A Programmable Parallel Accelerator for Learning and Classification

Srihari Cadambi Abhinandan Majumdar Michela Becchi
Srimat Chakradhar Hans Peter Graf
NEC Laboratories America, Inc.
4 Independence Way, Princeton NJ 08540. USA.
{cadambi, abhi, mbecchi, chak, hpg}@nec-labs.com

ABSTRACT

For learning and classification workloads that operate on large amounts of unstructured data with stringent performance constraints, general purpose processor performance scales poorly with data size. In this paper, we present a programmable accelerator for this workload domain. To architect the accelerator, we profile five representative workloads, and find that their computationally intensive portions can be formulated as matrix or vector operations generating large amounts of intermediate data, which are then reduced by a secondary operation such as array ranking, finding max/min and aggregation. The proposed accelerator, called MAPLE, has hundreds of simple processing elements (PEs) laid out in a two-dimensional grid, with two key features. First, it uses in-memory processing where on-chip memory blocks perform the secondary reduction operations. By doing so, the intermediate data are dynamically processed and never stored or sent off-chip. Second, MAPLE uses banked off-chip memory, and organizes its PEs into independent groups each with its own off-chip memory bank. These two features together allow MAPLE to scale its performance with data size. This paper describes the MAPLE architecture, explores its design space with a simulator, and illustrates how to automatically map application kernels to the hardware. We also implement a 512-PE FPGA prototype of MAPLE and find that it is 1.5-10x faster than a 2.5 GHz quad-core Xeon processor despite running at a modest 125 MHz.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems – *Microprocessor/microcomputer applications*.

General Terms

Design, Experimentation, Measurement, Performance.

Keywords

Accelerator-based systems, parallel computing, heterogeneous computing, machine learning

PACT'10, September 11-15, 2010, Vienna, Austria.

1. INTRODUCTION

Applications that examine raw, unstructured data in order to draw conclusions and make decisions are becoming ubiquitous. Banks and credit cards companies, for instance, analyze withdrawal and spending patterns to prevent fraud or identity theft. Online retailers study website traffic patterns in order to predict customer interest in products and services based upon prior purchases and viewing trends. Semantic querying of text and images, which has wide-ranging, mass market uses such as advertisement placement [1] and content-based image retrieval [2], is another fast growing application domain.

Such applications extensively use learning and classification techniques. With increasing amounts of data, the computational load imposed by these techniques becomes severe as they must be executed under stringent performance constraints. Scaling application performance with data assumes importance. As an example, for semantic text search, a server using a learning algorithm such as Supervised Semantic Indexing [3] must search millions of documents at a few milliseconds per query. Another example is face and object recognition in high resolution video that is often done with Convolutional Neural Networks (CNNs) [4]. A server performing this task must search VGA (640x480) or higher resolution images at rates of 24 or more frames per second. Often, economic considerations dictate that multiple video streams be processed simultaneously on one server. Our fastest parallelized software implementation on a quad-core 2.5 GHz Xeon server processes about 7 VGA frames per second, while GPU implementations [11] can reach 10 frames per second, both falling far short of requirements. Other similar workloads include digital pathology [15], automotive applications to predict failures and reduce recalls, financial analytics and cognitive databases.

Motivated by this gap between workloads and state-of-the-art computing platforms, we investigate a parallel accelerator for learning and classification applications, and an accompanying tool to automatically map application kernels to the accelerator hardware. To design the accelerator, we profile five representative workloads: Supervised Semantic Indexing [3], Convolutional Neural Networks [4], K-means [5], Support Vector Machines [6] and Generalized Learning Vector Quantization [7], and find that their computational kernels exhibit two common characteristics. First, they can be formulated as matrix or vector operations producing large intermediate data (potentially leading to many off-chip memory accesses), that are then reduced by a secondary operation such as array ranking, finding min/max and aggregation. Second, they exhibit coarse-grained as well as fine-grained parallelism, i.e., the computations can be partitioned into

parallel streams with little communication between them, with each stream processed by hundreds of simple parallel processing elements.

With this in mind, we architect MAPLE (MAssively Parallel Learning/Classification Engine), an accelerator with hundreds of simple vector processing elements (PEs) and two key features that directly address the above workload characteristics. First, MAPLE's on-chip memories are capable of in-memory processing which allows the large intermediate data to be processed on-the-fly thereby reducing off-chip memory accesses. Second, MAPLE uses banked off-chip memories with each memory bank serving a separate group of PEs, thereby creating processor-memory channels that can process the coarse-grained, independent computation streams. These two features make MAPLE's performance scale more easily with problem and data size.

While several prior efforts have developed FPGA and GPU implementations of individual algorithms such as SVMs [8][9], CNNs [10][11] and K-means [14], to the best of our knowledge, a more general, programmable architecture that is optimized across a range of learning and classification workloads has not yet been published. We believe the study and development of accelerators for this domain will become necessary as learning and classification techniques become ubiquitous.

To this end, we make the following contributions in this paper. We present the architecture of MAPLE, a parallel accelerator for learning and classification, and evaluate the use of in-memory processing for learning and classification applications. We present a strategy to automatically map application kernels to MAPLE. Using an FPGA prototype, we compare MAPLE's performance against parallel, optimized software implementations of learning and classification algorithms on multi-cores and GPUs.

The rest of the document is organized as follows. We discuss related work in Section 2, and describe our workloads in Section 3. In Section 4, we describe the MAPLE architecture, explore its design space and present a compilation strategy. In Section 5, we present our FPGA prototype and performance measurements. We conclude in Section 6.

2. RELATED WORK

Prior work in accelerating learning and classification workloads can be classified broadly into four categories: (i) optimized, parallel libraries for multi-core CPUs, (ii) optimized implementations on graphics processors (GPUs) [9][10][11][14], (iii) algorithm-specific accelerators on FPGAs [8] and (iv) other embedded and analog hardware implementations.

Multi-core CPUs and many-core GPUs [18][21] accommodate diverse learning and classification workloads through programmability. However multi-cores cannot avail of the fine-grained data parallelism inherent in these workloads due to thread synchronization overheads and inadequate memory bandwidth. In addition, GPUs do not have banked memory-processor channels, and require multiple independent parallel streams to be coalesced and synchronized. Neither CPUs nor GPUs have enough on-chip storage to handle the large intermediate data generated by these applications. In this paper, we quantitatively compare MAPLE to both CPU and GPU implementations, using optimized software libraries such as Intel MKL BLAS and NVIDIA's CUBLAS.

Several prior efforts have developed algorithm-specific implementations of SVMs [27], CNNs [23] and deep learning [26]. There are also architectures [19] and FPGA implementations that accelerate matrix computations [24][25]. MAPLE is not algorithm-specific, not restricted to matrix operations, and can be programmed for different learning and classification algorithms. We compare MAPLE's performance with published algorithm-specific numbers from [23] and [27].

