FPGAs in the Datacenters: the Case of Parallel Hybrid Super Scalar String Sample Sort

Mikhail Asiatici
EPFL
Lausanne, Switzerland
mikhail.asiatici@epfl.ch

Damian Maiorano
Politecnico di Torino
Torino, Italy
damian.maiorano.93@gmail.com

Paolo Ienne
EPFL
Lausanne, Switzerland
paolo.ienne@epfl.ch

***Abstract*—String sorting is an important part of database and MapReduce applications; however, it has not been studied as extensively as sorting of fixed-length keys. Handling variable-length keys in hardware is challenging and it is no surprise that no string sorters on FPGA have been proposed yet. In this paper, we present Parallel Hybrid Super Scalar String Sample Sort (pHS^5) on Intel HARPv2, a heterogeneous CPU-FPGA system with a server-grade multi-core CPU. Our pHS^5 is based on the state-of-the-art string sorting algorithm for multi-core shared memory CPUs, pS^5, which we extended with multiple processing elements (PEs) on the FPGA. Each PE accelerates one instance of the most effectively parallelizable dominant kernel of pS^5 by up to 33% compared to a single Intel Xeon Broadwell core running at 3.4 GHz. Furthermore, we extended the job scheduling mechanism of pS^5 to enable our PEs to compete with the CPU cores for processing the accelerable kernel, while retaining the complex high-level control flow and the sorting of the smaller data sets on the CPU. We accelerate the whole algorithm by up to 10% compared to the 28 thread software baseline running on the 14-core Xeon processor and by up to 36% at lower thread counts.**

I. Introduction

Sorting is among the most studied [12] and ubiquitous [9], [4], [5] problems in computer science. For the simplest and yet common case of sorting fixed-length keys such as integers, a number of implementations has been proposed on CPUs [8], GPUs [10], and FPGAs [13], [14]. FPGAs typically provide the best performance per watt [14]; however, the maximum dataset size is often bound by the on-chip memory available on the FPGA [13], which a few tens of megabytes at most and cannot be expanded. On the other hand, there are important classes of applications such as suffix sorting algorithms, MapReduce tools, and database index construction [3] which require to sort strings lexicographically. While parallel string sorting algorithms have been proposed on CPUs [2] and GPUs [15], there has been much less work compared to sorting of fixed-length keys [3] and, to the best of our knowledge, no FPGA accelerator for this problem has been proposed yet. Indeed, string sorting is especially challenging because keys can be long (which makes comparisons expensive) and of variable length (which makes keys hard to handle, especially in hardware).

Heterogeneous CPU-FPGA SoCs for embedded systems have been on the market for a few years already. Previous work [20], [7] has shown that, on those platforms, it is possible to take advantage of the best of the two worlds: massive parallelism and energy efficiency of FPGAs, flexibility and high performance on serial task execution of CPUs. High-performance heterogeneous systems with server-class instead of embedded CPUs are now being deployed in datacenters [11], [16]. In this context, the competition for performance is much fiercer than it has ever been in CPU-FPGA SoC platforms because here processors run at clock speeds up to an order of magnitude faster than FPGAs and have easily a dozen physical cores. It is natural then to wonder how much one has a chance to speedup significantly highly studied and optimized applications and, if so, at which price in terms of silicon real-estate and energy consumption. The results of the Catapult project [16] are well known and very encouraging, but are the result of hundreds of man years of work by arguably one of the most skilled group of engineers in the field. What can be expected with a few man months of averagely skilled hardware designers?

In this paper, we present a hybrid CPU-FPGA system for parallel string sorting implemented on the Intel's HARPv2 experimental platform. We have chosen to accelerate the open-source state-of-the-art parallel algorithm for string sorting on multi-core CPUs, Parallel Super Scalar String Sample Sort (pS^5) [3]. We believe that pS^5 is algorithmically representative of modern CPU-optimized algorithms because of its irregular structure (compared to the straightforwardness of the problem), its nontrivial codebase (about 4,000 lines of C++ code), and its opportunistic mix of several sort algorithms to obtain the best for each dataset and at each point of the sorting process. It is also representative of the quality of code that highly-skilled performance-aware software programmers can produce—e.g., exploiting in every possible way the actual dimensions and properties of processor caches. Our effort targeted what is arguably the single most time expensive and effectively parallelizable kernel—which, despite its relative simplicity, took considerable development and testing time to port on the FPGA. We succeeded on having a processing element on the FPGA compute its job faster than a CPU core, inclusive of all data transfers and even including the fact that some necessary data preparation remains in software. We integrated the kernel into the job scheduling mechanism of the original software (originally meant to ship jobs to CPU cores and now shipping them to both CPUs and our processing elements) to achieve what we think is the first FPGA-based system for string sorting

2160-052X/20/$31.00 ©2020 IEEE
DOI 10.1109/ASAP49362.2020.00031

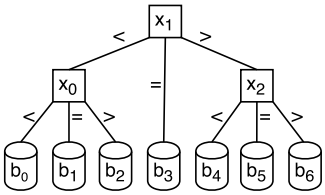


Fig. 1: Example of ternary tree with depth d = 2 used during S^5 string classification. Based on the result of the comparison with at most d *splitters*, each input string is classified into one of $2^d - 1$ *buckets* (adapted from Bingmann et al. [3]).

acceleration and one of the few FPGA sorters whose maximum dataset size is not bounded by the FPGA on-chip memory.

