Energy-Efficient String Search Architectures on a Fine-Grained Many-Core Platform

Emmanuel O. Adeagbo and Bevan M. Baas
Department of Electrical and Computer Engineering
University of California, Davis
{eoadeagbo, bbaas}@ucdavis.edu

***Abstract*—This paper presents three energy-efficient methods for searching and filtering streamed data on a fine-grained many-core processor array: parallel, serial, and all-in-one. All three architectures aim to provide programmable flexibility with low energy consumption. Experimental results show that for one keyword search, the parallel and serial architectures consume 2× less energy per workload than the all-in-one architecture. For two or more keyword searches, the all-in-one architecture achieves up to 2.6× higher throughput per area over the parallel architecture, and 25.6× over the serial architecture. Scaled results show that the serial and parallel designs provide 211× increased throughput per area, and yield 155× energy reduction when compared to a traditional processor (Intel Core i7 3667U). The proposed architectures are modular and easily scalable.**

I. Introduction

Exact match string searching matches one or more occurrences of a keyword within a set of input data, and is widely used in many datacenter applications such as large string databases [1], network intrusion detection systems [2], [3], and search engines [4]. As demand for datacenter performance continues to increase, energy consumption has gone up by nearly 4× within the last decade [5]. Previous work in string search has used Field Programmable Gate Arrays (FPGA) [6], [7], traditional CPUs [1], and Graphics Processing Units (GPU) [8], with throughput often the primary focus. FPGAs and GPUs can provide high performance but typically have high energy demands compared to traditional CPUs and fine-grained many-core arrays [9]. Traditional CPUs and GPUs offer ease of programming, while fine-grained many-core processor arrays can compute complex workloads with high performance and high energy efficiency while being smaller than the aforementioned platforms [10]. String search may be also be augmented to other applications such as sorting on the same processor array [11], where the first phase would use string search to filter out undesired data then sort the remaining data.

II. String Search Architectures

The primary component of the proposed string search is the filter, whose main operation is to match a keyword to input data. The core code is simple and easily replicated, only requiring 52 assembly instructions. The pseudocode of the basic filter algorithm is shown in Algorithm 1. The filter starts by reading *inputData* into a buffer. Since a successful search requires a match of all the characters in a keyword, the minimum *buffer* size is equal to the keyword length. The strings are scanned using the buffer rather than using more computationally expensive schemes such as string B-tree data structures or hash tables [12], [8]. Once filled, the buffer entries are compared to individual characters of the keyword. This process is repeated for as long as the following conditions are true: 1) the number of matches is less than the keyword length, 2) there is more input data to process. Since partial matches are possibilities during mismatches, most of the buffer entries must be preserved while replacing the earliest entry with a new one. The output control block sends out a "1 (True)" when the entire keyword matches, or "0 (False)" when the input data terminates prior to a keyword match. A natural requirement of the proposed string search is that the entire set of strings (e.g. a document) must be preserved when finding multiple keywords. The need to preserve the document presents some challenges for small-memory processors if the data is too large to fit in a processor's local memory.

Algorithm 1 Filter

```
while true do
    $buffer \leftarrow inputData[0 : keywordLength - 1]$
    $i \leftarrow 0$
    $localMatch \leftarrow 0$
    while $inputData \neq$ EOF and $localMatch == 0$ do
        if $buffer[i] == keyword[i]$ then
            if $i == keywordLength$ then
                $localMatch \leftarrow 1$
            else
                $i++$
            end if
        else
            $buffer$ << 1 char
            $buffer\ [0] \leftarrow inputData[0]$
            $i \leftarrow 0$
        end if
    end while
    if $firstFilter$ then
        $output \leftarrow localMatch$
    else
        Wait for $inMatch$
        $output \leftarrow (inMatch$ and $localMatch)$
    end if
end while
```

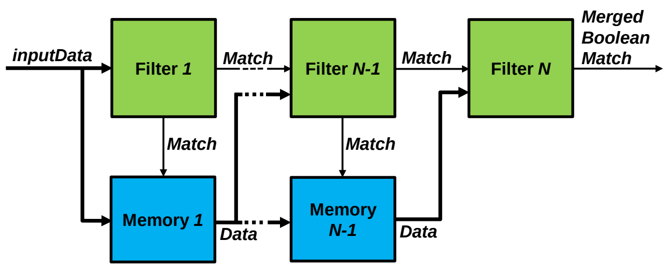

Fig. 1. Serial architecture data flow highlighting the major control signals of each filter and the on-chip memory. The architecture is pipelined with each filter a processor running Algorithm 1.