3. WORKLOAD ANALYSIS

We use five learning and classification workloads to help architect MAPLE. In this section we (i) profile these workloads to identify computational bottlenecks and make the case for an accelerator, (ii) study the nature of the computational bottlenecks (compute or memory bound), (iii) reformulate the computational bottlenecks using a set of common primitives and (iv) identify broader characteristics common to all the reformulated computational bottlenecks that the accelerator architecture must support.

The five algorithms we use are Supervised Semantic Indexing (SSI) [3], Convolutional Neural Networks (CNNs) [4], K-means [5], Support Vector Machines (SVMs) [6] and Generalized Learning Vector Quantization (GLVQ) [7]. SSI ranks a large number of documents based on their semantic similarity to the queries. CNNs are 2-dimensional neural networks used for pattern recognition in applications such as object and face detection [10][11], and recently even semantic text search [12]. K-means clusters points into K clusters, and is commonly used in computer vision for image segmentation. SVM training finds support vectors that separate given training data into distinct classes indicated by the training data labels. GLVQ is a supervised learning algorithm to classify an input into one of several classes.

We profile each algorithm using typical data set sizes (Table 1, column 3), and summarize the characteristics in Figure 1. The table shows the core computations in each workload and the fraction of the total running time they are responsible for. The execution profiles were measured on a 2.5 GHz quad core Xeon. It is clear that significant speedups are achievable by accelerating the core computations. The table also shows whether the workload is compute or memory bound, and the number of computations per memory operation. A memory bound workload performs one or fewer computations per memory load or store. MAPLE targets these core computations, providing adequate processing and I/O resources for both compute and memory bound workloads.

We now examine the computational bottlenecks of these workloads in more detail to find common characteristics and a set of primitives that may be used to design the accelerator. Figure 1 shows the five workloads, their typical parameters and how the computational bottleneck may be transformed into a common set of primitives.

In SSI [3], given D documents, we find K semantic best matches for each of Q concurrent queries. This amounts to a series of dot-products between the document and query vectors, followed by a ranking process to extract the top K matches. These operations may be transformed into matrix multiplications

Table 1: Workload characteristics

Workload	Core computations	% time (profile)	Characteristic	Compute ops per memory operation
SSI	Series of dot products, array rank	> 99%	Dot prod: compute bound Array rank: memory bound	Dotprods: 25-50 Array rank: 0.001
CNN	1D, 2D, 3D convolutions	> 99%	Compute bound	16-100
K-means	Minimum Euclidean dist.	~96%	Marginally compute bound	1-3
SVM	Large matrix-vector mult.	85-95% [27]	Memory bound	1
GLVQ	Minimum Euclidean dist.	> 99%	Memory bound	<1

(by reorganizing the document and query vectors into matrices) which produces a large intermediate result matrix, and array ranking to rank each column of the intermediate matrix and produce the final result.

CNN [4] convolves images with "kernels", which are small weight matrices that are part of a given CNN network. We express convolutions as matrix operations by creating matrices out of different parts of the input images, multiplying with the kernels and using the result matrices to update different portions of the output image. This requires specialized memory access patterns that mimic a convolution operation.

In K-means, the computational bottleneck is finding the closest of mean for *N* points. This can also be expressed as a matrix multiplication followed by a procedure to find the minimum element in each row of the intermediate result matrix. SVM's core computation is a large matrix-vector multiplication, where the matrix is typically too large for on-chip caches. Finally, GVLQ requires a matrix-vector multiplication followed by a minimum finding operation.

From Figure 1 we note that: (i) matrix operations are a common primitive, but matrix sizes vary from very small (CNN kernels) to very large (SVM), (ii) one matrix operand is constant while the other changes, (iii) a large intermediate result is produced before being reduced to a relatively small final output, (iv) the primitives used to reduce the intermediate result (array rank, find minimum) can be implemented using in-memory processing and (v) specialized memory access patterns are required (e.g., CNN). We architect MAPLE with these requirements in mind.

4. MAPLE ARCHITECTURE AND COMPILATION SCHEME

In this section, we present the MAPLE architecture, explore its design space and sensitivities with a simulator and present ways of automatically mapping application kernels to the hardware.

