Overview

Accelerating data mining workloads: current approaches and future challenges in system architecture design

Alok N. Choudhary,* Daniel Honbo, Prabhat Kumar, Berkin Ozisikyilmaz, Sanchit Misra and Gokhan Memik

Conventional systems based on general-purpose processors cannot keep pace with the exponential increase in the generation and collection of data. It is therefore important to explore alternative architectures that can provide the computational capabilities required to analyze ever-growing datasets. Programmable graphics processing units (GPUs) offer computational capabilities that surpass even high-end multi-core central processing units (CPUs), making them well-suited for floating-point- or integer-intensive and data parallel operations. Field-programmable gate arrays (FPGAs), which can be reconfigured to implement an arbitrary circuit, provide the capability to specify a customized datapath for any task. The multiple granularities of parallelism offered by FPGA architectures, as well as their high internal bandwidth, make them suitable for low complexity parallel computations. GPUs and FPGAs can serve as coprocessors for data mining applications, allowing the CPU to offload computationally intensive tasks for faster processing. Experiments have shown that heterogeneous architectures employing GPUs or FPGAs can result in significant application speedups over homogenous CPU-based systems, while increasing performance per watt. © 2011 John Wiley & Sons, Inc. *WIREs Data Mining Knowl Discov* 2011 1 41–54 DOI: 10.1002/widm.9

DATA MINING AND DOMAIN-SPECIFIC ARCHITECTURES

Latest trends indicate beginning of a new era in data analysis and information extraction. Today's 'connect anytime and anywhere' society based on the use of digital technologies is fueling data growth, which is doubling every two months (if not faster), akin to 'Moore's law for data'.[1] This growth is transforming the way business-, science-, and digital technology-based world function. Various businesses are collecting vast amounts of data to make forecasts and intelligent decisions about future directions. The world's largest commercial databases are over the 100 TB mark, whereas the database sizes on hybrid systems are approaching the PB mark.[2] Some of these large databases are growing by a factor of 20 every year. In addition, millions of users on the Internet are making data available for others to access. Countless libraries and databases containing photographs, movies, songs, etc., are available to a common user. In addition to the increasing amount of available data, other factors make the problem of information extraction particularly complicated. First, users ask for more information to be extracted from their datasets, which requires increasingly complicated algorithms. Second, in many cases, the analysis needs to be done in real-time to reap the actual benefits. For instance, a security expert would strive for real-time analysis of the streaming video and audio data in conjunction. Managing and performing run-time analysis on such datasets is appearing to be the next big challenge in computing.

Data mining is the process of automated extraction of predictive information from large datasets. With the ever-increasing computational demands, current state-of-the-art central processing unit (CPU) systems fail to keep pace. This performance gap arises due to the fact that CPUs devote lots of resources to keep relatively small number of execution units busy, which is great for sequential code. Further, to mitigate the effects of nonuniform memory

*Correspondence to: chhoudhar@eecs.northwestern.edu
Northwestern University. http://www.eece.northwestern.edu
DOI: 10.1002/widm.9

access, conventional CPUs reserve a large percentage of the silicon resource for caches. However, for applications that have a very structured memory access pattern, an efficient design would call for a larger percentage of the total available resources to be dedicated for processing. These limitations of the current CPU systems have led application developers to explore new architectures for improving performance. Recent technological advancements have witnessed domain-specific architectures penetrating the realm of general-purpose computing. Graphics processing units (GPUs), which were initially introduced for the task of texture mapping and rendering, have considerably increased in complexity. GPUs devote more silicon 'real estate' to data processing rather than data caching and flow control. They also have become much more programmable compared to the previous generation of fixed-pipeline graphics units. Similarly, field-programmable gate arrays (FPGAs) came into existence as a prototyping testbed to reduce the time-to-development for application-specific integrated circuit (ASIC) chips. However, the rapid progress in technology overcame the speed, size, and power limitations of the previous generation FPGAs and presented them as a system that offers the flexibility of on-the-fly reconfiguration. Exposing the lowest level on silicon, these devices equip system architects to utilize them as powerful computational engines. The future era will keep producing new architectures that will have characteristics catering to a subset of the vast domain of applications. This heterogeneous pool of architectures presents an exciting opportunity for designers to reinvent the concept of high-performance application development. Data mining applications are computationally expensive and have a structured memory access pattern that makes them a suitable candidate for utilizing these architectures as high performance coprocessors.

CHARACTERIZING DATA MINING APPLICATIONS

What makes the data mining applications unique is a mix of high data rates combined together with high computation power requirements. Typically, data mining applications oscillate between such phases regularly. Previous studies prove that current processors and architectural optimizations need to be enhanced further in order to handle such unique data-intensive applications.[3–5] In addition, the tremendous increase in data collection adds to the problem. As a result, the gap between the expected performance for data mining applications and the delivered performance of processor architectures is bound to get worse. This problem can be alleviated if current computer architectures are optimized or redesigned to accommodate data mining applications. A data mining benchmark called MineBench,[6] designed for enabling research in data mining architectures, provides insight for the directions of such optimizations.

The architectural characteristics of data mining applications share resemblance to each other and are markedly different from applications that fall under the domain of compute-intensive, multimedia, streaming, and database applications. Table 1 shows the classification of data mining applications.

TABLE 1 | Applications in MineBench

Algorithms	Category	Description
k-Means	Clustering	Mean-based data partitioning method
Fuzzy *k*-Means	Clustering	Fuzzy-logic based data partitioning method
BIRCH	Clustering	Hierarchical data segmentation method
HOP	Clustering	Density-based grouping method
Naive Bayesian	Classification	Statistical classifier
ScalParC	Classification	Decision tree based classifier
Apriori	ARM	Horizontal database, level-wise mining based on Apriori property
Eclat	ARM	Vertical database, equivalence class based method
SNP	Bayesian network	Hill-climbing search method for DNA dependency extraction
GeneNet	Bayesian network	Microarray-based structure learning for gene relationship extraction
SEMPHY	Expectation maximization	Phylogenetic tree based structure learning method for gene sequencing
Rsearch	Pattern recognition	Stochastic context-free grammar-based RNA sequence search method
SVM-RFE	Support vector machines	Recursive feature elimination based gene expression classifier
PLSA	Dynamic programming	Smith–Waterman optimization method for DNA sequence alignment
Utility	ARM	Utility-based association rule mining

TABLE 2 | Architectural Characteristics of Various Benchmark Suites

	Benchmark Suite				
Parameter†	SPEC-INT	SPEC-FP	MediaBench	TPC-H	MineBench
Data references	0.8071	0.5502	0.5676	0.4831	1.1032
Bus accesses	0.0303	0.0344	0.0027	0.0104	0.0371
Instruction fetches	1.6427	1.6325	1.0000	1.3193	2.7247
ALU operations	0.2550	0.2920	0.2650	0.3000	0.3080
Cycles	1.7583	1.7077	1.1185	1.3625	1.4724
L1 misses	0.1789	0.2107	0.0472	0.2965	0.3976
L2 misses	0.0013	0.0019	0.0003	0.0019	0.0056

