A Parallel Accelerator for Semantic Search

Abhinandan Majumdar, Srihari Cadambi, Srimat T. Chakradhar and Hans Peter Graf
NEC Laboratories America, Inc., Princeton, NJ, USA.
Email: {abhi, cadambi, chak, hpg}@nec-labs.com

***Abstract*- Semantic text analysis is a technique used in advertisement placement, cognitive databases and search engines. With increasing amounts of data and stringent response-time requirements, improving the underlying implementation of semantic analysis becomes critical. To this end, we look at Supervised Semantic Indexing (SSI), a recently proposed algorithm for semantic analysis. SSI ranks a large number of documents based on their semantic similarity to a text query. For each query, it computes millions of dot products on unstructured data, generates a large intermediate result, and then performs ranking. SSI underperforms on both state-of-the-art multi-cores as well as GPUs. Its performance scalability on multi-cores is hampered by their limited support for fine-grained data parallelism. GPUs, though beat multi-cores by running thousands of threads, cannot handle large intermediate data because of their small on-chip memory. Motivated by this, we present an FPGA-based hardware accelerator for semantic analysis. As a key feature, the accelerator combines hundreds of simple processing elements together with in-memory processing to simultaneously generate and process (consume) the large intermediate data. It also supports "dynamic parallelism" - a feature that configures the PEs differently for full utilization of the available processin logic after the FPGA is programmed. Our FPGA prototype is 10-13x faster than a 2.5 GHz quad-core Xeon, and 1.5-5x faster than a 240 core 1.3 GHz Tesla GPU, despite operating at a modest frequency of 125 MHz.**

I. INTRODUCTION

Semantic querying of text and images has wide-ranging, mass market uses such as advertisement placement [1] and content-based image retrieval [2]. A popular algorithm used in semantic text search is Supervised Semantic Indexing [3] which retrieves a small set of documents from a large database based on their semantic similarity to a text query. SSI's performance targets are dictated by application-level Quality-of-Service (QoS) requirements and Service Level Agreements (SLAs). For example; it must be capable of semantically searching millions of documents with a latency of 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, however, processes at a rate of 9.5ms/query.

Motivated by this gap between performance and state-of-the-art computing platforms, we investigate a custom accelerator for semantic search. SSI involves matrix or vector operations on large unstructured data producing a large intermediate result (potentially leading to off-chip storage with many off-chip memory accesses). The intermediate data then undergoes a memory-intensive ranking operation. Despite using heavily-optimized BLAS libraries for high-performance linear algebra [4] and SSE2/4 functional units [5], the multi-cores as well as the GPUs are both hampered by the presence of this large intermediate data.

To design the accelerator, we profile the computational kernels of SSI, identify performance bottlenecks, and use the FPGA's features to specifically address the drawbacks of the multi-core and the GPU when it comes to this specific workload. In particular, the multi-core lacks support for large amounts of fine-grained parallelism as well as the inability to handle large intermediate data (several GB) on-chip without incurring off-chip accesses. While the GPU can support thousands of threads efficiently, it incurs even more off-chip accesses due to its smaller caches and lack of in-memory processing. For example, SSI that processes 64 queries to retrieve the top 64 matching entries from a database of 2M documents generates 512MB of intermediate data. A 2.5 GHz dual-core Xeon with 12MB L2 cache, as well as a 1.3 GHz Tesla GPU with 16KB blocks of on-chip shared memory cannot cache this intermediate data. Our accelerator uses multiple Processing Elements (PEs) implemented using FPGA DSPs to handle the highly parallelizable compute-intensive portion of SSI. For the intermediate data, we architect smart memories using FPGA's on-chip block RAMs (BRAMs) along with surrounding logic to perform on-the-fly ranking without storing the data off-chip.

To implement a prototype of the SSI accelerator, we use an off-the-shelf FPGA board from Alpha-Data [12] that contains one Xilinx Virtex 5 SX240T FPGA with 1056 DSPs and 514 BRAMs [13]. The board has four DDR2 2GB memory banks and two 256MB SRAM banks. On this FPGA, we architect an SSI processor with these key features. First, 512 DSP elements are laid out as two-dimensional grid providing fine-grained parallelism for the dot-products. Second, the on-chip memories execute in-memory ranking which allows the large intermediate data to be processed on-the-fly thereby requiring no off-chip memory accesses. Third, our accelerator implements a dynamic parallelization mechanism based on the number of SSI queries in order to maximally utilize the available computational logic.