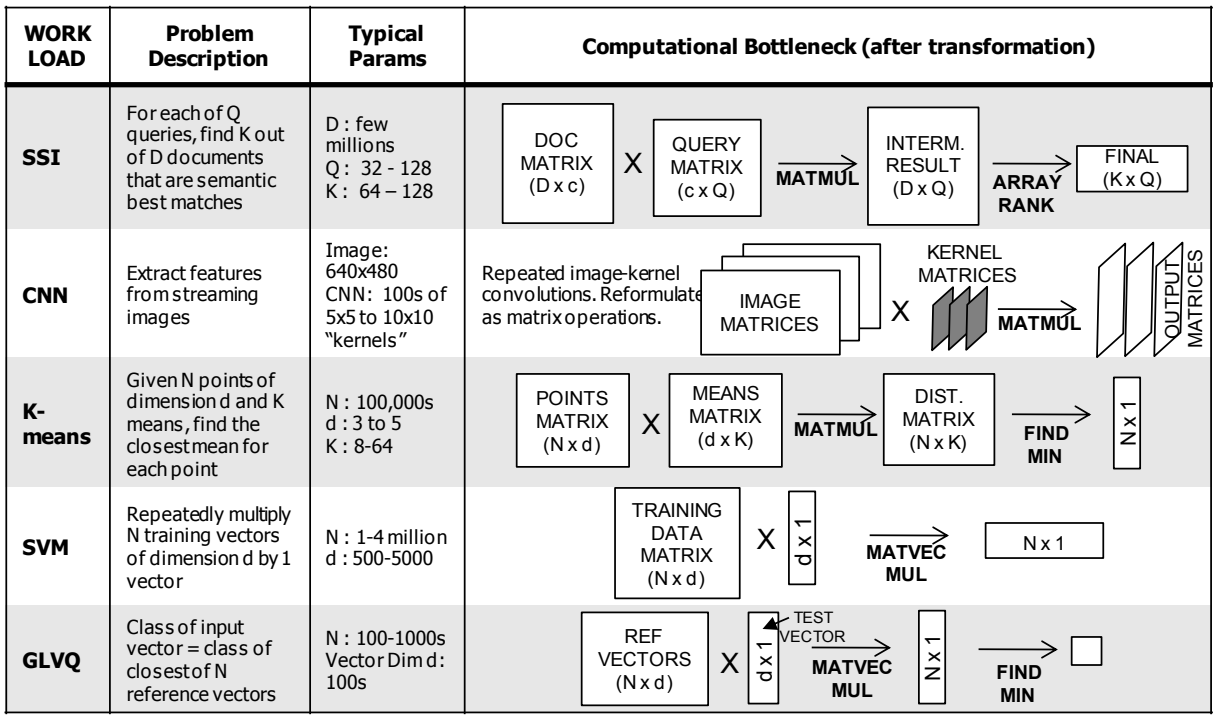

Figure 1: Transforming each workload's bottlenecks to a common set of primitives

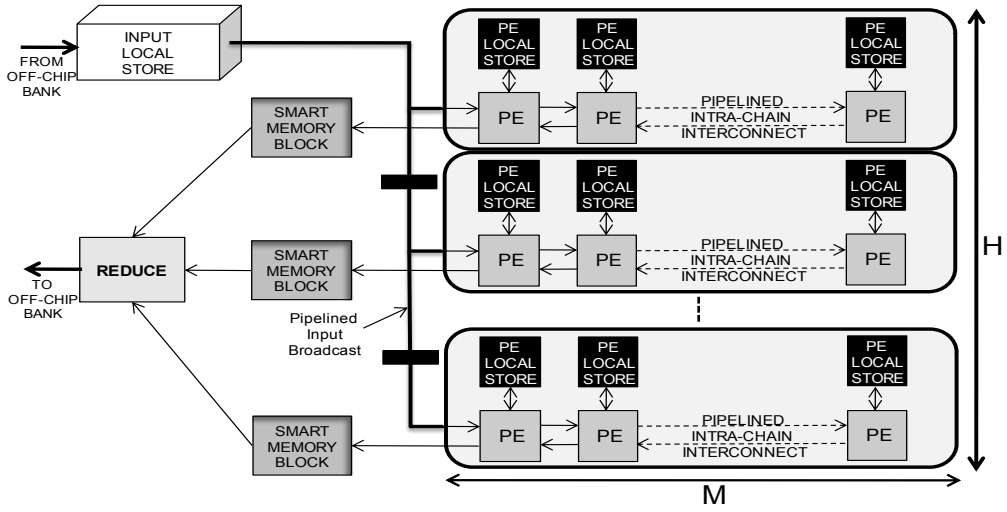

Figure 2: Architecture of a MAPLE processing core

4.1 Architecture

From the workload analysis, we find the architecture must support matrix and vector operations (both large and small matrices), handle large intermediate data and perform reduction operations such as array ranking, finding max/min and aggregation. These requirements lead us to the following design decisions.

First, matrix and vector operations are implemented by streaming data through a two-dimensional array of fine-grained vector processing elements (PEs). This allows minimizing instruction overhead and accelerating operations involving small matrices as well large matrices. Second, we use in-memory processing to handle the intermediate data on-the-fly. By performing reduction operations using on-chip memory blocks, we obviate the need for off-chip accesses to store and reload intermediate data.

We spatially lay out the PEs so that each PE produces a few elements of the output matrix. Each PE has its own local storage. By distributing the columns of one matrix across all PEs and streaming the rows of the other matrix through each PE, matrix multiplication is performed naturally (with each PE performing a multiply-accumulate of the streaming row and its stored column). PEs stream results into "smart memories" that perform in-memory processing of the intermediate data (e.g., finding minimums, ranking arrays, etc). Finally, to support complex access patterns that may result when applications such as CNN are cast as matrix operations, we provide an input buffer that may be addressed by the processing fabric and a memory controller that can support custom access patterns from off-chip.

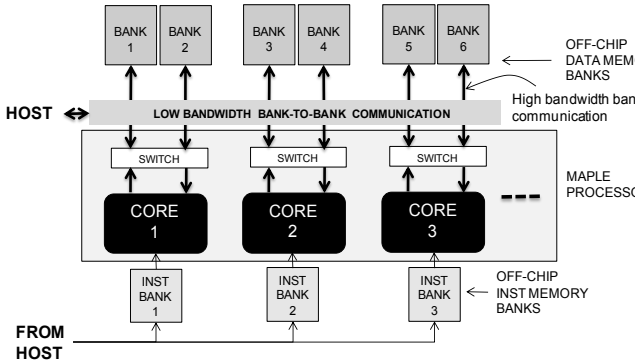

Figure 3: The MAPLE processor architecture

Figure 2 shows the details of the core architecture of MAPLE. A core has P vector PEs organized as H processing chains of M PEs each (P=H*M). Each chain has a bi-directional nearest neighbor interconnect ("intra-chain") along which inputs are propagated from left to right and outputs from right to left. The first PE in every chain accepts inputs from an input buffer (labeled input local store). Each PE also has a private local store which can be written with data from off-chip. A PE chain sends its outputs to the smart memory block, one of which is available to each processing chain, which performs in-memory processing like array ranking, finding min/max or aggregation. Each PE takes two vector operands as inputs, one from its local store, and the other streaming from the input buffer.

MAPLE (Figure 3) has C processing cores each connected to two off-chip memory banks. It is connected to a general-purpose host computer via a communication interface such as PCI. A high bandwidth bus connects each memory bank to its corresponding core. A switch enables the core to alternate between memory banks for inputs and outputs, or use both banks for inputs or outputs. Each core also has a separate instruction memory bank that is written by the host. The host can also write to the data memory banks via a slower bank-to-bank interconnection network. The architecture is tailored to applications that can be parallelized into separate memory-processor core "channels", with infrequent communications across the channels. The memory architecture can scale by increasing the number of banks.

Processing Elements and their Interconnection: The PEs, shown in Figure 4(A), perform standard ALU, multiply and multiply-accumulate operations in a single cycle. A PE is a simple vector processor with two operands - one from the PE on its left via the intra-chain interconnect, and the other from its private local store. The intra-chain interconnect bus is M words wide and matches the number of PEs in the chain. Thus the PE chain can perform up to M vector operations at a time. A PE can select any word from its intra-chain interconnect bus, leading to

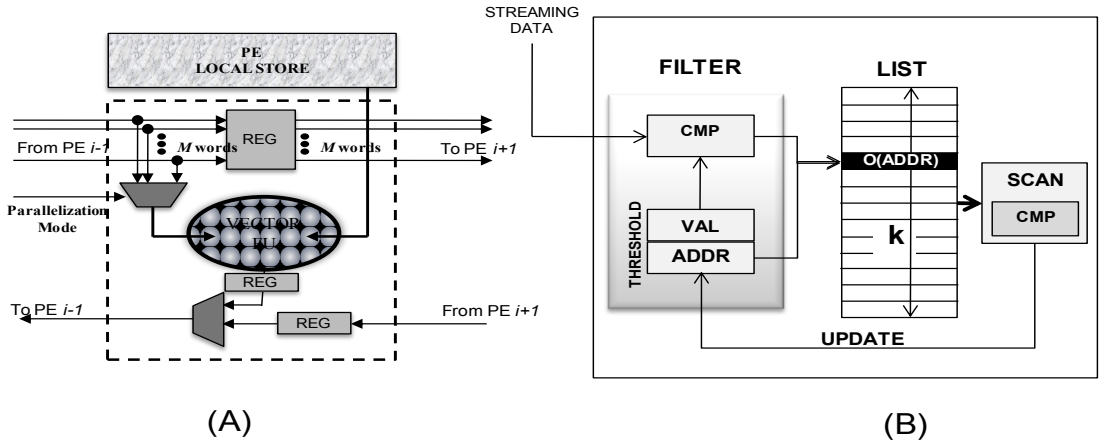

Figure 4: The MAPLE (A) PE and (B) smart memory block

different parallelization modes for the chain: the M PEs in a chain can operate on M different streaming words as well as on the same word. The PE stores outputs to its smart memory block and can continue processing the next vector in the next cycle. Unless the smart memory block issues a stall, a store takes M cycles. This latency can be hidden by the next vector operation if the vector size is large enough. To facilitate indexing of results (a feature required for K-means and GVLQ), a PE also sends its ID along with the results to be stored.