A comparative study presented in Table 2 shows the core differences between MineBench and other application benchmarks: SPEC-INT, SPEC-FP[7] (benchmark suite for processor manufacturers), MediaBench,[8] and TPC-H[9] (benchmark suite for media and communication processors). One key attribute that signifies the differences is the number of data references per instruction retired. For data mining applications, this rate is 1.103, whereas for other applications, it is significantly less. In Table 2, the number of bus accesses originating from the processor to the memory (per instruction retired) verifies this fact as well. These results reinforce the hypotheses that data mining is data-intensive by nature. Another important difference is the fraction of total instruction fetches to the instructions retired. This measure, instruction fetches in Table 2, includes the noncached instruction fetches, branch prediction fetches, and also those fetched as a result of wrong predictions. Besides, the number of arithmetic logic unit (ALU) (computation) operations per instruction retired is also surprisingly high for data mining applications, which indicates the extensive amount of computations performed in data mining applications. What really makes the data mining applications unique is this combination of high data rates with high computation requirements.

Traditional CPU architectures are unable to provide very scalable performance to data mining applications owing to their limited computation power and memory bandwidth. The focus has thus shifted to parallel multithreaded architectures such as the recently introduced GPUs by NVIDIA (GeForce; Santa Clara, CA, USA) and ATI (Radeon; Sunnyvale, CA, USA), and FPGAs. GPUs devote a large fraction of their resources for computations and hence can provide orders of magnitude higher performance. Higher memory bandwidth results in further improvements. On the other hand, FPGAs can be customized 'on-the-fly' depending on the application requirements and hence can challenge GPUs with better computational power, but they lack the memory bandwidth provided by the GPUs. Because of their unique capabilities, both these architectures can be utilized to enhance the overall performance (in terms of output quality, scalability, and execution times) of data mining applications compared to traditional CPUs. The following sections dive in details of these architectures and present their potential benefits in the data mining domain.

KEY COMPUTATIONAL BLOCKS IN DATA MINING—IMPORTANCE OF KERNEL

Data mining applications have several similarities with streaming applications because a consistently changing set of data is read for processing. But they are different from pure streaming applications by the fact that there are bursts of streaming data instead of data arriving at a consistent arrival rate. Figure 1 shows a generic data flow in such an application. These applications, therefore, can be characterized as multiphase, with each phase consisting of one or more kernels. These kernels form pieces of core operations, for example, histogram, distance calculation, correlations, tree-search, etc. These phases repeat many times, and the data that are consumed in each phase may change their execution characteristics. In other words, kernel is defined to be an abstract representation of a set of operations that are performed frequently in an application. Here, we try to extract the kernels in data mining applications in our quest for understanding their nature. Extraction of such kernels also helps in identifying how the kernel actually maps to the underlying architecture components, including the processor, memory, and other resources.

Table 3 presents the top three kernels of each application considered in this study. For each

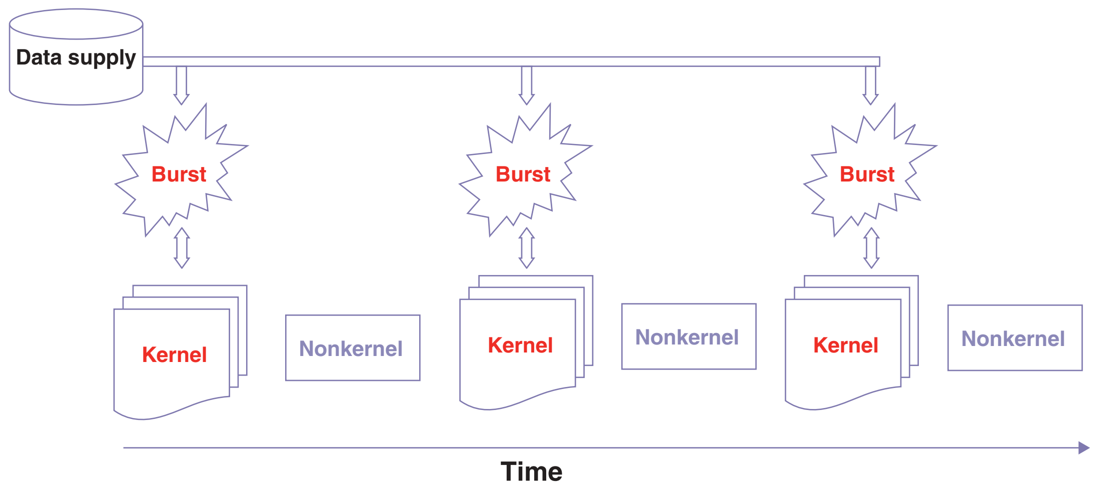

FIGURE 1 | Multiphased *data burst and kernel* operations seen in data mining.

application, the name of the kernel and the percentage of the system time spent executing the kernel are presented. The last column shows the cumulative sum of the three kernels. It is evident that the top three kernels constitute to most of the execution time (ranging from 71% to 99%); in some cases, even a single kernel can constitute a majority of the execution. For example, in *k*-means, distance calculation takes 68% of the execution time. On the other hand, in applications such as HOP, the execution is evenly distributed among several kernels. Overall, we can conclude that these applications spend a majority of their execution time in a relatively small number of kernels, as depicted in Table 3.

OVERVIEW

General-purpose computation on graphics processing unit (GPGPU) is a relatively new concept in high performance computing. Although mass-market three-dimensional graphics accelerators existed as far back

TABLE 3 | Top Three Kernels for Every Application in MineBench and Their Contributions to Total Execution

	Top Three Kernels (%)			
Application	Kernel 1 (%)	Kernel 2 (%)	Kernel 3 (%)	Sum (%)
k-Means	Distance (68)	Clustering (21)	minDist (10)	99
Fuzzy *k*-Means	Clustering (58)	Distance (39)	fuzzySum (1)	98
BIRCH	Distance (54)	Variance (22)	Redistribution (10)	86
HOP	Density (39)	Search (30)	Gather (23)	92
Naive Bayesian	ProbCal (49)	Variance (38)	dataRead (10)	97
ScalParC	Classify (37)	giniCalc (36)	Compare (24)	97
Apriori	Subset (58)	dataRead (14)	Increment (8)	80
Eclat	Intersect (39)	addClass (23)	invertClass (10)	71
SNP	CompScore (68)	updateScore (20)	familyScore (2)	90
GeneNet	CondProb (55)	updateScore (31)	familyScore (9)	95
SEMPHY	bestBrnchLen (59)	Expectation (39)	lenOpt (1)	99
Rsearch	Covariance (90)	Histogram (6)	dbRead (3)	99
SVM-RFE	quotMatrx (57)	quadGrad (38)	quotUpdate (2)	97
PLSA	pathGridAssgn (51)	fillGridCache (34)	backPathFind (14)	99
Utility	dataRead (46)	Subsequence (29)	Main (23)	98

as the mid-1990s, at that time, they implemented highly specialized, fixed functions and were therefore useful only for graphics.

