Hardware Acceleration for Similarity Measurement in Natural Language Processing

Prateek Tandon* Jichuan Chang† Ronald G. Dreslinski*
Vahed Qazvinian* Parthasarathy Ranganathan† Thomas F. Wenisch*
*Department of Computer Science and Engineering, University of Michigan †HP Labs

***Abstract*—The continuation of Moore's law scaling, but in the absence of Dennard scaling, motivates an emphasis on energy-efficient accelerator-based designs for future applications. In natural language processing, the conventional approach to automatically analyze vast text collections—using scale-out processing—incurs high energy and hardware costs since the central compute-intensive step of similarity measurement often entails pair-wise, all-to-all comparisons. We propose a custom hardware accelerator for similarity measures that leverages data streaming, memory latency hiding, and parallel computation across variable-length threads. We evaluate our design through a combination of architectural simulation and RTL synthesis. When executing the dominant kernel in a semantic indexing application for documents, we demonstrate throughput gains of up to 42× and 58× lower energy per similarity-computation compared to an optimized software implementation, while requiring less than 1.3% of the area of a conventional core.**

***Keywords*-hardware acceleration, cosine similarity, natural language processing**

I. Introduction

Whereas technology trends indicate that transistor dimensions will likely continue to scale for several technology generations, the anticipated end of CMOS voltage (a.k.a. Dennard) scaling has led many researchers and industry observers to predict the advent of "dark silicon"; that is, that much of a chip must be powered off at any time [3], [9], [17], [24]. This forecast has renewed interest in domain specific hardware accelerators that drastically improve the energy-efficiency of compute intensive tasks to create value from otherwise dark portions of a chip.

One target domain for such accelerators is natural language processing (NLP). With the explosive growth in electronic text, such as emails, tweets, logs, news articles, and web documents, there is a growing need for efficient automatic text processing (e.g., summarization, indexing, and semantic search). The conventional approach to analyze vast text collections—scale-out processing on large clusters with frameworks such as Hadoop—incurs high costs in energy and hardware [11]. We propose and evaluate a hardware accelerator that addresses one of the most data- and compute-intensive kernels that arises in many NLP applications: calculating similarity measures between millions (or even billions) of text fragments [1], [4], [6], [19], [21].

We develop this accelerator in the context of a motivating NLP application: constructing an index for semantic search (search based on similarity of concepts rather than string matching) over massive text corpora such as Twitter feeds, Wikipedia articles, logs, text messages, or medical records. The objective of this application is to construct an index where queries for one search term (e.g., "Mitt Romney") can locate related content in documents that share no words in common (e.g., documents containing "GOP candidate"). The intuition underlying semantic search is that the relationship among documents can be discovered automatically by clustering on words appearing in many documents (e.g., "GOP" frequently appearing in documents also containing "Romney"). Such a search index can be constructed by generating a graph where nodes represent documents (such as tweets) and edges represent their pairwise similarity according to some distance measure (e.g., the number of words in common) [8], [16]. A semantic search can then be performed by using exact text matching to locate a node of interest in this graph, and, thereafter, using breadth-first search, random walks, or clustering to navigate to related nodes.

Constructing the search graph nominally requires a distance calculation (e.g., cosine similarity) between all document pairs, and is hence quadratic in the number of documents. This distance calculation is the primary computational bottleneck of the application. As an example, over half a billion new tweets are posted to Twitter daily [25], implying roughly 10^{17} distance calculations per day, and this rate continues to grow. Clever pre-filtering can reduce the required number of comparisons by an order of magnitude; nevertheless, achieving the required throughput on conventional hardware remains expensive. For example, based on our measured results of an optimized C implementation of this distance calculation kernel running on Xeon-class cores, we estimate that a cluster of over 2000 servers, each with 32 cores is required to compare one day's tweets within a 24-hour turnaround time.

Instead, we develop an accelerator that can be integrated alongside a multicore processor, connected to its last-level cache, to perform these distance calculations with extreme energy efficiency at the bandwidth limit of the cache interface. The accelerator performs only the distance calculation kernel; other algorithm steps, such as tokenization, sorting, and pre-filtering, have runtimes that grow linearly in the number of documents and are easily completed in software. Our design is inspired by the latency hiding concepts of multi-threading and simple scheduling mechanisms to maximize functional unit utilization. The accelerator comprises a window of active threads (each corresponding to a single document pair), a simple round-robin functional unit scheduler, and three kinds of functional units: intersection detectors (XDs), which identify matching tokens (words) in documents; floating point

multiply-accumulate units (MACs), which perform distance calculations; and floating point multiply-divide units (MDIVs), which normalize the distance measure before it is written back to memory. We evaluate the design through a combination of cycle-accurate simulation in the gem5 framework (performance analysis) and RTL-level synthesis (energy analysis). For Twitter and Wikipedia datasets, our accelerator enables 36×-42× speedup over a baseline software implementation of the distance measurement kernel on a Xeon-like core, while requiring 56×-58× lower energy.

