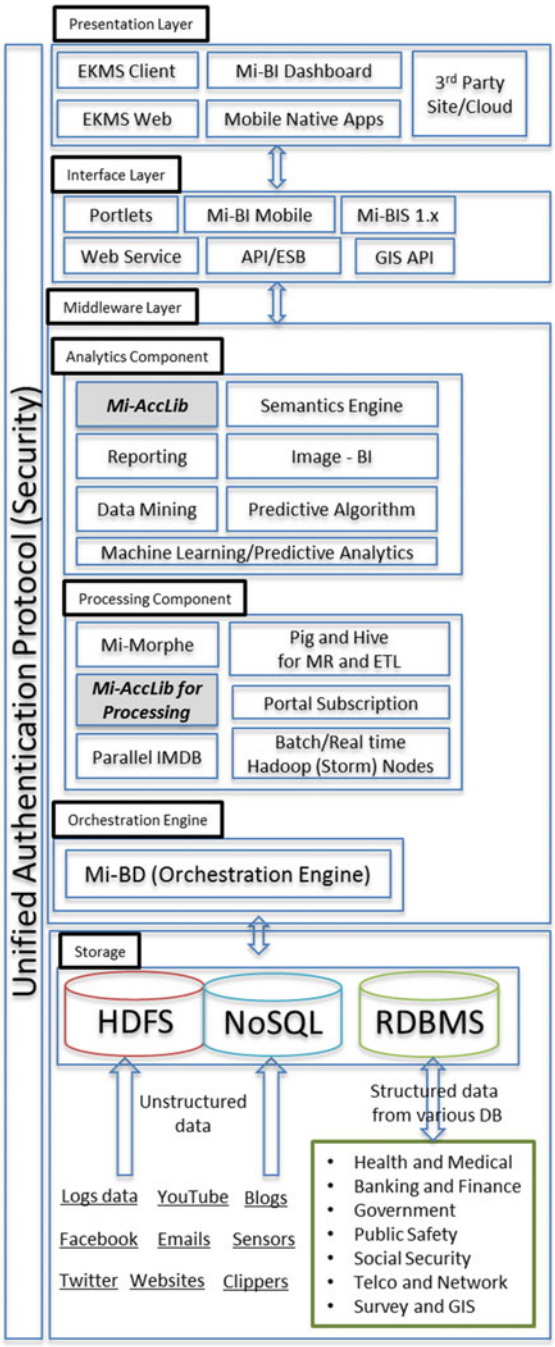

Fig. 12.1 Architecture of middleware framework

needs to be transferred from host (PC) processor to GPUs (device) for processing. Even though there are performance gains by using GPUs, functional-specific algorithms and application-specific algorithms exploiting GPU architecture need to be designed for optimum data processing. Thus, we have designed a set of library suites named as "MIMOS Accelerator Libraries" (Mi-AccLib) for various domain-specific (analytics component) and generic applications (processing component). These algorithms are categorized into different groups, namely, "Common Mi-AccLib," "Finance Mi-AccLib," "Text/String Mi-AccLib," "DB Mi-AccLib," etc. These libraries are designed and developed using Mi-AccLib framework such that the code can run seamlessly on different processor (GPU and multi-core for now) architectures exploiting underlying parallelization capabilities. The processed information in turn is fetched and displayed at presentation layer facilitated by interface layer (refer to Fig. 12.1).

In order to exploit current GPU computing capabilities for Mi-AccLib, we have to take into consideration the characteristics of the GPU and how it can cooperate with the CPU. One such consideration is the disparity of the computation capabilities between versions of the NVIDIA GPU cards. As such, a chunking and load balancing mechanism that splits and distributes data to different GPU cards in the system based on their computing capability has to be developed. Secondly, the I/O delays due to moving data to and from the hard disk, RAM, and GPU cards need to be considered when designing the framework in order to ensure that the overall system performance (multi-core CPU and GPU) is actually better by at least an order of magnitude compared to a multi-core CPU alone. Otherwise, there is no justification for multi-architecture development.

To meet these requirements, we designed the Mi-AccLib framework for multiple GPU support along with CPU synchronization. Our initial goal was to exploit GPUs for text-based processing and analytics work. In order to evaluate our middleware framework, we implemented one search and one sort algorithm for text processing on our framework and demonstrate how we can utilize these algorithms for data cleansing application. We then evaluate these algorithms against multi-core GPU versions.

Section 12.2 describes related works, while Sect. 12.3 details our Big Data framework and system architecture embracing our Mi-AccLib libraries via analytic and processing components. We outline our implementation in Sect. 12.4 and show the results of our algorithms on various different GPU cards in Sect. 12.5. Finally, we conclude in the last section.

12.2 Related Work

The MapReduce [5, 6] framework for distributed computing has been widely adopted in large-scale data processing. Mars [4] applies a flexible parallel programming in managing tasks partitioning and data distribution by using a GPU, which is an accelerated run-time system. However, Mars only works by distributing the data

set over streaming processors on a single GPU card. MapCG [7] provides source code level portability between CPU and GPU for a high-level programming model. Nevertheless, this implementation has scarified the usage of shared memory and constant memory in GPUs due to the compiler support issues. There are more MapReduce on GPUs implementation [8–10], yet, these systems have faced the overhead issues on data transfer and kernel launching issues. The proposed Big Data middleware framework using Mi-AccLib has a more macro-level data distribution orientation that works by chunking data into multiple GPU cards.

