Journal of Signal Processing Systems
https://doi.org/10.1007/s11265-021-01686-8

How Many CPU Cores is an FPGA Worth? Lessons Learned from Accelerating String Sorting on a CPU-FPGA System

Mikhail Asiatici[1] · **Damian Maiorano**[2] · **Paolo Ienne**[1]

Received: 15 October 2020 / Revised: 9 April 2021 / Accepted: 26 July 2021

Abstract

String sorting is a fundamental kernel of string matching and database index construction; yet, it has not been studied as extensively as fixed-length keys sorting. Because processing variable-length keys in hardware is challenging, it is no surprise that no hardware-accelerated string sorters 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 CPU. Our pHS^5 extends pS^5, the state-of-the-art string sorting algorithm for multi-core shared memory CPUs, by adding multiple processing elements (PEs) on the FPGA. Each PE accelerates one instance of the most effectively parallelizable among the dominant kernels of pS^5 by up to 33% compared to a single Intel Xeon Broadwell core despite a clock frequency that is 17 times slower. Furthermore, we extended the job scheduling mechanism of pS^5 to schedule the accelerable kernel not only among available CPU cores but also on our PEs, while retaining the complex high-level control flow and the sorting of the smaller data sets on the CPU. Overall, we accelerate the entire algorithm by up to 10% with respect to the 28-thread software baseline running on the Xeon processor and by up to 36% at lower thread counts. Finally, we generalize our results assuming pS^5 as representative of software that is heavily optimized for modern multi-core CPUs and investigate the performance and energy advantage that an FPGA in a datacenter setting can offer to regular RTL users compared to additional CPU cores.

Keywords String sorting · FPGA · HARP · Heterogeneous computing

1 Introduction

Sorting is one of the most studied problems in computer science [1] and a fundamental building block of countless algorithms and applications [2–5]. Especially in its simplest and yet most common form of fixed-length key (e.g., integer) sorting, a plethora of highly optimized parallel implementations have been proposed on multiple compute platform: CPUs [6], GPUs [7, 8], and FPGAs [9–11]. While FPGAs generally provide the best performance per watt [10], the maximum dataset size is often bound by the amount of on-chip memory [9, 12], which is limited to a few tens of megabytes and cannot be expanded. In addition, many sorting algorithms are based on recursion, which cannot be directly implemented in hardware.

Compared to fixed-length sorting, many fewer solutions have been proposed for sorting variable-length strings lexicographically. This is a building block of suffix sorting, used in string matching, and database index construction [13]. Parallel string sorting algorithms have been proposed on CPUs [14] and GPUs [15], however, to the best of our knowledge, no hardware accelerator for this problem has been made available yet. Indeed, handling variable-length keys in hardware is not only challenging per se but also involves key comparisons that can become expensive as keys are arbitrarily long.

Previous work [16–19] has shown that heterogeneous CPU-FPGA platforms can benefit from the best of both worlds: flexibility and high performance on serial tasks execution from CPUs paired with energy efficiency and massive parallelism from FPGAs. Heterogeneous platforms,

✉ Mikhail Asiatici
mikhail.asiatici@epfl.ch

Damian Maiorano
damian.maiorano.93@gmail.com

Paolo Ienne
paolo.ienne@epfl.ch

[1] Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, 1015 Lausanne, Switzerland

[2] Politecnico di Torino, 10129 Torino, Italy

Published online: 23 September 2021

originally targeting embedded systems, are now colonizing datacenters as well [20, 21]. In this context, the competition for performance is much stiffer than it has ever been in the embedded world because here processors have easily a dozen physical cores that run at clock speeds up to an order of magnitude faster than FPGAs and exploit all of the best microarchitectural techniques known (instruction-level parallelism, speculation, simultaneous multithreading, etc.). Therefore, a high-performance software developer may wonder to what extent FPGAs can contribute to significantly accelerate 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 [21] are well known and very encouraging, but are the result of years of work by arguably one of the most skilled group of engineers in the field. What can be expected from a few months of averagely skilled hardware designers' work?