Smart Memory Blocks: Each chain in a MAPLE core has a memory block capable of two atomic store operations. The first is a variable latency store for selection and ranking operations in large arrays, and the second a read-modify-write. The memory block can be written to only by the processing chain, and is read by the reduce network. Figure 4(B) shows relevant architectural components of the smart memory architecture for selecting the top k elements in an array given a function to compare two elements: (i) a filter with the programmed compare function (CMP), a threshold value (VAL) and threshold address (ADDR), (ii) a list of k elements (LIST) and (iii) a hardware list scanner. The array elements are streamed in and compared with threshold VAL. If the comparison succeeds for an array element, it replaces VAL located at ADDR in LIST. The scanner then scans LIST to find a new threshold value and address and updates the filter. If the comparison fails, the element is discarded. When k is small compared to the array size, there are more discards than insertions. In the event of an insertion, the store operation stalls the processor in order to scan LIST and update the filter.

4.2 Mapping applications onto MAPLE

In order to program MAPLE, the user expresses application kernels in terms of a primary operation (i.e., matrix multiplication) and a reduction function (e.g., finding max/min for K-means and GLVQ, ranking arrays for SSI). At a low level, MAPLE is programmed using specialized assembly. In order to free the programmer from low level programming issues, we provide a tool that, given the input matrices and the reduction function: (i) maps the data onto input- and PE- local stores, (ii) maps data and reduction operations onto the smart memory blocks, and (iii) generates the assembly used to program MAPLE.

MAPLE's design goals are to handle matrices of various sizes, and minimize intermediate data by performing reduction operations using smart memory blocks. Data placement and smart memory configuration are key aspects. The mapping algorithm determines the data placement by analyzing the sizes of input matrices A and B. Assuming that A is streamed from the input local store and B is stored in the PE local stores, a fundamental output of the mapping process is the parallelism mode parameter that determines how B is split across PEs.

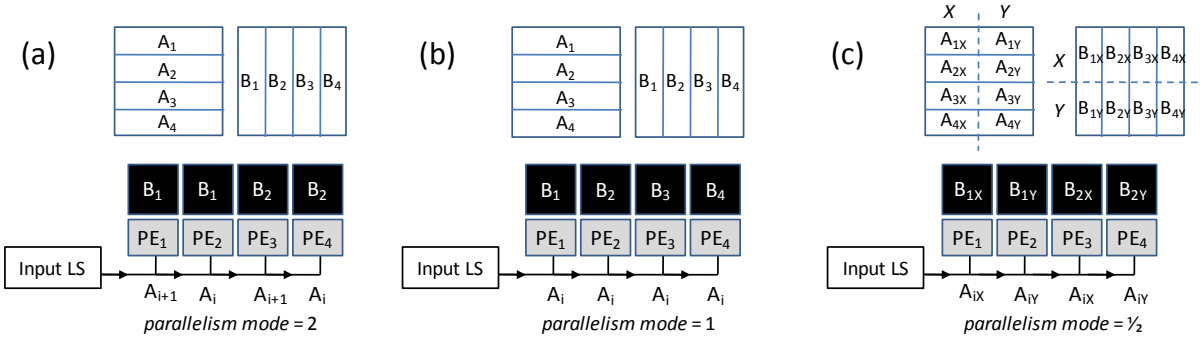

Figure 5: Mapping matrices to MAPLE with different *parallelism modes:* (a) each column of B duplicated across 2 PEs (parallelism doubled), (b) 1 column of B per PE and (c) columns of B split into X and Y to fit the PE local stores (parallelism halved).

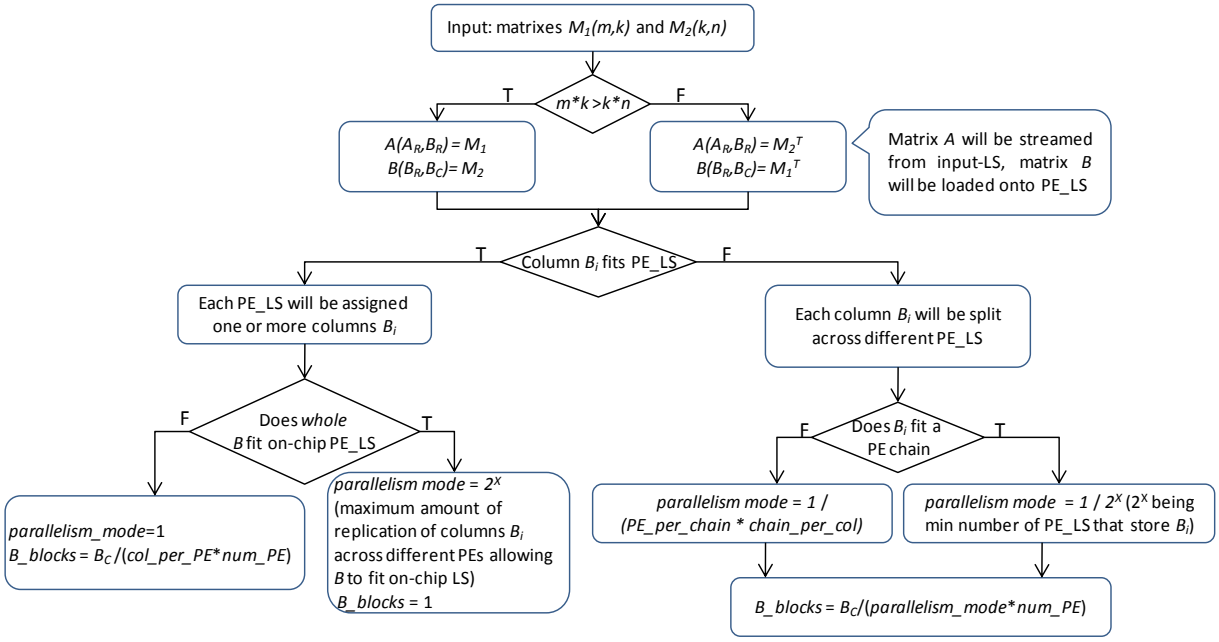

Figure 6: Block diagram showing the mapping of matrices M_1 and M_2 onto the accelerator

Figure 5 illustrates how the parallelism mode *pm* affects the mapping. If *pm* is greater than one, each column of the matrix B is replicated on pm PEs. Those PEs will process different rows of matrix A concurrently, and the results will relate to different rows of the output matrix. If *pm* is less than one, each column of B is split into 1/*pm* portions stored on different PEs. Rows of A are split as well and properly distributed to the PEs. In this case, the smart memory will need to accumulate results from the different PEs processing the same column before performing the reduction operation. Therefore, the parallelism modes affects: (i) the data placement in the PE local stores, (ii) the data distribution from the input local store to the PEs, (iii) the smart memory configuration.