II. Preliminaries

A. Terminology

We adopt the same terminology defined by Bingmann et al. for the original pS^5 algorithm [3]. A string sorting algorithm classifies a set $\mathcal{S} = \{s_1, \ldots, s_n\}$ of n strings with N characters in total. A string s is an array of $|s|$ characters from the alphabet $\Sigma = \{1, \ldots, \sigma\}$. Given two strings s_1 and s_2, lcp(s_1, s_2) denotes their *longest common prefix* (LCP), that is, the length of the longest sequence of initial characters that is shared by s_1 and s_2 (e.g., lcp('abacus', 'aboriginal') = 2 as they share the prefix 'ab'). D represents the *distinguishing prefix size* of $\mathcal{S}$, that is, the minimum number of characters that have to be inspected in order to determine the lexicographic ordering of $\mathcal{S}$. For example, D = 3 for $\mathcal{S}$ = 'aboriginal', 'article', 'abacus' as sorting requires inspecting at least 'abo', 'ar' and 'aba'. Sorting algorithms based on single character comparisons have a minimum runtime complexity $\Omega(D+n \log n)$ [3]. This bound can be lowered by using *super-alphabets*, i.e. by grouping w characters which are compared and sorted at once.

B. Super Scalar String Sample Sort (S^5)

S^5 is a string sorting algorithm based on sample sort. Sample sort is a generalized quicksort with k - 1 pivots (*splitters*) $x_1 \leq \cdots \leq x_{k-1}$ which classifies strings into k *buckets* $b_1 \leq \cdots \leq b_k$. Splitters are chosen by randomly sampling $\alpha k - 1$ strings from the input, sorting them, and then taking every α-th element, where α is the *oversampling* factor. The output sorted set is obtained by concatenating the sorted buckets.

S^5 uses a super-alphabet with $w = 8$ characters to exploit word parallelism on 64 bit CPUs. In the first classification, the *common prefix* of the whole set is initialized to $l = 0$ and the algorithm considers the first w characters of both the strings and the splitters. When recursively sorting each bucket b_i, the starting index of the w characters to be compared (l) is incremented by lcp(x_{i-1}, x_i) (i.e., the LCP of the splitters delimiting the bucket), which is a lower bound on the LCP of each couple of strings in the bucket. This minimizes the total number of character comparisons by effectively reusing as much of the information gained in the upstream classification as possible.

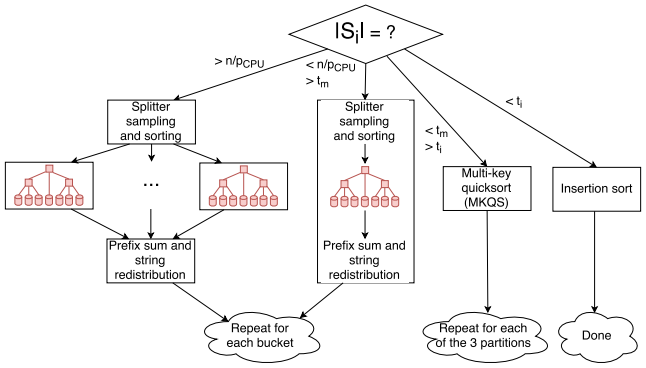


Fig. 2: Sorting sub-algorithm selection in pS^5 depending on the size of the string subset $\mathcal{S}_i$. Each box represents a *job* which can be processed by a different thread.

Splitters are arranged in a search tree and classification is done by descending the tree. *Equality buckets* are defined for strings whose next w characters are the same. At each node, an equality check between the w characters of interest of splitter and string is added. By definition, all strings in an equality bucket share the first w characters which therefore can be skipped altogether when the equality bucket is recursively sorted. Therefore, $v = 2^d - 1$ splitters are arranged in a ternary search tree and define $k = 2v+1$ buckets as shown in Figure 1. String sample sort with implicit ternary tree and super-alphabet has runtime complexity O($\frac{D}{w} \log v + n \log n$)) [3].

The output of the classification kernel is a *bucket counting vector* with k elements containing the number of strings in each bucket and an *oracle* vector with one element per input string, whose element o_i contain the index of the bucket of string s_i. After computing a prefix sum of the bucket counting vector, the string pointers are redistributed in the respective bucket. v is chosen to ensure that both the splitter tree and the bucket counting array fit in the L2 cache of each processor. For a 256 KB cache, this results in $v = 8191$, corresponding to a tree with $d = 13$ levels.

C. Parallel S^5 (pS^5)

pS^5 spawns one thread for each of the p_{CPU} CPU logical cores and invokes four different sub-algorithms depending on the size of the string (sub)set to be sorted $\mathcal{S}_i$, where $\mathcal{S}_i$ initially corresponds to entire string set $\mathcal{S}$ and, as sorting progresses, to the buckets that are recursively sorted. The algorithm selection criteria are summarized in Figure 2. For the largest sets with $|\mathcal{S}_i| > \frac{n}{p_{CPU}}$, a fully parallel version of S^5 described below is used, for $t_m < |\mathcal{S}_i| \leq \frac{n}{p_{CPU}}$ the sequential S^5 described in Section II-B is invoked, for $t_i < |\mathcal{S}_i| \leq t_m$ a parallel version of caching multikey quicksort (MKQS) [17] is run, and insertion sort is called when $|\mathcal{S}_i| \leq t_i$, where $t_m = 2^{20} = 1$ Mi and $t_i = 64$ have been determined empirically by Bingmann et al.

The fully parallel version of S^5 consists of four stages: sampling the splitters to generate the ternary tree, classification, global prefix sum and string redistribution. Classification is the only parallel stage, where strings in $\mathcal{S}_i$ are split evenly among

$p\prime = \left\lceil 2 \times \frac{|\mathrm{S}_i|}{\max\left(t_m, \frac{n}{p_{CPU}}\right)} \right\rceil$ jobs Global prefix sum starts as soon as all classification jobs terminate; therefore, the total execution time of the fully parallel S^5 is determined by the classification job that is completed last. For each instance of the other sub-algorithms, as well as for the serial stages of the fully parallel S^5, a single job is created.