NVIDIA's NV20 architecture,[10] released in 2001, marked the first advance toward GPGPU with the introduction of programmable shaders. Using graphics-specific application programming interfaces (APIs), short programs could be used to control the vertex- and pixel-shading stages of the graphics pipeline. With programmable shaders, GPGPU became technically possible, but mapping an application to a GPU was a very cumbersome process because the application needed to be coaxed into conforming to the rigid graphics pipeline and the limited APIs.

By 2006, flexibility and performance demands for GPUs had led to the adoption of a unified shading architecture. Instead of a rigid, feed-forward pipeline in which vertex and pixel shaders are implemented separately, a unified shading architecture employs a flexible control mechanism with a pool of generic computational units capable of handling any shader operation. Adding support for general-purpose programming languages made GPGPU practical, as a GPU could then be repurposed as a many-core, multithreaded, general-purpose coprocessor.

At present, both AMD/ATI and NVIDIA, the two largest discrete GPU manufacturers, support GPGPU for their high-end product offerings, allowing the massive floating-point capabilities of commodity GPUs to be applied toward accelerating a broad range of applications. GPU-accelerated applications are written with C-style syntax, with the aid of either a proprietary set of extensions or the standards-based OpenCL framework.[11]

MAPPING APPLICATIONS OF GPUs

GPUs offer approximately an order of magnitude greater floating-point computational capabilities and an order of magnitude greater memory bandwidth than CPUs.[12] Because of the high compute capabilities of GPUs, data parallel applications often map well to GPUs. But compared to a CPU, the GPU lacks the robust control logic required to keep the processing units busy for all but the simplest sequences of operations. Also, the cost of moving input data and results between the GPU's memory and the CPU's memory weighs against speedups afforded by the GPU's computational capabilities. In general, floating-point intensive applications with abundant parallelism, predictable memory accesses, data reuse, and little or no divergent control flow are the best candidates for GPU acceleration.

EXAMPLE: NVIDIA GPU ARCHITECTURE

The G80 architecture, released in 2006, was NVIDIA's first GPU family to employ a unified shader architecture. From the GPGPU standpoint, the G80 architecture resembles a many-core single instruction multiple data (SIMD) processor.[12] Each core, or streaming multiprocessor, is composed of a set of SIMD processing units and a shared memory region. The GPU card has its own external memory, which is connected to the GPU core *via* a high-bandwidth, low latency link.

The top-of-the-line consumer G80 card, the 8800 Ultra, has 128 processing units, configured as 16 streaming processors of eight processing units each, and 768 MB GDDR3 RAM.[13] The peak single-precision performance of this card is 384 Gflop/s and the memory bandwidth is 104 GB/s. NVIDIA's latest GPU series, the GT200, is available in single-chip configurations up to 240 processing units, with as much as 2 GB GDDR3 RAM. Peak performance for these cards can exceed 1 Tflop/s and the memory bandwidth is as high as 159 GB/s.[14] Adding to this list, NVIDIA released their newest Tesla Processor based on Fermi architecture with 448 processing units and 1.03 Tflop/s peak floating-point performance. The ATI's Radeon series provide similar performance. To put these figures into perspective, CPUs released in similar timeframes are rated at approximately an order of magnitude lower floating-point performance and memory bandwidth.

PROGRAMMING NVIDIA GPUs

Work is dispatched to the multiprocessors as independent blocks of threads. Compared to a thread running on the CPU, GPU threads are lightweight and simple. A single CPU thread will far outperform a single GPU thread, but the GPU as a whole is capable of outperforming the CPU because it can execute instructions from hundreds of threads simultaneously.

NVIDIA describes its recent GPUs as single instruction multiple thread (SIMT) architectures.[12] From the programming perspective, SIMT is similar to the single program multiple data (SPMD) programming model used commonly in distributed computing applications. In both cases, tasks are broken down into identical threads that operate on different data. Threads have independent control and are free to diverge from one another at any time, implying task parallelism capabilities.

The underlying GPU architecture is what makes SIMT distinct. The multiprocessors in the GPU are inherently SIMD, and the SIMT model, which explicitly calls for scalar operations within each thread, maps groups of threads to each multiprocessor for concurrent execution. Threads within a group have the same physical instruction fetch and data cache, and can share data through a local store on the multiprocessor. This is in contrast to a standard SPMD model, in which threads are assumed to require asynchronous communication to share data and otherwise operate completely independently of one another. In SIMT, when the active threads on a multiprocessor are all executing the same instruction, they execute concurrently, enabling fine-grained data parallelism. When threads have diverged from one another, their execution is serialized in the multiprocessor, providing the flexibility to implement task parallelism at the thread level, although at the cost of reduced utilization.

NVIDIA provides a proprietary framework for programming its GPUs, called compute unified device architecture (CUDA),[12] as well as an OpenCL driver. In either case, C-style syntax is used to specify the behavior of the GPU and orchestrate overall application control flow. The GPU acts as a coprocessor, so an application running on the host CPU is required to manage the overall application control flow, configure the GPU for computation, and schedule memory transfers between the GPU and the host system.

BASIC STATISTICS ON GPUs

Computational scientists generate large amounts of data from experimental, observational, and computational simulation. These data need to be analyzed and interpreted to gain insight. The basic operations that are applied to the data are called descriptive statistics. They provide simple summaries about the sample. Together with simple graphics analysis, they form the basis of further more complex quantitative analysis of data. A few of these basic statistical functions include min, max, mean, standard deviation, variance, histogram, and summation functions. The first two statistics, min and max, provide the range of the data. Mean is the arithmetic average of the data observations. The standard variation is the most commonly used measure of the spread or dispersion of data around the mean. The standard deviation is defined as the square root of variance (the expected squared deviation from the mean). A histogram shows the shape of the probability distribution function ID data, checks for homogeneity, and suggests possible outliers. These functions have different characteristics/computational requirements that result in varied performance gains when implemented on GPUs.

The GPU implementation of the summation function follows a tree-based approach. In this approach, the datapoints are summed in pairs at the root node. The results follow the similar pattern of pair summation and continue till the final result is obtained. The other functions, except histogram, can be implemented by simple modifications to the tree approach. Histograms are traditionally quite difficult to compute efficiently on GPUs due to nonuniform memory access pattern and provide limited speedup compared to other functions.[15]

DATA MINING APPLICATIONS FOR GPUs

Clustering and classification algorithms are composed of kernels that are computationally very demanding. As an example, *k*-means[16] is a clustering algorithm that divides the input dataset into *k* clusters. It is an iterative algorithm composed of two main kernels: distance computation and cluster update. The distance computation kernel calculates the distance of every input data point to the *k* cluster centers and assigns it to the cluster that is closest to it. This is followed by the cluster update kernel that calculates the new cluster centers for the next iteration by computing the mean of all the datapoints in a cluster. In the first kernel, the computation of distance of a datapoint is independent of other datapoints. Hence, the massive number of threads of GPUs can be very suitable to accelerate the distance computation kernel. The latter kernel is similar to the histogram (mentioned in the basic statistics section) implementation and its performance is limited by the memory access pattern.