In this paper, we present a hybrid CPU-FPGA parallel string sorter implemented on the Intel's HARPv2 experimental platform. Our implementation is based on the open-source state-of-the-art parallel algorithm for string sorting on multi-core CPUs, Parallel Super Scalar String Sample Sort (pS^5) [13]. Besides the interest of this solution for the specific problem of string sorting, we believe that pS^5 is representative of modern CPU-optimized algorithms because of 1) its large codebase (about 4,000 lines of C++ code), 2) its irregular structure (compared to the straightforwardness of the problem), and 3) its opportunistic blend of multiple sort algorithms to obtain the best performance in the largest possible set of scenarios. Moreover, pS^5 is also representative of the code quality that experienced software programmers can produce—e.g., exploiting in every possible way the actual dimensions and properties of processor caches. We accelerated what is arguably the single most effectively parallelizable and time-consuming kernel—which, despite its relative conceptual 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, even when taking into account all data transfers and despite the fact that some necessary data preparation remains in software. We integrated the hardware kernel into the job scheduling mechanism of the original software (originally meant to dispatch jobs to CPU cores and now dispatching them to both CPUs and our processing elements) to build what we think is the first FPGA-based system for parallel string sorting whose maximum dataset size is not bounded by the FPGA on-chip memory. We study the speedup that our FPGA accelerator achieves versus the speedup that every processor core brings, and use this to understand the potentials of FPGAs in datacentres for regular users who do have the competence to design hardware in RTL but can afford only a few months of effort to compete with highly optimized software.

2 Preliminaries

After introducing some terminology that we will use throughout the paper, we introduce S^5 and pS^5, upon which we built our hybrid pHS^5. We finally present some details about the memory layout of the strings in the input dataset that have important consequences on the performance, which will be discussed in Section 4.

2.1 Terminology

In the reminder of the paper, we generally adopt the same terminology defined by Bingmann et al. for the original pS^5 algorithm [13]. 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, $D = 8$ for $\mathcal{S}$ = 'aboriginal', 'article', 'abacus' as sorting requires inspecting at least 8 characters: 'abo', 'ar', and 'aba'. Sorting algorithms based on single character comparisons have a minimum compute complexity $\Omega(D + n \log n)$ [13]. String sorting can be accelerated by using *super-alphabets*, i.e. by grouping w characters which are compared and sorted at once.

2.2 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. By doing so, the statistical distribution of the splitter set will be an approximation of that of the input set, resulting in bucket sizes more uniform than those produced by radix or bucket sort [22]. The final sorted set is obtained by concatenating the sorted buckets.

To exploit the word parallelism of the CPU, S^5 uses a super-alphabet with $w = 8$ characters, which fit into a 64-bit machine word. 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.

Splitters are arranged in a search tree and classification is done by descending the tree. Using a binary search tree to perform the classification would not reveal much information if many strings share the same next w characters. To efficiently handle these cases, *equality buckets* are defined for strings whose next w characters are the same. This is implemented by adding an equality check between the w characters of interest of splitter and string at each node. When recursively sorting an equality bucket, the common prefix l can be increased by w: indeed, the w characters that have been found to be equal in all strings can be skipped altogether in the subsequent steps. Therefore, $v = 2^d - 1$ splitters are arranged in a ternary search tree and define $k = 2v + 1$ buckets as shown in Fig. 1. Such string sample sort with ternary tree and super-alphabet has runtime complexity $O(\frac{D}{w} \log v + n \log n))$ [13].

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 contains 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. The number of splitters v is chosen to ensure that both the splitter tree (of size $w \times v$ bytes) and the bucket counting array (of size $8 \times k = 16v + 8$ bytes) 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.