For dynamic load balancing, pS^5 uses a central job queue polled by all threads. In this way, multiple jobs ready for processing can be handled in parallel by different threads. All the sequential jobs use an explicit recursion stack and implement voluntary work sharing: as soon as another thread is idle, an atomic global flag is set, which causes other threads to release the bottom of their stacks (with the largest subproblems) as independent jobs.

D. Memory System Considerations

The string set is represented as an array of pointers to the first character of each string. Because of this indirection, scanning the input dataset is much less cache efficient than in atomic sorting. During pS^5 initialization, the string characters are written to memory contiguously in the same order as they appear in the file. Initially, the pointers to the beginning of each string are arranged in memory in the same order as the string characters. As strings are being sorted, only the pointers are moved in memory and thus the order of the strings and characters arrays will differ. As a result, the performance of the memory system will be higher during the first sorting step compared to all the following: initially, reads are still sequential and may hit on the same cache line of the previous string, or at least take advantage of hardware prefetching. As string pointers start to be rearranged in memory, reads become completely random and have a high chance of cache miss on datasets that do not fit in the cache.

III. Design Methodology

We decided to accelerate the classification steps of both the parallel and sequential steps of S^5 because, as discussed in Section IV-B, it is one of the two dominant kernels of the whole sorting algorithm. Moreover, classification is massively parallel in itself, as each string can be classified independently. Finally, sample sort classification can be seen as a generalization of the three-way partitioning step of MKQS. As a result, it is easier for a classification accelerator to be extended to also handle MKQS partitioning in the future rather than the opposite. Our accelerator contains a number of processing elements (PEs), each capable of handling the entire classification step of a single S^5 job.

A. Hardware Platform

HARPv2 is a shared memory heterogeneous system, consisting of a 14-core Intel Xeon Broadwell CPU and an Intel 10AX115N4F45I3SG Arria 10 FPGA. The FPGA logic is divided in an FPGA Interface Unit (FIU) provided by Intel and an Accelerated Functional Unit (AFU) designed by us. The FIU implements platform capabilities such as the interface logic for the links between CPU and FPGA and exposes a Core Cache Interface (CCI-P) and a Memory Mapped I/O (MMIO) interface to the AFU. The MMIO interface is used by the CPU to initiate read or write transfers to the AFU registers, whereas the AFU reads and writes 64 byte cache lines from/to the system memory through CCI-P.

The AFU sees a three-level memory hierarchy: a 64 KB first level cache inside the FPGA itself and managed by the FIU, the 35 MB processor's last level cache (LLC), and the 64 GB system memory. To access the system memory, the AFU can use virtual addresses, provided that the buffers that will be shared with the AFU are allocated using a special allocator.

B. Parallel Hybrid S^5 (pHS^5)

To fully exploit the additional parallelism available for the parallel S^5 steps of our Parallel Hybrid S^5 (pHS^5), we replaced p_{CPU} with $p_{CPU} + p_{AFU}$ when computing the threshold for the parallel S^5 steps and $p\prime$ (see Section II-C), where p_{AFU} is the number of PEs in the AFU.

pS^5 implements voluntary work sharing to achieve workload balancing among CPU cores. Ideally, one would enable the additional processing elements (PEs) on the FPGA to directly push and pop jobs from the same shared job queue. However, the current version of CCI-P does not support the atomic memory operations that are necessary to use the lock-free job queue. Moreover, the additional cores that the AFU introduces can only process a kernel present in two kinds of jobs: classification jobs from fully parallel S^5 steps and sequential S^5 steps. Lastly, although the AFU can access the same system memory using the same virtual addresses as the software, buffers that must be shared with the AFU must be allocated with a special allocator. In the case of pS^5, these would include the entire dataset array, the string pointers and a large number of temporary buffers. Changing the way all these buffers are allocated would require major modifications to the pS^5 code which we could not perform in the limited time frame we set.

Given all the constraints above, we resorted to a secondary work sharing mechanism inside the two accelerable S^5 jobs. An array of *AFU workspaces* is allocated as a CPU/AFU shared memory buffer. Whenever a CPU thread reaches the classification kernel, it checks *AFU job count*, an atomic global variable that counts the number of jobs currently attributed to the AFU. If the AFU job count is less than a given maximum value, the job is attributed to the AFU and the CPU (1) copies the 8 characters of interest of all job's splitters and strings to one of the AFU workspaces and (2) sends a *job descriptor* to the AFU via MMIO. The job descriptor contains the pointers to the input splitter and string arrays in the AFU workspace, as well as to the output oracle buffer. The availability of each AFU workspace is managed by an array of atomic boolean flags; sending the job descriptor via MMIO, which requires some tens of microseconds, is the only operation that requires the acquisition of a spinlock.