To this end, we make following contributions in this paper:

- We present the architecture of the FPGA-based SSI accelerator.
- We explore its optimal architectural parameters and implement a working prototype on an off-the-shelf FPGA PCI-card.
- We present a high-level API to program the SSI processor, as well as a tool that can automatically generate assembly used to program the accelerator.

- We compare its performance against parallel, optimized software implementations on multi-cores and GPUs.

At a higher-level, the concepts this work brings out are the use of in-memory processing in combination with massively parallel processing, and dynamic parallelism. While these are presented and evaluated specifically in the context of SSI, they may also find utility in other application domains.

The rest of the document is organized as follows. Section II profiles and analyzes the computational bottlenecks of SSI. In Section III, we describe the architecture of the SSI accelerator, its core components and the APIs responsible for mapping SSI onto to the hardware. We describe its architectural implementation on an FPGA, explore its design space and present the FPGA resource utilization in Section IV. In Section V, we compare the performance of our FPGA accelerator with the optimized implementation on Xeon and GPU. We discuss related work in Section VI and conclude in Section VII.

II. Background and Application Profile

The SSI algorithm ranks documents based on their semantic similarity to text-based queries. Each document and query is represented by a vector and each vector element is the product of Term Frequency (TF) and Inverse Document Frequency (IDF) [3][11]. By multiplying a query or document vector with a weight matrix generated by training, we produce a smaller vector that contains relevant information for document-query matching. The SSI matching process multiplies the query vector with all document vectors, generates a large intermediate matrix, and identifies those documents whose vectors produced the top few results.

TABLE I
PROFILING OF SSI ALGORITHM

Problem Description	Typical Para-meters	% time of the total execution (profile)	Characteristic	Compute ops per memory operation
For each of Q queries, find *k* out of D documents that are semantic best matches	D: few millions Q: 32-128 *k*: 64-128	> 99%	Dot product: compute bound Array rank: memory bound	Dot prods: 25-50 Array rank: 0.001

Table I shows the core computations of SSI and the fraction of the total execution time they consume as measured on a 2.5 GHz quad-core Xeon. The two core computations, Dot Product and Array Rank, are responsible for over 99% of the execution time. It is clear that significant speedups are achievable by accelerating these core computations. Table I also shows which portions of SSI are compute or memory bound, and the number of computations per memory operation.