2.3 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 Fig. 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 2.2 is invoked, for $t_i < |\mathcal{S}_i| \leq t_m$ a parallel version of caching multikey quicksort (MKQS) [23] 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.[1]

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 $p\prime = \left\lceil 2 \times \frac{|\mathcal{S}_i|}{\max\left(t_m, \frac{n}{p_{CPU}}\right)} \right\rceil$ jobs[2]. 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.

2.4 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. Indeed, even though the first 8 characters of each string are not placed contiguously in memory (unless the string length is ≤ 8), reads during the first sorting step will be monotonous and may hit on the same cache line of the previous string,

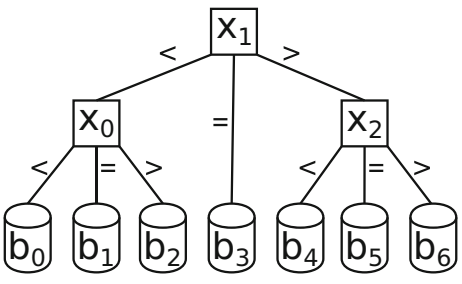


Figure 1 Example of ternary tree with depth $d = 2$ used during S^5 string classification. Based on the result of the comparison with at most d *splitters*, each input string is classified into one of $2^d - 1$ *buckets* (adapted from Bingmann et al. [13]).

[1] The values for t_m and $p\prime$ in the paper [13] and in the released source code [24] are not consistent. We used the values in the source code as they provided higher performance in our tests.

[2] See Footnote 1

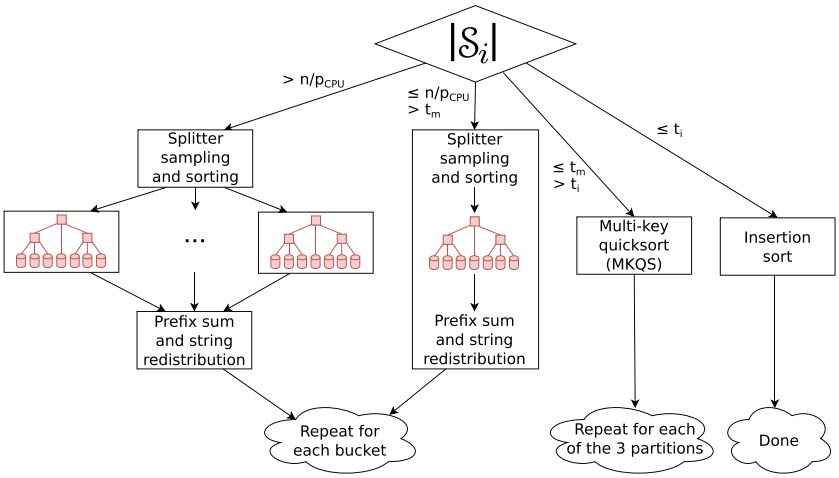


Figure 2 Sorting sub-algorithm selection in pS^5 depending on the size of the string subset $\mathcal{S}_i$. Each box represents a *job* which can be processed by a different thread.

or at least take advantage of hardware prefetching. As string pointers start to be rearranged in memory, reads during all the subsequent sorting steps become more and more irregular and have a high chance of cache miss, unless the dataset is small enough to fit in the cache.

3 Design Methodology

We decided to accelerate the classification steps of both the parallel and sequential steps of S^5 because, as discussed in Section 4.2, 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.

3.1 Hardware Platform

HARPv2 (Heterogeneous Architecture Research Program, version 2) 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, together with three clocks (100, 200 and 400 MHz) for the AFU logic. 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. Communication between the FPGA and the CPU's LLC is ensured by two PCI-Express and one QPI physical channels. 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.

3.2 Parallel Hybrid S^5 (pHS^5)

To fully exploit the additional parallelism available for the parallel S^5 steps of our hardware-accelerated 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 2.3), 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 PEs 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.

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 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.