A modified *k*-means algorithm is fuzzy *k*-means.[17] The distinguishing feature of fuzzy *k*-means is that it allows each datapoint to have a degree, in terms of probability, of membership to each cluster. This means that instead of assigning a datapoint to a particular cluster, every datapoint belongs to a cluster with certain probability. The computation of this probability makes fuzzy *k*-means more compute-intensive compared with *k*-means. Further, the memory access pattern is more uniform in the cluster update kernel compared with *k*-means that results in fuzzy *k*-means having a higher performance improvement on GPUs.

Another example of a data mining application, which is suitable for GPUs, is principal component analysis (PCA).[18] PCA is a powerful tool to identify patterns in high dimensional data. It aims at finding principal components that are representative of the input dataset. The algorithm can be divided into three kernels: tridiagonalization kernel, eigenvalue kernel, and principal component kernel. The first kernel transforms the covariance matrix of the input dataset into a tridiagonal matrix. The second kernel uses the tridiagonal matrix to compute the eigenvalues of the input covariance matrix. (Converting the matrix to a tridiagonal form and computing the eigenvalues is more efficient than computing the eigenvalues of the matrix directly.) The principal component kernel takes the first k eigenvalues and computes the corresponding eigenvectors that form the principal components of the input dataset. The kernels involve matrix operations that are shown to perform an order of magnitude higher on the GPUs as compared to the traditional CPU implementation.

GPU IMPLEMENTATION

As mentioned in the previous section, k-means comprises of a distance computation kernel and a cluster update kernel, both of which can be separately implemented on GPU. The first kernel can have various possible implementations resulting in different performance benefits. A naive way of implementation would be to assign each datapoint to a GPU thread and let it compute the cluster membership of that datapoint (i.e., the thread computes the distance of the datapoint from the k-cluster centers and assigns it to the cluster closest to it). The NVIDIA GPUs have different levels of memory with different latencies and sizes. Hence, the performance depends heavily on where the cluster center data for the current iteration are stored. If the cluster centers are stored in the device memory of the GPU, then there will be a lot of memory conflicts as all threads will try to read the cluster center data simultaneously, which will result in poor performance. To improve the performance, the cluster centers can be copied to the on-chip shared memory, which is an order of magnitude faster than the device memory. Further, because the threads are grouped as blocks, and each block has its own shared memory, replicating the cluster centers will reduce the number of conflicts, thus boosting the performance. The implementation of cluster update closely follows the histogram implementation. Because the averaging is done for all the attributes present in the dataset, the computation is distributed across arrays of blocks, with each array handling one attribute.

NVIDIA provides a library of basic linear algebra subprograms (BLAS) functions in the form of CBLAS library.[19] The library offers a rich implementation of various operations on matrices and vectors. The kernels in PCA algorithm involve a combination of matrix–matrix, matrix–vector, and vector–vector operations. Thus, GPU implementation provides an order of magnitude higher speedup compared with the CPU implementation.

GPU VERSUS CPU PERFORMANCE

Experiments performed on NVIDIA's GeForce 8800GT show performance gains of the applications mentioned above, which outperform the traditional single-CPU implementation. Figure 2 shows

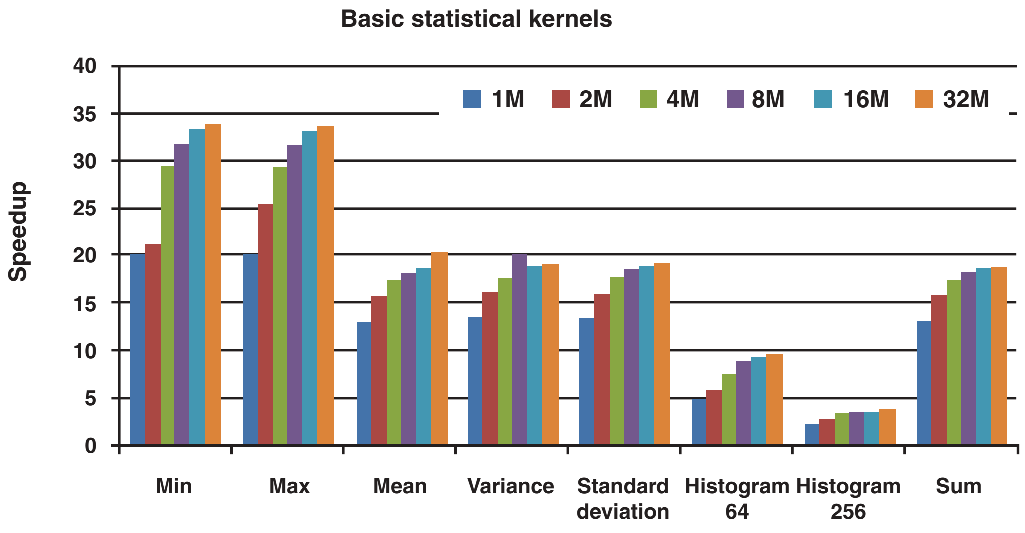

FIGURE 2 | Speedup of GPU versus CPU for basic statistical kernels for different data sizes.

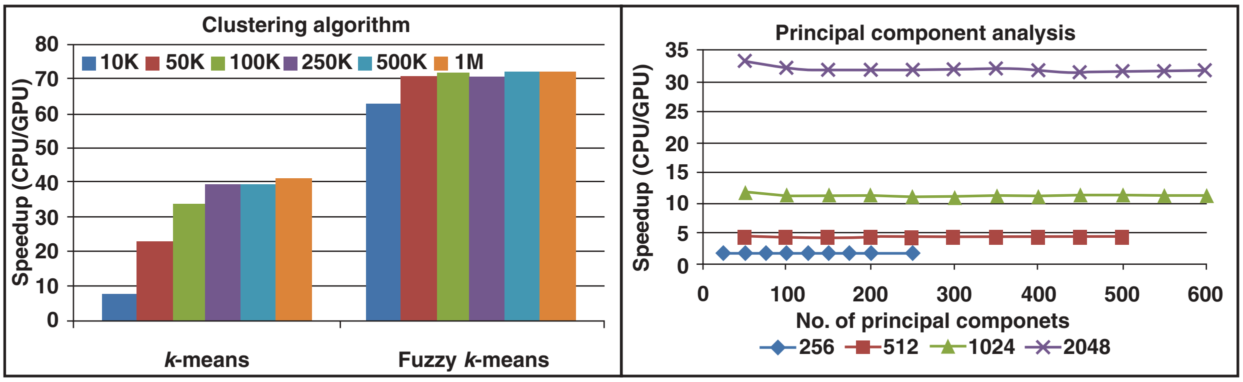

FIGURE 3 | Speedup of GPU versus CPU for different data mining algorithms.

the speedups for the various basic statistical kernels. Figure 3(a) shows the results of the clustering algorithms with 20 attributes and different size of the input data ranging from 10,000 to 1 million data records. Similarly, Figure 3(b) shows the results from PCA algorithm with different size of the input data and different number of principal components. Limited memory on the GPU device restricts the maximum number of data elements that can be processed by the GPU device.