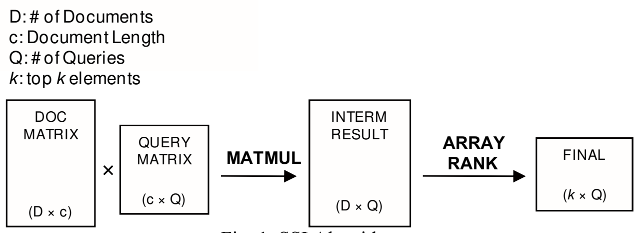


Fig. 1. SSI Algorithm

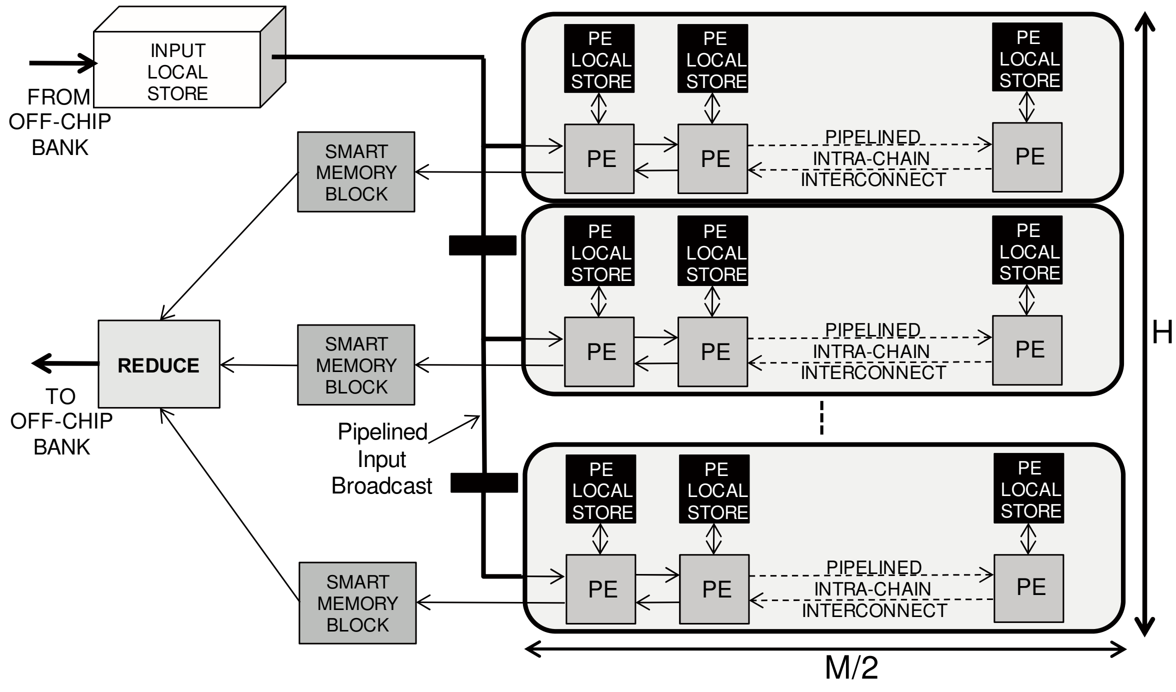


Fig. 2. Core architecture of the SSI accelerator

From Fig. 1, we see that SSI generates a large intermediate result as an outcome of matrix multiplication, which then undergoes a memory-intensive ranking operation to generate a smaller matrix. General purpose processors cannot cache the large intermediate result and often require fetching this data from off-chip. GPUs also do not demonstrate good performance because of their limited on-chip shared memory. This motivates us to design a specialized accelerator for SSI by addressing the architectural limitations of multi-cores and GPUs.

III. Core Architecture of The SSI Accelerator

In this section, we present the core architecture of the SSI accelerator, its internal components and describe the software-level API that maps SSI to the hardware.

A. *Core Architecture*

From the analysis of SSI algorithm, we find that the architecture must support matrix operations and should perform array ranking on large intermediate data without accessing off-chip memory. 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 matrix multiplication. 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 with its stored column). PEs stream results into the “smart memories” that perform in-memory ranking of the intermediate data.

Fig. 2 shows the architectural details of one processing core of the SSI accelerator. A core has ***P*** processors (implemented using FPGA DSPs) organized as ***H*** processing chains of ***M/2*** vector PEs (each PE uses two FPGA DSPs and one dual-ported BRAM) such that ***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 the *input local store*. Each PE also has a private *local store* that can be written with data from off-chip. Each PE takes two vector operands as inputs, one from its *local store*, and the other streaming from the input buffer. A PE chain sends its outputs to the smart memory block, one of which is available to each processing chain, which performs in-memory array ranking.

B. *SSI Mapping and Dynamic Parallelism*

The user expresses the SSI parameters (number of documents, length of a document vector, number of queries, top *k*) using a high-level API. The API functions allow users to allocate space in the off-chip memory, transfer data and assembly code between the host and accelerator, initialize the instruction pointer and send control signals to start SSI or poll completion. At a low level, these APIs generate assembly instructions that the accelerator reads from its instruction memory in order to execute SSI. The assembly instructs the accelerator to transfer data between off-chip and accelerator memories. The documents and queries are initially stored by the user into the off-chip. As SSI execution proceeds, the documents are buffered in the *input local store*, while the queries are stored in the *PE local store*.

The way the queries are organized and the documents are processed depends on the dynamic parallelism. The assembler generates a parameter called *parallelism mode* based on the number of SSI queries. The parallelization scheme here is dynamic because the PE usage is reconfigured after the FPGA is programmed, to allow maximal processor utilization for a specific SSI problem. This is done in following ways: First, it allows processing multiple documents in parallel by duplicating the queries among the *PE local stores* within a chain. Second, the bandwidth of the *input local store* is dynamically configured (after the FPGA is programmed) to broadcast that many number of documents to all processing chains through a pipelined network as shown in Fig. 2. Third, the smart memories are programmed to rank the streaming results differently such that final top *k* documents are the outcome of one or multiple queries depending on the *parallelism mode*.