3.3 AFU Design

The top level architecture of our AFU is shown in Fig. 3. Each processing element handles one classification job, which consists of a set of strings to be classified with a single splitter tree and corresponds to the workload that, in pS^5, is processed by a CPU thread. The CPU uses MMIO to enqueue jobs to the *jobs FIFO* (1), where they are consumed by the *dispatcher* whenever a PE is available. Once a PE receives a job (2), it requests access to the *CCI-P interface* via the *arbiter* (3). 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 (4), it sends CCI-P read requests to the *read address FIFO* for the job splitters and strings (5), which will be forwarded via the CCI-P read data channel (6). Classification starts as soon as the first 64 byte cache line is received. When the oracle buffers inside the PEs are full, the PE requests the interface again (7, 8) and sends out the oracles via the *write address FIFO* (9) and the CCI-P write data channel (10).

Although the PE 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 at job execution time described in Section 3.2 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 Fig. 4, each PE contains 8 classification *sub-PEs*, one per 8 B string in a 64 B cache line. Splitters are replicated in four dual port on-chip memories, each serving two sub-PEs in parallel.

Figure 5 shows the internal structure of a sub-PE. Between input and output FIFOs, a successive

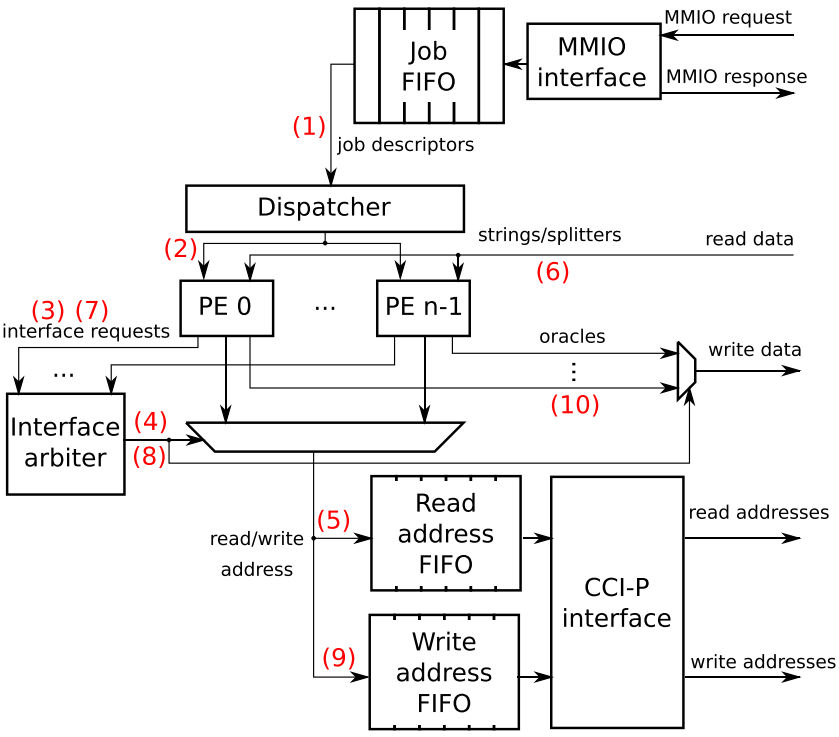


Figure 3 System-level architecture of our AFU. Jobs are received through the MMIO interface (1) and dispatched to an idle PE (2). PEs request splitters and input strings (3, 4, 5, 6), classifies them and return the oracles via CCI-P (7, 8, 9, 10). Multiple PEs, each processing a separate classification job, are needed to fully utilize the CCI-P I/O bandwidth.

