A Massively Parallel, Energy Efficient Programmable Accelerator for Learning and Classification

ABHINANDAN MAJUMDAR, SRIHARI CADAMBI, MICHELA BECCHI,
SRIMAT T. CHAKRADHAR, and HANS PETER GRAF, NEC Laboratories America, Inc.

Applications that use learning and classification algorithms operate on large amounts of unstructured data, and have stringent performance constraints. For such applications, the performance of general purpose processors scales poorly with data size because of their limited support for fine-grained parallelism and absence of software-managed caches. The large intermediate data in these applications also limits achievable performance on many-core processors such as GPUs. To accelerate such learning applications, we present a programmable accelerator that can execute multiple learning and classification algorithms. To architect such an 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. Our 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 dynamic in-memory processing where on-chip memory blocks perform the secondary reduction operations. 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 allow MAPLE to scale its performance with data size. We also present an Atom based energy-efficient heterogeneous system with MAPLE as the accelerator that satisfies the application's performance requirements at a lower system power. This article describes the MAPLE architecture, explores its design space with a simulator, illustrates how to automatically map application kernels to the hardware, and presents its performance improvement and energy benefits over classic server-based implementations. We 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 clock rate. With MAPLE connected to a 1.6GHz dual-core Atom, we show an energy improvement of 38-84% over the Xeon server coupled to a 1.3 GHz 240 core Tesla GPU.

Categories and Subject Descriptors: C.1.2 [Processor Architecture]: Multiple Data Stream Architectures (Multiprocessors); C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems

General Terms: Design, Performance

Additional Key Words and Phrases: Accelerator-based computing, parallel computing, heterogeneous computing, machine learning, architecture

ACM Reference Format:

Majumdar, A., Cadambi, S., Becchi, M., Chakradhar, S. T., and Graf, H. P. 2012. A massively parallel, energy efficient programmable accelerator for learning and classification. *ACM Trans. Archit. Code Optim.* 9, 1, Article 6 (March 2012), 30 pages.

DOI = 10.1145/2133382.2133388 <http://doi.acm.org/10.1145/2133382.2133388>

This article is an extension of the conference paper, "A programmable parallel accelerator for learning and classification," presented at the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10).

Authors' address: A. Majumdar, S. Cadambi, M. Becchi, S. T. Chakradhar, and H. P. Graf, NEC Laboratories America Inc., 4 Independence Way, Suite 200, Princeton, NJ, 08540; email: {abhi, cadambi, mbecchi, chak, hpg}@nec-labs.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permission may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1544-3566/2012/03-ART6 \$10.00

DOI 10.1145/2133382.2133388 <http://doi.acm.org/10.1145/2133382.2133388>

1. INTRODUCTION

Applications that examine raw, unstructured data in order to draw conclusions and make decisions are becoming ubiquitous. Banks and credit card companies, for instance, analyze withdrawal and spending patterns to prevent fraud or identity theft. Online retailers study Web site traffic patterns in order to predict customer interest in products and services based on prior purchases and viewing trends. Semantic querying of text and images, which has wide-ranging, mass market uses such as advertisement placement [Mei et al. 2007] and content-based image retrieval [Datta et al. 2008] 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 [Bai et al. 2009] must search millions of documents at a few milliseconds per query. From our own experiments, a 2.5 GHz quad-core Xeon server processes 64 queries with a throughput of 61 ms/query when it searches for top 64 semantically similar documents from a database of 2M documents. A 1.3 GHz 240 core Tesla GPU, on the other hand, processes at a rate of 9.5ms/query. Another example is face and object recognition in high resolution video, that is often done with Convolutional Neural Networks (CNNs) [Lecun et al. 1998]. A server performing this task must search VGA (640×480) 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 [Nasse et al. 2009] can reach 10 frames per second, both falling far short of requirements. Other similar workloads include digital pathology [Cosatto et al. 2008], automotive applications to predict failures and reduce recalls, financial analytics and cognitive databases.

Motivated by this gap between machine learning 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 [Bai et al. 2009], Convolutional Neural Networks [Lecun et al. 1998], K-means [MacQueen 1967], Support Vector Machines [Platt 1999] and Generalized Learning Vector Quantization [Sato and Yamada 1995], 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, that is, 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 a many-core accelerator called MAPLE (Massively Parallel Learning/Classification Engine) which consists of hundreds of simple vector processing elements (PEs) with two key features that directly address the above workload characteristics. First, its 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 [Catanzaro et al. 2008; Graf et al. 2008], CNNs [Chellapilla et al. 2006; Nasse et al. 2009] and K-means [Hall and Hart 2004], 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.

As such workloads become more widely used, such computations will be increasingly performed not only in high-end servers, but also in embedded platforms such as automobiles and store cameras. Typical compute servers comprising high-end CPUs and GPUs are too power hungry to be deployed in such scenarios, which have tight power budgets. For example, an Intel® XEON® X7460 consumes 130W of power [X7460 Power] while NVIDIA's Tesla C2050/C2070 GPU is rated as 247W [C2050/C2070 Power], both far too high to sustain in a car or camera. As a natural extension to our study in this article, we also investigate if the MAPLE accelerator can be used as a low-power server that executes learning applications without compromising performance. Specifically, we present a heterogeneous system constructed with MAPLE coupled to a low power Atom processor [Intel Atom]. Using three representative learning and classification workloads, we find that our Atom-based system consumes much lesser energy while largely meeting performance constraints, thereby making it suitable for possible deployment within embedded platforms.

To this end, in this article we make the following contributions.

- We present the architecture of MAPLE, a parallel accelerator for learning and classification.
- We evaluate the use of in-memory processing for learning and classification applications.
- We present a strategy to automatically map application kernels to the MAPLE architecture.
- Using an FPGA prototype, we compare MAPLE's performance against parallel, optimized software implementations of learning and classification algorithms on multicores and GPUs.
- We present a low-power system consisting of the MAPLE accelerator with an Intel Atom processor as the host and show its energy benefits.

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. Section 5 describes the details of MAPLE programming, and Section 6 explores the architectural design space of MAPLE. In Section 7, we present our FPGA prototype and performance measurements, while Section 8 evaluates energy of the Atom-based MAPLE-accelerated system. We conclude in Section 9.

2. RELATED WORK

Prior work in accelerating learning and classification workloads can be classified broadly into four categories: (i) optimized, parallel libraries for multicore CPUs, (ii) optimized implementations on graphics processors (GPUs) [Catanzaro et al. 2008; Chellapilla et al. 2006; Hall and Hart 2004; Nasse et al. 2009], (iii) algorithm-specific accelerators on FPGAs [Graf et al. 2008] and (iv) other embedded and analog hardware implementations.