Figure 6 shows how two matrices are mapped onto MAPLE. The mechanism can be generalized to multiple matrices. It is based on the larger matrix (A) streaming row-wise from the input local store, and the smaller matrix (B) mapped column-wise onto the PE local stores. Either matrix may be blocked if it is too large to fit its memory. For matrix B, each local store will potentially accommodate one or more columns. If B is small, the same column is mapped onto different PEs (*pm* > 1). In this case, PEs containing the same column of B will process different rows of A concurrently. If the columns of B are too large to fit in a single PE local store, they are split over multiple PEs (*pm* < 1). During operation, the rows of A will be split and directed to the correct PE.

```
SET_PARALLEL_MODE parallelism_mode          ; sets the parallelism mode
SET_SM_REDUCTION reduction_operation        ; configures the reduction performed by smart memories
SET_A_NUM_ROWS a_num_rows                   ; sets number of A rows present in each A_block
SET_B_COL_SZ b_col_size                     ; sets the size of the portion of B column fitting a PE local store
SET_B_NUM_COLS b_num_cols                   ; sets the number of B columns stored in each PE local store
for each A_block {                          ; A_blocks consist of A rows
   WRITE_A A_block                          ; transfers an A_block from DRAM into input local store
   for each B_block {                       ; B_blocks consist of B columns
      WRITE_B B_block                       ; transfers a B_block from DRAM into input local store
      SET_INPUT_LS_ADDR 0                   ; resets the active address in input local store
      for each A_row_group {                ; A_row_group consists of A rows processed concurrently
         for b_col: 0..b_num_cols {
            SET_PE_LS_ADDR b_col*b_col_size ; sets the address in PE_mem to load the B data from
            SET_SM_ADDR result_addr         ; sets the address in smart memory for partial results
            MULT_ACC_DUMP b_col_size        ; performs b_col_size MACC and sends result to smart mem
         }
         INC_INPUT_LS_ADDR sz(A_row_group)  ; increments the address in input local store to read from
      }
   }
   DUMP_SM                                  ; dumps the content of smart memory
}
```

WORK LOAD	Off-chip Data	On-chip Data	Computational Bottleneck (after transformation)
SSI	D documents stored as D x C matrix	Q queries stored as C x Q matrix	• Distribute query matrix: 1 column per PE chain • Stream document matrix through all PE chains • Each PE chain computes distance between query and all documents • Each PE chain streams results into its smart memory • Smart memory extracts top K on-the-fly
CNN	Input images	CNN kernels	• Distribute kernels column-wise across PEs • Stream images through all PEs • Program smart memory to aggregate results in-place
K-means	N points stored as N x D matrix	K means stored as D x K matrix	• Distribute means: 1 mean per PE chain • Stream points matrix through all PE chains • Each PE chain streams distance of mean to all points into its smart memory • Smart memory identifies closest mean to each point
SVM	N training vectors stored as N x D matrix	1 vector of size D	• Vector stored in all PEs • Training vector streamed / broadcast to all PEs • Each PE computes one result of the output vector • Smart memory collects results in batches and sends off-chip
GLVQ	N reference vectors stored as N x D matrix	T test vectors stored as T x D matrix	• Distribute test vectors: 1 test vector per PE chain • Stream reference vectors through all PE chains • Each PE chain computes distance between test vector and all reference vectors • Smart memory identifies closest and farthest reference vector to each point

Figure 7: Mapping our workloads onto MAPLE.

The pseudo-code above shows the assembly code generation. Bold keywords represent generated assembly directives, and italicized keywords represent configuration variables provided by the user or parameters produced by the mapping tool. A_blocks and B_blocks are row-wise and column-wise portions of A and B matrixes fitting the input- and the PE-local stores, respectively. The example assumes partial results computed on A-blocks fit the smart memory. SET_PARALLEL_MODE affects how B is mapped onto the PE local stores, potentially with column replication (*pm* > 1) or splitting (*pm* < 1). It also affects how the rows of A are distributed to the PEs, as well as the smart memory configuration. SET_SM_ADDR instructs the first PE of each chain; the remaining PEs are automatically configured based on *pm*. If B is a (*BR*,*BC*) matrix, b_col_size is equal to *BR* if *pm* is greater than one, and to *BR***pm* otherwise.

Taking SSI as an example, the user may provide the following inputs: document matrix of size 2M document x 64 categories, query matrix of size 64 categories x 64 queries, and reduction operation equal to top-64 ranking. The mapping phase will set matrix A to be document matrix, and matrix B to be the query matrix. Additionally, assuming 4B data, 32 chains of 8 PEs each, 2KB PE local stores and 64KB input local stores, the mapping algorithm will produce the following parameters to configure the assembly generator code: *pm* = 8, *B_block* = 1, *A_block* = 7813, *a_num_rows* = 256 (zero-padding is performed in the last *A_block*), *b_col_size* = 64 and *b_num_cols* = 2. Figure 7 summarizes the outcome of the mapping process on the considered workloads.

Table 2: Architectural exploration setup

Type	Parameter	Value
Off-chip Memory Organization	Number of banks	4
	Bandwidth per bank	8 words/cycle
Processor	PE budget	512
	Cores (C)	2
	Chains (H)	Variable
	Processors/chain (M)	Variable

4.3 Architectural exploration

We developed a C++ cycle-accurate simulator that takes as input assembly code for applications mapped to MAPLE and an architectural configuration file that specifies the off-chip memory architecture (banks and bandwidth) and processor layout, and produces an estimate of MAPLE's execution time.

Table 2 shows the different parameters of MAPLE. We seek to find, given an off-chip memory organization and processor budget, the processor layout (chain size and number of chains) that maximizes performance for different applications. We use different instances of SSI, CNN and K-means as examples to explore the architectural design space.