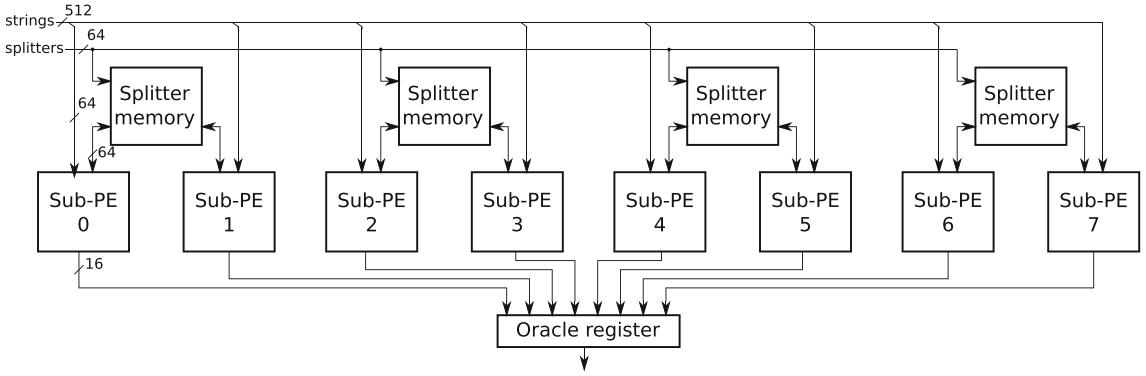


Figure 4 Architecture of one of the PEs shown in Fig. 3. A PE contains eight *sub-PEs*, each classifying one of the eight strings contained in a 64 B cache line. Splitters are replicated into four dual port on-chip RAMs each shared by two sub-PEs.

approximation register (SAR) is used to descend the classification tree stored in the splitter memory. Descending one level requires four cycles as the on-chip memory has a 2-cycle latency and the comparator and the SAR have one each. 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

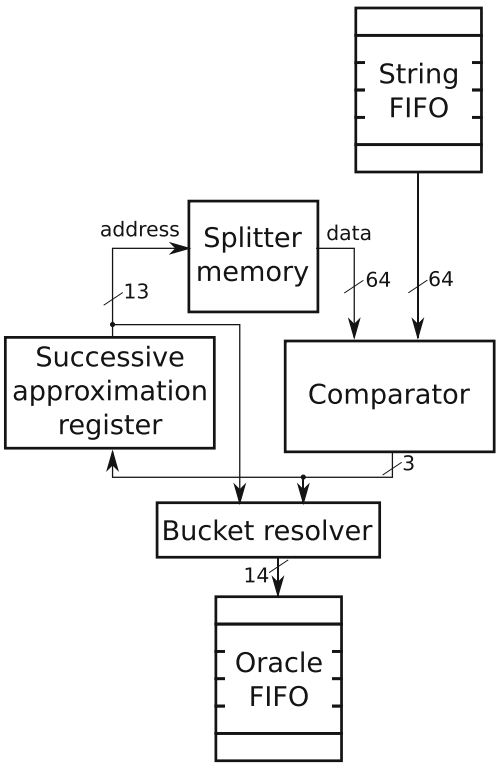


Figure 5 Architecture of one of the sub-PEs shown in Fig. 4. Starting from the root of the splitter tree, the sub-PE traverses the tree by iteratively comparing strings to splitters. Based on the result of each comparison, the successive approximation register updates the address of the next splitter to fetch. Once the classification is completed, the bucket resolver computes the oracle index based on the result of the last comparison. To always fully utilize all functional units, we classify four strings in an interleaved fashion.

per string per sub-PE: $v = 13$ cycles for the tree descent plus 2 cycles to fill and flush the pipeline. A PE with 8 sub-PEs 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.

4 Experimental Results

All tests have been performed on the HARP system described in Section 3.1. 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 [13] which we take as representative of datasets with qualitatively different statistics:

- **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. While about 25% of the strings which consist or start with XML tags are very similar to each other, the remaining 75% of strings are lines of natural text and have a more uniform distribution.

Table 1 Properties of the datasets used to evaluate our system.

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

– **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 1 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.

4.1 Resource Utilization

Table 2 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.

Table 2 Resource utilization of our FPGA design.

	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%)

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. Note that the AFU is constrained within a LogicLock region which contains 78% of ALMs and 70% of M20K blocks. The maximum number of PEs is not limited by the resource utilization but by timing closure at 200 MHz

4.2 Profiling of Single Thread pS^5

Figure 6 shows the results of profiling a single thread execution of pS^5 on our benchmarks. Classification, which is the step 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.