Multicore CPUs and many-core GPUs [Owens et al. 2007; Seiler et al. 2008] accommodate diverse learning and classification workloads through programmability. However, multicores cannot avail the fine-grained data parallelism inherent in these workloads due to thread synchronization overheads and inadequate memory bandwidth. In addition, GPUs do not have in-memory processing, and require multiple independent parallel streams to be coalesced and synchronized. Uncoalesced loads and stores from random off-chip locations result in increased memory latency, and thus lead to reduced performance. Furthermore, neither CPUs nor GPUs have enough on-chip storage to handle the large intermediate data generated by these applications. For instance, a semantic search algorithm that processes 64 queries to retrieve the top 64 matching entries from a database of 2M documents generates 512MB intermediate data. A 2.5GHz dual-core Xeon with 12MB L2 cache, as well as a 1.3 GHz Tesla GPU with 16KB blocks of on-chip storage cannot cache this intermediate data. In addition, the poor caching behavior of some of these applications on general purpose processors and the lack of in-memory processing in GPUs result in many off-chip accesses making performance worse. In this article, 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 [Cadambi et al. 2009], CNNs [Sankaradas et al. 2009] and deep learning [Raina et al. 2009]. There are also architectures [Burger et al. 2004] and FPGA implementations that accelerate matrix computations [Rousseaux et al. 2007; Zhuo et al. 2005]. 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 [Sankaradas et al. 2009] and [Cadambi et al. 2009].

For energy level optimization, FAWN [Vasudevan et al. 2009] targets I/O and seek-bound applications, while Lim et al. [2008] suggest embedded processors for both server and media applications for optimal power and cost. Reddi et al. [2010] investigates the use of the Atom processor for Web search applications, while companies like SeaMicro are already marketing low-power Atom-based clusters for server applications [SeaMicro].

In our work, we perform a systematic study of the performance bottlenecks in representative learning and classification workloads, architect an accelerator to address those bottlenecks, prototype the accelerator and measure its performance comparing it with traditional server-class systems. As an extension, we use the accelerator to build a low-power system for embedded learning and classification applications.

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.

We use the five following algorithms.

- *Supervised Semantic Indexing (SSI)* [Bai et al. 2009]. Given a large number of documents and text-based queries, SSI ranks the documents based on their semantic similarity to the queries.
- *Convolutional Neural Networks (CNNs)* [Lecun et al. 1998]. CNNs are 2-dimensional neural networks used for pattern recognition in applications such as object and face

Fig. 1. Transforming each workload's bottlenecks into a common set of primitives.

Table I. 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	Dot prods: 25–50 Array rank: 0.001
CNN	1D, 2D, 3D convolutions	>99%	Compute bound	16–100
K-means	Minimum Euclidean distance	~ 96%	Marginally compute bound	1–3
SVM	Large matrix-vector multiplication	85–95% [Cadambi et al. 2009]	Memory bound	1
GLVQ	Minimum Euclidean distance	>99%	Memory bound	<1

detection [Chellapilla et al. 2006; Nasse et al. 2009], and recently even semantic text search [Collobert and Weston 2008].

- *K-means* [MacQueen 1967]. K-means clusters N points into K clusters. It is commonly used in computer vision for image segmentation.
- *Support Vector Machines (SVMs)* [Platt 1999]. Given labeled training data, SVM training finds support vectors that separate the data into distinct classes indicated by the labels.
- *Generalized Learning Vector Quantization (GLVQ)* [Sato and Yamada 1995]. GLVQ is a supervised learning algorithm that classifies an input into one of several classes.

We profile each algorithm using typical data set sizes (Figure 1, column 3), and summarize the characteristics in Table I. 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 2.5GHz 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 has 1 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, and find common characteristics as well as a common set of primitives that may then 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 [Bai et al. 2009], given D documents, we find k semantic best matches for each of Q parallel 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. As shown in Figure 1, these operations may be transformed into a matrix-matrix multiplication (by reorganizing all the document and query vectors into matrices). The matrix multiply operation produces a large intermediate result matrix, and array ranking then ranks each column of the intermediate result matrix to produce the final result.

CNN [Lecun et al. 1998] performs many convolutions between images and “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 to fit on-chip caches. Finally, GLVQ 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

In this section, we describe the architecture of MAPLE as an accelerator for learning and classification algorithms. From the workload analysis, we find that the architecture must support matrix and vector operations (both large as well as many 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 many small matrices as well a few large matrices. Second, we use in-memory processing to handle the intermediate data on-the-fly. By performing reduction operations using on-chip

Fig. 2. Architecture of a MAPLE processing core.

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 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.

The overall MAPLE accelerator is a coprocessor (Figure 3) with 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/PCI-X. A high bandwidth bus connects each memory bank to its corresponding core. A switch enables the core to alternate between its memory banks for inputs and outputs, or use both banks as inputs or as outputs. Each core also has its own 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 allows easy scalability by increasing the number of banks.

Fig. 3. The entire MAPLE processor architecture.

Fig. 4. The MAPLE (a) PE and (b) smart memory block.

4.1 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 word 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 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.

Fig. 5. MAPLE stall mechanism.

4.2 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 is 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 array element e , e will replace 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.

MAPLE's stall mechanism is exemplified in Figure 5 showing the H chains in a MAPLE core. Each chain has a COMPUTE phase that lasts for L cycles, L being the operand vector-size, and a STORE phase where the outputs of the M processors in the chain are stored in the memory block. Successive chains are separated by 1 cycle due to pipelining. If chain S incurs a stall, it can continue computing the next result but cannot store until the stall resolves. Other chains can also compute during a stall, but must wait to store their results. Thus if any chain stalls, all chains continue computing but wait to store results. Thus, input streams that are already being processed are not interrupted by the stall, but the stall prevents new input streams from being read. Moreover, if vector size L is larger than the number of cycles for the stall, the stall penalty is effectively hidden because by the time the COMPUTE phase completes, the stall would have resolved (MAPLE takes L cycles to compute the result of a vector operand of length L). When multiple chains stall, the stall cycles can be overlapped reducing the overall stall penalty. The OR gate in Figure 5 broadcasts a global STALL

signal generated from the individual chains. This signal can be pipelined since it only has to stall the next input vector, and can therefore reach the first chain as its current COMPUTE cycle completes.

We now analyze the stall probability and penalty in the average case. We assume each memory block needs to extract the top k elements from an array of size n . Since a memory block corresponds to one of H processing chains, we can process H arrays in parallel. The goal of this analysis is therefore to compute the average number of processing cycles (i.e., cycles for computation as well as the wait cycles due to stalls) required to extract the top k elements from H arrays. We do this by first computing the stall probability for each of H chains. Since the MAPLE core stalls when any of its H chains stall, we can then compute the stall probability for the entire core.