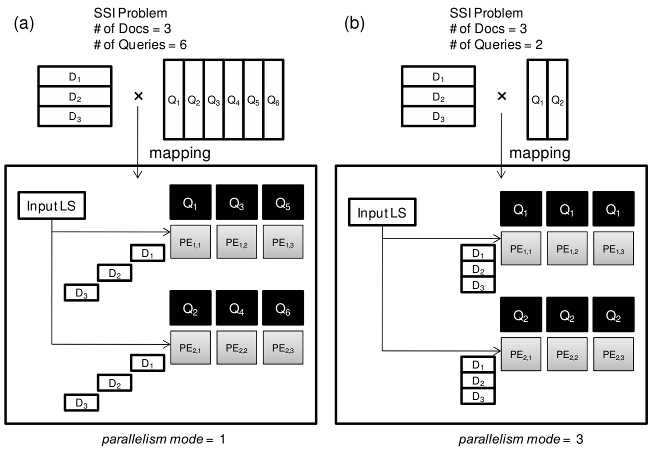


Fig. 3. SSI mapping and illustration of dynamic parallelism

Fig. 3 demonstrates the mechanism of dynamic parallelism for a SSI core with 2 chains each with 3 PEs. For this example, we assume that a PE is architected using one DSP, and the *PE local store* uses single-ported BRAM. When the queries exceed the number of chains as per Fig. 3(a), individual queries are stored in each *PE’s local store* and the documents are sequentially streamed in row-wise manner from *input local store*. However, when the queries are lesser than number of chains, the *parallelism mode* becomes equal to 3 and spreads the matrix multiplication across multiple DSPs, as illustrated in Fig. 3(b). In this case, queries are replicated within a chain and the *input local store* broadcasts 3 documents in parallel. This results in 3x performance increase for matrix multiplication.

The bus-width between *input local store* and the PE chain is hard-wired to ***M*** words and is reconfigured as per the *parallelism mode*. When the number of documents sent from the *input local store* is less than ***M***, unused buses transmit zeros and are ignored by the PE chain during the MAC operation. In this way, dynamic parallelism reconfigures a programmed FPGA to a particular SSI problem-size and guarantees full utilization of the PEs during the execution.

C. *Internal Core Components*

Now, we describe the individual core components of our SSI accelerator and how they are implemented on an FPGA.

Input Local Store: The *Input Local Store* is architected using on-chip FPGA memories. Each local store consists of dual-ported on-chip BRAMs which allow simultaneous buffering of streaming data coming from off-chip and broadcast the stored data to the processing elements through a pipelined network. All writes and evictions are executed serially while reads can access random address locations. The *parallelism mode* also configures its output bandwidth to broadcast 1 to ***M*** words per cycle. Additional NOP cycles are added initially in the assembly to cache the data in the *local store* before it is broadcasted to the processing chain.

Processing Elements and their Interconnection: The PEs, shown in Fig. 4, perform standard 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 local store benefits from FPGA's on-chip dual-ported BRAM which feeds to the vector processor implemented using two DSPs. The DSP uses 16-bit fixed-point signed multiplication and generates a 32-bit result with a latency of 1 cycle. This result streams out through the chain to the smart memory for array ranking.