A. Serial Implementation

The serial implementation is a pipelined architecture with additional memory external to the processor(s). In Fig. 1, each filter is a processor running Algorithm 1. *inputData* streams into both Filter *1* and Memory *1* in parallel. If Filter *1* outputs a "True" *Match*, it is sending a "1" signal to both Memory *1* and Filter *2*. *Data* streams from Memory *1* to Filter *2* and Memory *2* in parallel after which Filter *2* starts processing the next . Filter-memory pairs in the latter parts of the chain conditionally run based on match results of previous filter-memory pairs. Filter *N* produces a "True" *Merged Boolean Match* if all subsequent searches were a success. Due to the sequential nature of the serial architecture, if less common or rare keywords are programmed into earlier filters in the chain, subsequent filters are less likely to run as frequently due to the restrictions on the early filters.

B. Parallel Implementation

In Fig. 2, each filter is a processor running Algorithm 1 where *InputData* is streamed to all of them in parallel for processing. Each *Match* output is boolean merged to *Matches* of subsequent filters and Filter *N* produces a "True" *Merged Boolean Match* if all subsequent searches are a success. Filters shutdown either if they find a match and wait for other processors, or if *InputData* is empty. The modularity of the parallel architecture enables it to easily scale to larger search queries.

C. All In One Implementation

The all-in-one (AIO) architecture combines multiple keyword search operations into the minimum required filter typically one processor. The processor runs Algorithm 1 on multiple keyword searches by using it's data memory to manage the keywords and internal *Matches*. When *Match* is "True", a flag corresponding to the matched keyword is asserted thereby disabling future searches of the keyword. When all keyword flags are asserted or when *inputData* is empty AIO produces *Merged Boolean Match*. Multiple AIO architectures working together allow for dense search queries.

III. Data Generation and Test conditions

The string search architecture performances are evaluated using a list of keywords containing ~350,000 words that are

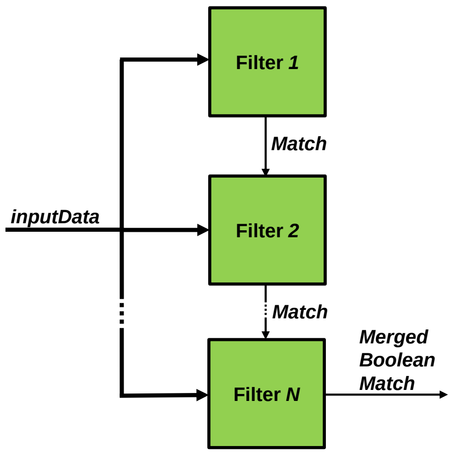

Fig. 2. Parallel architecture data flow highlighting the major control signals of each filter. Each filter is a processor running Algorithm 1 directly on *inputData*.

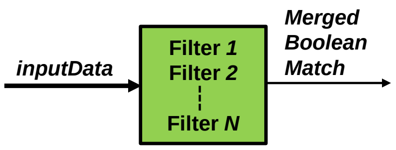

Fig. 3. AIO architecture data flow with combined multiple keyword search operations. The structure is a processor running Algorithm 1 on multiple keywords.

randomly generated from the English dictionary. *inputData* is generated in 8 KB sizes. For a set of keywords, a page excluding these keywords is first generated. A real dictionary is used instead of generating a page of random characters because real words result in more realistic performance data that closely matches real world workloads.

inputData for the architectures are generated using three parameters. The first parameter is the number of keywords, and it sets the amount of filters per keyword. The second parameter is the keyword length which sets the size of a keyword at one byte per character. The last parameter is the location of a keyword in a data page. When a keyword is chosen, it is assigned a random location on a page, favoring the middle of the page. 1000 iterations are carried out to produce consistent averaged results.