We define and review a few symbols first.

- n : the size of each array
- k : the number of top elements to be extracted from each array
- $P_{chain-stall}(i, j)$: probability of chain j stalling when storing element i of its array
- C_{stall} : the number of cycles for a stalled store into the smart memory block
- $C_{nostall}$: the number of cycles for a nonstalled store into the smart memory block
- $P_{stall}(i)$: the probability of the entire core of H chains stalling when the first chain processes element i of the array. The first chain processes element i , but due to pipelining, the other $H-1$ chains process elements $i-1$ through $i-H+1$ of their respective arrays.
- TC : the total number of cycles to process H arrays

We first compute $P_{chain-stall}(i, j)$, the stall probability of chain j when processing element i of its array. A chain with M processors stores M words during its store phase. The chain stalls if any of the M stores stall. Since the n -element array is streaming, when element i is stored, the probability that element i will be in the top k is k/i . Therefore:

$$P_{chain-stall}(i, j) = \max \left[1, \sum_{l=1}^M \frac{k}{(i+l)} \right].$$

The entire MAPLE core stalls when any of its H chains stall. When the first chain processes element i of its array, the other $H-1$ chains process elements $i-1$ through $i-H+1$ of their respective arrays. Therefore, the probability of the MAPLE core stalling when the first chain processes element i of its array is given by:

$$P_{stall}(i) = \max \left[1, \sum_{j=1}^H P_{chain-stall}(i-j+1, j) \right].$$

Given the probability of the core stalling for a certain element i , we can compute the average number of processing cycles required to process the entire set of H chains in parallel, each array consisting of n elements:

$$TC = \sum_{i=1}^n [P_{stall}(i) \cdot C_{stall} + (1 - P_{stall}(i)) \cdot C_{nostall}].$$

This implies, intuitively, that for small i , stalls will be frequent, but will taper off as we process more elements of the array. Based on the above equations, Figure 6(a) shows the stall probability of MAPLE with 1 core, 32 chains and 8 PEs per chain, for an array with 4M elements when selecting k top elements. The horizontal axis represents the element index being stored, and the vertical axis the probability that one of the

Fig. 6. (a) Probability of MAPLE stalling and (b) MAPLE execution cycles for array sub-set selection.

smart memories will issue a stall. Figure 6(b) shows the simulated performance when selecting the top k elements of an array of size n . Each element of the array is assumed to take C cycles to produce (for example, by a vector dot-product with vectors of size C). We find that even though the stall probability is not small (for e.g., around 10% for the first 1M elements of a 4M element array when $k = 512$), MAPLE's performance is largely insensitive to k . This is because the stall cycles are effectively hidden by overlapping them with the C compute cycles of the next array element. Thus, we find that MAPLE can use the memory blocks to compute array selection efficiently.

5. MAPLE PROGRAMMING

In this section, we describe MAPLE instruction set and API, show how MAPLE can be programmed using a high-level API, and present ways to automatically map application kernels to the hardware.

5.1 MAPLE Instructions and API

MAPLE can be programmed using a high level API consisting of general purpose functions as well as algorithm-specific libraries. All parallelization and synchronization issues are hidden from the user within the libraries, which are implemented at the assembly level. In the next section, we present a basic compilation framework that can abstract some of the low-level details from the programmer, and perform automatic application mapping. We first describe the MAPLE instruction set. There are five classes of instructions for MAPLE.

Off-chip to Input Local Store. These instructions program MAPLE's memory controllers to move data from the off-chip memory banks to the on-chip input local store. They allow the user to initiate a burst fetch to efficiently stride and step through off-chip memory data at the rate of one memory clock per fetch width. This hides off-chip memory latency for our considered data access patterns. An example is shown in Figure 7(a), where the dark points from the inner cube can be burst-fetched, skipping the clear white points. The memory controller is programmed with lengths (nX , nY and nZ) and strides ($Xstride$, $Ystride$ and $Zstride$) along three dimensions. The fetched data are gathered in the input local store. This access pattern is used in CNNs to process every other input image pixel.

Fig. 7. (a) A sub-set of inner points of a cube that MAPLE's memory controllers can burst-fetch from off-chip. (b) Populating the PE local stores with off-chip data. (c) Striding through the input local store.

Off-chip to PE Local Store. These instructions program MAPLE's memory controllers to fetch arbitrary data from off-chip into a specific location of a particular PE's local store. Such data placement (Figure 7(b)) can schedule operations on different PEs thereby extracting parallelism.

On-chip Smart-Memory Blocks to Off-chip. These instructions program the reduce network in MAPLE to collect data from various smart memory blocks, operate on them and store them off-chip. The reduce operations can be either aggregations (used in CNNs) or comparisons (used in GLVQ or K-means). Output storage patterns are similar to the input patterns shown in Figure 7(a).

Input Local Store to Processing Elements. The fourth class of instructions specifies data access patterns from input local store into the processing fabric. The input local store access can also be programmed to skip and stride across data so that access latency is hidden and the processing fabric is never waiting for data. All local stores on MAPLE are software managed and operations such as data eviction are explicitly specified by instructions.

PE Instructions. These instructions program the PE to load data (from its local store), compute (which implicitly starts a stream from the input local store) and store results in the smart memory block. Stores from PEs also indicate the smart memory block operation to be performed, such as ranking or read-modify-write.

The preceding instructions are abstracted from the programmer and are used internally by the assembler to generate machine-level binary code for MAPLE. A set of high-level API functions are exposed to the programmer. These functions allow setting up computational patterns and data transfers between the host and MAPLE memory. Below we show a sample code to execute SSI on MAPLE; the bold keywords indicate the MAPLE API available to the programmer. MAPLESetup allocates memory to hold the database of documents and transfer the documents from the host to MAPLE memory. Note that, since the document database is static, this operation is performed only once. The MAPLERun function performs the following tasks: it a) transfers queries from the host to MAPLE memory, b) generates assembly specific to the particular reduce operation specified as argument to MAPLEGenerateASM,