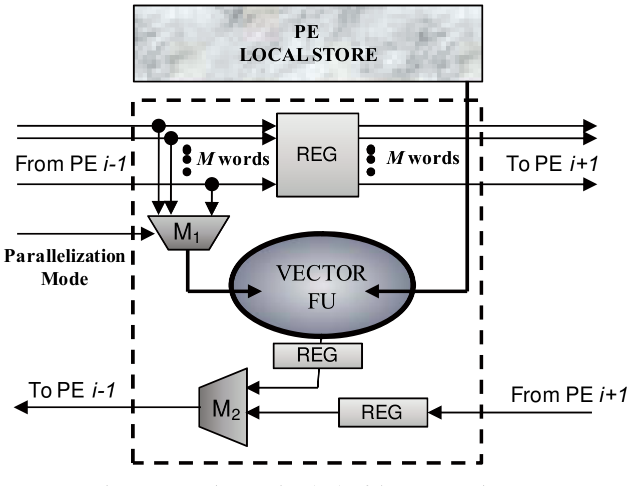


Fig. 4. Processing Engine (PE) of the SSI Accelerator

Therefore a PE with a chain length of ***M/2*** uses ***M*** DSPs, and performs ***M*** vector operations at a time. The intra-chain interconnect bus is also ***M*** words wide and matches the number of DSPs in the chain. A PE can select any word from its intra-chain interconnect bus based on the *parallelism mode*: the ***M*** DSPs in a chain can operate on ***M*** different streaming words as well as on the same word. The PE stores the output data to its smart memory block and can continue processing the next vector in parallel. However, this overlapping of streaming output data with next vector operation benefits only when the incoming vector length is sufficiently long. When the vector length is smaller than the chain length, additional NOP cycles are added in order to avoid the shared bus conflict on the output bus PE_{i-1}. As shown in Fig. 4, PE_{i-1} is shared by two different sources – VECTOR FU and PE_{i+1} – through mux M_2. This results in shared bus conflict when both the units attempt to send their output through PE_{i-1}. To avoid this conflict, additional NOP cycles are necessary for correct execution so that the output from VECTOR FU successfully transmits through PE_{i-1} before the output of PE_{i+1} appears.

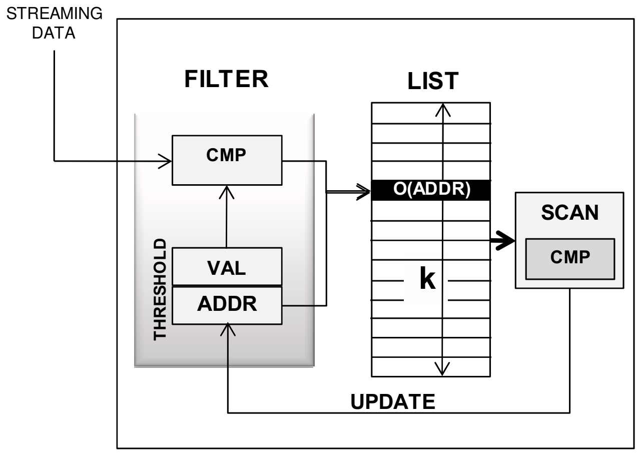


Fig. 5. Smart Memory Block of the SSI Accelerator

Smart Memory Blocks: Each chain consists of a memory block executing a variable latency store for selection and ranking operations in large arrays. The memory block is written to only by the processing chain, and is read by the reduce network. Fig. 5 shows relevant components of the smart memory architecture for selecting the top *k* 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*, 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. If a stall doesn't happen, a store takes ***M*** cycles. In the worst case, the smart memory stalls for ***M*k*** cycles. However, this latency is effectively hidden since we overlap the processing of the next vector with the store. The smart memory is also programmed to maintain separate lists for individual query based on *parallelism mode*. During full parallelism, the smart memory maintains one list and assumes all the streaming output to be of one query (duplicated within the chain).

Thus the smart memory ranks the large intermediate data streamed out from the processing chain and avoids accessing off-chip memory. This combination of FPGA's on-chip memory along with surrounding processing logic results in SSI acceleration. The smart memory block is implemented using one dual-ported BRAM. We save extra cycles by building a pipelined read-modify-write functionality on top of this BRAM. The smart memory has a read latency of one cycle and read-modify-write latency of two cycles. In addition of storing the streamed values, a portion of BRAM also stores the respective document indexes that are generated internally. These indexes are used by the host to identify the top *k* documents.

Reduce: The reduce logic scans through each smart memory and retrieves the indexes of top *k* documents for each query. The reduce logic packages the unsorted indexes as per the off-chip memory bandwidth before writing it back. The host, after receiving the results, uses these indexes to identify the documents and sorts them up for final display. The overhead involved in sorting *k* documents (with *k*=32 or 64) by the host is minimal, and thus it saves us from designing a complex hardware to systolically sort the streaming results on the FPGA.

IV. FPGA Implementation

This section describes the FPGA prototype along with the host interface, architectural exploration within the core, and the FPGA processor and memory utilization.