After sending the job descriptor, the CPU thread enqueues a new *polling job* to the central job queue. In the polling job, the CPU will poll a done bit in the respective AFU workspace: if the job is done, the CPU reads back the oracle array and

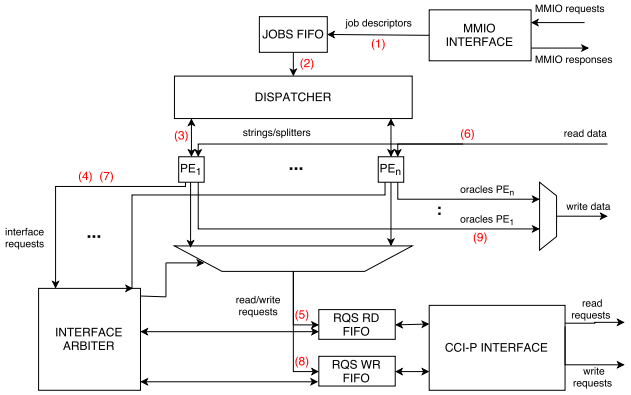


Fig. 3: System-level architecture of our AFU. Jobs are received through the MMIO interface (1, 2) and dispatched to an idle PE (3). PEs request splitters and input strings (4, 5, 6), classifies them and return the oracles via CCI-P (7, 8, 9). Multiple PEs, each processing a separate classification job, are needed to fully utilize the CCI-P I/O bandwidth.

proceeds with the prefix sum and string permutation as in the standard pS^5; if not, it enqueues a new polling job. By using a separate polling job, the CPU thread can process other jobs in the queue, if any, while the AFU is busy. We empirically determined $2 \times p_{AFU}$ to be a good value for the maximum number of jobs that can be assigned to the AFU.

C. AFU Design

The top level architecture of our AFU is shown in Figure 3. The CPU uses MMIO to enqueue jobs to the *jobs FIFO* (1), where they are consumed by the *dispatcher* (2) whenever a PE is available. Once a PE receives a job (3), it accesses the *CCI-P interface* via the *arbiter* (4). The arbiter uses a simple round-robin policy to provide fair I/O access to each requesting PE. Once the PE has been granted access to the interface, it sends CCI-P read requests to the *read request FIFO* for the job splitters and strings (5), which will be forwarded via the CCI-P read channel (6). Classification starts as soon as the first 64 B cache line is received. When the oracle buffers inside the PEs are full, the PE requests the interface again (7) and sends out the oracles via the *write request FIFO* (8) and the CCI-P write channel (9).

While PEs could, in principle, handle a larger super-alphabet than the CPU as its word-level parallelism is not locked to 64 bit, doing so would make S^5 jobs for the two platforms not compatible with each other. This would require our dynamic scheduling described in Section III-B to be replaced by some form of early job scheduling at job creation time. Even if we cannot increase parallelism at *character* level, we increased parallelism at *string* level by classifying multiple strings concurrently. As shown in Figure 4, each PE contains 8 classification *cores*, one per 8 B string in a 64 B cache line. Splitters are replicated in four dual port on-chip memories, each serving two cores in parallel.

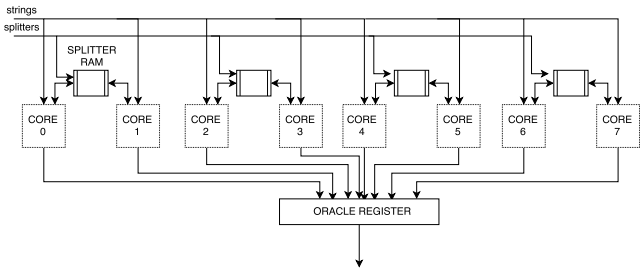


Fig. 4: Internal structure of one of the PEs shown in Figure 3. A PE contains eight *cores*, each classifying one of the eight strings contained in a 64 B cache line. Splitters are replicated into four dual port on-chip RAMs each shared by two cores.

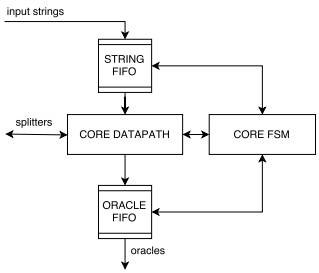


Fig. 5: Internal structure of one of the cores shown in Figure 4. Strings are collected in the input FIFO, classified and oracles are returned. To utilize all functional unit during 100% of the cycles, four strings are classified in an interleaved fashion.

Figure 5 shows the internal structure of a core. Between input and output FIFOs, a successive approximation register (SAR) is used to descend the classification tree stored in the splitter memory. Descending one level requires four cycles (the on-chip memory has a 2-cycle latency, plus one for the comparator and one for the SAR); therefore, four strings are classified in an interleaved fashion in order to fully utilize all the units. The resulting classification has a latency of 15 clock cycles at 200 MHz *per string per core*: $v = 13$ cycles for the tree descent plus 2 cycles to fill and flush the pipeline. A PE with 8 cores classifies 8 strings simultaneously and thus has a latency of 15 cycles *per cache line* with 8 strings, or 1.875 cycles per string.

Given that the CCI-P interface can supply up to one cache line per clock cycle, we instantiate multiple PEs to match computation to I/O throughput. Any time a PE requests the CCI-P interface, the lock is granted to read the splitter set, 8,192 input strings, or to write 8,192 oracles before being given to the next requesting PE.

IV. Experimental Results

All tests have been performed on the HARP system described in Section III-A. The CPU has 28 logical cores and the system is equipped with 64 GB of RAM. We compiled our software with gcc 6.3.1 with `-O2 -march=broadwell`. We evaluate our pHS^5 on three of the benchmarks that have been used by Bingmann et al. in the original pS^5 paper [3] which we take as representative of datasets with qualitatively different statistics:

	URLs	Wikipedia	Random
n	161M	131M	617M
$\frac{D}{N}$	96.3%	29.8%	58.8%
avg. string length	66.9	81.9	17.4
parallel S^5 steps	12	1	1
sequential S^5 steps	69	21	0

TABLE I: Properties of the datasets used to evaluate our system.

	ALMs	M20Ks
CCI-P interface	793 (<1%)	0
FIFOs	119 (<1%)	38 (1%)
6 PEs	22,133 (5%)	1,164 (43%)
Total AFU	24,109 (6%)	1,202 (44%)
HARP infrastructure (FIU)	78,900 (18%)	351 (13%)
Total	103,009 (24%)	1,553 (57%)

TABLE II: Resource utilization of our FPGA design. No DSPs were used. The AFU breakdown only contains the largest modules and thus the total AFU resource utilization is larger than the sum of that of the listed modules. In parenthesis: percentage of total available resources.