c) transfers assembly instructions from the host to the instruction memory of MAPLE, d) initializes MAPLE's program pointer and waits for the completion of the execution by polling MAPLE's status, e) copies the top- k results from MAPLE to host memory. Both MAPLESetup and MAPLERun require the matrix dimensions and the type of reduction operation to be performed. For SSI, we specify *num_doc* and *num_cat* as the dimension of document matrix, and *num_queries* for the query matrix. We also set *RANKING* as the reduction operation thus instructing MAPLE's smart memory to perform in-memory ranking.

```
MAPLESetup (uint16_t *docs, int num_doc, int num_cat) {
    MAPLEMEM *m_docs;
    //Allocate MAPLE memory
    MAPLEMalloc (m_docs, num_doc*num_cat*sizeof(uint16_t));
    //Transfer documents from host to MAPLE memory
    MAPLEMemCpy (m_docs, docs, sizeof(docs), HOST2MAPLE);
}
```

```
MAPLERun (uint16_t *topk, uint16_t*queries, int num_doc, int num_cat, int num_queries,
reduction_type RANKING) {
    MAPLEMEM *m_queries, *m_topk;
    uint16_t *asm = (uint16_t*) malloc (MAX_ASM_LENGTH);
    MAPLEASM *m_asm;
    int asm_length;

    //Allocate MAPLE memory
    MAPLEMalloc (m_queries, num_queries*num_cat*sizeof(uint16_t));
    MAPLEMalloc (m_topk, num_queries*top_k*sizeof(uint16_t));

    //Transfer queries from host to MAPLE memory
    MAPLEMemCpy (m_queries, queries, sizeof(queries), HOST2MAPLE);

    // Generate Assembly
    asm_length = MAPLEGenerateASM (asm, num_doc, num_cat, num_queries, RANKING);
    MAPLEMalloc (m_asm, asm_length);

    //Transfer assembly from host to MAPLE instruction memory
    MAPLEMemCpy (m_asm, asm, asm_length, HOST2MAPLE);

    //Poll until completion
    while (MAPLEExecute()!=MAPLE_FINISH);

    // Transfer top-k values from MAPLE memory to host
    MAPLEMemCpy (topk, m_topk, sizeof(topk), MAPLE2HOST);

    MAPLEFree (m_queries);
    MAPLEFree (m_topk);
    MAPLEFree (m_asm);
}
```

Fig. 8. Expressing CNN convolutions as “sliding” matrix operations.

5.2 Mapping Applications onto MAPLE

In this section, we describe how the algorithms from Section 3 can be programmed on MAPLE.

SSI classification. We recall that SSI involves finding documents that are semantically the closest to a text query. The core computation is the multiplication (dot-product) of a compacted query vector with a set of compacted document vectors and the identification of the k documents with the highest dot-product. The document vectors can be expressed as a single matrix with N rows and C columns where N is the number of searchable documents, and C is the number of “concepts” along which similarity has to be identified. Each processor chain in MAPLE is programmed to evaluate one query, which is loaded into the PE local stores. The chain’s M processors compute the distance between the query and M documents in SIMD mode. All document vectors are stored in off-chip memory and packed so that every memory read streams in M different documents, which are sent to the M processors in the chain. At the end of the stream, M dot-products computed by the chain are written to its smart memory block which maintains a list of the top k dot-products and their corresponding documents IDs. Multiple queries are handled by assigning one query to each of H chains in a MAPLE core; each chain processes the same M document streams obviating the need for additional memory fetches.

CNN classification. CNN consists of 1D or 2D convolutions followed by arithmetic operations and subsampling. The core computation in one layer is the convolution of In input images with In kernels and their pixel-wise summation to produce one output image. This is repeated for On output images, each with the same In inputs but a different set of weights. MAPLE’s supported data access patterns (Figure 7) allow us to express convolutions as matrix operations.

Figure 8 shows how convolutions in a CNN layer can be expressed as matrix operations. The In images, each of size $Ir \times Ic$ form the “image cube” on the left, and the On output images each of size $Or \times Oc$ form the “output cube” on the right. The $In \times On$ kernels, each of size $Wr \times Wc$, are arranged as shown in the figure. While a detailed discussion of this is beyond the scope of this article, from the architectural point of view, the operations amount to a set of $(Ir - Or + 1) \times (Ic - Wc + 1) \times In$ repeated matrix-matrix multiplications $\{A\} \times \{B\} = \{C\}$ where $\{A\}$, $\{B\}$ and $\{C\}$ are sets of input,

Fig. 9. Matmul on MAPLE. The PE schedule shown on top has 2 PEs working in parallel, while the one below has 4 PEs in parallel, assuming 2 rows can be simultaneously streamed from off-chip.

kernel and output matrices obtained by “sliding” across the image, kernel and output cubes as indicated by the black arrows in the figure. MAPLE’s memory and input local store controllers can be programmed for these data access patterns. Since the kernels are small and do not change, we place kernel data in the PE private local stores, stream in the input matrices and stream out the output matrices. Each matrix-matrix operation $A \times B$ is parallelized along the columns of B : each column of B is stored in the PE local stores, and the matrix rows are streamed in. This is exemplified in Figure 9.

The top part of Figure 9 shows one method of parallelization where each column of matrix B is loaded into the local stores of PEs (0,0) and (1,0), i.e., the first PEs in chains 0 and 1. The image rows are streamed in one by one and broadcasted to the two chains, resulting in PE (0,0) and PE (1,0) computing column 0 and column 1 of the output, respectively. Another schedule is shown in the bottom of the figure where columns 0 and 1 of matrix B are duplicated in 2 PE local stores in each chain. Both rows are streamed in together (assuming sufficient memory bandwidth); therefore all 4 output elements are computed simultaneously making it twice as fast. Thus, if the number of columns of matrix B is smaller than the number of PEs, the PEs can be kept busy by duplicating columns and streaming multiple input rows.

K-means. In K-means, MAPLE computes Euclidean distances between K means and all points, and finds the closest mean to each point. Each PE computes the distance between a mean and a stream of points. The K means, each a vector, are loaded into the PE local stores, and the points are streamed through the PEs. The PEs write the Euclidean distances to the smart memory block, which stores them along with the point’s ID obtained from the PE locator which is also attached to the store. The reduce network then reads the points one by one and discovers the mean that is closest to each point. Given the closest mean to each of the n points, the host computes the next set of K means for the subsequent iteration.

SVM Training. The performance bottleneck in SVM training is the “kernel calculation” which involves the multiplication of training vectors with a large training matrix. This is a vector-matrix multiplication where the matrix under consideration is very large and must reside off-chip. We perform this computation on MAPLE by transferring the training or test vectors directly from the host to the PE local stores and streaming in the matrix from off-chip. SVM training is a memory-bound problem: if the memory bandwidth allows fetching R elements of the matrix in one cycle, no more than $2R$ PEs can execute in parallel since at most 2 vectors are given to multiply the matrix with in an iteration.

Fig. 10. Mapping matrices to MAPLE with different parallelism modes: (a) each column of B is duplicated across 2 PEs (doubles the parallelism), (b) 1 column of B per PE and (c) columns of B are split into two portions (X and Y) to fit the PE local stores (halves the parallelism).

GLVQ classification. GLVQ involves finding the closest reference vector to the given query vector. All reference vectors are loaded on the PE local stores, and the queries streamed from off-chip one by one. Each PE writes its result to the smart memory block, along with a “PE locator”. The reduce network computes the closest reference vector by determining the lowest value recorded by the PEs.