A. FPGA Prototype

To implement the SSI accelerator, we use an off-the-shelf ADM-XRC-5T2 board from Alpha Data [12]. We specifically chose the Xilinx Virtex 5 SX240T FPGA because of its high memory-to-logic ratio and enhanced DSP blocks which provide parallel processing for DSP and memory-intensive applications [13]. This board comes with four off-chip DDR2 banks with a bandwidth of 8 words-per-cycle per bank and two off-chip SRAMs with a bandwidth of 4 words-per-cycle per bank. The FPGA contains 514 BRAMs each of 36Kbits and 1056 DSPs. Since a PE has a resource requirement of one dual-ported BRAM for every two DSPs, the limited number of BRAMs restricts our design to use 512 DSPs. Thus the prototype can support 256 such PEs (with a total of 512 DSPs) and 256 BRAMs in total. The *input local store* uses 8 BRAMs per core (with a total capacity of 36KB) to match the bandwidth of off-chip data bank. We use the remaining BRAMs for smart memory.

Limited by the number of off-chip SRAMs available on the FPGA board, our accelerator uses two SSI cores each using one SRAM bank as instruction memory. We utilize the off-chip memory and implement two cores each with 128 PEs (256 DSPs) as shown in Fig. 6. Each processing core is connected to two off-chip DDR2 memory banks and a SRAM bank through a high bandwidth bus. The DDR2 serves as a data memory while the assembly instructions are stored in the SRAM. Within the core, a switch enables the accelerator to alternate between its memory banks for inputs and outputs. The off-chip banks are also connected to a general-purpose host computer via a PCI-X interface. The read/write access to the off-chip memory – either by the host or by the accelerator – is controlled by our API. The SSI execution using our accelerator is as follows: a) API function provides host access to the FPGA memory; b) Host transfers all documents to the data banks of all cores, c) Host distributes queries to the data banks, d) Host transfers assembly code to SRAM banks of all cores, e) API function gives memory access to the accelerator; f) API function initiates accelerator execution and polls for its completion, g) When completed, API function gives memory access back to host; h) Host reads back the indexes of top *k* documents from the off-chip data banks to host memory.

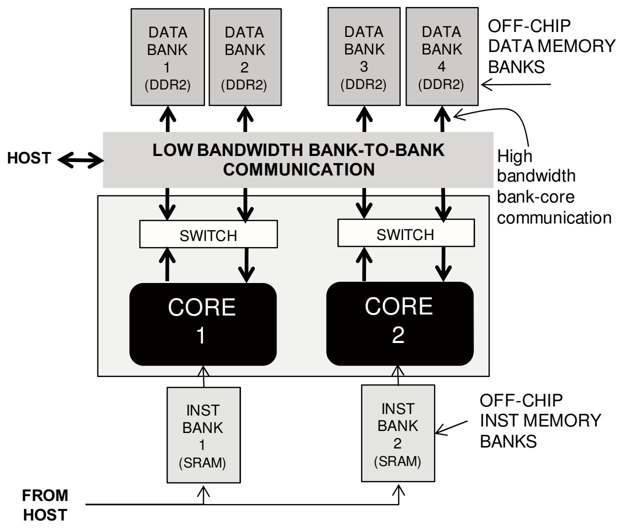


Fig. 6. Architecture of the SSI accelerator with off-chip memory interface

B. FPGA Prototype

The FPGA's resource constraints dictate the accelerator to be implemented as two cores accessing four DDR2 memory banks and two off-chip SRAMs. However, with a total budget of 256 PEs (256x2 = 512 DSPs), we seek to determine the optimal layout of a core in terms of number of chains and chain size that maximizes SSI's performance. We use a C++ simulator and simulate all processor and memory latencies and provide a cycle-accurate estimate of the execution time based on the processor layout. The off-chip memory latencies are kept similar to that from functional simulation, and its effect on actual execution is minimal since all the off-chip accesses are serialized. Table II shows the different parameters of our accelerator considered for this simulation. We simulate SSI for 1024 queries (allowing full parallelism by processing ***H*** queries at a time) to rank 2M documents each with a vector size of 100.