II. Related Work

Hardware accelerators for text processing, clustering, semantic search, and database applications have been the focus of extensive research in the architecture community. Tan and Sherwood present a specialized, high-throughput string matching architecture for intrusion detection and prevention [23]. Chen and Chien investigate low-power and flexible hardware architectures for k-means clustering [5]. Fushimi and Kitsuregawa describe a co-processor with hardware sorters for database applications [10], and Moscola et al. implement reconfigurable hardware that extracts semantic information from volumes of data in real-time [13]. Roy et al. present an algorithm for frequent item counting that leverages SIMD instructions [18].

Our accelerator relies on a fast set intersection detector, a topic of much prior work. Wu and co-authors demonstrate a GPU-based solution for set intersection detection on the CUDA platform [26]. Schlegel et al. propose an algorithm for sorted set intersection computation that speculatively executes comparisons between sets using SIMD instructions available on modern processors [20]. Ding and Konig develop linear space data structures to represent sets such that their intersection can be computed in a worst-case efficient way and within memory [7]. In contrast to these works, we propose custom hardware to perform set intersection that is particularly suited to the NLP domain.

Perera and Li have done extensive work in the area of hardware support for distance measurement computation [14], [15]. Their work targets FPGAs and smaller, fixed length vectors. Our proposed design, however, overcomes the drawbacks of FPGAs, and targets variable vector lengths, which is important when dealing with documents larger than a few words.

III. Design

We briefly describe the overall problem of constructing a semantic search index and then focus on the dominant kernel, distance calculation between documents, and how our hardware accelerates this operation.

A. Constructing a Semantic Search Index

The motivating context for our accelerator is the problem of constructing a semantic search index over snippets of text. We implement an algorithm based on the text similarity quantification work of Erkan and Radev [8]. The full application is described in informal psuedo-code in Figure 1. In the first step, the textual documents are transformed into a vector representation to reduce their memory footprint. Each word in a document is replaced with a tuple comprising a token id and a weight that represents the information content of the word (based, e.g., on the word's a priori appearance frequency in English text). We then sort the tokens so that the set intersection of two documents can easily be determined with a merge join.

The similarity calculation step nominally must compare all documents pairs, however, documents that share no word in common have a similarity score of zero. The total number of comparisons can be reduced by an order of magnitude by first bucketizing documents (step 2), i.e., adding a pointer to the document into a bucket corresponding to each token in the document. Hence, each bucket contains only documents sharing at least one word in common.

The similarity calculation step then processes each bucket, calculating the similarity of each document pair via a merge join. Our hardware design accelerates this step. Following common practice [8], [16], we use *cosine similarity* (the normalized dot product of the two weight vectors) as the distance measure, but our hardware architecture could easily implement other distance measures by replacing the multiply-accumulate operation with an appropriate alternative.

Once the complete similarity matrix of all document pairs has been calculated, the final step is to construct a graph where nodes correspond to documents, and edges connect together documents with similarity scores above some fixed threshold. Then, a conventional search index, mapping search terms to nodes for documents containing those words, is constructed. Starting from these exact-word-match nodes, additional related documents can be discovered through traversal of the graph (e.g., via random walk).

Whereas GPUs are often used for problems that exhibit large-scale parallelism, they are not well-suited to calculating distance measures using the method described in Figure 1. Because input documents vary in length, the merge-join set intersection operation does not lend itself to SIMT parallelism, since loop bounds for each document-pair depend on document length. It is unclear how to stage the input data to avoid substantial thread divergence and many idle GPU threads. It is also unclear how to lay out data in memory to enable coalesced accesses, which are crucial to high GPU performance.