- **URLs** contains a list of URLs crawled breadth-first from the pS^5 authors' institutional web page. This set has the largest $\frac{D}{N}$ as all keys start with either `http://` or `https://` followed, in many cases, by a small set of labels such as `www`, `en` or `de`. For a given N, this dataset is close to the worst case for a string sorting algorithm as almost all characters must be checked in order to establish the order of the strings.
- **Wikipedia** is the XML dump of all pages of the English Wikipedia as of June 1st, 2012. Except for about 25% of the strings which consist or start with XML tags and are very similar to each other, all the other strings are lines of text and have a more uniform distribution.
- **Random** is a list of randomly generated numbers of 16 to 19 digits. Both the digits and the length are uniformly distributed, which results in an uniform distribution of keys and thus of bucket sizes.

We consider the first 10 GB of each dataset; we noticed that the trends are much more dependent on the dataset statistics than on the dataset size, at least above a few gigabytes. Table I summarizes the main properties of the datasets, together with the number of parallel and sequential S^5 invocations that are called by pS^5 at 28 threads.

A. Resource Utilization

Table II shows the resource utilization of the entire FPGA design and a breakdown of the largest modules of our AFU. Each PE consumes less than 1% of the ALMs and 7.1% of the M20K memory blocks of the Arria 10 FPGA. All other AFU modules have negligible resource utilization. The main bottleneck for increasing the number of PEs is not the resource utilization per se but rather timing closure. With more than 6 PEs in the design, we could not achieve timing closure with a 200 MHz clock; falling back to the 100 MHz clock and duplicating the number of PEs was not an option as 12 PEs would not have fit in the AFU LogicLock region which has only 70% of the total M20K blocks.

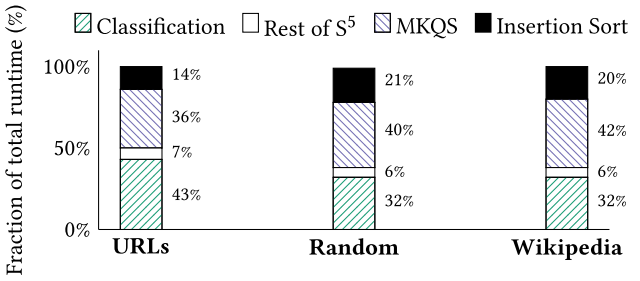


Fig. 6: Profiling of single thread pS^5 run on our benchmarks. We accelerate the classification part (dashed) of S^5 (white), which one of the two dominant kernels of the entire application.

B. Profiling of Single Thread pS^5

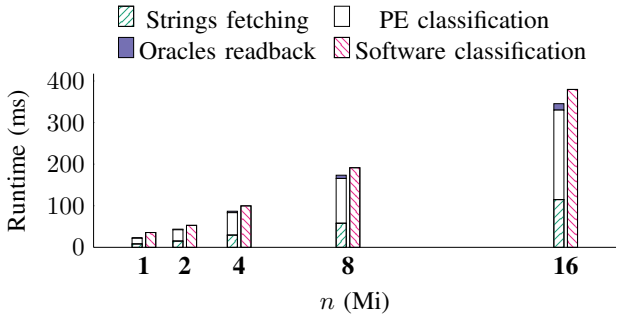
Figure 6 shows the results of profiling a single thread execution of pS^5 on our benchmarks. Classification (the step that we accelerate) is either the first or the second most dominant kernel of the entire application. We expect the runtime share of classification to increase even further for larger datasets as more and more string subsets become larger than t_m. Moreover, our S^5 classification can easily be extended in the future to handle MKQS where strings are essentially classified into three buckets by a single splitter, whereas extending an MKQS accelerator to handle classification would require more important adaptations.

C. Performance Evaluation

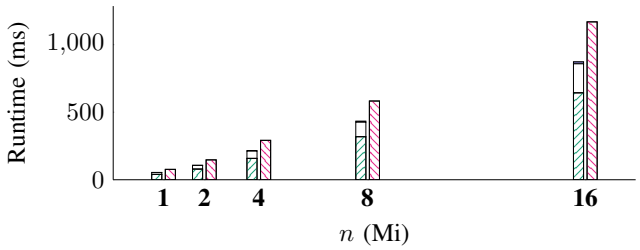
Kernel acceleration. To confirm the interest to accelerate classification, Figure 7 compares the runtime of the classification kernel on a CPU core and on one of our PEs. We distinguish the case of sequential reads (Figure 7a) in the first sorting step from that of random reads (Figure 7b) which applies to all subsequent sorting steps. We only consider datasets bigger than $t_m = 1$ Mi as smaller jobs are handled by other sorting algorithms.

For large n, one PE is 10% and 33% faster than a Xeon CPU core in the case of sequential and random reads respectively. String fetching from main memory is $5.6\times$ slower on random than on sequential reads. This makes string fetching the dominant step of all accelerated S^5 jobs that are not part of the first parallel S^5 step. When the classification is done in software, the three stages (input reading, actual classification, and output writing) are finely intermixed with each other and their runtime cannot be measured separately. The overall performance hit on random reads is nevertheless clearly visible on the software classification runtime ($3\times$ for n = 16 Mi).

As expected, PE execution time and oracle readback take the same time irrespective of the sparsity of the input strings. For large n, the PE classification throughput tends to 2.56 AFU clock cycles per string. If 1.88 cycles are expected to be for the actual classification (see Section III-C), we can estimate that 0.54 of the remaining 0.68 cycles are for reading the 8 input



(a) Sequential reads: string pointers and characters have the same order in memory. Fetching the string characters involves non-contiguous but sequential reads.