FPGA OVERVIEW

An FPGA is a type of integrated circuit whose functionality can be specified after the fabrication process. Because it is designed to have the flexibility to implement any logical function, an FPGA can be used to prototype ASICs or replace them altogether.

At the core of an FPGA's architecture is an array of logic blocks. The specific architecture of these logic blocks varies between manufacturers and product lines, so a generic example is shown in Figure 4. The LookUp table in the logic block can be programmed to implement an arbitrary combinational logic function, and the register allows each logic block to hold state. Multiple logic blocks can be used collectively to specify very complex logic functions.

Signal routing between logic blocks is accomplished with a hierarchy of programmable interconnect resources. Short lines, connecting neighboring blocks, paired with long lines, connecting distant blocks, allow signals to move freely within the FPGA. FPGAs also implement dedicated clock and reset lines designed to minimize timing problems.

Modern FPGAs typically embed fixed-function blocks, such as block RAM elements, digital signal processor modules, high-speed I/O transceivers, and even general-purpose processors, into the FPGA. Embedded function blocks are faster and more space-efficient than equivalent implementations using reconfigurable resources, and enable FPGAs to implement complex, customizable System-on-Chip architectures.

With sufficient resources, an FPGA can implement an arbitrarily complex circuit. And because many FPGA solutions allow run-time reconfiguration, the functionality provided by a single FPGA is limited only by one's ability to specify configurations. Compared with ASICs, products targeting FPGAs have a much shorter time to market, require much lower nonrecurring engineering costs, and can be updated for bug fixes or improved functionality at any time. They are, however, significantly slower than ASICs and more expensive at high volumes.

RECONFIGURABLE COMPUTING

The design flexibility afforded by FPGAs, in addition to their ever-growing gate counts and comparatively low overall power consumption, has generated recent interest in reconfigurable computing. In the realm of high-performance computing, reconfigurable architectures typically employ one or more FPGAs as coprocessors of the CPU. Software running on the CPU drives the execution of the application as a whole and offloads computationally intensive tasks to the FPGA.

Although FPGAs operate at relatively low clock speeds (200 MHz is common), they are capable of performing many operations in parallel and can be tailored to the specific data flow of a task requiring acceleration. Also, the vast programmable interconnect network of an FPGA provides very high internal bandwidth for moving inputs and intermediate data

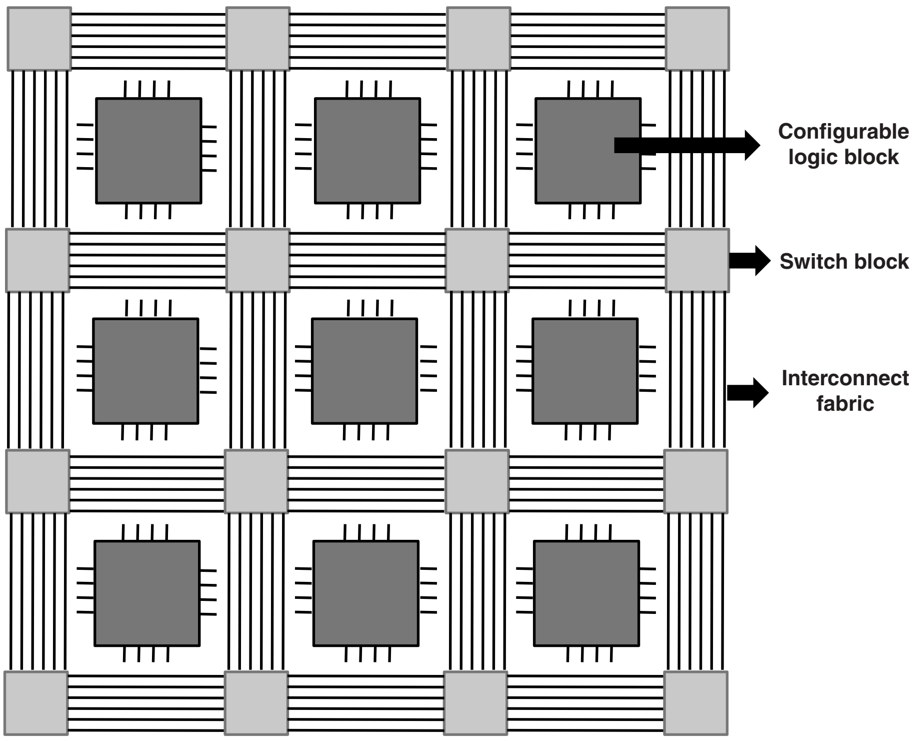

FIGURE 4 | Generic implementation of FPGA logic cells.

within the FPGA. Taking these factors into account, certain data-intensive tasks can be performed more quickly and efficiently on an FPGA than on a CPU.

PROGRAMMING FOR FPGAs

FPGA configurations are commonly specified using a hardware design language (HDL) such as VHDL or Verilog. The desired functionality of the FPGA can be specified down to the gate level, and FPGA manufacturers and third parties provide commonly used functional blocks, such as arithmetic, memory controllers, and communications, which can be integrated into the HDL description. A tool chain specific to the target device maps the HDL description to the FPGA and produces a configuration file, which can be loaded onto the FPGA when necessary.

The reconfigurable nature of FPGAs offers some of the high-performance customizability of ASICs with the flexibility of software solutions. Programming an accelerator for an FPGA using an HDL is a much simpler and shorter process than creating an ASIC accelerator, but is much more difficult and time-consuming than developing software for a general-purpose processor. High-level synthesis tools address the difficulty associated with creating HDL descriptions by providing an algorithmic-level abstraction of the hardware description. These tools allow a designer to specify the desired functionality of a circuit using a high-level language, such as C/C++, and rely on automated tools to transform the specification into an HDL description.

MAPPING APPLICATIONS TO FPGAs

Determining the suitability of an application for FPGA acceleration requires an analysis of the application's control flow and data usage patterns. The best candidates for FPGA acceleration are tasks featuring a simple control flow, abundant data parallelism, and low input data requirements. Mapping a task to an FPGA generally amounts to the specification of a custom datapath for that task, the goal of which is to leverage the parallel computational capabilities and high internal bandwidth of the FPGA to offset the low operating frequency. Tasks in which an input dataset can be streamed through a datapath are often good candidates for FPGA acceleration because they reduce data storage requirements and performance-degrading memory access logic.