GPUMiner [1] is a parallel data mining framework for using GPUs for data mining work. It is composed of three parts – a storage and buffer management module, a visualization module, and a mining module. GPUMiner utilizes DirectX for visualization and CUDA for the data mining module. Chidchanok [11] works on an experimental framework for searching large Resource Description Framework (RDF) and performing the semantic query using JCUDA.[1] It takes advantage of GPUs parallel thread and block for retrieving, joining, and finding operations to the corresponding RDF graph. While the focus of these systems are on data mining, Mi-AccLib components utilize the GPU for a wide range of string processing functions including, but not limited to, data mining, analytics, and in-memory database like operations.

OpenAcc is a standard for the directives and programming model which has been developed by CAPS, Cray, The Portland Group, and NVIDIA [12]. There are two commercialized directive compilers integrating with NVIDIA NVCC compilers, which are CAPS (HMPP) and PGI (PGCC) [13] compilers. Directive-based high-level programming model is a simple and portable method to parallelize loops in C code. This intermediate high-level code is compiled by the NVCC compiler. Subsequently, it converts to a CUDA assembler source (PTX[2]) and optimizes the defined code. Then, it generates the final CUDA binary (a .cubin file). Ghosh, Liao, Calandra, and Chapman [14] evaluated the GPU directive compilers, which resulted 1.5× to 1.8× improvement in performance for both ISO and TTI kernels in single GPU against multi-core CPU by using OpenMP. In addition, they concluded this reduce efforts in code optimization with pragmas. Directive approach is complementary to Mi-AccLib rather than a competitor since it works on a different level of parallelization.

OpenCL [15] is a platform-independent standard for programming heterogeneous systems. OpenCL programs are compiled just in time for execution and can be used together with Mi-AccLib or other run-time libraries. These works [16–18] experienced a performance penalty on the NVIDIA GPU, due to the OpenCL abstraction layer. Thus, we have disabled OpenCL support as it is not optimized

[1] It is a CUDA binding for the Java language, which exploiting the features of NVIDIA GPU computing from Java-based applications.

[2] Parallel Thread Execution (PTX) is a pseudo-assembly language used in NVIDIA CUDA programming environment.

for GPUs at the moment, and real gains on GPUs can only be seen through optimized code as there are additional overheads from data movement.

CUDA [19, 20] or Compute Unified Device Architecture is NVIDIA's parallel computing architecture for their GPU cards. It is an intuitive and scalable programming model which is an extension of C [21]. Additionally, it provides the entire GPU platform accessing for developers. This architecture unifies the devices of CPU and GPU by performing a heterogeneous computation system [22]. It has rapidly evolved and scaling parallel performance since 2007. There are sets of libraries that are mostly for non-graphics-related processing. Mi-AccLib is built on CUDA for the GPU computation parts.

12.3 Middleware Framework Design

The proposed Big Data application framework in this chapter comprises of presentation layer, interface layer, middleware layer, and storage component. Middleware layer is further decomposed to analytics component, processing component, and orchestration engine. Mi-AccLib libraries are positioned at both analytics component and processing component. The **presentation layer** includes business intelligence (BI) dashboard and the **interface layer** includes MIMOS business intelligence suite (Mi-BIS 1.x API). Meanwhile, off-the-shelf technology will be used for the **storage layer**.

The following subsection describes these frame layers and middleware components in details, followed by specifically focusing on GPU-based solutions.

12.3.1 Big Data Needs

Digital data explosion has exceeded the petabytes and entered into the zettabyte era, based on IDC Digital Universe Study 2011 [23] as shown in Fig. 12.2. As of year 2011, as a society, we have generated and consumed ~1.8 zettabytes of data. But the question is, was the data analyzed for useful information in a timely manner for instantaneous usage? The ultimate value of a big data implementation will be judged based on one or more of these three criterions:

- Able to provide more useful information
- Able to improve the reliability of the information
- Able to improve the timeliness of the response

Thus, a Big Data application framework which meets the above three criteria is inevitable to provide reliable, useful, and timely information, enabling quick response by the data owner. Otherwise, Big Data is worthless.

Meeting the growing demand for Big Data processing, large-scale parallel processing for data mining and analytics has sparked innovative solutions both in

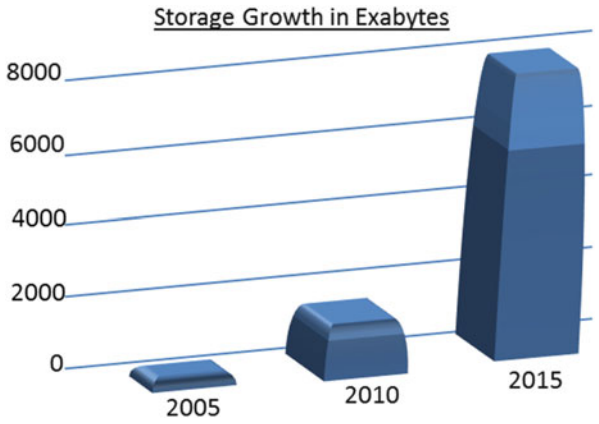

Fig. 12.2 Digital data growth in terms of storage size with forecasted data

commercial and scientific domain. Some of the commercial applications (e.g., Netezza, Teradata, Vertica) are computationally fast and cater for terabytes of data processing in milliseconds, however, relatively expensive to be used by small and medium enterprises. On the other hand, generally scientific communities rely on the MapReduce framework like applications such as Apache Hadoop which is free and stable for large-scale data processing. Following this open-source success, many applications are designed and developed in a parallel manner. Usage of parallel computing hardware such as GPGPU and Intel MIC (Many Integrated Core) coupled with parallel computing capability aware middleware/application can provide another less expensive approach of Big Data processing.