We used an SSI instance of 2M documents, each expressed as a vector of size 100, and extracted the top 32, 64 and 128 best matches (i.e., K=32, 64, 128). We simulated this for 1024 queries across various chain lengths (Figure 8(A)). Because of MAPLE's smart memories that rank arrays dynamically, the number of cycles was largely insensitive to K. The processor layout with the best performance was when the chain size was 6-8 processors. This is because in our SSI mapping, each PE chain compares the same set of documents with a different query. Since the documents are broadcast from the off-chip memory bank, the

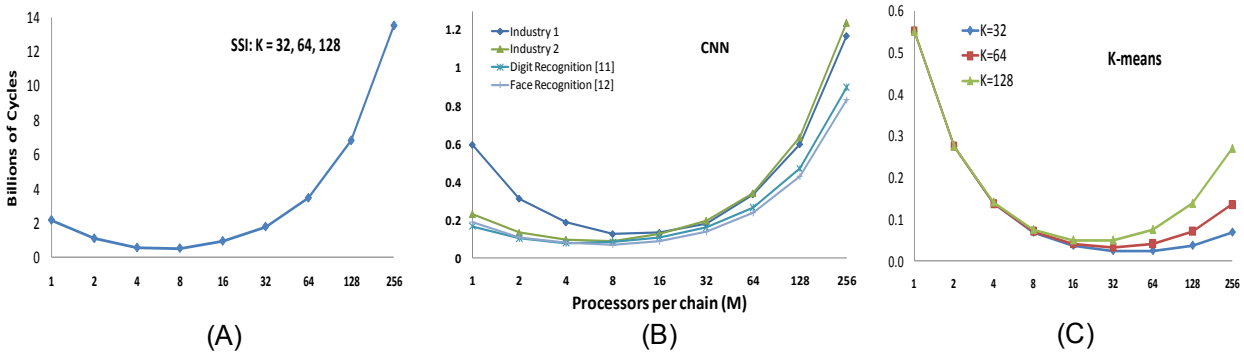

Figure 8: MAPLE architectural exploration for SSI, CNN and K-means.

Table 3: Effect of MAPLE's in-memory processing on off-chip load/stores

Workload	Parameters	Off-chip Accesses (Bytes)		Off-chip Bus Transactions		Reduction
		No SM	With SM	No SM	With SM	
SSI	2M docs, 64 queries, 32-128 top K	687M	419M	42M	26M	1.64x
CNN	5 networks (Figure 9(B))	993M	12.9M	62M	0.81M	76x
K-means	200K points, 32-128 means	64.7M	25.2M	40.5M	1.57M	25.7x

Table 4: Speedup due to in-memory processing

Workload	Parameters	Execution Time (sec)		Speedup
		No SM	With SM	
SSI	2M docs, 64 queries, 32-128 top K	4.12	0.24	17.2x
K-means	200K points, 32-128 means	1.52	0.61	2.46x

performance is best when the chain can consume as many documents as the memory can provide.

Figure 8(B) shows MAPLE's performance for 4 CNN networks. The best performer ranges between M=4 to M=16. This is due to the fact that CNNs are compute-bound, and the amount of parallelism across networks. For instance, networks with many kernels can be converted into a large matrix-matrix multiplication, for which a larger chain is better.

We present K-means in Figure 8(C). The best chain sizes ranges from 64 down to 16 as K varies from 32 to 128. This is because when we map K-means to MAPLE, the number of points that can be multiplied with the means in parallel decreases as the number of means increases. Therefore a "tall, skinny" configuration with small chain sizes suits large values of *K*.

4.4 Effect of in-memory processing

The primary advantage of in-memory processing is reduced off-chip accesses and, as a consequence, improved performance. Using the simulator, we evaluate the extent of off-chip accesses reduced, as well as the performance boost this provides MAPLE. Table 3 shows the number of bytes loaded and stored to off-chip memory with and without in-memory processing (i.e., with and without the smart memory). It also shows the number of bus transactions. We consider SSI, CNN and K-means, and average the results across different instances of each application. For SSI, the smart memory performs array ranking, while for CNN it performs aggregation and for K-means, it computes the minimum. The reduction in the number of off-chip accesses ranges from 1.6x to 76x.

We also use the simulator to compute the actual execution time of SSI and K-means with and without in-memory processing. Table 4 shows that without performing in-memory processing for array ranking, the SSI execution time on MAPLE increases by a factor of 17. For K-means, if the minimum computation is not performed in-memory, the execution time increases by almost 2.5x. The speedups can be attributed to reducing off-chip memory accesses as well as to overlapping the secondary and primary operations.

5. PROTOTYPE AND EXPERIMENTAL RESULTS

In this section, we present the MAPLE prototype and its measured performance. We built the prototype using an off-the-shelf FPGA board from AlphaData. Our architectural design space exploration, along with FPGA constraints, determined the specific prototype architecture. We implemented each of the five workloads on the prototype.

We compare the prototype to (i) optimized parallel software implementations, (ii) available and published GPU implementations of SSI, CNN and SVM from [11] and [9] and (iii) FPGA-based algorithm-specific implementations of CNN and SVM from [23] and [27]. Table 5 shows the experimental setup.

Table 5: Experimental setup

MAPLE FPGA Prototype	PEs (total)	512 organized as 2 cores, 32 chains/core, 8 PEs/chain
	Clock speed	125 MHz
	Memory	4 banks DDR2, bandwidth 8 words per bank (8GB/s total)
	FPGA	Xilinx Virtex 5 SX240T
	Host interface	64-bit, 66MHz PCI
Software	2.5 GHz quad-core Xeon, Intel MKL library	
GPU	NVIDIA Tesla C870, 1.3GHz, 128 cores, CUDA 2.3 with CUBLAS library	

5.1 SSI and CNN

We implemented SSI in software using BLAS for matrix multiplication, and an optimized multi-threaded implementation for array ranking. We compared this to a MAPLE implementation of SSI. Figure 9(A) shows the performance in ms/query for document database sizes ranging from 256K to 10M, and for K=32 and K=128. For each, the 2 bars show optimized software speed and measured prototype speed to process 64 text queries. We find the MAPLE prototype to be up to 50% faster than the optimized software, We also used the prototype to search 1.8 million Wikipedia documents, a dataset from [3], and obtained a speed of 4.63 to 4.88 ms/query for K=32 and K=128 respectively.

We measured speeds of five CNNs performing face and digit recognition, surveillance and automotive safety for 640x480 images. Figure 9 (B) shows the speed in milliseconds per frame measured with the parallel software and the MAPLE prototype.

Now we compare MAPLE's performance with available GPU implementations of SSI and CNN. Table 6 compares software, GPU and the MAPLE prototype. For SSI, we used NVIDIA's CUBLAS library and array rank routines, while for CNN, we use GPU numbers from [11]. Compared to the GPU, the MAPLE prototype is 3x faster for SSI and about 50% faster for CNN.