The capacity of FPGAs has grown to the point at which they can feasibly be used for floating-point arithmetic. But the computational intensity of the candidate application needs to be very high to achieve

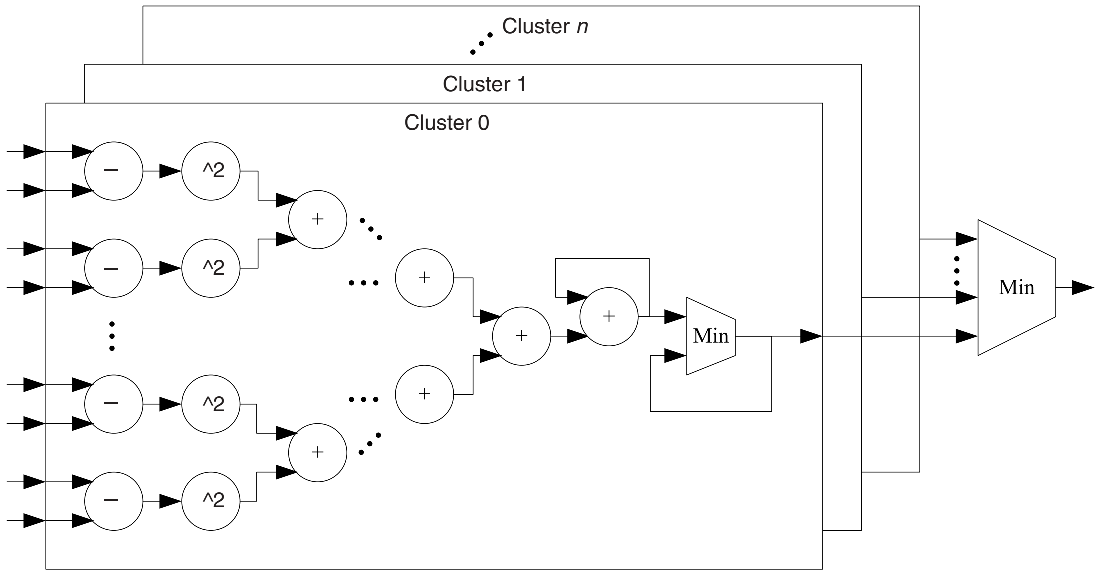

FIGURE 5 | *k*-means implementation on FPGA.

decent performance gains because the floating-point capabilities of even the largest FPGAs are not vastly better than commodity CPUs.

IMPLEMENTATION OF *k*-MEANS ON FPGA

Consider, for example, the distance calculation phase of *k*-means. At first glance, it seems like a prime candidate for FPGA acceleration due to embarrassing parallelism, very simple control flow, and low storage requirements. The computation maps easily to the datapath shown in Figure 5. The datapath is capable of simultaneously calculating squared distances for multiple dimensions between a single point and multiple clusters. The input dataset streams through the FPGA once per iteration.

The problem with this datapath is that the floating-point operations consume a significant amount of resources, so its computational capabilities are not much better than a software implementation on a high-end CPU, especially when the overhead of transferring data between the FPGA and CPU is taken into account. In this case, the task maps well to an FPGA, but its performance is constrained by limitations on available resources.

Depending on the accuracy requirements of the application and the nature of the input dataset, the resource utilization, and accordingly the performance, of this datapath can be improved by moving to fixed-point representations or implementing a simpler distance metric. Either solution would reduce the complexity of the datapath, allowing the number of functional blocks to be increased.

DECISION TREE CLASSIFICATION

Decision tree classification (DTC) is a basic technique for generating a predictive classification model. Given an input dataset consisting of multiple records of a number of attributes, a decision tree model recursively splits the dataset based on the value of a particular attribute.

Consider, for example, a dataset with two continuous attributes, A and B, and a binary class ID attribute, C. We wish to use the dataset to generate a predictive model for C, based on the values of A and B.

To create this model, the dataset is divided into two partitions, P_0 and P_1, where P_0 is initially empty and P_1 contains all records in the dataset. Gini score, which measures the quality of the partitioning scheme, is calculated for the current partitioning scheme, based on the following formula:

$$\text{Gini} = \sum_{i=0}^{1} \left\{ \frac{R_i}{R} \left[1 - \sum_{j=0}^{1} \left(\frac{R_{ij}}{R_i} \right)^2 \right] \right\}$$

where R is the number of records in the dataset, R_i is the number of records in partition i, and R_{ij} is the number of records in partition i bearing the class label j.

The record in P_1 associated with the smallest value of A is then moved into P_0 and a new Gini score

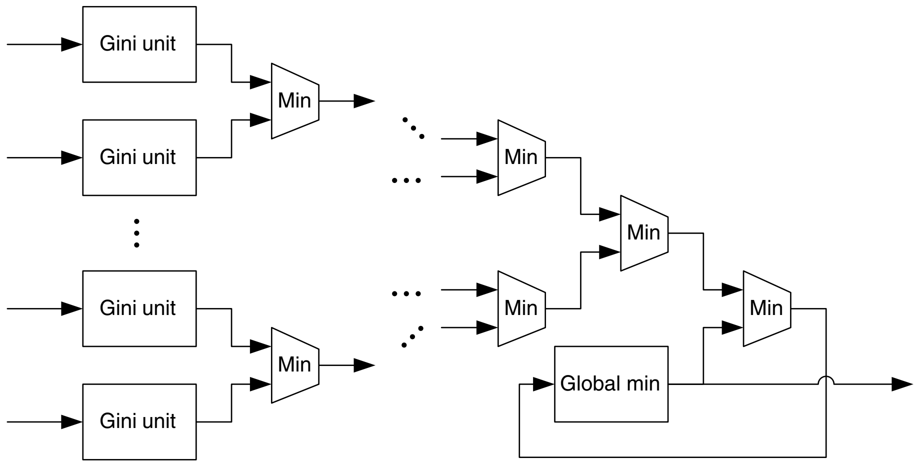

FIGURE 6 | DTC implementation on FPGA.

is calculated. This process is repeated until all records have been moved into P_0. The lowest Gini score calculated during this process is associated with the best partitioning, or split position, of the dataset for attribute A. The same process of sorting the dataset and finding the best partitions is repeated for attribute B, and the minimum Gini score for A and B is compared.

The dataset is split on the basis of the attribute and position of the resulting global minimum Gini score and the entire process is repeated on each partition for the remaining attribute. The result is a tree describing the predictive model for C. Nodes in the tree represent a splitting decision for a particular attribute, and leaves in the tree identify the predicted class ID.

DTC: FPGA IMPLEMENTATION

A significant portion of the overall execution time of the decision tree induction task is consumed in determining the minimum Gini score for each attribute. The determination of a global minimum splitting attribute and split position is a computationally intensive process requiring a sequence of integer arithmetic operations to be performed for each record over each attribute in the dataset. The relative simplicity of the arithmetic operations required to calculate Gini scores, paired with the straightforward control flow of the minimum score calculation process, makes this operation a good candidate for FPGA coprocessing.

To accelerate optimal split position determination on the FPGA, an architecture is designed to simultaneously compute the minimum Gini score for multiple attributes,[20] as shown in Figure 6. The total number of Gini units implemented in the datapath is a power of 2 and is limited by the available FPGA resources.