IV. Analysis

A. Experimental Results

The serial, parallel and AIO architectures are simulated on a simulator that uses measured values from the AsAP2 chip [9] operating at a supply voltage of 1.3 V, with programmable processors running at 1.2 GHz. Energy per workload for each architecture is defined as the total energy consumed when processing *inputData* divided by the total number of bytes in *inputData*. Fig. 4 shows that for one keyword, the serial and parallel implementations consume $2\times$ less energy per workload than the AIO implementation. For five keywords, the AIO implementation consumes $1.5\times$ less energy per workload

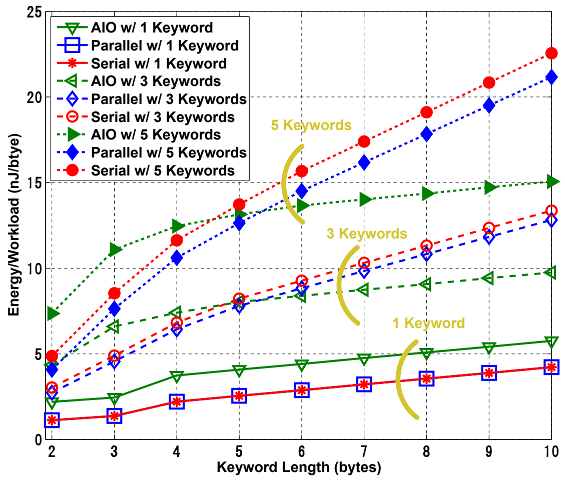

Fig. 4. Energy per workload versus keyword length for different keywords.

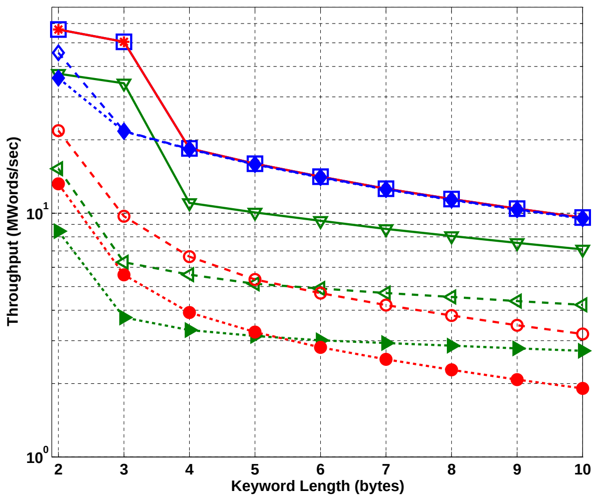

Fig. 6. Throughput comparison at each keyword length for different keywords. See Fig. 4 for legend.

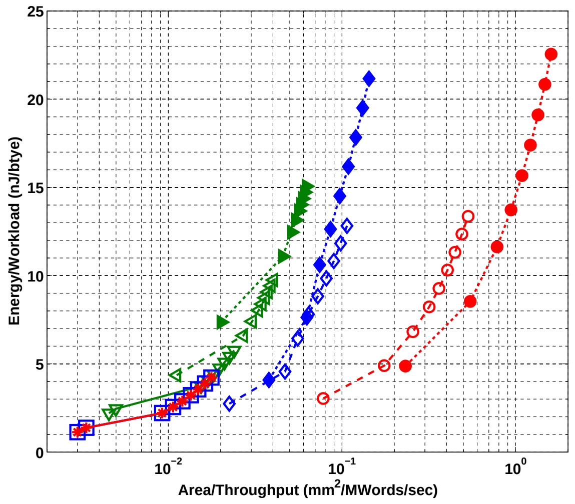

Fig. 5. Energy per workload versus area per throughput for different keywords. See Fig. 4 for legend.