Table 6: SSI and CNN performance comparisons

Workload Details	Software	GPU	MAPLE Prototype
SSI: 2M docs, 32 queries, 128 top K	52.8 ms/query	11.4 ms/query	3.76 ms/query
CNN: Face Recognition [11]	5 fps	9.5 fps ([11])	13 fps

5.2 K-means

Figure 10 shows data for K-means. MAPLE finds the closest mean for every point and transfers that information to the host, which averages all the closest points to each mean. Since successive iterations cannot be overlapped, we consider the data transfer time and the host component to compute the effective speedup. Figure 10(A) breaks down the running time of K-means on MAPLE into the core MAPLE execution, data transfer and host execution. The data transfer time increases with the number of points, but the host execution is larger and responsible for reducing speedups. Figure 10 (B) shows speedups of the prototype over parallel software. The data shows that MAPLE's speedup is largely independent of the number of points (and that is due to the fact that K-means can be easily parallelized by partitioning the points), but increases with the number of means.

5.3 GLVQ

We implemented GLVQ [7] training and testing on the prototype, and used it for eye detection in images. The data had 128 images each represented as reference vectors of dimension 512. 64 vectors represented different eye images and 64 non-eye images. The training data set had 5400 images, and the testing data 240 images. Training images are processed sequentially, as each incrementally modifies the model. This also means considerable host-accelerator communication. Table 7 shows the performance for the eye detection data. Considering the substantial transfer time, the projected speedup for training is 3x, but is much higher (9.5x) for testing where data may be transferred in bulk.

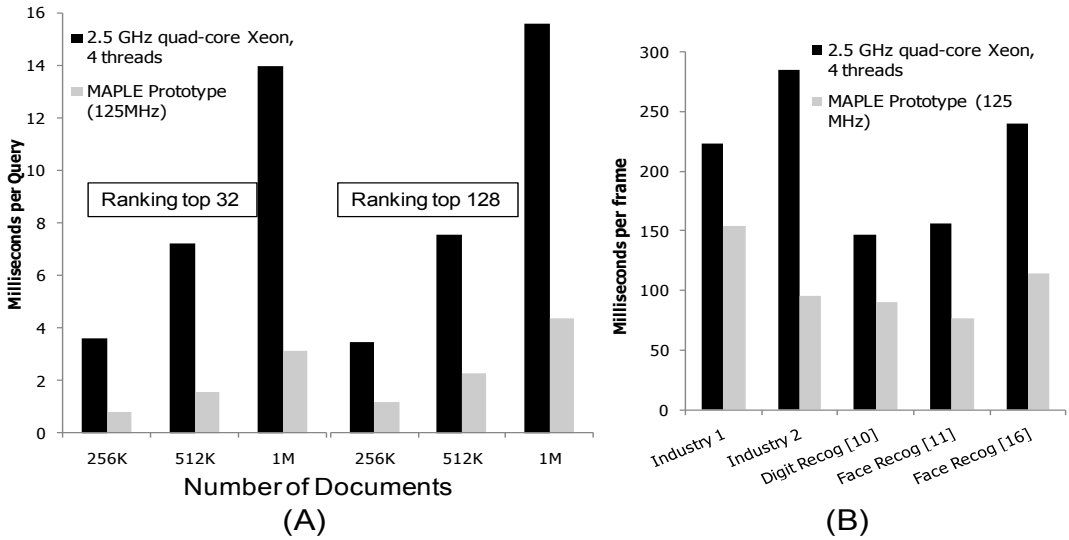

Figure 9: MAPLE prototype performance vs optimized software for (A) SSI and (B) CNN

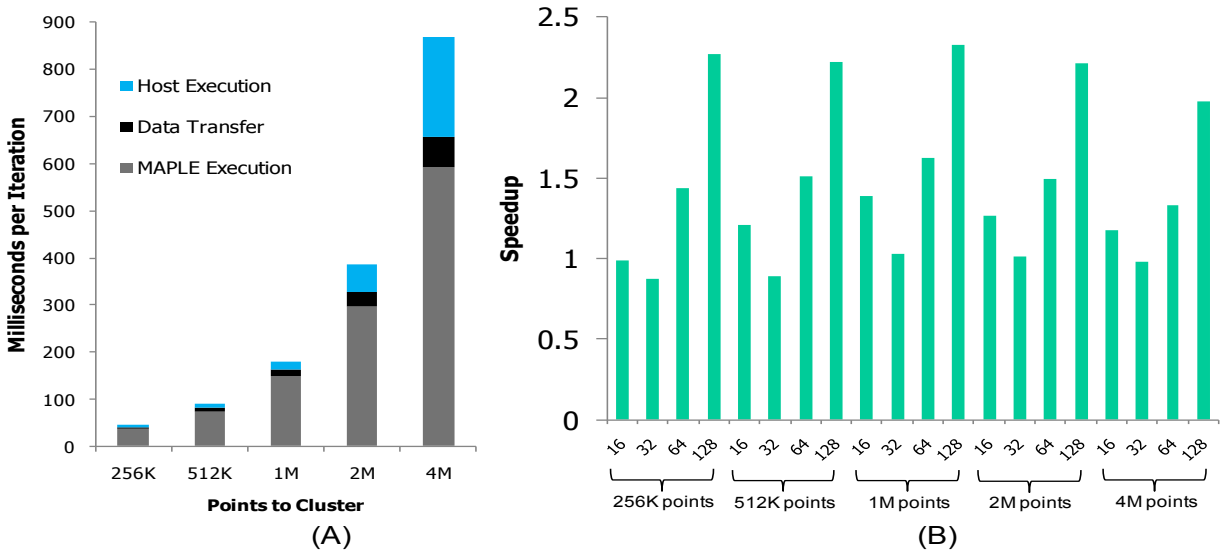

Figure 10: K-means performance on MAPLE prototype

Table 7: GLVQ training and testing performance (in seconds) for the eye-detection data set

	# Vectors	Vector Size	Classes	Vectors per Class	Software Time (s)	MAPLE Prototype 512 PEs, 125 MHz			
						Data Xfer	SW	HW	Speedup
Training	5400	512	2	64	1.7304	0.5196	0.03	0.55	2.97
Testing	230	512	2	64	0.2357	0.0118	0	0.01	9.58

5.4 SVM

In SVM, the "kernel computation", which involves multiplication of test or training vectors with the large training or support vector matrix, is mapped to MAPLE. We compare MAPLE with a recent GPU implementation of SVM [9] as well as an FPGA implementation [27]. An iteration involves multiplying the training set matrix with 2 training vectors; typically tens of thousands of iterations are required to complete training. Table 8 shows SVM training performance in milliseconds per iteration for six data sets [9]. The large size of the training matrix renders this problem memory bound. While the prototype underperforms the GPU implementation, we note the GPU data sets from [9] are small. For instance, MNIST uses only 60K training vectors. As we show in the next section, MAPLE's performance scales well for MNIST with 2M training vectors. Further, compared to the FPGA prototype, the GPU has a higher memory bandwidth and faster, custom circuitry. If a custom MAPLE processor were built, it could benefit from similar considerations.