TABLE II
ARCHITECTURAL EXPLORATION SETUP

Type	Parameter	Value
Off-chip Memory Organization	Number of DDR2 banks	4
	DDR2 bandwidth per bank	8 words-per-cycle
Processor	Total DSP budget	512
	Number of cores (***C***)	2
	DSP budget per core (***P***)	256
	Number of chains (***H***)	Variable
	DSPs per chain (***M***)	Variable

Fig. 7 shows the performance variation of SSI as the chain size varies from 1 to 256 DSPs. We notice that the number of cycles remains largely insensitive to *k*. This is due to dynamic ranking of the arrays by smart memories along with the gradual decrease in stall probability as the algorithm progresses, which gets hidden along with the next vector operation. We also observe that the accelerator achieves best performance when the chain uses 8 DSPs. This is because the chain consumes as many documents as the off-chip memory can provide i.e. ***M*** matches the DDR2 memory bandwidth of 8 words-per-cycle. For the extreme cases, smaller chain length leads to lower *parallelism mode* and results in limited fine-grained data parallelism for matrix multiplication. Increase in chain length increases parallelism, which improves performance. However, when the chain length increases beyond 8 DSPs, the chain consumes documents faster than the rate at which DDR2 can supply. This results in all the chains to wait before the documents are buffered from the DRAM on the *input local store*. For higher number of DSPs beyond 64, the performance loss is more severe because of added NOP cycles to the processor chain. This is done in order to compensate for the relatively smaller document length when compared to the larger chain size. Based on these results, we architect one core of the SSI accelerator by laying out 256 DSPs in a two-dimensional grid - 32 chains each containing 8 DSPs. The processor layout or the number of processors is independent of the document size or number of queries. The way the PEs of the already programmed FPGA are going to be used is determined by the *parallelism mode* set by the assembler.

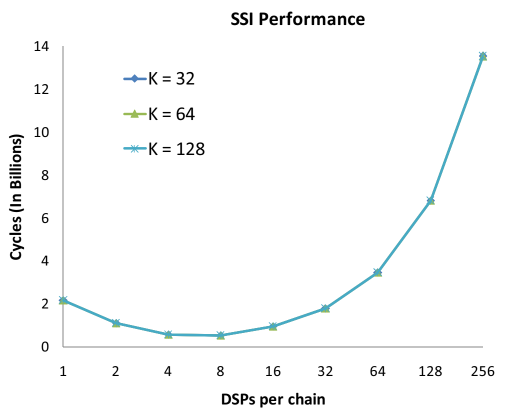


Fig. 7. Architectural exploration of the SSI accelerator

Fig. 8: Virtex 5 SX240T FPGA Card

C. *FPGA Resource Utilization*

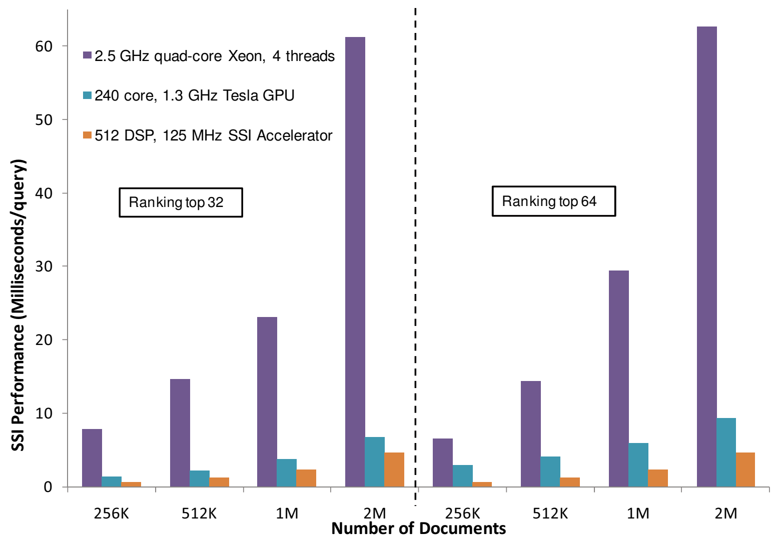
We use Xilinx ISE 12.1 tool to synthesize and implement the SSI accelerator on Virtex 5 SX240T FPGA device [13]. Fig. 8 shows the FPGA card that implements the SSI accelerator with the resource utilization shown in Table III. The accelerator is connected to the host through a 66 MHz, 64 bit PCI-X interface, and runs at a frequency of 125 MHz. With each BRAM of 36Kbits, the total on-chip memory of SSI accelerator is 1.7MB. This amount of memory available to the SSI accelerator is sufficient because of the in-memory processing of smart memories, which ranks the arrays as and when the intermediate data is generated.

TABLE III
FPGA Resource Utilization

V. Experimental Results

In this section, we present the measured performance of our SSI accelerator and compare it with (i) optimized parallel software implementations using a 2.5 GHz Xeon quad-core, (ii) GPU implementations on a 240-core 1.3 GHz Tesla C1060 [14] running CUDA 3.0 with CUBLAS library.