5.3 Mapping Matrix Multiplication onto MAPLE

We recall that in order to program MAPLE, the user expresses application kernels using MAPLE APIs 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 for SSD). At a low level, MAPLE is programmed through 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.

Two of MAPLE’s design goals are to handle matrices of various sizes, and minimize intermediate data by performing reduction operations using smart memory blocks. To this end, data placement and smart memory configuration are key aspects in the mapping operation. 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 10 illustrates how the parallelism mode pm affects the mapping. In the base case (pm equal to one) each PE processes a different column of B against the same row of A . 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 11 shows how two matrices that have to be multiplied are mapped onto the accelerator. The mechanism can be generalized to multiple matrix multiplications. The mapping is based on the following idea: the larger matrix (A) is streamed from the input local store, whereas the smaller matrix (B) is mapped onto the PE local stores. A is streamed row-wise; if it is too large to fit the input local store, row-wise blocking is

Fig. 11. Block diagram showing the mapping of matrices M_1 and M_2 onto the accelerator.

performed (and A is loaded into the input local store in multiple passes). The mapping of matrix B is more complex. Each local store will potentially accommodate one or more columns of B . If B is small, the same column is mapped onto different PEs, leading to parallel processing (*parallelism mode* > 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 (*parallelism mode* < 1). During operation, the rows of matrix A will be split as well and directed to the proper PE. If B cannot fit the PE local stores, column-wise blocking is performed. The output of the mapping process is a mapping of matrices to input and PE local stores as well as a set of parameters used to automatically generate assembly to program the accelerator.

If matrices A and B do not fit the on-chip resources (e.g., the number of rows of matrix A exceeds the smart memory size or the columns of matrix B are more than the PEs in a core), the processing is blocked and done sequentially. In this circumstance, however, the intermediate data are never stored to and reloaded from off-chip memory for further processing. In fact, the smart memory blocks accumulate partial results. As a consequence, the performance penalty involves only instruction and control overhead, which contributes to less than 2% of the overall execution time.

The pseudo-code below shows the assembly code generation (MAPLEGenerateASM function invoked by the MAPLERun API function). Bold keywords represent the generated assembly directives, and italicized keywords represent configuration variables provided by the user or parameters produced by the mapping tool. In the pseudo-code, A_blocks and B_blocks are (row-wise and column-wise) portions of the A and B matrices fitting the input- and the PE-local store, respectively. The code assumes that partial results computed on A_blocks fit the smart memory. SET_PARALLEL_MODE affects how B is mapped onto the PE local stores, potentially leading to column replication (*parallelism mode* > 1) or splitting (*parallelism mode* < 1). It also affects the way 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 depending on the value of the parallelism mode parameter. If B is a (B_R, B_C) matrix, b_col_sz is equal to B_R if pm is greater than one, and to $B_R * pm$ otherwise.

```

SET_PARALLEL_MODE parallelism_mode ; sets the parallelism mode
SET_SM_REDUCTION reduction_operation ; configures the reduction performed by SMs
SET_A_NUM_ROWS a_num_rows ; sets number of A rows present in each A_block
SET_B_COL_SZ b_col_sz ; 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 {
  WRITE_A A_block ; transfers an A_block from DRAM into input
; local store
  for each B_block {
    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_sz ; sets the PE_mem address to load the B data
        SET_SM_ADDR result_addr ; sets the address in SM for partial results
        MULT_ACC_DUMP b_col_sz ; performs b_col_sz MAC and sends result to SM
      }
      INC_INPUT_LS_ADDR sz(A_row_group) ; increments the input local store address
    }
  }
}
DUMP_SM ; dumps the content of smart memory
}

```

Taking SSI as an example, the user may provide the following inputs: document matrix of size $2M \text{ document} \times 64 \text{ categories}$, query matrix of size $64 \text{ categories} \times 64 \text{ queries}$, 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: *parallelism_mode* = 8, one *B_block*, 7813 *A_blocks*, *a_num_rows* equal to 256 (zero-padding is performed in the last *A_block*), *b_col_sz* equal to 64 and *b_num_cols* equal to 2. Figure 12 summarizes the outcome of the mapping process on the considered workloads.

6. ARCHITECTURAL EXPLORATION

In this section, we explore the design space and sensitivities of MAPLE accelerator with the help of an architectural simulator.

6.1 Effect of Processor Layout in MAPLE Processing Core

We developed a C++ simulator that takes an input assembly code for applications mapped onto MAPLE and an architectural configuration file that specifies the off-chip memory architecture (banks and bandwidth) and processor layout. It then simulates all processor and memory latencies, and provides a cycle-accurate estimate of the execution time.

Table II shows the different parameters of MAPLE. For this article, we seek to find, given an off-chip memory organization and processor budget, the processor layout

WORK LOAD	Off-chip Data	On-chip Data	Computational Bottleneck (after transformation)
SSI	D documents stored as $D \times C$ matrix	Q queries stored as $C \times Q$ matrix	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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 $N \times D$ matrix	K means stored as $D \times K$ matrix	<ul style="list-style-type: none"> 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 $N \times D$ matrix	1 vector of size D	<ul style="list-style-type: none"> 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 $N \times D$ matrix	T test vectors stored as $T \times D$ matrix	<ul style="list-style-type: none"> 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

Fig. 12. Mapping our workloads onto MAPLE.

Table II. Architectural Exploration Setup

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

(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 matching documents (i.e., $k = 32, 64, 128$). We simulated this for 1024 queries across various chain lengths (Figure 13(a)). Because of the dynamic array ranking within MAPLE's smart memories, the number of cycles was largely insensitive to k . The processor layout with the best performance was that with chain size equal to 6–8 processors. We recall that, in our SSI mapping, each PE chain compares the same set of documents with a different query. Since the documents are broadcasted from the off-chip memory bank, the performance is best when the chain can consume as many documents as the memory can provide, i.e., when M matches the memory bandwidth.

Figure 13(b) shows MAPLE's performance for 4 CNN networks. The best performer in this case ranges between $M = 4$ to $M = 16$. This is due to the fact that CNNs are compute-bound, and the amount of parallelism differs from network to network. For instance, networks with intermediate layers with many kernels can be converted into a large matrix-matrix multiplication, for which a larger chain is better.

We finally present K-means in Figure 13(c). The best chain size 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

Fig. 13. MAPLE architectural exploration for SSI, CNN and K-means.

Table III. 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 15(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 IV. 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

the number of means increases. Therefore a “tall, skinny” configuration with small chain sizes suits large values of K .