(b) Random reads: string pointers and characters do not have the same order. Fetching the string characters involve random reads.

Fig. 7: Runtime of classification when executed on the CPU and on a PE. Strings have been extracted from the first 2 GB of URLs, which contain 33 million strings. Writing the splitters takes a negligible time in all cases (10 µs). For $n \geq 16$ Mi, a PE is 10% and 33% faster than a Xeon core on sequential and random reads respectively.

characters and 0.14 to write the oracle from/to the CPU/AFU shared memory (8 and 2 bytes respectively, both accessed sequentially and contiguously), assuming that read and write bandwidth between PE and shared memory are equal and if we neglect the partial overlaps between I/O and classification.

pS^5 Acceleration. To isolate the contribution of each of our modification to the pS^5 code to the overall performance, we compare four different scenarios:

1) pS^5: original pS^5
2) pS^5-add_jobs: the same as pS^5 where p_{CPU} has been replaced by $p_{CPU} + p_{AFU}$ (see Section III-B), which results in parallel S^5 invocations with more, smaller classification jobs.
3) pHS^5-block_sequential: the same as pS^5-add_jobs where jobs are dispatched to the PEs whenever possible but the software thread waits in a polling loop after offloading the smaller sequential S^5 jobs and a separate polling job is enqueued only after offloading a classification job from a parallel S^5 step.
4) pHS^5-no_block: pHS^5 as described in Section III-B. Compared to pHS^5-block_sequential, a separate polling job is created in every case.

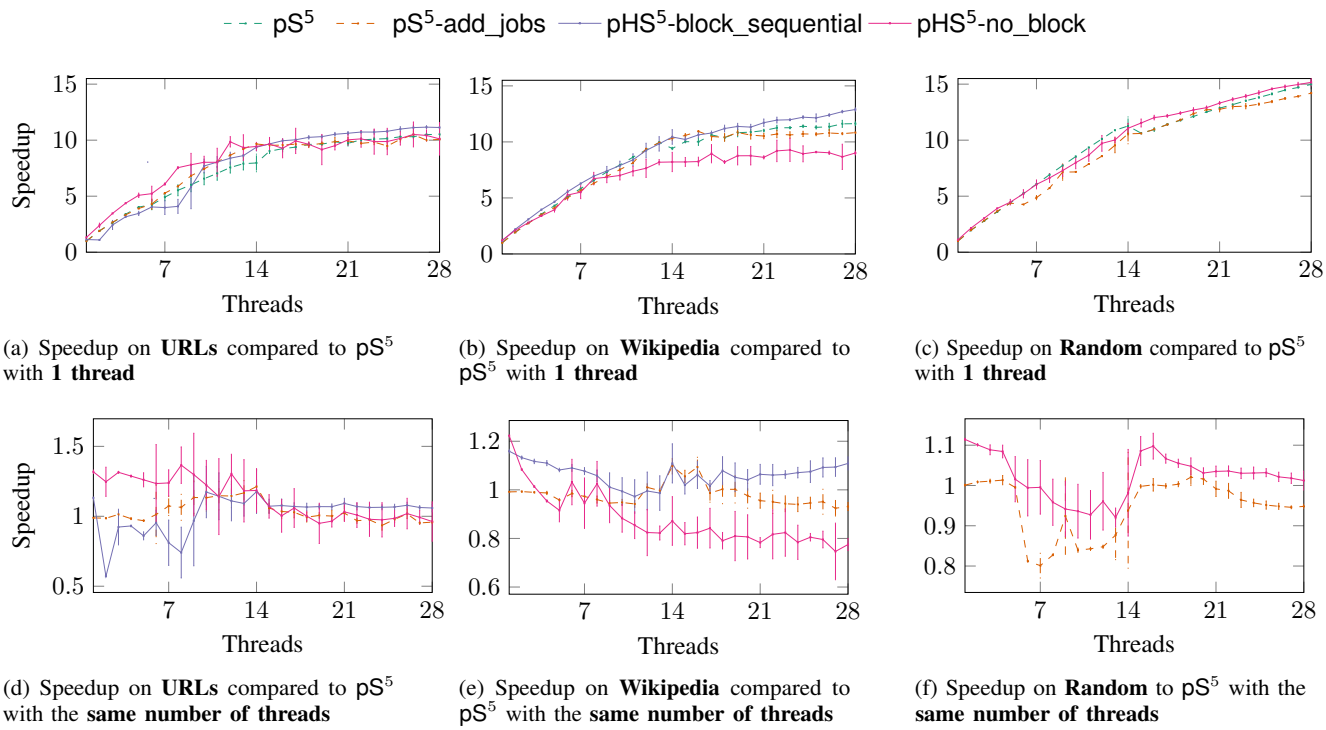
Figure 8a-8c show the speedup of whole algorithm compared to a single thread execution of pS^5 and Figure 8d-8f compared to pS^5 with the same number of threads.

The results vary greatly depending on the input dataset and on the number of CPU threads. In URLs, strings are very similar to each other and, in the first iterations, most strings are classified in a few large buckets. Indeed, parallel and sequential sample sort are invoked 81 times overall, and the fraction of accelerable code is the highest of all benchmarks. At low thread counts, pHS^5-no_block on URLs provides the highest acceleration compared to pS^5 that we measured, peaking at 36% at 8 threads. Between 5 and 15 threads, part of the benefit is due to splitting parallel S^5 invocations in more classification jobs, and having additional resources in the FPGA to handle them with limited overhead on the CPU cores gives further advantage. On more than 15 threads, pHS^5-block_sequential becomes the best performing algorithm, providing a 6-8% acceleration compared to the baseline.