B. Accelerator Architecture

Our accelerator implements step three of Figure 1 entirely in hardware. Figure 2 shows a block diagram of our accelerator. The accelerator is connected to the L2 bus and reads from the system's L2 cache. Since the accelerator never reads its own output, it writes memory, via the L2 bus, with non-cacheable transactions that bypass L2. The CPU controls the accelerator by preparing a region of memory with an array of document-pair descriptors. Each descriptor contains the address of the vector representing each document and a destination address for the similarity calculation result. The accelerator is activated through programmed I/Os that provide the start address and length of the descriptor array. The CPU can then sleep until an interprocessor interrupt from the accelerator is delivered to indicate completion.

```
[Step 1: Build vocabulary, tokenize, and sort]
  1.   For all documents:
  2.      Split into sub-strings at whitespace and punctuation
  3.      Replace each sub-string with {token id, IDF weight}; add new tokens as needed
  4.      Sort tokens in ascending order; replace duplicates with count
  5.      Write out documents as sorted vectors of { token, weight = count * IDF }
[Step 2: Pre-filter and bucketize]
  1.   For all documents:
  2.      For all tokens in document:
  3.         Insert pointer to document into a bucket corresponding to the token
[Step 3: Similarity calculation]
  1.   For all buckets:
  2.      For all document pairs {d1, d2} in bucket:
  3.         while d1 or d2 has more tokens:
  4.            if d1.token == d2.token:                          //XD
  5.               numerator = numerator + d1.weight * d2.weight  //MAC
  6.               pop front token from d1 and d2
  7.            else:
  8.               pop front token with lower token id
  9.         similarity[d1,d2] = numerator / ( ||d1|| * ||d2|| )  //MDIV
[Step 4: Build similarity graph]
  1.   Construct graph with node for each document and edges connecting documents with similarity > threshold
  2.   Traverse graph (e.g., via random walk) to discover related documents and build index
```

Fig. 1: High-level description of semantic search index construction

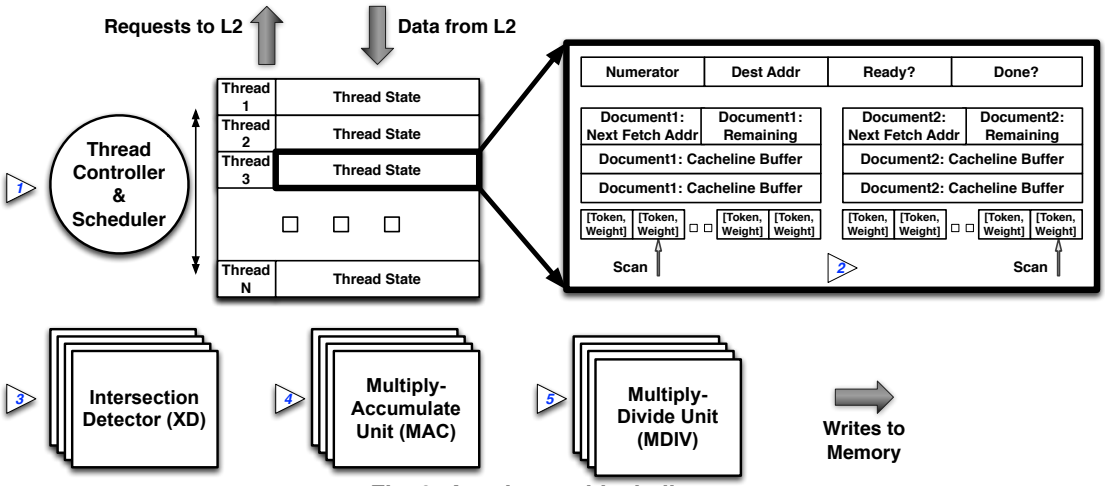


Fig. 2: Accelerator block diagram.

The accelerator is architected much like an in-order core and has six major architectural blocks: a memory read interface, a thread controller/scheduler, intersection detectors (XDs), multiply-accumulate units (MACs), multiply-divide units (MDIVs), and finally, a memory write interface. As most NLP algorithms represent concepts like document similarity with floating-point values, we use floating-point functional units. We describe the operation of the accelerator by walking through a simple example. Numerical labels in Figure 2 correspond to the steps described below.

(1) Fill thread window. The thread controller maintains a window of active threads (each thread corresponding to a document pair specified in a descriptor), and schedules threads to functional units using a simple round-robin scheduler. Each thread window entry comprises thread status information (addresses for the next data fetches, destination address, remaining indices to be scanned, partial sum) and several cacheline-sized data blocks for each input document. If any thread window entry is empty, the controller fills it with the next descriptor in the input array. The controller then iterates over all active threads and issues requests to L2 to fill all available buffer space for each document. While our simulations ignore virtual memory, in a practical implementation, virtual addresses must be translated by either a dedicated TLB or by the TLB of a core neighboring the accelerator.