Companies like Intel and NVIDIA are on track to realize many-core and multi-core parallel computing hardware with increasing number of parallel cores. Intel MIC equips with 60 cores and 244 threads for hyper-threading. NVidia Tesla K20c offers 14 SMX (streaming multiprocessor extension) with 2496 CUDA cores. Both Intel and NVIDIA claim their processor is much faster compared to the others. The fact is that this competition is important for total paradigm shift in hardware enables parallel computing.

For example, Kepler architecture of Tesla series of NVIDIA GPU promises higher 1.3 teraflops (double precision), while Intel MIC Xeon Phi provides 1.2 teraflops with both having 6 GB memory for big data analytics. Streaming functionality enables seamless data movement between CPU and GPGPU for ultrahigh speed data processing. Leveraging the hardware technological capabilities, MIMOS is building various solutions including Mi-AccLib to enable ultra-speed big data processing with data and process parallelism. MIMOS Mi-AccLib libraries reside at both analytics component and processing component of the middleware, being part of the overall building block of the MIMOS Big Data (Mi-BD) processing solution.

12.3.2 Presentation Layer

The presentation layer is responsible to provide precise, concise, and simple visualization capability for the processed big data, enabling the user to make sense of the data in an informative manner. MIMOS business intelligence and Big Data framework provides client, web, dashboard, and native mobile solutions for easy representation of data for various audiences with visually comprehensible format. Usually the information is presented in chart and graph forms. It also allows easy integration of other 3rd-party software tools.

12.3.3 Interface Layer

This layer provides the interfacing between the middleware layer and the presentation layer. It caters for various types of application programming interface (API) toward presenting on various types of user interfaces, such as through business intelligence dashboards, where the orchestration engine interacts with Mi-BIS 1.x to output results through tabular, graphical, and charts display. At the same time, portlet, web service, ESB (enterprise service bus), and Mi-Mobile BIS outputs result on third-party Web sites, third-party cloud, and MIMOS web EKMS (MIMOS interactive dashboard) and also on native Windows applications and native Android/IOS applications for smartphones and tablets. GIS API is responsible for producing mapping results on web, MIMOS web, native client, and mobile display.

12.3.4 Middleware Layer

Middleware layer consists of orchestration engine, analytic component, and processing component. The orchestration engine is responsible to orchestrate the entire process from acquiring/ingesting data contained in the storage; processing data; providing the required analytic library requested by the user interface, in order to produce the desired results, which are mapped via API of the interface layer; and finally presenting results through web, mobile, native client, and cloud connections.

The analytics component, which would be utilized to handle selected types of analytics, algorithm, statistical analysis, and prediction depending on the required user needs, currently, consists of:

- Mi-Acclib (GPGPU libraries)
- Reporting and OLAP (online analytical processing) for business intelligence (BI)
- Data mining libraries
- Machine learning and predictive analytics

- Semantic engine
- Image BI for video/image analytics
- Predictive algorithm suites

The processing component, which handles data processing such as extracting, transforming, and loading (ETL) data from databases, before passing to analytics components or passing data directly to the API layer as per instructions from the orchestration engine, consists of:

- Mi-Morphe (MIMOS ETL tool)
- Mi-Acclib for fast processing
- Parallel in memory DB
- Pig and Hive for big data ETL
- Portal subscription
- Batch (Hadoop) and real-time (Storm [24] and Impala [25]) big data processing

Processed data before analysis and after analysis are stored in the storage components in the form of data warehouse for BI API to utilize. The orchestration engine will determine, based on user selection from the presentation layer or predefined configuration, which analytics model and processing model to be utilized. The orchestration engine will ingest incoming data (structured or unstructured) and pass the result to interface layer which will be mapped to the display channels, such as MIMOS dashboard (EKMS), BI dashboard, or native mobile application.

12.3.5 Orchestration Engine (with Example of Use Case)

As mentioned above, the orchestration engine plays a significant part to process, analyze, and send the processed data to the respective display channels via the appropriate interface API layers. Through the API on the interface layer, the orchestration engine could also call hybrid technologies. For example, orchestration engine may access libraries with various algorithms for preconfigured purposes such as utilizing GPGPU libraries to edit distance algorithm and using Pig with Hadoop to perform MapReduce function and processing structured data from RDBMS (relational data bases management system) data while grouping the unstructured result within the same system to achieve the aggregated task.

The orchestration engine could very well be used for different scenarios. The following paragraph explains the implementation of Mi-BIS for video analytics (or image analytics) as an example, where the purpose is to identify the various types of events such as "event detected," "face detected," and "motion detected" per given camera location for given time stamp information as reported by the video analytics system. This data when analyzed on real time requires real-time processing speed, volume, and complexity of aggregating from other structured database tables such as listing the names of guards in charge during the occurrence