On the Wikipedia dataset, pHS^5-block_sequential always outperforms pHS^5-no_block except at one thread and has the highest acceleration at 28 threads (10%). With more than 8 threads, pHS^5-no_block is actually slower than pS^5, by 20-25% at high thread counts. Using more classification jobs does not provide the same clear benefit as in URLs and is even counter productive at high thread counts. As for the random dataset, there is no distinction between the two pHS^5 versions as there are no sequential S^5 invocations: the only accelerable jobs are those of the single parallel S^5 invocation at the beginning of the algorithm. On this dataset, the AFU provides acceleration on either low thread counts, or when the thread count exceeds the number of physical CPU cores.

At low thread counts, the relative increase of parallelism provided by the 6 PEs is larger than at high thread counts. In the case of URLs, increasing the number of classification jobs seems to be beneficial in itself, perhaps due to the superlinear runtime complexity of a single string sample sort (see Section II-B). This effect may not appear on the other benchmarks given the smaller number of parallel S^5 invocations, and any gains may be offset by having a number of jobs that is not any more divisible by the number of threads. This results in an increase of runtime of the slowest thread due to load unbalance, which causes a slowdown of the overall parallel S^5 step. The AFU provides instead additional resources to handle those jobs with limited overhead on the CPU cores.

Overall, we expected pHS^5-no_block to always outperform pHS^5-block_sequential as the former provides a better use of parallelism by enabling the thread that transferred the job data to the AFU to process other jobs while the AFU is busy. This actually holds when the number of threads is low and blocking one of them in polling results in a significant reduction of available computing resources. However, as the number of threads placing jobs in the queue increases, the polling job might end up being processed a significant time after the completion of the classification job by the PE. This results in (1) the AFU job queue slot being occupied for longer than necessary, wasting AFU resources and (2) delaying the creation, and thus the completion, of the 16,381 jobs to sort the



(a) Speedup on **URLs** compared to pS^5 with **1 thread**

(b) Speedup on **Wikipedia** compared to pS^5 with **1 thread**

(c) Speedup on **Random** compared to pS^5 with **1 thread**

(d) Speedup on **URLs** compared to pS^5 with the **same number of threads**

(e) Speedup on **Wikipedia** compared to pS^5 with the **same number of threads**

(f) Speedup on **Random** to pS^5 with the **same number of threads**

Fig. 8: Sorting speedup compared to pS^5 with one thread (8a, 8b, 8c) and to pS^5 with the same number of threads (8d, 8e, 8f, note the different vertical scales). The data points are the average of 5 runs; error bars show their standard deviation. Depending on the dataset and on the number of threads, either pHS^5-no_block or pHS^5-block_sequential accelerate pS^5 by up to 36%.

sequential job's buckets. These effects have a smaller impact on URLs than on Wikipedia because the number of sequential S^5 jobs in URLs is much greater than p_{AFU}, which ensures that there are always enough sequential S^5 jobs processed by CPU threads whose recursive subjobs are generated as soon as the classification finishes.

V. Related Work

A. Parallel String Sorting and Sorting on FPGAs

Besides pS^5 [3], which is a state-of-the-art string sorting algorithm for multi-core shared memory machines, Bingmann et al. also analyzed string sorting parallelization on NUMA machines [2]. For those architectures, they proposed to run independent pS^5 sorters on each NUMA node and then merging the results with a multiway mergesort that uses the LCP information from pS^5 to skip over common characters during merging. On both shared memory and NUMA machines, they observed that the statistical properties of the dataset have a large impact on the effectiveness of parallelization. This is consistent with fixed-length key sorting and with our findings on a heterogeneous shared memory CPU-FPGA machine.

On GPUs, Neelima et al. [15] proposed a parallel MKQS that uses dynamic parallelism to recursively sort the partitions as they are created, which result in an exponentially increasing amount of GPU threads. Deshpande et al. [10] adapted the radix sort for fixed-length keys provided as a part of the CUDA Thrust library to handle variable-length keys by iteratively extracting a fixed number of characters and treating them as integers, which we see as a form of super-alphabet. Both works use datasets not larger than a few hundreds of megabytes.

To the best of our knowledge, no other string sorting implementations on FPGA or on heterogeneous CPU-FPGA systems have been proposed. Sorting of fixed-length keys on FPGA has been extensively studied; however, none of the solutions we found can be easily extended to handle variable-length keys as they rely on storing entire keys on the FPGA on-chip memory and comparing them at once. Koch et al. [13] proposed a FIFO-based merge sorter followed by a tree-based merge sorter to maximize the maximum dataset size that an FPGA can sort. Although partial runtime reconfiguration can increase the maximum dataset size, it is still bounded by a function of the total FPGA on-chip memory. Matai et al. [14] proposed a framework that generates sorting architectures. The work focuses on design automation and on simplifying design space exploration; some of the generated architectures have been evaluated on datasets that are at most on the order of hundreds of thousands of keys.