(2) Token data arrives. Once document data arrive for a particular thread, processing can begin. Each document is represented as an array of $\{token, weight\}$ tuples. Each cycle, a ready thread arbitrates for an XD unit which will compare the next two tokens sent to it.

(3) XD comparison. The operation of the XD units is conceptually similar to a sort-merge join. In a particular cycle, the XD unit compares the token ids of the next tokens from each document. Recall that tokens in each document have been sorted. Hence, if the token ids do not match, the head pointer for the document with the smaller token is advanced and updated tokens are compared again in the next cycle. If the tokens match, the thread is marked and will arbitrate for a MAC unit in the next cycle.

(4) MAC calculation. When an XD unit reports a match, the MAC unit performs the floating-point multiply-accumulate

Parameter	*Range*
Number of XDs	1-32
Number of MACs	1-32
Number of MDIVs	1-32
Thread Window Size	1-64
XD Delay	1 cycle
MAC Delay	3 cycles
MDIV Delay	7 cycles

TABLE I: Simulation Parameters.

Statistic	*Twitter*	*Wikipedia*
Number of documents in dataset	10,000,000	100,000
Number of documents in bucket of interest	2,500,000	15,000
Average document length (tokens)	9.3	511.5
Minimum document length (tokens)	2	24
Maximum document length (tokens)	39	4,107
Average number of intersections	1.01	200.9

TABLE II: Statistics for Twitter and Wikipedia Datasets

Unit	*Area (μm^2)*	*Delay (ns)*	*Power (mW)*
XD	1,143	0.5	0.5
MAC	14,232	1.5	4.93
MDIV	18,216	3.5	3.95
Controller+Thread Window	293,878	0.5	69.4

TABLE III: Synthesis Results.

to calculate an updated numerator for the document similarity computation. The input weight values from the two documents are multiplied and summed with the current numerator, and the result is stored back into the active thread entry.

(5) Normalization. When the end of either input document is reached, the merge-join set-intersection operation is complete. The thread then arbitrates for an MDIV unit to normalize the accumulated numerator by the product of the magnitudes of the input documents, and then arbitrates for the store interface unit to write its output value to the destination address in memory. The thread window entry is then freed.

IV. Methodology

We use a two-pronged approach to evaluate our design relative to an optimized software baseline. We measure performance of both the pure software implementation and hardware-accelerated kernel using the gem5 architectural simulator [2]. To investigate energy savings and area overheads, we implement our design in Verilog and synthesize using industrial 45nm standard cells.

A. Simulation

To compare the performance of our accelerator to a CPU baseline, we extend the gem5 simulator with a device model for our accelerator. The accelerator connects to the L2 interface. It can read and write 64-byte cache blocks from L2 and is controlled via programmed I/O to special memory locations. We vary the hardware parameters of our design (number of threads, XDs, MACs, and MDIVs) to determine the minimum hardware needed to saturate L2 and/or main memory bandwidth, which ultimately limits the performance of the accelerator. Table I shows the various parameters of the design space we explore. We determine functional unit delays from synthesized timing results targeting a 2 GHz clock.

We contrast our hardware design with a SSE-accelerated C implementation of the cosine similarity kernel compiled with gcc -O3. We model a 4-wide out-of-order processor running at 2 GHz with 64kB L1 caches and an 8MB L2. As operating system interactions and I/O do not contribute significantly to the runtime of this workload, we use gem5's syscall emulation mode. Note that this simulation mode does not model virtual memory; nevertheless, because of the high data locality, TLB misses are unlikely to significantly affect the runtime of the baseline or hardware-accelerated execution. We validate that the CPU runtimes reported by gem5 are, on average, within 6% of the runtimes observed on a comparable 8-core Xeon-class server. For consistent energy comparisons between the CPU baseline and our hardware design, we report the gem5 results.