Each Gini unit calculates the best split value and split position for a particular attribute, from which the comparator tree identifies the global optimum splitting attribute. The Gini score calculation for binary class IDs can be rewritten as:

$$\text{Gini} = \frac{2 \cdot R_{00} \cdot R_{01}}{R_0 \cdot (R_0 + R_1)} + \frac{2 \cdot R_{10} \cdot R_{11}}{R_1 \cdot (R_0 + R_1)}$$

R_0 is equivalent to $R_{00} + R_{01}$, R_1 is equivalent to $R_{10} + R_{11}$, and $R_0 + R_1$ is constant over the processing of an attribute, so the following modified formula generates the same split position with less hardware:

$$\text{Gini}' = \frac{R_{00} \cdot R_{01}}{R_{00} + R_{01}} + \frac{R_{10} \cdot R_{11}}{R_{10} + R_{11}}$$

A further refinement is based on the observation that, given two partitions of known record counts and class memberships, the number of records in a particular partition with a particular class label changes by either 0 or 1 when a record is moved from one partition to the other. Product terms can therefore be calculated with addition and subtraction operations instead of multiplications. The resulting optimized Gini unit architecture for binary class IDs is shown in Figure 7.

The overall application flow with the FPGA coprocessor proceeds as follows. First, software running on the host CPU generates a representative bitmap of class IDs. Each column of the bitmap holds a copy of the class ID field of the dataset after the dataset is sorted based on a distinct attribute. Software initializes the Gini units on the FPGA accelerator by writing initial values of R_0, R_1, R_{00}, R_{01}, R_{10}, and R_{11}. The class ID bitmap is then streamed to the FPGA accelerator, which distributes each column of the

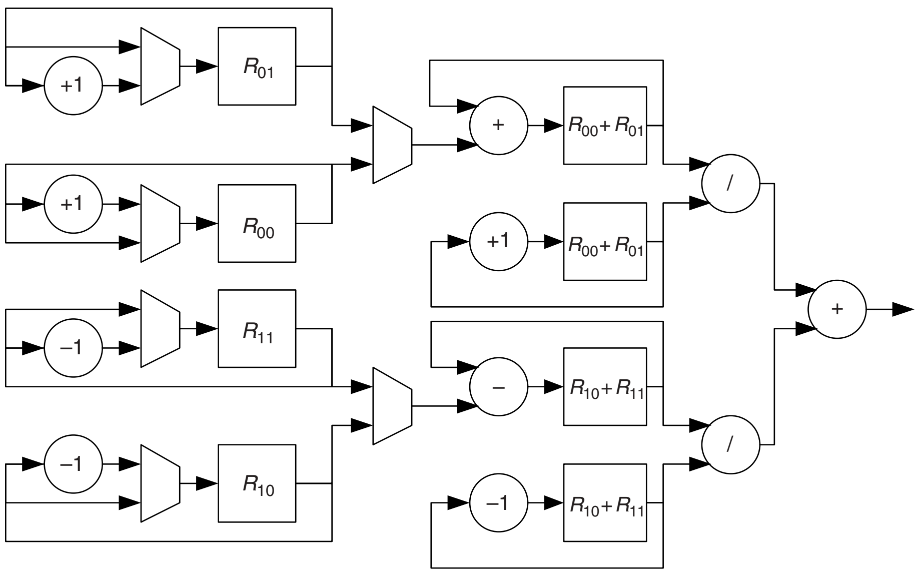

FIGURE 7 | Logic implementing Gini calculation.

bitmap to a separate Gini unit. Once the FPGA accelerator has processed the bitmap in its entirety, the software application reads the optimal splitting attribute and split position from the FPGA and uses it to perform the actual partitioning of the dataset. The process is repeated recursively for each partition until a full predictive model has been generated.

Compared with a pure software implementation, a 6× speedup of the Gini calculation task can be obtained with this accelerator on a small FPGA.[20] Using this accelerator on an XD1000 development system from XtremeData, Inc., Schaumburg, IL, USA, which features a much larger FPGA and a high-bandwidth, low latency link to the CPU,[21] a speedup of 30× is possible. Taking only an accelerated Gini calculation into account, the overall application speedup for ScalParC,[22] a popular DTC algorithm, would be limited to 1.5× because the split determination phase accounts for about one-third of the overall algorithm execution time. The overall speedup can be significantly improved by accelerating other time-consuming operations, such as dataset sorting, using an FPGA or GPU.

PCA: FPGA IMPLEMENTATION

As has already been mentioned earlier, PCA produces a set of principal components, which are orthonormal eigenvalue/eigenvector pairs. In other words, it projects a new set of axes that best represent the data.

The FPGA implementation of PCA can be divided into two parts. The first part takes each data element (i.e., vector x) and projects it along the new set of orthonormal axes. In most data mining applications such as classification, the eigenvalues and eigenvectors need to be calculated only once during the initial training phase, which is frequently done offline. But each new data item needs to be projected along the new axes. Often, this needs to be done online for high throughput requirements. Hence, this is an ideal candidate for FPGA implementation. The second part takes two vectors x and y projected along the principal axes and calculates the distance between them.

Each eigenvalue of a principal component corresponds to the relative amount of variation it encompasses. The larger the eigenvalue, the more significant is its corresponding projected eigenvector. Therefore, the principal components are sorted in the decreasing order of significance. If any two data items are projected along the upper set of the significant principal components, it is likely that we can get a good estimate of distance without projecting along all the principal components. Hence, only a subset of the most important principal components is needed to estimate the distance between any two vectors.

The high level architecture of the principal component score pipeline (PCSP)[23] is shown in Figure 8. There are many levels of parallelism to exploit in the PCSP pipeline. They are depicted in the dashed line boxes in Figure 8. First of all, subtracting the vector y from the vector x is done in parallel. If each data point has p attributes $[x = (x_1, x_2 \ldots x_p)]$, then p operations are performed in parallel. The next phase

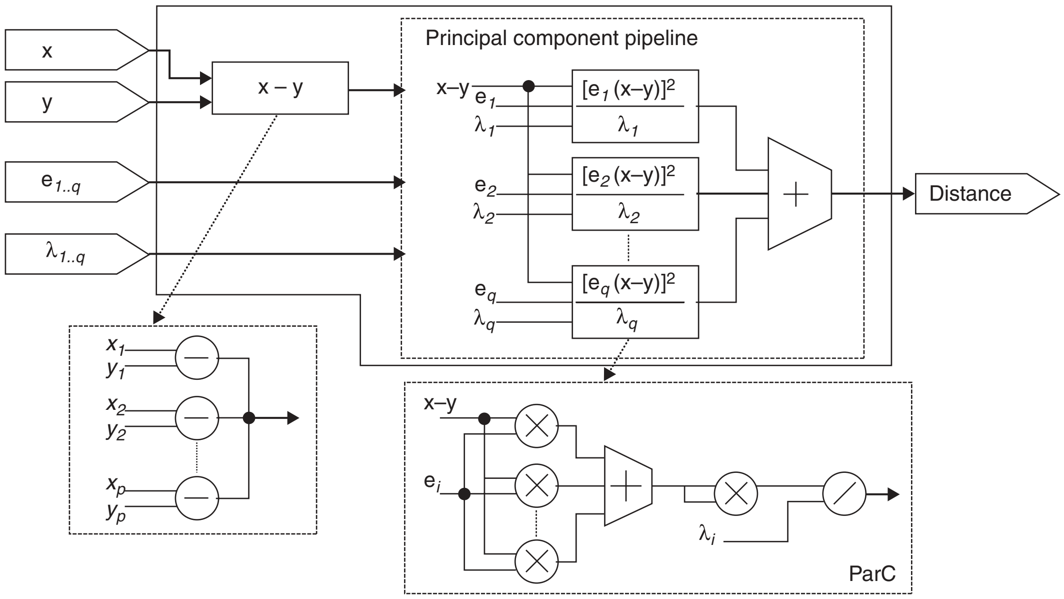

FIGURE 8 | FPGA implementation of principal component analysis.