4.3 Performance Evaluation

Kernel acceleration 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 (Fig. 7a) in the first sorting step from that of random reads (Fig. 7b) which applies to all subsequent sorting steps for the reasons discussed in Section 2.4. 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× 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.

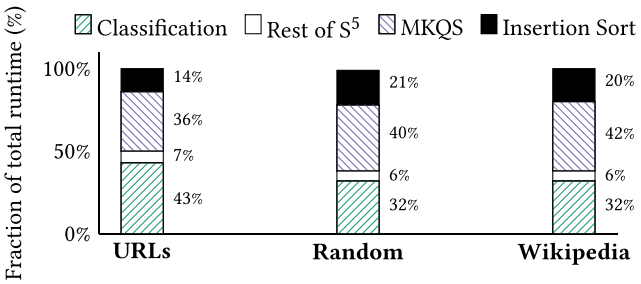
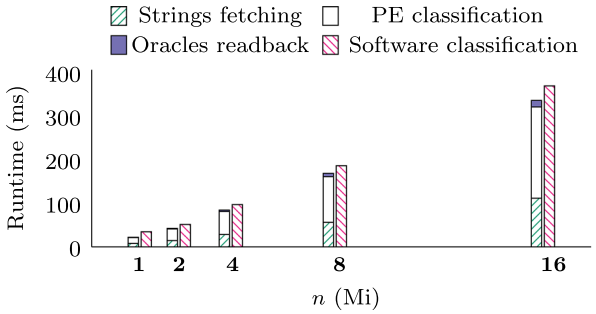
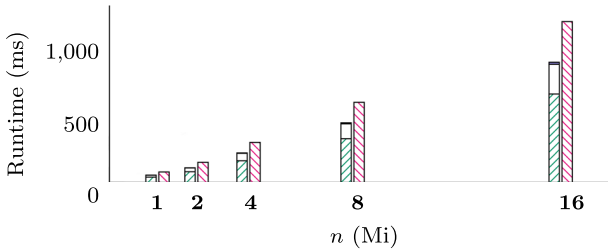


Figure 6 Profiling of single thread pS^5 run on our benchmarks. We accelerate the classification part (dashed green) of S^5 (dashed green and white), which is one of the two dominant kernels of the entire application.



(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.

Figure 7 Classification runtime when executed on the CPU and on a PE. Strings are extracted from the beginning of the URLs benchmarks. Writing the splitters takes a negligible time in all cases (10μ s). For $n \geq 16$ Mi, a PE is 10% and 33% faster than a Xeon core on sequential and random reads respectively.

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× 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 3.3), we can estimate that 0.54 of the remaining 0.68 cycles are for reading the 8 input 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. With the major pS^5 software modifications required to allocate the whole input dataset and the oracle buffers on the CPU/AFU shared memory, and if the string pointer indirection was done by the PE, we could estimate the ideal hardware-accelerated S^5 runtime to be the current one without the oracle readback time and 0.54 AFU clock cycles per string. This would make our PE 33% and 43% faster than a Xeon core for sequential and random reads respectively, besides increasing parallelism at a system level by making the CPU available during the whole accelerated S^5 job. We plan to implement this improvement as a part of our future work.