6.2 Effect of In-Memory Processing on Off-Chip Transfers

The primary advantage of in-memory processing is that it reduces off-chip accesses thereby improving performance. Using the simulator, we evaluate the extent of the reduction in off-chip accesses, as well as the consequent performance boost of MAPLE.

Table III shows the number of bytes loaded from 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, as shown in the second column of the table. For SSI, the in-memory processing performs array ranking, for CNN it performs aggregation, and for K-means it computes the minimum. As can be seen, the reduction in the number of off-chip accesses ranges from 1.64x 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 IV 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.

Table V. Experimental Setup—Performance

MAPLE FPGA Prototype	PEs (total)	512 organized as 2 cores, 32 chains/core, 8 PEs/chain
	On-chip Memory	464 Block RAMs 36 Kb/BRAM = 2MB
	Clock Speed	125MHz
	Off-chip data memory	4 banks (2GB) DDR-II SDRAM, bandwidth 8 words per bank (8GB/s total)
	Off-chip instruction memory	2 banks (512MB) DDR-II SRAM, bandwidth 4 words/bank (4GB/s)
	FPGA	Virtex 5 SX240T
	Host-Interface	64-bit, 66MHz, PCI-X
Host	2.5GHz Xeon quad-core with Intel MKL (BLAS) library	
GPU	240-core 1.3GHz Tesla C1060 running CUDA 3.0 with CUBLAS library	

7. 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 [Alpha-Data]. Our architectural design space exploration, along with FPGA constraints, determined the specific prototype architecture. The PE local store, Input local store, and smart memories are implemented using FPGA's on-chip block RAMs each of 36Kb. We use 464 BRAMs out of 514 available BRAMs and the total on-chip memory available to MAPLE prototype is 2MB. However, because of smart-memories' in-memory processing, intermediate data is reduced as soon as it is generated. Therefore reduction of intermediate data is executed well within MAPLE's available on-chip storage without accessing off-chip. We implemented each of the five workloads on the prototype and compared the prototype's performance with (i) optimized parallel software implementations, (ii) available and published GPU implementations of SSI, CNN and SVM from Nasse et al. [2009] and Catanzaro et al. [2008] and (iii) FPGA-based algorithm.-specific implementations of CNN and SVM from Sankaradas et al. [2009] and Cadambi et al. [2009]. Table V shows the experimental setup and Figure 14 shows a snapshot of the Virtex 5 FPGA card used.

7.1 Performance of SSI and CNN

We implemented SSI in software using multithreaded BLAS for the matrix multiplication, followed by an optimized multithreaded implementation of array ranking. We compared this to a MAPLE implementation of SSI. Figure 15(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 Bai et al. [2009], and obtained a speed of 4.63 to 4.88 ms/query for $k = 32$ and $k = 128$ respectively.

We measured the performance of five CNN workloads that perform face and digit recognition, and surveillance and automotive safety on 640×480 images. Figure 15(b) shows the speed in milliseconds per frame measured with the parallel software and the MAPLE prototype.

Now we compare MAPLE's performance to available GPU implementations of SSI and CNN. Table VI 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 Nasse et al. [2009]. Compared to the GPU, the MAPLE prototype is 2x faster for SSI and about 50% faster for CNN.

Fig. 14. Virtex-5 SX240T FPGA Card.

Fig. 15. MAPLE prototype performance vs. optimized software for (A) SSI and (B) CNN.

7.2 K-Means Performance

Figure 16 shows data for K-means. MAPLE finds the closest mean for every point. Then it transfers that information to the host, which averages all the closest points to each mean. Since successive iterations cannot be overlapped, we must consider the data transfer time and the host component to compute the effective speedup. Figure 16(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 longer and responsible for speedup reduction.

Table VI. SSI and CNN Performance Comparisons

Workload Details	Software	GPU	MAPLE Prototype
SSI: 2M docs, 64 queries, 64 top k	31ms/query	9.3ms/query	4.6ms/query
CNN: Face Recognition [Nasse et al. 2009]	5.9 fps	9.5 fps ([Nasse et al. 2009])	13.3 fps

Fig. 16. K-means performance on MAPLE prototype.

Table VII. GLVQ Training and Testing Performance (in Seconds) for the Eye-Detection Dataset

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

Figure 16(b) shows the speedup 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.

7.3 GLVQ Performance

We implemented GLVQ [Sato and Yamada 1995] training and testing on the prototype, and used it for eye detection in images. This relatively simple model 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 were processed sequentially, as each incrementally modifies the model. This also means considerable host-accelerator communication. Table VII 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.

7.4 SVM Performance

In SVM, the “kernel computation,” which involves multiplication of test or training vectors with the large training or support vector matrix, is mapped onto MAPLE.

Table VIII. SVM Training Performance (Milliseconds/Iteration)

Data Set	Training Set Size	Dim	MAPLE Prototype	[Catanzaro et al. 2008]
Adult	32,561	123	3.2	0.76
Web	49,749	300	9.27	3.56
MNIST	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

Table IX. Comparing the MAPLE Prototype with Published Algorithm Specific Accelerators

	Algorithm Specific	MAPLE Prototype
SVM: MNIST	9.3 billion MACs per second [Cadambi et al. 2009]	4 billion MACs per second
CNN: Face Recognition	10 frames/sec [Sankaradas et al. 2009]	10.5 frames/sec

We compare our work with a recent high performing GPU implementation of SVM [Catanzaro et al. 2008] as well as an FPGA implementation from Cadambi et al. [2009]. An iteration involves multiplying the training set matrix with 2 training vectors; typically tens of thousands of iterations are required to complete training. Table VIII shows SVM training performance in milliseconds per iteration for 6 data sets [Catanzaro et al. 2008]. 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 Catanzaro et al. [2008] 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.

7.5 MAPLE versus Algorithm Specific Implementations

We compare the MAPLE prototype performance with algorithm-specific FPGA-based implementations of SVM and CNN from Cadambi et al. [2009] and Sankaradas et al. [2009]. For SVM, Cadambi et al. [2009] 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 Sankaradas et al. [2009].

7.6 PCI Bandwidth Utilization

Table X shows the PCI bandwidth utilization of SSI, CNN and K-Means applications for a given workload. In this simulation, the measured bandwidth is determined by the total amount of input and output data sent per unit time, which includes the data transfer and the execution time of MAPLE for a specific application running a clock frequency of 125 MHz. PCI bandwidth utilization compares the measured bandwidth against the theoretical maximum bandwidth of PCI that is, 4 Gbps. The results demonstrate that all considered workloads are compute-bound with the data transfer time reflecting a small fraction of the overall processing time. SSI, in particular, has the lowest PCI utilization because it transfers a small data set, that is, queries as input and top- k indexes as output. For CNN and K-Means, where the data size is reasonably large, data transfers still take a small portion of the PCI bandwidth.