of certain type of events for a period of 1 year pertaining to the camera locations. This would help the security companies or public safety organization to place their staff at strategic locations within their planned coverage area for patrolling, based on predictive analytics. Thus, probability of similar occurrence could be observed within certain time frame. In terms of implementation, the sources of video files from all the cameras being monitored are stored in HDFS of Hadoop nodes. Mi-BD (MIMOS Big Data) will sqoop in the files based on schedule time with the various types of events detected and performs ETL using Pig/Hive. Hadoop is used here for batch MapReduce processing, where the results would be stored in the storage component for later date. When a business user logs in through the display channel of Mi-BIS dashboard, the user could construct the required dimensions to view the report chart such as events, day, month, year, personnel names, and camera locations from the various data sources (structured or unstructured as stored in the storage components). Mi-BIS presentation/display layer will communicate with Mi-BD via the interface layer; Mi-BIS 1.x. Mi-BD will determine if this request is for real time or could utilize any previously stored preprocessed data. If real-time big data (from large video files of many camera sources) is required, then Storm would be used. In this example, the batch Hadoop ETL data that was pre-produced earlier is utilized by the BI dashboard in order to view the relationship in a bar chart (or tabular format) between time of year, type of events, camera locations, and guards in charge. The user can view an image and an 8-s video clip of the event (6 s before and 2 s after the event) when each individual event is clicked. If the user decides to open any statistical analysis tools, then Mi-BD, the orchestration engine, would be responsible to call machine learning/predictive analysis libraries or Mi-Acclib to drill down on the statistics of occurrences and pass the result back to the presentation layer via API from the interface layer.

12.3.6 Storage

The storage component consists of storage for structured and unstructured data. RDBMS such as SQL, MySQL, and Postgres supports the storage of structured data, while HDFS (Hadoop Distributed File System) and NonSQL such as Mongo DB or Cassandra could support unstructured data such as live feed from Twitter, Facebook, and log files. The storage component also store processed and analyzed data in the storage or as data warehouse in order to be used by API to display results in the various display channels and devices. In the next section, Mi-Acclib as an analytics component with GPGPU libraries is further elaborated.

12.3.7 Mi-AccLib and Analytics Component

We designed the Mi-AccLib framework to be modular in order for it to be extensible. Mi-AccLib is divided into two layers, which are an application-specific layer (analytic component) and a functional algorithm layer (processing component) as shown in Fig. 12.3. The Mi-AccLib framework is built to run on top of different processor architectures. One of the challenges of such a system is the need to support libraries written on different languages for different architectures. However, we focus on our work with GPU in this chapter.

The library interface wrapper layer provides a common interface for users to utilize functions that have been implemented for various hardware processors and coprocessors. For example, a search function can be used on either a GPU card or a multi-core CPU card or on both as the user requires. The functions exposed to the users at this layer share a common format as shown in Fig. 12.4.

The function takes in a variable number of parameters. All these parameters provide users with a fine-grained level of control when executing tasks. However, they can also leave the parameters to the default value for the framework to determine the best parameters for performance based on the available resources, function profiling, and data size.

The application-specific libraries (analytic component) are a set of basic functions that have been linked together to perform a certain task. An example of such a

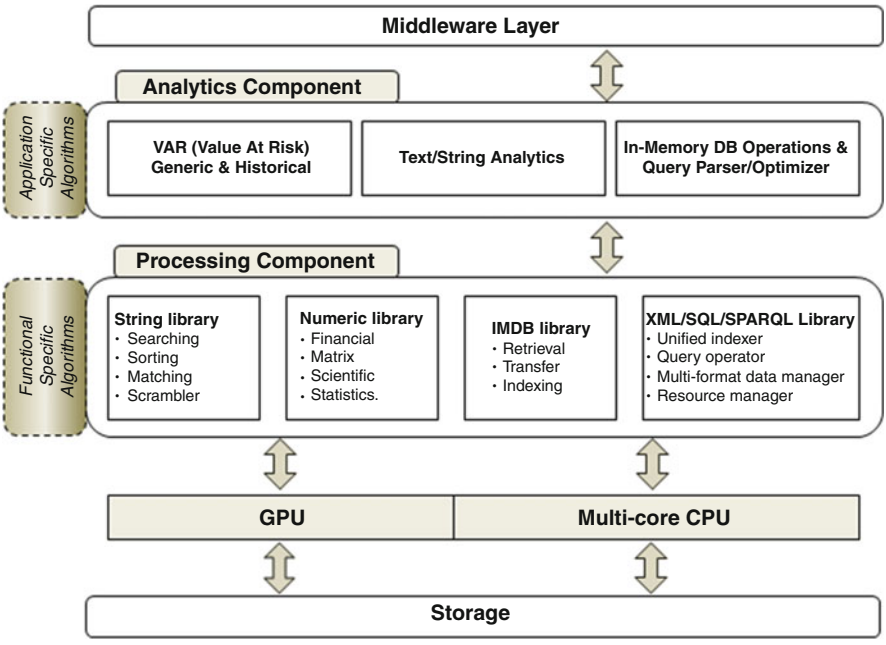

Fig. 12.3 Mi-AccLib framework architecture

```
miacclibError_m MiAccLib::MiLoad(
     char*                      filename,       /// specifies the input data file path
     miacclibComputePlatform_m  platform,       /// specifies the computing architecture
     migpuaccMode_m             gpumode,        /// specifies the GPU execution model
     unsigned int*              retTotalRows,   /// returns total of rows
     unsigned long long*        retTotalBytes)  /// returns total of bytes
{
     ...
}

miacclibError_m MiAccLib::MiExactKeywordSearch(
     unsigned char*  keyword,           /// specifies keyword to match
     unsigned char*  retResult)         /// returns results
{
     ...
}

miacclibError_m MiAccLib::MiEditDistanceKeywordSearch(
     unsigned char*  keyword,           /// specifies keyword to match
     unsigned char*  approval_level,    /// specifies edit distance approval level
     unsigned char*  retResult)         /// returns results
{
     ...
}
```

Fig. 12.4 Format of function

task is a financial calculation of value at risk (VAR) [26]. The VAR application-specific library (finance Mi-AccLib) takes in a set of data and first preprocesses the data for sorting using a preprocessing kernel (processing component) (e.g., changing floating point numbers to unsigned integers), sorts the preprocessed data using a sorting kernel, sends the sorted data to a percentile kernel to obtain the result, and converts the final result back into a floating point value.