We implemented SSI in software using Intel MKL (BLAS) for the dot-products, followed by an optimized multi-threaded implementation of array ranking. Fig. 9(A) compares this with that of FPGA accelerator's performance in ms/query, whereas Fig. 9 (B) shows the respective speedups. We process 64 text queries in order to search for top *k*=32 and *k*=64 documents from a database of 256K to 2M documents. We find our design to be up to 10-13x faster than the optimized software. We also implement SSI on a 240 core 1.3 GHz Tesla GPU. The GPU implementation uses CUBLAS to compute dot-products. The top *k* is computed by CUDA threads each ranking a set of documents in a map-reduce fashion such that there are no incoherent memory loads and stores. For this application, the number of threads per CUDA block is limited by GPU's on-chip shared memory resources, which makes the FPGA faster by 1.5-5x. We also use the accelerator to search 1.8 million Wikipedia documents each with a vector length of 100 using a dataset from [3]. With this real dataset, we obtain a throughput of 3.33 ms/query for *k*=32 and 64, which is a 13x improvement over Xeon implementation. For all the performance measurement of the SSI accelerator, we include the time to transfer the queries from the host to the FPGA memory, and the FPGA setup time to distribute the queries to all its *PE local stores*. However, we consider transferring the documents to FPGA memory as one time operation, and do not include that in measuring the SSI performance. One point to note here is as *k* increases, speedup also increases. This is because of the additional time spent by both CPU and GPU to rank more documents relative to the SSI accelerator which is largely insensitive to *k* variation, as demonstrated in Fig. 7.



(A) (B)

Fig. 9. (A) FPGA prototype's performance over Xeon and GPU, (B) Speedup of the SSI Accelerator over Xeon and GPU

VI. 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)[4][7][8][9], (iii) algorithm-specific accelerators on FPGAs [10]and (iv) other embedded and analog hardware implementations. SSI training has been accelerated over GPUs by using best-effort approach that relaxes accuracy in order to achieve performance [11]. However, no work has been published for accelerating SSI classification to the best of our knowledge. Our accelerator prototype demonstrates a speedup of 1.5-5x over a GPU, and operates on 16-bit fixed-point signed numbers without compromising accuracy.

VII. Conclusion

We presented an application specific accelerator for semantic text analysis. In order to design the accelerator, we profiled a specific semantic analysis algorithm called SSI and identified its performance bottlenecks. The SSI accelerator uses hundreds of simple parallel processing elements together with in-memory processing so that all intermediate data is simultaneously generated and consumed, without the need for going off-chip. Further, the accelerator's PEs can be logically reorganized to achieve different "levels" of parallelism, essentially maximizing utilization for different input data.

We built a working FPGA prototype of the SSI accelerator, and showed speedups over parallelized multi-core as well as GPU implementations of SSI. Besides intricacies involving mapping such an accelerator to an off-the-shelf FPGA card, the higher-level contributions of this work are the use of in-memory processing together with massively parallel many-core PEs to handle semantic analysis.

References

[1] Mei, T.; Hua, X.; Yang, L.; Li, S., "VideoSense: towards effective online video advertising," In *Proceedings of the 15th International Conference on Multimedia*. MULTIMEDIA '07, 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*. 2009.

[4] Basic Linear Algebra Subprograms (BLAS) http://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

[5] Streaming SIMD Extensions (SSE) http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

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

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

[8] 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,* Proceedings. LNCS 2009.

[9] 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.

[10] 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 (NIPS), 23rd Conference on*, Dec. 2008.

[11] Byna, S.; Meng, J.; Chakradhar; S. T.; Raghunathan, A.; and Cadambi, S., "Best Effort Semantic Document Search on GPUs", *Third Workshop on General-Purpose Computation on Graphics Procesing Units*, Pittsburgh, PA, Mar 2010

[12] Alpha Data ADM-XRC-5T2 Board specification http://www.alpha-data.com/products.php?product=adm-xrc-5t2

[13] Xilinx Virtex 5 SXT FPGA Board specification http://www.xilinx.com/products/virtex5/sxt.htm

[14] NVIDIA Tesla C1060 GPU specification http://www.nvidia.com/object/product_tesla_c1060_us.html