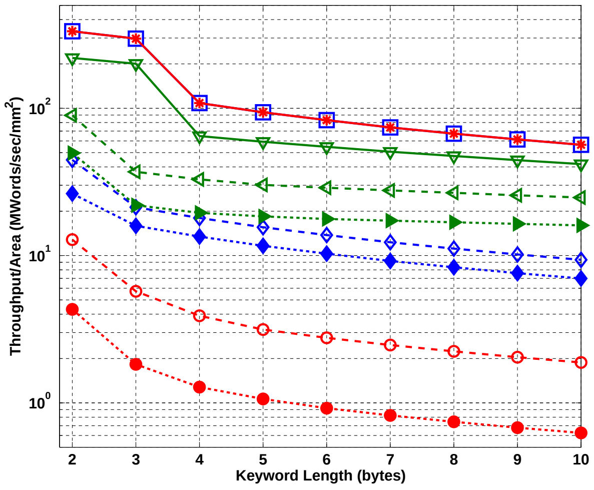

Fig. 7. Throughput per area vs keyword length for different keywords. See Fig. 4 for legend.

over the serial and parallel implementations with majority its energy consumption from branching overhead. The serial, parallel, and AIO implementations consume 22.55 nJ/byte, 21.16 nJ/byte, and 15.06 nJ/byte, respectively, making the AIO implementation the most energy efficient. The energy overhead in the serial architecture comes from the energy required for communication between its filters and the inclusion of the memories.

For a given architecture, area per throughput is defined as the area occupied by the programmable processors plus memory used divided by how quickly they processes *inputData* in units of mm^2/(MWords/sec), where a word is 16 bits wide. Fig. 5 plots the trade offs between energy per workload vs area per throughput for each implementation. For one keyword, the serial and parallel architectures consume approximately 2$\times$ less energy per workload and 1.5$\times$ less area per throughput than the AIO architecture. For three keywords, AIO occupies approximately 2$\times$ and 7$\times$ less area per throughput than the parallel and serial architectures respectively. In contrast, the serial and parallel architectures consumes approximately 2.6$\times$ less energy per workload than the AIO architecture, with similar trends at five keywords.

Fig. 6 shows that for one keyword, the parallel and serial architectures achieve 1.6$\times$ higher throughput than the AIO architecture. For five keywords, the parallel architecture is 5.8$\times$ higher in throughput than the AIO architecture, and 5$\times$ over serial. Longer keyword lengths require more processing, which lead to an average throughput drop from 33 to 6 MWords/sec.

In Fig. 7, for one keyword, the parallel and serial architectures achieve up to 1.6$\times$ higher throughput per area than the AIO architecture. For two or more keywords, the AIO architecture achieves up to 2.6$\times$ throughput per area over the parallel architecture, and 25.6$\times$ over the serial architecture. Although the serial architecture is pipelined, there is still memory read latency after a match result thereby reducing

Table I
ENERGY PER WORKLOAD AND THROUGHPUT PER AREA FOR KEYWORD LENGTH OF 6. SCALED COLUMN VALUES ARE SCALED TO 22 NM [13].

	Architecture	Unscaled Energy/Workload (nJ/byte)	Scaled Energy/Workload (nJ/byte)	Unscaled Throughput (MWords/sec)	Scaled Throughput (MWords/sec)	Unscaled Throughput/Area ((MWords/sec)/mm^2)	Scaled Throughput/Area ((MWords/sec)/mm^2)
1 Keyword	Intel Core-i7 3667U	77	77	408	408	13.8	13.8
	Serial	2.8	**0.50**	14.1	55	83.0	**2910**
	Parallel	2.8	**0.50**	14.1	55	83.0	**2910**
	AIO	4.4	0.80	9.31	36	54.7	1920
5 Keywords	Intel Core-i7 3667U	41	41	256	256	9.02	9.02
	Serial	16	2.8	2.82	11	0.920	35.1
	Parallel	15	2.6	14.0	54	10.3	362
	AIO	14	**2.4**	3.00	12	17.7	**621**

the over data throughput of the serial architecture.

In the case of one keyword, the parallel and serial architectures have the exact same energy, throughput, and area because the serial architecture only requires memory for more than one keyword. The AIO architecture for the one keyword case still has a complexity overhead and therefore consumes slightly more energy with a slightly lower throughput.

B. Comparisons

As a reference point for how well the architectures perform, similar data inputs are processed in C++ on an Intel Core i7 3667U processor (22 nm fabrication technology) for comparison. The fabrication technology for the serial, parallel, and AIO architectures are 65 nm. The results in Table I are for 6 char keyword lengths (6 bytes) showing both unscaled and scaled results. In the scaled columns, the values are scaled from 65 nm to the 22 nm node to match the many core platform on which the workload was performed to the Intel Core i7 [13]. Table I shows for one keyword that the serial and parallel architectures provide 155× in energy savings, and with a 211× increased throughput per area over the Intel Core i7 3667U. For five keywords, the AIO architecture provides 17× in energy savings, and with 69× in increased throughput per area over the Intel Core i7 3667U.