The functional algorithm libraries (processing component) are a collection of kernels that perform tasks in similar areas and are the basic building blocks for the library framework. For example, the string processing library contains a set of different search algorithms that are exposed through the library wrapper interface. Each search algorithm gives a different set of performance that users can try to use for different application purposes. Sets of kernels from different functional algorithm libraries can also be integrated to perform a larger task by the application-specific libraries (analytic component).

In order to achieve an overall improvement of the whole system rather than emphasizing just faster execution of parts of the workflow, a holistic view needs to be taken during system architecture design. This is especially true when considering latency for I/O and load balancing for data distribution to GPUs of varying capabilities.

The first step is to decide which parts of the workflow are more suitable to be executed on the CPU or GPU. Typically, functions that require a lot of calculations that can be parallelized or have a lot of uncorrelated data to be processed are suitable for execution on the GPU. For example, matching data from columns of two different tables is very suitable for GPU processing as each entry of a column can be compared independently from any other entry. This allows the system to leverage on the parallel nature of the GPUs.

The second step is to determine how to order the data transfer and processing so that latency can be hidden. Mi-AccLib provides methods to split data into chunks that are readily transferrable to multiple GPU cards based on their available

memory and computing power profiles. This ensures the data is distributed and processed optimally so that delays are kept to a minimum. Data chunking also needs to be done in a manner that allows the data structure to be preserved so that each chunk can be processed individually. The data in the chunks will normally need to be converted into a structure of arrays for faster processing. This is identical to the normal column format in databases, so data will need to be transposed when copied to the GPU.

12.4 Implementation

We have currently implemented a few different functional algorithm libraries into our framework. We will discuss some of the implementation details and challenges in this section.

One of the biggest limitations to using GPU cards for text processing is the large amounts of data that must be moved through the PCI-e bus to and from the GPU and also the reading and writing of data from the hard disk, which is five times or more slower than the PCI-e bus. Streaming is the method commonly used to hide or minimize the data transfer latency, where data is sent and received in small chunks using direct memory access (DMA) methods in the background, while data processing is performed on the chunks already received by the GPU.

To enhance the parallel processing of data and memory transfer, profiling of kernels needs to be performed beforehand to determine the duration of the kernel execution time. Based on the kernel execution time, the size of the data chunks used can be determined using the PCI-e data transfer speed as well as the kernel execution time. On initial observation, it may seem that chunking data to the minimum size chunk would seem like the logical choice to minimize overall delay as that will give the most overlap between data transfer and data processing. However, the transfers of many small chunks give rise to additional overhead time between data transfers. For example, the transfer of a single chunk size of 64 MB gives the highest data transfer throughput, but multiple transfers of 64 MB chunks incur a 33 % overhead on that of a single chunk transfer as we observed.

Getting outputs from kernel execution on the GPU is another trade-off issue that has to be considered during implementation. For example, an algorithm returning multiple search results in an array needs to ensure that the global variable that serves as the index of the array is not accessed simultaneously by multiple threads. Since there is no concept of critical sections in CUDA, a mutex must be implemented using the "atomicCAS" instruction to allow CUDA cores to lock the variable for reading and incrementing before releasing it for other cores.

While this works well if there are only a few results to be returned, searches with many results will cause many threads to be executed serially for this section. Besides this, atomic instructions in CUDA are slower than other instructions since they need to access global memory every time a read and write is performed,

and a timeout may occur if too many threads in the same warp try to lock the same global variable.

A trade-off using additional memory can be done by allocating an array of bytes or bits equivalent to the size of search data. Each thread can mark the equivalent byte or bit in the result array without the need for a mutex. However, this requires a much larger memory allocation for the output and a larger delay when moving the results back to the CPU.

As an alternative HPC solution to GPU, we have implemented Big Data applications using the proposed middleware framework. In this experiment, we have incorporated a sample scenario using the presentation layer (Mi-BI dashboard), via Mi-BiS 1.x (API and connectors), middleware, storage, and the orchestration engine with the MapReduce functions managing the nodes. We have configured seven virtual machines with one master node (8 cores) and six worker nodes (4 cores each) running on a few of HP DL380p G8 servers installed with Apache Hadoop, Cloudera's Hadoop, and Impala. We have also installed Postgres on another same model of HP server with 8 GB RAM with 4 cores and another high-end HP machine with 96 GB RAM and 48 cores. At the same time, we have installed a GPU card, NVIDIA Tesla K20c, on a DELL Precision T5500 workstation with Windows Server 2008 R2 Enterprise SP1 64-bit operating system, running on Intel Xeon E5630@2.53 GHz processor, with 12GB RAM, 1 TB SATA hard drive (7,200 rpm). The Kepler GK110 GPU card provides 2496 CUDA cores. Next, we have written scripts within the orchestration engine to import ~120 million records from four significant tables of inpatient record table, state code names table, disease code names, and age groups from the hospital database, originally residing in Postgres database of our high-end physical server, into the storage of the Hadoop clusters which is in HDFS (Hadoop distributed file storage) format. Through our libraries in the middleware framework, our ETL processing component uses Hive to extract, cleanse, transform, and load the dataset to be accessed for the new data warehouse. Mi-BiS 1.x uses the cleansed data from the warehouse using connectors (API that had been developed for the interface), given by the business user, when logged on through BI dashboard. For example, the user could request for information such as the type of disease and age group distribution on a pie chart or view the trend of the selected disease for the duration of several years. Mi-BIS dashboard creates the reports for the types of query (or SQL select statements) for the different API/connectors. The processing time has been recorded with at least five trials for each of the different setup. The results of the average processing time are shown in Table 12.1.