B. Heterogeneous CPU-FPGA Platforms

Zhang et al. [20] presented a CPU-FPGA sorter for HARPv1. The dataset is split into blocks sorted by a merge sorter on the FPGA; blocks are eventually merged by the CPU. The first blocks can be merged by the CPU while the FPGA is sorting the next ones. Using the CPU to merge the blocks makes the maximum dataset size independent from the FPGA on-chip memory; however, the on-chip memory still limits the block

size, which makes the runtime dominated by the CPU merge for datasets larger than 256 MiB. A similar idea of pre-sorting on the FPGA and merging with the CPU has also been evaluated by Chen et al. [7] on a Zynq CPU-FPGA platform, obtaining a similar conclusion that the CPU becomes the bottleneck for problems that are large compared to the amount of FPGA on-chip memory. In addition, the CPU merge algorithms that have been proposed are sequential and heavily underutilize modern multi-core CPUs. Our pHS^5 is built on top of a state-of-the-art multithreaded sorter that fully exploits the parallelism of modern CPUs and extends it with additional specialized processing cores on the FPGA. Moreover, the maximum dataset size that can be processed by one of our PEs is only limited by the system memory and not by the FPGA on-chip memory.

Umuroglu et al. [18] proposed a hybrid breadth-first search (BFS) implementation on a Zynq CPU-FPGA system. Weisz et al. [19] analyzed pointer chasing on three CPU-FPGA systems including HARPv1. On single-linked lists with data payload accessible through a pointer, the best results are achieved when the CPU performs the indirections to visit the list nodes and streams the payload pointers to the FPGA for processing. Both works suggest that the highest performance on a CPU-FPGA system are achieved when the CPU is used on irregular and serial computations and the FPGA on massively parallel processing involving large amounts of data. These results inspired the high-level CPU-FPGA partitioning of our pHS^5, where we offload the most parallel kernel operating on the largest data subsets to the FPGA while the CPU keeps handling recursion and sorting of small datasets.

Chang et al. [6] presented an FPGA accelerator on HARPv1 for the SMEM seeding algorithm of DNA sequencing alignment. SMEM involves a large amount of short, random reads and its bottleneck resides in memory latency rather than computation. The authors propose a many-PE architecture that issues as many outstanding reads as possible to hide the long latency of memory accesses. This is the work that is the most similar to ours in that they also accelerate one of the dominant kernels of a complex, multithreaded software on a cache-coherent CPU-FPGA system. However, the FPGA can only service one CPU thread at a time, which the authors cite as a possible limiting factor for the acceleration of the entire algorithm. The AFU of our pHS^5 can instead accelerate the work coming from as many CPU threads as there are PEs. Moreover, the main purpose of using PEs on the FPGA in Chang et al. is to have as many in-flight random reads as possible rather than accelerating the computation itself as in pHS^5. Their results suggest that we could expect further speedups if we were to also delegate the random reads, i.e. the string indirections, to the FPGA.

VI. Conclusion

We presented pHS^5, to our knowledge the first hardware-accelerated string sorter, which has been implemented on the Intel HARPv2 CPU-FPGA heterogeneous system. Our pHS^5 extends pS^5, a string sorting software that has been extensively optimized for multi-core CPUs. One of our processing elements accelerates one of the dominant kernels in pS^5 by up to 33% compared to a single Xeon core, and 6 PEs accelerate the entire application by up to 10% compared to pS^5 running in its fully parallel version on a 14-core Xeon CPU. We believe there is potential for future FPGAs that promise clock frequencies closer to those of CPUs, such as Intel Agilex and Xilinx Versal, to improve upon the modest overall speedup that we could achieve with today's FPGA technology.

References

[1] T. Bingmann. Engineering parallel string sorting for multi-core systems.

[2] T. Bingmann, A. Eberle, and P. Sanders. Engineering parallel string sorting. *Algorithmica*, 77(1):235–286, 2017.

[3] T. Bingmann and P. Sanders. Parallel string sample sort. In *European Symposium on Algorithms*, pages 169–180, 2013.

[4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[5] J. Casper and K. Olukotun. Hardware acceleration of database operations. In *Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 151–160, 2014.

[6] M.-C. F. Chang, Y.-T. Chen, J. Cong, P.-T. Huang, C.-L. Kuo, and C. H. Yu. The SMEM Seeding Acceleration for DNA Sequence Alignment. In *Proceedings of the 24th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 32–39. IEEE, 2016.

[7] R. Chen and V. K. Prasanna. Accelerating Equi-Join on a CPU-FPGA Heterogeneous Platform. In *Proceedings of the 24th IEEE Symposium on Field-Programmable Custom Computing Machines*.

[8] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] A. Deshpande and P. Narayanan. Can GPUs sort strings efficiently? In *Proceedings of the 20th International Conference on High Performance Computing (HiPC)*, pages 305–313, 2013.

[11] P. K. Gupta. Accelerating Datacenter Workloads, 2016. Keynote presented at FPL 2016.

[12] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. 1998.

[13] D. Koch and J. Torresen. FPGASort: a High Performance Sorting Architecture Exploiting Run-Time Reconfiguration on FPGAs for Large Problem Sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 45–54, 2011.

[14] J. Matai, D. Richmond, D. Lee, Z. Blair, Q. Wu, A. Abazari, and R. Kastner. Resolve: Generation of High-Performance Sorting Architectures from High-Level Synthesis. In *Proceedings of the 24th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 195–204, 2016.

[15] B. Neelima, B. Shamsundar, A. Narayan, R. Prabhu, and C. Gomes. Kepler GPU Accelerated Recursive Sorting Using Dynamic Parallelism. *Concurrency and Computation: Practice and Experience*, 29(4), 2017.

[16] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceedings of the 24th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2014.

[17] T. Rantala. A collection of string sorting algorithm implementations.

[18] Y. Umuroglu, D. Morrison, and M. Jahre. Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform. In *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications*, pages 1–8, 2015.

[19] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe. A Study of Pointer-Chasing Performance on Shared-Memory Processor-FPGA Systems. In *Proceedings of the 24th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 264–273, 2016.

[20] C. Zhang, R. Chen, and V. Prasanna. High throughput large scale sorting on a CPU-FPGA heterogeneous platform. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 148–155, 2016.