V. CONCLUSIONS

Three energy-efficient architectures are presented utilizing a fine-grained many-core processor array for searching and filtering streamed data. The serial architecture is suited for small keyword searches while the parallel architecture is well suited for larger keyword searches. The all-in-one architecture combines filtering operations and ensures the smallest area footprint. The designs achieve 211× increased throughput per area, and yield 155× energy reduction when compared to a traditional processor (Intel Core i7 3667U).

ACKNOWLEDGMENTS

The authors gratefully acknowledge support from SRC GRC Grant 2321.001, NSF CCF Grant No. 1321163, 1018972, 0903549, 0430090, C2S2 Grant 2047.002.014, Jon Pimentel, Aaron Stillmaker, and Brent Bohnenstiehl.

REFERENCES

[1] J. Yang, W. Wang, and P. Yu, “BASS: approximate search on large string databases,” in *Scientific and Statistical Database Manage., 2004. Proc. 16th Int. Conf. on*, June 2004, pp. 181–190.

[2] L. Tan and T. Sherwood, “A high throughput string matching architecture for intrusion detection and prevention,” in *Comput. Architecture, 2005. ISCA ’05. Proc. 32nd Int. Symp. on*, June 2005, pp. 112–122.

[3] N.-F. Huang *et al.*, “A deterministic cost-effective string matching algorithm for network intrusion detection system,” in *Commun., 2007. ICC ’07. IEEE Int. Conf. on*, June 2007, pp. 1292–1297.

[4] S. Rus, R. Ashok, and D. Li, “Automated locality optimization based on the reuse distance of string operations,” in *Code Generation and Optimization (CGO), 2011 9th Annu. IEEE/ACM Int. Symp. on*, April 2011, pp. 181–190.

[5] M. Pedram, “Energy-efficient datacenters,” *Comput-Aided Des. Integr. Circuits Syst., IEEE Trans.*, vol. 31, no. 10, pp. 1465–1484, Oct 2012.

[6] I. Sourdis and D. Pnevmatikatos, “Pre-decoded cams for efficient and high-speed NIDS pattern matching,” in *Field-Programmable Custom Comput. Mach., 2004. FCCM 2004. 12th Annu. IEEE Symp. on*, April 2004, pp. 258–267.

[7] J. Divyasree, H. Rajashekar, and K. Varghese, “Dynamically reconfigurable regular expression matching architecture,” in *Application-Specific Syst., Architectures and Processors, 2008. ASAP 2008. Int. Conf. on*, July 2008, pp. 120–125.

[8] W. Ong *et al.*, “A parallel bloom filter string searching algorithm on a many-core processor,” in *Open Syst. (ICOS), 2013 IEEE Conf. on*, Dec 2013, pp. 1–6.

[9] D. Truong *et al.*, “A 167-processor computational platform in 65 nm CMOS,” *Solid-State Circuits, IEEE J. of*, vol. 44, no. 4, pp. 1130–1144, April 2009.

[10] ——, “A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling,” in *VLSI Circuits, 2008 IEEE Symp. on*, June 2008, pp. 22–23.

[11] A. Stillmaker, L. Stillmaker, and B. Baas, “Fine-grained energy-efficient sorting on a many-core processor array,” in *Parallel and Distrib. Syst. (ICPADS), 2012 IEEE 18th Int. Conf. on*, Dec 2012, pp. 652–659.

[12] P. Ferragina and R. Grossi, “The string b-tree: A new data structure for string search in external memory and its applications,” *J. ACM*, vol. 46, no. 2, pp. 236–280, Mar. 1999.

[13] A. Stillmaker, Z. Xiao, and B. Baas, “Toward more accurate scaling estimates of CMOS circuits from 180 nm to 22 nm,” VLSI Computation Lab, ECE Department, University of California, Davis, Tech. Rep. ECE-VCL-2011-4, Dec. 2011.