The orchestration engine is responsible to interlink the storage layer toward the self-service report creation from the Mi-BIS dashboard or running the real-time analytics via the dashboard. Upon identification of large data request, the predefined orchestration engine will use HDFS and run the search either in Impala, GPU, or by combining the queries as hybrid parallel process and presenting the output to the Mi-BIS dashboard.

Table 12.1 Comparison of processing time for types of search query using RDBMS, Big Data Hadoop/Impala nodes, and GPU parallel DB

No	Description of search query vs. average processing time (seconds)	SQL (8GB/ 4Core)	SQL (96GB/ 48Core)	Hadoop-Hive	Impala-Hive	GPU-parallel DB
1	Selecting sum from one column of 120 million records	1,466.7 s	218.7 s	347.6 s	3.7 s	0.3 s
2	Selecting a name column, counting the name and ordering by top 10 names	7,901 s	1612s	505 s	64.2 s	NA
3	Selecting state code, years from hospital patient records with one disease code selected, group by years and state code, order by years and state code	1,464.7 s	103.6 s	383.5 s	3.5 s	3 s
4	Selecting state, years, disease name from hospital patient records where one disease name type is selected and joining disease code with disease name and state code with state names; grouping by years, states, and disease names; ordering by years and state code	1,688.7 s	102.7 s	N/A	2.9 s	1.6 s
5	Inserting the results of selecting state code, years from hospital patient records with ALL disease type, group by years and state code, order by years and state code	Failed	7,878 s	557.3 s	10.1 s	N/A
6	Selecting state, years, disease name from hospital patient records where three disease name types are selected and joining disease type with disease name and state code with state names; grouping by years, states, and disease names; ordering by years, states, and disease names	1,893s	704 s	N/A	3.7 s	6.3 s

12.5 Results

Two of the most important features of text processing sorting and matching of processing component and edit distance of analytics component are explored. In this section, we discuss the results from our implementations of the string matching in our Mi-AccLib framework, while results of edit distance operations are illustrated for various configurations. Finally, we present the results of experiment based on the different setups using the middleware framework.

We implemented a string matching algorithm, which matches all the characters in a keyword to a string from a large text file. This search is O(n) in complexity,

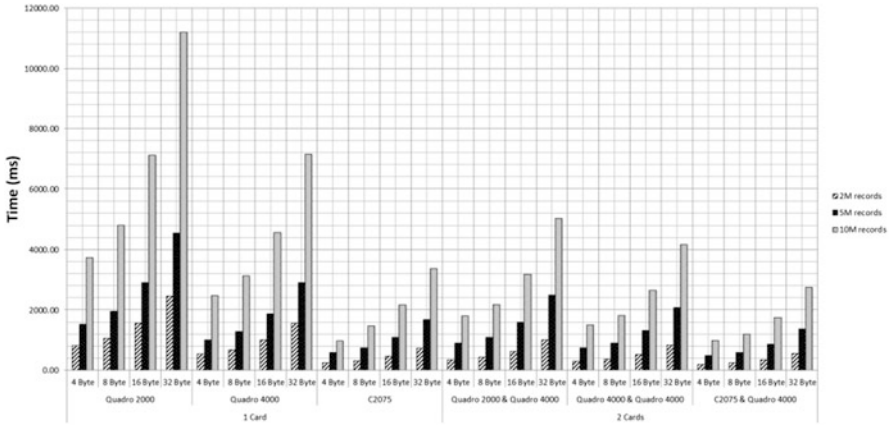

Fig. 12.5 Performance of single and dual GPU cards for string matching

where n is the number of characters to be searched. To speed up matching, we copy chunks of the text into the shared memory of the GPU, which is much faster than accessing global memory. Then, each thread in a warp searches for the keyword at different points of the text. The same search is repeated for other chunks of memory at other streaming multiprocessors throughout the GPU. The results of the search are returned either in array size of the search text or in arrays of integers pointing to the positions of the characters in the search text, as detailed in the previous section.

From our results in Fig. 12.5, we can see that the algorithm scales well according to the number of CUDA cores and shading processor speeds of the GPUs. For example, the Tesla C2075 GPU is three times as fast as the Quadro 2000 due to having almost three times as many CUDA cores as well as having a much higher memory bandwidth between the global memory and the shared memory. This allows it to complete memory-intensive jobs, such as matching and sorting, much faster than the Quadro 2000.

When we distribute the matching load between two cards, the throughput for the searches is slightly lower than the sum of the throughput of each card individually. The reason for this is mainly due to the overhead of distributing data to two separate cards on the same PCI-e bus. For example, performing a search for a 4-byte keyword from two million characters takes an average of 275.85 ms on a C2075 and 545.84 ms on a Quadro 4000, but distributed matching on a Quadro 4000 and C2075 takes only 198.80 ms. This gives us a throughput of 7.25 MB/s and 3.66 MB/s for the C2075 and Quadro 4000, respectively, and a combined throughput of 10.91 MB/s. However, the distributed matching on both cards gives a throughput of 10.06 MB/s.