Table 8: SVM training performance (millisec / iteration)

Data Set	Training Set Size	Dim	MAPLE Prototype	[9]
Adult	32,561	123	3.2	0.76
Web	49,749	300	9.27	3.56
MNST	60,000	784	25.69	12.76
USPS	7,291	256	1.22	0.15
Forest	561,012	54	35.26	6.09
Face	6,977	381	1.6	0.27

5.5 Algorithm Specific Implementations

We compare the MAPLE prototype performance with algorithm-specific FPGA-based implementations of SVM and CNN from [27] and [23] (Table 9). For SVM, [27] reports 9.3 billion MACs per second for the MNIST dataset with 2M training vectors. The MAPLE prototype achieves nearly half that speed. For CNN, the MAPLE prototype matches the speed reported in [23].

6. CONCLUSION

We described a programmable parallel accelerator that can handle several learning and classification algorithms. We profile and analyze five representative workloads to identify their computational bottlenecks. We find the core computations of these workloads can be transformed into a matrix or vector operation producing large intermediate data which is then reduced by a secondary operation. We architect the accelerator to leverage this characteristic by provisioning many simple, parallel PEs and in-memory processing. The in-memory processing obviates the need for off-chip memory loads and stores. We also present a compilation scheme to automatically map application kernels to the accelerator. An FPGA-based prototype of the

Table 9: MAPLE prototype vs algorithm specific accelerators

	Algorithm specific	MAPLE Proto.
SVM: MNIST	9.3 GMACs/sec [27]	4 GMACS/sec
CNN: Face Recog.	10 frames / sec [23]	10.5 frames / sec

accelerator demonstrates measured speedups over optimized, parallel software implementations as well as GPU implementations of some of our learning and classification workloads.

REFERENCES

[1] Mei, T., Hua, X., Yang, L., Li, S., "VideoSense: towards effective online video advertising," *Proc. 15th International Conference on Multimedia* 2007, pp 1075-1084.

[2] Datta, R., et al.,"Image retrieval: Ideas, influences, and trends of the new age," *ACM Comput. Surv.* 40,2,Apr 08.

[3] Bai, B., Weston, J., Grangier, D., Collobert, R., Sadamasa, K., Qi, Y., Chapelle, O., Weinberge, K., "Learning to Rank with (a lot of) word features," *Special Issue: Learning to Rank for Information Retrieval. Information Retrieval,* 2009.

[4] Lecun, Y., Bottou, L., Bengio, Y., Haffner, P., "Gradient-based learning applied to document recognition," *Proc. of the IEEE*, vol.86, no.11, pp.2278-2324, Nov 1998.

[5] MacQueen, J. B., "Some methods for classification and analysis of multivariate observation," *Proc. Berkeley Symp. on Math. Stat. and Prob.*, pages 281–297.

[6] Platt, J., "Fast Training of Support Vector Machines Using Sequential Minimal Optimization," in *Advances in Kernel Methods – Support Vector Learning*, MIT Press 1999.

[7] Sato, A., Yamada, K., "Generalized learning vector quantization", *Neural Information Processing Systems,* pp.423-429, 1995.

[8] Graf, H. P., Cadambi, S., Durdanovic, I., Jakkula, V., Sankaradass, M., Cosatto, E., Chakradhar, S. T., "A Massively Parallel Digital Learning Processor," *Neural Information Processing. Systems*, Dec. 2008.

[9] Catanzaro, B., Sundaram, N., Keutzer, K., "Fast Support Vector Training and Classification on Graphics Processors," *Machine Learning, 25th International Conference on, (ICML 2008)*, Jul. 2008.

[10] Chellapilla, K., Puri, S., Simard, P., "High Performance Convolutional Neural Networks for Document Processing," *Tenth International Workshop on Frontiers in Handwriting Recognition* (2006).

[11] Nasse, F., Thurau, C., Fink, G. A., "Face Detection Using GPU-Based Convolutional Neural Networks," *Computer Analysis of Images and Patterns, 13th International Conference, CAIP 2009,* Proc.. LNCS 2009.

[12] Collobert, R., Weston, J., "A unified architecture for natural language processing: deep neural networks with multitask learning," *Proc. of the 25th International Conference on Machine Learning*, vol. 307, pp.160-167, Jul 2008.

[13] Lloyd, S.P., "Least squares quantization in PCM," *IEEE Transactions on Information Theory* 28 (2): pp 129–137.

[14] Hall, J. D., Hart, J. C., "GPU Acceleration of Iterative Clustering," *The ACM Workshop on General Purpose Computing on Graphics Processors* and SIGGRAPH 2004 poster, Aug 2004.

[15] Cosatto, E., Miller, M., Graf, H. P., Meyer, J., "Grading Nuclear Pleomorphism on Histological Micrographs," *Proc. Int. Conf. Pattern Recognition*, 2008.

[16] Lawrence, S., Giles, C.L., Ah Chung Tsoi, Back, A.D., "Face recognition: a convolutional neural-network approach," *Neural Networks, IEEE Transactions on*, vol.8, no.1, pp.98-113, Jan 1997.

[17] M D Taylor et al, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," IEEE Micro, vol. 22, no. 2, pp. 25-35, Mar./Apr. 2002.

[18] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M,, Krueger, J., Lefohn, A.E., Purcell, T.J., "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, 26(1):80–113, 2007.

[19] Burger, D, et al. "Scaling to the End of Silicon with EDGE Architectures," *IEEE Computer*, **37**(7), pp. 44-55, July 2004.

[20] Diamond, J. R., Robatmili, B., Keckler, S. W., van de Geijn, R., Goto, K., and Burger, D. 2008. "High performance dense linear algebra on a spatially distributed processor," Proc. *13th ACM SIGPLAN PPoPP 2008.*

[21] Seiler, L., et al., "Larrabee: a many-core x86 architecture for visual computing," In *ACM SIGGRAPH 2008*.

[22] Kapasi, U.J., Rixner, S., Dally, W.J., Khailany, B., Jung Ho Ahn, Mattson, P., Owens, J.D., "Programmable stream processors," IEEE *Computer*, vol.36, no.8, pp. 54-62, Aug. 2003.

[23] Sankaradas, M., et al, "A Massively Parallel Coprocessor for Convolution Neural Networks", In *Proc. 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP),* 2009, Boston, MA.

[24] Zhuo, L., Prasanna, V. K., "High Performance Linear Algebra Operations on Reconfigurable Systems", in *ACM/IEEE Conference on Supercomputing*, *Proc. of the 2005,* November 2005.

[25] Rousseaux, S., Hubaux, D., Guisset, P., Legat, J., "A High Performance FPGA-Based Accelerator for BLAS Library Implementation," *Proc. of the Third Annual Reconfigurable Systems Summer Institute (RSSI'07).*

[26] Raina, R., Madhavan, A., Ng, A. Y., "Large-scale deep unsupervised learning using graphics processors," Proc. *26th Annual international Conference on Machine Learning* 2009.

[27] Cadambi, S., et al, "A Massively Parallel FPGA-based Coprocessor for Support Vector Machines", Proc. *IEEE Symposium on FCCM* 2009*,* Napa, CA.