We construct benchmarks for Twitter and Wikipedia from databases of 10 million tweets and 100,000 articles respectively. Table II shows various statistics for the datasets we use. The software pre-processing steps of the semantic index construction algorithm (tokenization, sorting, and bucketizing) are performed offline in advance; our measurements focus only on the dominant distance calculation step. From the Twitter database, we select the most frequently occurring token (corresponding to the string *"RT"*) and construct a bucket of all tweets containing this token (2.5 million entries, requiring 6.25 trillion distance calculations). As it is impossible to process this vast dataset in simulation, we simulate only the first 5 million tweet-pair comparisons and use the first 1 million tweet-pairs for warm-up. We follow a similar bucketization process for the Wikipedia data, and simulate 50,000 article-pair comparisons with the first 2000 pairs used for warm-up.

When processing the entire data set, the document-pair comparisons are blocked to maximize L2 locality. Thus, the computation will alternate between one phase where a large fraction of document accesses miss to main memory and a much longer phase where a block of documents is resident in L2 and there are no main memory accesses. The relative time spent in each phase depends on the relative size of the document bucket and the L2 cache. To ensure that our accelerator design hides latency and saturates available L2/memory bandwidth in both phases, we construct two test cases: *Fit*, wherein all documents are L2-resident, and *Spill*, wherein the L2 is empty and documents must be retrieved from memory. We report speedup of the accelerator relative to the CPU baseline for both phases. To avoid L2 cache pollution, and since the accelerator never reads its own output, outputs are written directly to main memory using uncacheable writes.

B. Timing, Power, and Area Analysis

We implement the accelerator in Verilog and synthesize using an industrial 45nm standard cell library to obtain delay, area, and power results assuming a 0.72V supply voltage. Table III shows the post-synthesis delays and areas for each of the sub-units of the accelerator (the thread window size is 6, the configuration we use in our final design). We use these synthesized delay results to set functional unit latencies within the gem5 model. Our floating-point multiply, multiply-

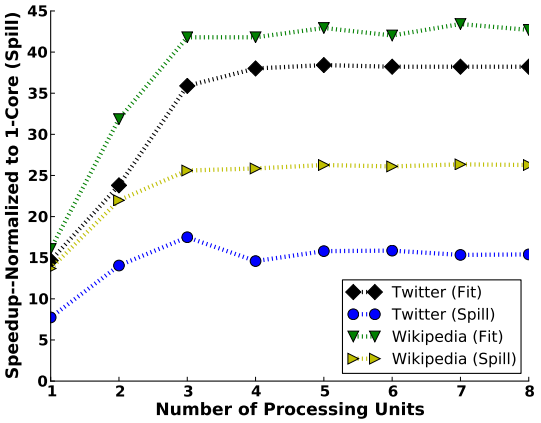


Fig. 3: Speedup. Normalized to 1-core *CPU (Spill)* case.

accumulate, and divide units are from the Synopsys DesignWare IP suite [22]. The delays reported in the table are rounded up to the next 0.5ns clock edge. We use system configuration and functional unit activity results from gem5 to generate estimates of CPU core and cache power using McPAT [12].

V. Results

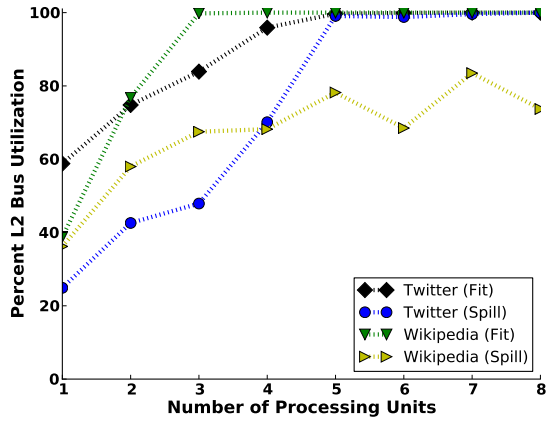
The following subsections outline the performance and energy improvements afforded by our design.

A. Performance

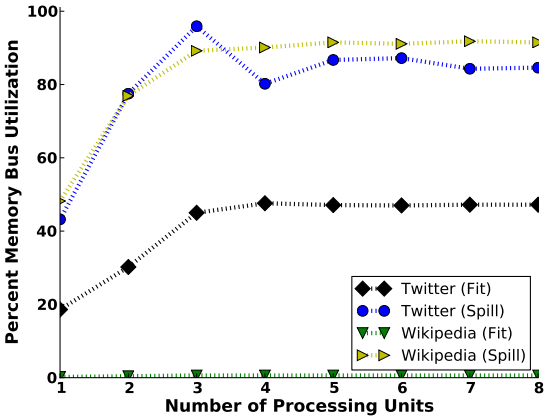
We first contrast the performance and performance scalability of the accelerator relative to the baseline cosine similarity software kernel running on conventional out-of-order CPU cores. Figure 3 shows the speedup provided by the accelerator for the Twitter and Wikipedia datasets for both the *Fit* and *Spill* scenarios normalized to each single-core *Spill* CPU baseline. On the horizontal axis, we vary the amount of hardware dedicated to the accelerator. To simplify presentation of the results, in this experiment, we vary the number of XD, MAC, MDIV units together, from 1 to 8. Each configuration has double the number of thread slots as XD units. These configurations all overprovision MAC and MDIV units relative to XD units; we further optimize the functional unit mix in subsequent experiments.