We also implemented Levenshtein distance (edit distance) matching on CPU and GPU. This parallel version of edit distance feature is part of the analytic component in Mi-AccLib. The following explanation describes how application-specific analytic component and functional-specific processing components are

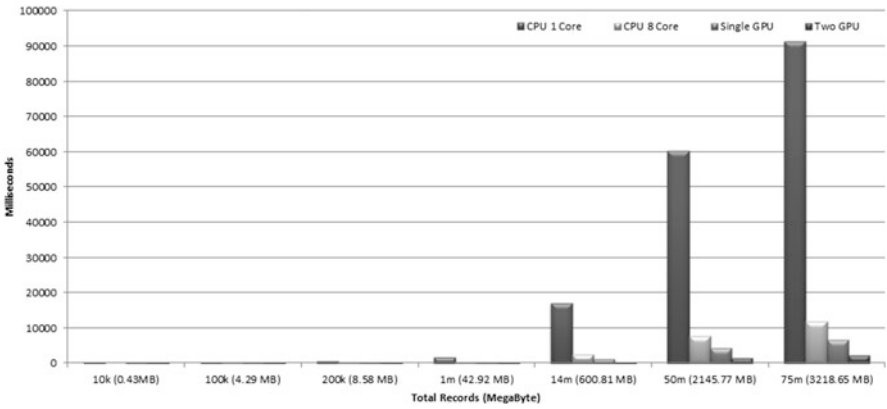

Fig. 12.6 Edit distance result for CPU and GPUs

used for data cleansing example using the middleware framework. It is certain these results prove GPU-based solution is an alternative to existing MapReduce-like application for Big Data processing.

It is defined to be the smallest number of edit operation (insertions, deletions, and substitutions) required to change one string into another [27]. Figure 12.6 shows one-to-many matching of execution time versus total records for the edit distance algorithm by utilizing CPU cores and GPUs (C2075). Single GPU and dual GPUs outperform the CPU multiple cores for processing much larger size. The speedup (CPU time/GPU time) on single CPU cores is 13.9×, and the 8 cores CPU is 1.78× for processing 75 million of records (3.14 GB). There is more speedup by utilizing dual GPUs, as utilizing the single CPU core is as high as 38.34×, and the 8 cores CPU is 4.92×. By comparing the speedup of single and dual GPUs processing, there is ~2.7× for the size of records from 14 to 75 million of records.

An application that we put together using sort and search was a data cleansing application project. In this project, we compared the national registration identification (NRIC) numbers from a database of 14 million records, which we call database A, against a clean set with 13 million records, which we call database B. Once the number matches, the kernel compares the names associated with the identification numbers at each table to confirm the match.

For this application, we first extracted the data from the two databases. Then we developed a kernel that first performs a sort on database B using the identification numbers as keywords. The identification numbers vary in length based on whether they are old, new, or army identification numbers. Then, for each record in database A, we search through database B for a match. We use a binary search algorithm to perform the search, and the brute force algorithm to perform the match. The binary search algorithm takes an average of O(logN − 1) to find if there is a match. After all the records from database A have been iterated through, we use the results to perform a brute force name matching from database A to database B. The total time taken using the GPU kernel for sorting and matching alone is below 15 s

compared to the time of over 12 h on an Intel Xeon Quad Core E5620 running MySQL. The total time including data extraction from the databases and preprocessing was below 5 min for the GPU processing.

The results are shown as in Table 12.1 for six types of searches for the different environment setups using our middleware framework including the Mi-BIS presentation layer.

The average time for each query type was analyzed for Postgres database with 8 GB server and 96 GB server, Hadoop with Hive (7 nodes), Impala with Hive (7 nodes), and GPU server processing in order to compare the processing time (in seconds). The result shows the performance comparison of various setups for real-time analytics processing of Big Data in the health sector, using Mi-BIS presentation dashboard to analyze ~120 millions of records in HDFS and Postgres (RDBMS) servers. Some of the results are reported as NA (not available) because the work is still in progress.

GPU parallel DB processing takes the shortest time to process ~120 million of records and the cost is also cheaper than implementing 7 nodes of Hadoop or SQL on Postgres (medium and high-performance fully tuned Postgres database server). SQL was not able to compute on big data especially for real-time analytics. It even failed for insertion of output data in the same database. Hadoop with Hive is not suitable for real-time processing and would only be useful for batch processing of big data. Impala-Hive is as beneficial as GPU for general queries and could be used to complement in the hybrid parallel processing approach especially led by the orchestration engine with predefined rules within the scripts, developed for the middleware layer. Impala-Hive is faster compared to GPU parallel DB when there are multiple tables to be joined and with huge strings operations to be performed.

Conclusion

We have presented a middleware framework for big data processing using data cleansing as an example application. It is certain the above results prove GPU-based solution is an alternative to existing MapReduce-like application for Big Data processing. Our layered middleware framework approach with GPU capable analytic and processing components has facilitated seamless integration of our Mi-AccLib. It allows users to exploit the powers of the GPU by providing the ability for efficient work distribution across multiple GPUs with regard to I/O access and load balancing. Using the Mi-AccLib framework, we implemented and tested radix sort and string matching algorithms on single and multiple GPU cards as part of processing component. On the other hand, the edit distance algorithm as part of analytic component used underlying processing component functionalities for application-specific needs. Our results show a significant improvement by using two GPU cards over single GPU cards and single GPU cards over multi-core CPUs for text data sorting, matching, and cleansing. The performance of the GPU

(continued)

implementation for data cleansing shows a speedup of over two orders of magnitude over the same operation done in MySQL on a multi-core machine. The proposed middleware framework can perform real-time analytical queries using the hybrid Impala and GPU libraries of ~120 million records for the selected hospital database, within less than 11 s.