Table X. PCI Bandwidth Utilization

	Input Data Size (bytes)	Output Data Size (bytes)	Data Transfer time (ms)	Execution time (ms)	PCI utilization (%)	# of MAPLE cards to fully utilize the PCI
SSI N = 2M Q = 64 k = 32	65536	4096	0.36	295.02	0.09	1140
CNN Face Recognition on 640×480 image	3072000	270592	17.72	262.36	4.18	24
K-Means N = 100K D = 3 M = 32	2560	2097152	15.80	297.23	2.38	42

8. USING MAPLE TO BUILD AN ENERGY-EFFICIENT SYSTEM

In this section, we propose an energy efficient system for embedded learning and classification workloads by coupling MAPLE to a low power Atom processor, and compare its energy benefits over a Xeon based server system. Our comparison against such state-of-the-art high end server is motivated by the performance requirements of these learning applications. These high performance systems meet performance requirements, but consume considerable power. Other low end processors operate at low power but compromise performance. Our low power Atom + MAPLE system meets the high-performance requirement of these applications while achieving high energy efficiency.

8.1 Low Power System Architecture

The low-power system for embedded learning and classification is architected using a dual-core Intel® Atom™ 330 [Intel Atom]. Specifically, we use the ASUS AT3N7A-I motherboard [AT3N7A-I] for our experiments. The motherboard has a PCI slot onto which we add the MAPLE prototype operating at 100MHz. The Atom functions as a host and control processor providing a platform for data transfer, and offloads computationally intensive portions of the workloads to MAPLE. We show that the system achieves energy efficiency by delivering better performance than a server comprising a Xeon processor and a Tesla GPU, but at significantly reduced power. Table XI summarizes the configuration of our low power system.

8.2 Performance

In order to understand the performance loss (slowdown) caused by the Atom processor, we compare our proposed embedded system to a server-class system. For the workloads with problem sizes shown in Table XII, Figure 17 shows the wall-clock application execution time on a GPU-accelerated server comprising a 2.5GHz quad core XEON coupled with a 1.3GHz 240-core NVIDIA Tesla C1070 GPU.

In Figure 17, the first bar shows the execution time using just the Xeon, while the second bar shows the reduced time when we accelerate the core computational kernels using the Tesla GPU. The third bar shows the time if the entire application runs just

Table XI. Low Power System Architecture

MAPLE FPGA Prototype	PEs (total)	512 organized as 2 cores, 32 chains per core, 8 PEs per chain
	On-chip Memory	464 Block RAMs $\times$ 36 Kb/BRAM = 2MB
	Clock Speed	100MHz
	Off-chip data memory	4 banks (2GB) DDR-II SDRAM, bandwidth 8 words per bank (8GB/s total)
	Off-chip instruction memory	2 banks (512MB) DDR-II SRAM, bandwidth 4 words/bank (4GB/s)
	FPGA	Virtex 5 SX240T
	Host-Interface	32-bit, 66MHz, PCI
Host	1.6GHz Atom dual-core with Intel MKL (BLAS) library	

Table XII. Benchmarks

Workload	Description	Problem Size
Supervised Semantic Indexing (SSI)	Semantic search algorithm to extract k best documents from a total of N documents based on Q concurrent text queries	$N = 256K$ $Q = 64$ $k = 32$
Convolutional Neural Network (CNN)	A pattern recognition algorithm for face and object detection, and semantic text search.	Two 640×480 images
K-Means	Image segmentation algorithm to cluster N points into K clusters.	$N = 100K$ $K = 32$

Fig. 17. Performance of Atom-based embedded vs. Xeon-based server system.

on the Atom processor and the fourth bar shows the improvement when MAPLE is coupled to the Atom.

The respective speedups of each system compared to the common baseline of just the Xeon processor are listed in Table XIII. Among all the configurations, using just the Atom processor results in slowest performance. However, the use of the MAPLE accelerator boosts performance and makes the embedded system competitive. We note that, for SSI, the use of MAPLE actually makes the Atom + MAPLE system faster than the Xeon + Tesla. For the other workloads, the Atom based system incurs a

Table XIII. Speedup over Xeon

	Xeon + Tesla	Atom	Atom + MAPLE
SSI	3.47	0.17	5.15
CNN	2.97	0.16	1.54
K-Means	1.86	0.25	0.74

Table XIV. Power Numbers (in Watts) of Atom and Xeon Based System

Workload	SERVER		EMBEDDED	
	Xeon	Xeon + Tesla	Atom	Atom + MAPLE
Idle	157	215	42	55
SSI	184	283	49	69
CNN	194	284	48	70
K-means	191	285	47	71

Fig. 18. Energy consumption of (A) Xeon with Tesla, and (B) Atom with MAPLE.

slight slowdown despite the accelerator. In all cases, we use the Intel MKL optimized libraries for the Atom and Xeon and nVidia's CUBLAS for the GPU. All numbers are results based on our actual implementation, except the CNN GPU data, which are derived from Nasse et al. [2009].

8.2.1 Power Profile. Table XIV shows the power numbers measured using a power meter [Watts-up Pro] for both the proposed Atom and the standard Xeon system when running the specified workloads with the problem size as specified in Table XII.

8.2.2 Energy Comparison. In this section, we evaluate the energy consumption of the Atom-based embedded system as well as the Xeon+Tesla system.

Energy consumption within a system. Figure 18 shows the energy reduction when the applications are executed on the accelerator compared to the execution on their respective host processors. Figure 18(a) shows the energy behavior of the Xeon system with Tesla accelerator. For SSI and CNN, the implementation on Tesla reduces the system energy by 56% and 51% respectively, whereas for K-means, energy decrease is only 20%. Figure 18(b) shows the energy consumption within the Atom system when the workloads are running on the Atom or on MAPLE. Compared to the sole Atom execution, the system with MAPLE is 95% and 85% more energy-efficient respectively

Fig. 19. Energy comparison of server vs. embedded system.

for SSI and CNN. For K-Means, the energy decrease is 49% because of limited performance speedup of K-Means on MAPLE.

System-level energy comparison. Figure 19 provides an energy level comparison of the proposed embedded system versus the server-class system by taking the best energy case from Figure 18(a) and Figure 18(b) respectively. Compared to the Xeon system, the energy of the Atom based system decreases by 84% and 53% for SSI and CNN respectively, while the energy decrease for K-Means is 38%.