Through a combination of simulation, and experiments on a Xeon-class server, we verify that the baseline CPU performance scales roughly linearly with the number of CPU cores, up to about an 8.5× speedup with 8 cores for the *Twitter (Fit)* scenario over the single-core *Twitter (Spill)* case. CPU performance is limited because of the overheads of instruction execution (memory addressing, loop flow control, etc.).

The accelerator enables substantial speedups. Even with only one of each functional unit, the accelerator can achieve speedups of 8× and 14× in the *Twitter (Spill)* and *Wikipedia (Spill)* scenarios respectively. In general, three XD units are required to achieve peak speedup. The accelerator improves performance because it eliminates all software overheads; e.g., in the *Twitter (Fit)* case, each XD unit can process a tweet-pair roughly every 24 clock cycles, while a CPU core on average requires 353 cycles to execute 469 instructions per tweet-pair.



(a) Percent Accelerator-L2 Bus Utilization



(b) Percent L2-Memory Bus Utilization

Fig. 4: L2 and Memory Bus Utilization. (a) L2 bus saturates for both *Spill* scenarios and the *Twitter (Spill)* case. (b) Memory bus is a bottleneck for the *Spill* configurations.

Performance generally saturates beyond three XDs since the accelerator fully utilizes either the L2 bus for *Twitter (Fit and Spill)* and *Wikipedia (Fit)* scenarios, or main memory bandwidth for both *Spill* scenarios. We show the relevant bus utilization results in Figure 4. *Twitter (Spill)* is L2- or memory-bandwidth bound depending on the number of processing units deployed; this configuration also demonstrates decreased performance with more than three processing units due to destructive interference effects. *Wikipedia (Fit)* shows little memory traffic since the number of writes to memory is negligible, and all reads are serviced by the L2. As a point of comparison, for the CPU case, even with 8 cores, the L2 and memory bus utilizations peak at 8.5% for *Wikipedia (Fit)* and at 19% for *Twitter (Spill)* respectively.

We find that the pareto-optimal design, when considering performance, energy, and area in conjunction, consists of three XD units, two MAC units, one MDIV unit, and six thread slots. The Wikipedia dataset tends to favor slightly more functional units compared to the Twitter dataset due to its larger document size. Further, larger thread windows are favored in the *Spill* scenarios since they must maintain more outstanding accesses to the memory system to hide the long delay to access main memory.

Accelerator Configuration		
[XDs, MACs, MDIVs]	[3, 2, 1]	
Issue Window	6	
Area		
Core	24.88 mm^2	
Accelerator	0.31 mm^2	
Power		
Core	6 W	
Uncore	14.8 W	
Accelerator	0.43 W	
Fit Energy & Performance – CPU 1-core Baseline		
	Twitter	*Wikipedia*
Core Energy/Document-Pair	1.18 μJ	96.5 μJ
Chip Energy/Document-Pair	4.09 μJ	339.4 μJ
Fit Energy & Performance – Accelerator		
	Twitter	*Wikipedia*
Accelerator Energy/Document-Pair	2.12 nJ	170.9 nJ
Chip Energy/Document-Pair	72.7 nJ	5.8 μJ
Chip Energy Ratio (Core:Accelerator)	56.3:1	58.5:1

TABLE IV: Power & Area Results.

B. Area and Energy

Table IV shows the area overhead and energy savings when using our accelerator in the [3 XD, 2 MAC, 1 MDIV] configuration with a thread window size of six. Note that, we assume that the accelerator is power-gated when cores are active and vice-versa. The accelerator only imposes an area overhead of 0.31 mm^2, less than 1.3% of the area of a core. Because of its simple microarchitecture and lack of instruction fetch/decode bottlenecks, the accelerator's power requirements are much lower than that of a core. The power savings translate to an even larger energy-efficiency gain, since the accelerator can also process document-pairs much faster (and hence incur less leakage overhead per processed document-pair). Overall, the accelerator improves energy efficiency by approximately two orders of magnitude relative to the CPU baseline.