Acknowledgement This research was done under joint lab of "NVIDIA-HP-MIMOS GPU R&D and Solution Center." This is the first GPU solution center in Southeast Asia established in October 2012. Funding for the work came from MOSTI, Malaysia. The authors would like to thank Prof. Simon See and Pradeep Gupta from NVIDIA for their support.

References

1. Fang, W., et al.: Parallel data mining on graphics processors. Technical Report (2008)
2. Gregg, C., Hazelwood, K.: Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 134–144 (2011). doi:10.1109/ISPASS.2011.5762730
3. Bakkum, P., Skadron, K.: Accelerating SQL database operations on a GPU with CUD. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp. 94–103. New York, NY: ACM (2010). ISBN: 978-1-60558-935-0, doi:10.1145/1735688.1735706
4. He, B., et al.: Mars: a MapReduce framework on graphics processors. In: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, pp. 260–269. New York, NY: ACM (2008). ISBN: 978-1-60558-282-5, doi:10.1145/1454115.1454152.
5. Dean, J., Ghemawat, S. MapReduce: simplified data processing on large clusters. Communications of the ACM, vol. 51, pp. 107–113. New York, NY: ACM (2008). ISSN: 0001-0782, doi:10.1145/1327452.1327492
6. Wolfe Gordon, A., Lu, P.: Elastic phoenix: Malleable MapReduce for shared-memory systems. In: Altman, E., Shi, W. (eds.) Network and Parallel Computing, vol. 6985, pp. 1–16. Springer, Heidelberg (2011)
7. Hong, C., et al.: MapCG: writing parallel program portable between CPU and GPU. In: Proceedings of the 19th international conference on Parallel architectures and compilation techniques, pp. 217–226. New York, NY: ACM (2010). ISBN: 978-1-4503-0178-7, doi:10.1145/1854273.1854303
8. Shirahata, K., Sato, H., Matsuoka, S.: Hybrid map task scheduling for GPU-based heterogeneous clusters. In: IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp. 733–740 (2010). doi:10.1109/CloudCom.2010.55
9. Stuart, J. A., Owens, J. D.: Multi-GPU MapReduce on GPU clusters. IEEE Computer Society. In: Proceedings of the 2011 I.E. International Parallel & Distributed Processing Symposium, pp. 1068–1079. Washington, DC. (2011). ISBN: 978-0-7695-4385-7, doi:10.1109/IPDPS.2011.102
10. Catanzaro, B., Sundaram, N., Keutzer, K.: A map reduce framework for programming graphics processors. In: Third Workshop on Software Tools for MultiCore Systems (STMCS) (2008)

11. Choksuchat, C., Chantrapornchai, C.: Experimental framework for searching large RDF on GPUs based on key-value storage. In: 10th International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 171–176 (2013). doi:10.1109/JCSSE.2013.6567340
12. NVIDIA Corporation. OpenACC. https://developer.nvidia.com/openacc (2011). Accessed 4 Aug 2013
13. Wolfe, M.: Implementing the PGI accelerator model. New York, NY: ACM. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp. 43–50 (2010). ISBN: 978-1-60558-935-0, doi:10.1145/1735688.1735697
14. Ghosh, S., et al.: Experiences with OpenMP, PGI, HMPP and OpenACC directives on ISO/TTI Kernels. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, pp. 691–700 (2012). doi:10.1109/SC.Companion.2012.95
15. Munshi, A.: The OpenCL specification. Khronos OpenCL Working Group. Technical Report (2009)
16. Torres, Y., Gonzalez-Escribano, A., Llanos, D.R.: Using fermi architecture knowledge to speed up CUDA and OpenCL programs. In: IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp. 617–624 (2012). doi:10.1109/ISPA.2012.92
17. Wezowicz, M., Taufer, M.: On the cost of a general GPU framework: the strange case of CUDA 4.0 vs. CUDA 5.0. In: High Performance Computing, Networking, Storage and Analysis (SCC), SC Companion, pp. 1535–1536 (2012). doi:10.1109/SC.Companion.2012.310
18. Shen, J., et al.: Performance traps in OpenCL for CPUs. In: 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 38–45 (2013). ISSN: 1066-6192, doi:10.1109/PDP.2013.16
19. NVIDIA Corporation.: CUDA C Programming Guide. s.l. NVIDIA Corporation (2012)
20. Sanders, J., Kandrot, E.: CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional. (2010). ISBN: 0131387685
21. Wilt, N.: CUDA handbook: a comprehensive guide to GPU programming. Addison-Wesley Professional, (2013). ISBN: 0321809467
22. Kirk, D.B., Hwu, W-m.W.: Programming massively parallel processors, second edition: a hands-on approach. Morgan Kaufmann, Burlington, MA (2012). ISBN: 0124159923
23. Hollis, C.: IDC digital universe study: Big data is here, now what? http://chucksblog.emc.com/chucks_blog/2011/06/2011-idc-digital-universe-study-big-data-is-here-now-what.html (2011). Accessed 18 July 2013
24. Storm—Distributed and fault-tolerant realtime computation. http://storm-project.net/ (2011). Accessed 10 Aug 2013
25. Impala—The platform for big data. http://www.cloudera.com/ (2013). Accessed 10 Aug 2013
26. Holton, G.A.: Value at risk: theory and practice. Academic Press, Amsterdam (2003). ISBN: 0123540100
27. Navarro, G. A guided tour to approximate string matching. ACM computing surveys, vol. 33, pp. 31–88. New York, NY: ACM. (2001). ISSN: 0360-0300, doi:10.1145/375360.375365