for PCA is calculating the partial component scores (parC).[24] The element-by-element multiplication, using fixed-point arithmetic, is performed in parallel. The first summation gives the projection of 'x'–'y' along each principal component. This is accomplished with an adder tree whose depth scales logarithmically with the number of attributes [$\log_2(p)$]. The result is then squared and divided by the eigenvalue of the ith principal component. The next step is the summation of all parC scores. This is again computed by an adder tree whose depth scales logarithmically with the number of principal components provided [$\log_2(q)$]. Using this accelerator on an XtremeData XD1000 development system, speedup of about two orders of magnitude can be achieved over pure software implementation.

CONCLUSION

The increasing complexity and programmability of high performance architectures, such as modern GPUs, and current generation of FPGAs have paved their way in the domain of applications exhibiting high computational requirements and high memory bandwidth. Data mining represents such a domain, in which application characteristics necessitate the embracement of coprocessing units to improve performance and provide scalability. The chapter attempts to provide an introduction of the characteristics of the applications as well as the characteristics of the GPUs and FPGA devices. It explores the different applications in detail and provides useful insight for implementation on these new architectures. Experimental studies show that depending on the characteristics of the applications, they can achieve speedups ranging from 40× to 150×. Because the overall execution time of a data mining algorithm is often dominated by a few computationally intensive kernels, applying GPUs and FPGAs can yield significant speedups over the application as a whole. However, these coprocessors are not without limitations. GPUs show poor performance if the data access pattern is not uniform, whereas FPGAs might lose their benefit if a complex datapath is implemented.

With the continuous advancement in the technology, we envision that the future systems will see the emergence of heterogeneous computing units which will include a collection of high performance architectures catering to different aspects of data analysis. These heterogeneous units will form the building blocks of ultrascalable systems. Such systems require more than just possessing high computational capabilities. Without a high bandwidth link and high performance storage, the overall performance of the system will largely be limited. Hence, the technology will progress in the direction of producing high performance compute nodes and high performance IO nodes along with a fast communication link between the nodes to provide performance gains which will be orders of magnitude higher than the state-of-the-art systems of the present era.

REFERENCES

1. Ramanathan R, Bruening F, Intel Corporation. *Architecting the Era of Tera—Technical White Paper*. Santa Clara, CA: Intel Corporation.
2. WinterCorp Consulting Services, 2005. Top Ten Program. Available at: http://www.wintercorp.com/VLDB/2005_TopTen_Survey/2005TopTenWinners.pdf (Accessed December 14, 2009). Cambridge, MA.
3. Pisharath J, Choudhary A. Design of a hardware accelerator for density based clustering applications. *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors*; July 2005. Washington DC: IEEE Computer Society, 101–106
4. Zambreno J, Ozisikyilmaz B, Pisharath J, Memik G, Choudhary A. Performance characterization of data mining applications using MineBench. *Proceedings of the International Workshop on Computer Architecture Evaluation using Commercial Workloads*; February 2006, 61–70.
5. Ozisikyilmaz B, Narayanan R, Zambreno J, Memik G, Choudhary A. An architectural characterization study of data mining and bioinformatics workloads. *Proceedings of IEEE International Symposium on Workload Characterization*. San Jose, CA: October 2006, 61–70.
6. Narayanan R, Ozisikyilmaz B, Zambreno J, Memik G, Choudhary A. MineBench: a benchmark suite for data mining workloads. *Proceedings of IEEE International Symposium on Workload Characterization*: San Jose, CA: October 2006, 182–188.
7. Standard Performance Evaluation Corporation. *SPEC CPU2000 V1.2, CPU Benchmarks*. 2001.
8. Lee C, Potkonjak M, Mangione-Smith WH. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. *Proceedings of the International Symposium on Microarchitecture*. Research Triangle Park, NC: 1997, 330–335.
9. Transaction Processing Performance Council. *TPC-H Benchmark Revision 2.0.0*. 2004.
10. NVIDIA Corporation. The Infinite Effects GPUs. Available at: http://www.nvidia.com/page/geforce3.html (Accessed December 14, 2009).
11. KHRONOS Group. OpenCL—The open standard for parallel programming of heterogeneous systems. Available at: http://www.khronos.org/opencl/ (Accessed December 14, 2009).
12. NVIDIA Corporation. *NVIDIA CUDA Programming Guide. Version 2.3.1*. 2009.
13. NVIDIA Corporation. GeForce 3. Available at: http://www.nvidia.com/page/geforce3.html (Accessed December 14, 2009).
14. NVIDIA Corporation. GeForce GTX 285. Available at: http://www.nvidia.com/object/product_geforce_gtx_285_us.html (Accessed December 14, 2009).
15. NVIDIA Corporation. *Histogram Calculation in CUDA. Version 1.1.1*. November 2007.
16. Lloyd SP. Least squares quantization in PCM. IEEE Trans Inf Theory 1982, 28:129–137.
17. Bezdek JC. *Pattern recognition with fuzzy objective function algorithms*. Norwell, MA: Kluwer Academic; 1981.
18. Pearson K. On lines and planes of closest fit to systems of points in space. Philosl Mag 1901, 2:559–572.
19. NVIDIA Corporation. *CUDA CUBLAS Library. Version 1.1*. September 2007.
20. Narayanan R, Honbo D, Zambreno J, Memik G, Choudhary A. An FPGA implementation of decision tree classification. *Proceedings of the IEEE International Conference on Design, Automation and Test in Europe (DATE)*. Nice, France: April 2007, 1–6.
21. XtremeData Inc. XD1000 Development System. Available at: http://old.xtremedatainc.com/index.php?option=com_content&view=article&id=109&Itemid=170 (Accessed December 14, 2009).
22. Joshi M, Karypis G, Kumar V. ScalParC: a new scalable and efficient parallel classification algorithm for mining large datasets. *Proceedings of the 11th International Parallel Processing Symposium (IPPS)*. Orlando, FL: 1998, 573–579.
23. Das A, Misra S, Joshi S, Zambreno J, Memik G, et al. An efficient FPGA implementation of principal component analysis based network intrusion detection system. *Proceedings of Design, Automation & Test in Europe (DATE)*. Munich, Germany: Munich; 2008, 1160–1165.
24. Jobson JD. *Applied Multivariate Data Analysis, Volume II: Categorical and Multivariate Methods*. New York: Springer-Verlag; 1992.