VI. Conclusion

The conventional approach of using scale-out methods to automatically analyze vast text collections incurs high energy and hardware costs since the central compute-intensive step of similarity measurement often entails pair-wise, all-to-all comparisons. We propose a custom hardware accelerator for similarity measures that leverages data streaming and parallel computation, and, due to its low-power requirements, utilizes dark silicon areas of the chip that would otherwise have to be powered down. Architectural simulations and RTL synthesis demonstrate throughput gains of up to 42× and 58× lower energy consumption compared to an optimized software implementation of cosine similarity calculation, while incurring minimal area overheads.

Acknowledgments

This work was partially supported by NSF CCF-0815457.

References

[1] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling Up All Pairs Similarity Search. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 131–140, 2007.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[3] S. Borkar and A. A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[4] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic Clustering of the Web. *Comput. Netw. ISDN Syst.*, 29(8-13), 1997.

[5] T.-W. Chen and S.-Y. Chien. Flexible Hardware Architecture of Hierarchical K-Means Clustering for Large Cluster Number. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(8), 2011.

[6] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding Replicated Web Collections. *SIGMOD Rec.*, 29(2):355–366, May 2000.

[7] B. Ding and A. C. König. Fast set intersection in memory. *Proc. VLDB Endow.*, 4(4):255–266, Jan. 2011.

[8] G. Erkan and D. R. Radev. LexRank: Graph-Based Lexical Centrality as Salience in Text Summarization. *J. Artif. Int. Res.*, 22(1), Dec. 2004.

[9] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Computer Architecture (ISCA), 2011 38th Annual Intl. Symposium on*, 2011.

[10] S. Fushimi and M. Kitsuregawa. GREO: A Commercial Database Processor Based on a Pipelined Hardware Sorter. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 449–452, 1993.

[11] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. *SIGOPS Operating Systems Review*, 44(1), 2010.

[12] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009.

[13] J. Moscola, Y. Cho, and J. Lockwood. Hardware-Accelerated Parser for Extraction of Metadata in Semantic Network Content. In *Aerospace Conference, 2007 IEEE*, pages 1–8, March 2007.

[14] D. Perera and K. F. Li. On-Chip Hardware Support for Similarity Measures. In *Communications, Computers and Signal Processing, 2007. PacRim 2007. IEEE Pacific Rim Conference on*, pages 354–358, 2007.

[15] D. Perera and K. F. Li. Hardware Acceleration for Similarity Computations of Feature Vectors. *Electrical and Computer Engineering, Canadian Journal of*, 33(1):21–30, 2008.

[16] V. Qazvinian and D. R. Radev. Scientific Paper Summarization Using Citation Summary Networks. In *Proceedings of the 22nd International Conference on Computational Linguistics*, COLING '08, 2008.

[17] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. Wenisch, and M. Martin. Computational Sprinting. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, 2012.

[18] P. Roy, J. Teubner, and G. Alonso. Efficient Frequent Item Counting in Multi-Core Hardware. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge discovery and data mining*, KDD '12, pages 1451–1459, 2012.

[19] M. Sahami and T. D. Heilman. A Web-Based Kernel Function for Measuring the Similarity of Short Text Snippets. In *Proceedings of the 15th International conference on World Wide Web*, WWW '06, 2006.

[20] B. Schlegel, T. Willhalm, and W. Lehner. Fast Sorted-Set Intersection using SIMD Instructions. *ADMS Workshop 2011*, 2011.

[21] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating Similarity Measures: A Large-Scale Study in the Orkut Social Network. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, KDD '05, pages 678–684, 2005.

[22] Synopsys. *DesignWare Building Blocks*. Synopsys Inc., 2011.

[23] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, 2005.

[24] M. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131–1136, 2012.

[25] D. Terdiman. CNET: Twitter hits half a billion tweets a day. http://news.cnet.com/8301-1023_3-57541566-93/report-twitter-hits-half-a-billion-tweets-a-day/.

[26] D. Wu, F. Zhang, N. Ao, F. Wang, J. Liu, and G. Wang. A Batched GPU Algorithm for Set Intersection. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, 2009.