9. CONCLUSION

We described a programmable parallel accelerator that can handle several learning and classification algorithms. By profiling and analyzing five learning and classification workloads, we identify their computational bottlenecks and find that they can be transformed into a matrix or vector operation producing large intermediate data which are then reduced by a secondary operation. We architect the accelerator to leverage this characteristic by provisioning it with a two-dimensional grid of simple PEs to provide fine-grained parallelism. The PEs are reconfigured dynamically based on matrix sizes; in-memory processing allows intermediate data to be simultaneously generated and consumed, without the need for going off-chip. Furthermore, the banked off-chip memories of MAPLE with each memory bank serving a separate group of PEs create processor-memory channels that can process the coarse-grained, independent computation streams. These features allow MAPLE to scale its performance with respect to data size.

In addition to the architecture, we present a compilation scheme to automatically map application kernels to the accelerator hardware. We also present an FPGA-based prototype of the accelerator and measure speedups over optimized, parallel software implementations as well as GPU implementations of some of our considered learning and classification workloads.

Finally, we use the MAPLE accelerator to build an energy efficient system for embedded learning and classification applications. The system comprises MAPLE coupled to an Atom processor that functions as the host. Using three representative workloads, we demonstrate the energy-efficiency of our system over standard server-class

architectures. We envision such programmable embedded solutions based on general-purpose CPUs and low power accelerators to be part of current and future systems.

Future work along this direction includes investigating a dynamic runtime strategy to further reduce the system energy consumption by optimally splitting the workload across the accelerators, and using such a system in a heterogeneous cluster environment targeting learning applications.

REFERENCES

- ALPHA-DATA. <http://www.alpha-data.com/products.php?product=adm-xrc-5t2>.
- AT3N7A-I. Specification. http://www.asus.com/product.aspx?P_ID=xrR7wto9Z5BL42aU&temple=2.
- BAI, B., WESTON, J., GRANGIER, D., COLLOBERT, R., SADAMASA, K., QI, Y., CHAPELLE, O., AND WEINBERGE, K. 2009. Learning to rank with (a lot of) word features. *Info. Retrieval J. 13* (Special Issue on Learning to Rank), 291–314.
- BURGER, D., ET AL. 2004. Scaling to the end of silicon with EDGE architectures. *IEEE Computer* 37, 7, 44–55.
- C2050/C2070 POWER. http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html.
- CADAMBI, S., ET AL. 2009. A massively parallel FPGA-based coprocessor for support vector machines. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 115–122.
- CATANZARO, B., SUNDARAM, N., AND KEUTZER, K. 2008. Fast support vector training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning (ICML)*. 104–111.
- CHELLAPILLA, K., PURI, S., AND SIMARD, P. 2006. High performance convolutional neural networks for document processing. In *Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition*.
- COLLOBERT, R. AND WESTON, J. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*. 160–167.
- COSATTO, E., MILLER, M., GRAF, H. P., AND MEYER, J. 2008. Grading nuclear pleomorphism on histological micrographs. In *Proceedings of the International Conference on Pattern Recognition*. 1–4.
- DATTA, R., JOSHI, D., LI J., AND WANG, J. Z. 2008. Image retrieval: Ideas, influences, and trends of the new age. *ACM Comput. Surv.* 1–60.
- 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. In *Proceedings of the 13th ACM SIGPLAN PPoPP*. 63–72.
- GRAF, H. P., CADAMBI, S., DURDANOVIC, I., JAKKULA, V., SANKARADASS, M., COSATTO, E., AND CHAKRADHAR, S. T. 2008. A massively parallel digital learning processor. In *Proceedings of the Neural Information Processing Systems (NIPS)*. 529–536.
- HALL, J. D., AND HART, J. C. 2004. GPU acceleration of iterative clustering. In *Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors* (SIGGRAPH poster).
- INTEL ATOM. <http://ark.intel.com/Product.aspx?id=35641>.
- LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11, 2278–2324.
- LIM, K., RANGANATHAN, P., CHANG, J., PATEL, C., MUDGE, T., AND REINHARDT, S. 2008. Understanding and designing new server architectures for emerging warehouse-computing environments. In *Proceedings of the International Symposium on Computer Architecture*. 315–326.
- LLOYD, S. P. Least squares quantization in PCM. In *IEEE Trans. Info. Theory* 28, 2, 129–137.
- MACQUEEN, J. B. 1967. Some methods for classification and analysis of multivariate observation. In *Proceedings of the Berkeley Symposium on Mathematics Statistics and Probability*. 281–297.
- MEI, T., HUA, X., YANG, L., AND LI, S. 2007. VideoSense: Towards effective online video advertising. In *Proceedings of the 15th International Conference on Multimedia (MULTIMEDIA'07)*. 1075–1084.
- NASSE, F., THURAU, C., AND FINK, G. A. 2009. Face detection using GPU-based convolutional neural networks. In *Proceedings of the 13th International Conference on Computer Analysis of Images and Patterns (CAIP)*. 83–90.

- OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRUEGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Comput. Graph. Forum* 26, 1, 80–113.
- PLATT, J. 1999. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods – Support Vector Learning*. MIT Press, 997–1001.
- RAINA, R., MADHAVAN, A., AND NG, A. Y. 2009. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*. 873–880.
- REDDI, V. J., LEE, B. C., CHILIMBI, T., AND VAID, K. 2010. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *Proceedings of the 37th International Symposium on Computer Architecture*. 314–325.
- ROUSSEAU, S., HUBAUX, D., GUISET, P., AND LEGAT, J. 2007. A high performance fpga-based accelerator for blas library implementation. In *Proceedings of the 3rd Annual Reconfigurable Systems Summer Institute (RSSI'07)*.
- SANKARADAS, M., ET AL. 2009. A massively parallel coprocessor for convolution neural networks. In *Proceedings of the 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. 53–60.
- SATO, A. AND YAMADA, K. 1995. Generalized learning vector quantization. In *Neural Information Processing Systems*. 423–429.
- SEAMICRO. <http://gigaom.com/2010/01/06/seamicros-secret-server-changes-computing-economics/>.
- SEILER, L., ET AL. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM SIGGRAPH'08* 27, 3, 1–16.
- TAYLOR, M. D., ET AL. 2002. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro* 22, 2, 25–35.
- VASUDEVAN, V., FRANKLIN, J., ANDERSEN, D., PHANISHAYEE, A., TAN, L., KAMINSKY, M., AND MORARU, I. 2009. FAWNdamantly power-efficient clusters. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS XII)*. 1–5.
- WATTS-UP PRO. <https://www.wattsupmeters.com/secure/products.php?pn=0&wai=228&more=2>.
- X7460 POWER. <http://www.intel.com/cd/products/services/emea/eng/processors/xeon7000/343718.htm>.
- ZHUO, L. AND PRASANNA, V. K. 2005. High performance linear algebra operations on reconfigurable systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 1–12.

Received September 2010; revised April 2011; accepted August 2011