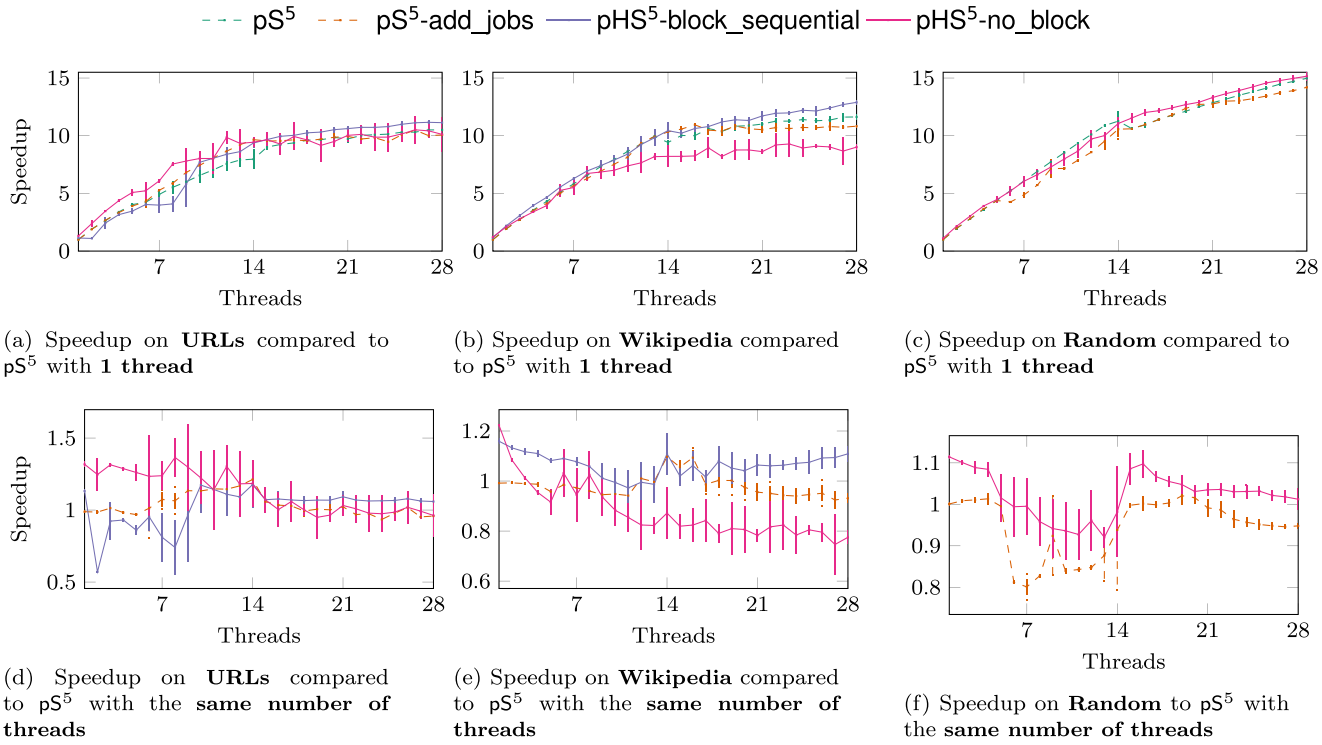
pS^5 Acceleration To isolate the contribution of each of our modification to the pS^5 code to the overall performance, we compare four different scenarios:

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 3.2), 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 to the central job queue only after offloading a classification job from a parallel S^5 step.
4. pHS^5-no_block: pHS^5 as described in Section 3.2. Compared to pHS^5-block_sequential, a separate polling job is created in every case.

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

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. Increasing the number of 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



(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**

Figure 8 Sorting speedup compared to pS^5 with one thread (**a, b, c**) and to pS^5 with the same number of threads (**d, e, f**, 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%.

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 2.2). 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 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.

4.4 Cost/benefit of FPGA and CPU Cores

In Section 4.3, we showed that pS^5 can be accelerated by either increasing the number of CPU cores or by exploiting dedicated hardware on the FPGA. In this section, we discuss costs and benefits of the two approaches in terms of performance per silicon area and energy consumption.

We strived to estimate parameters that are as realistic as possible only based on publicly available information. Table 3 summarizes the results of our analysis. As the exact CPU model name of the HARPv2 machine was not available, we used the Intel Xeon E5-2680 v4 which seems to be the most similar in terms of number of cores, base

Table 3 Parameters used for our cost/benefit analysis.

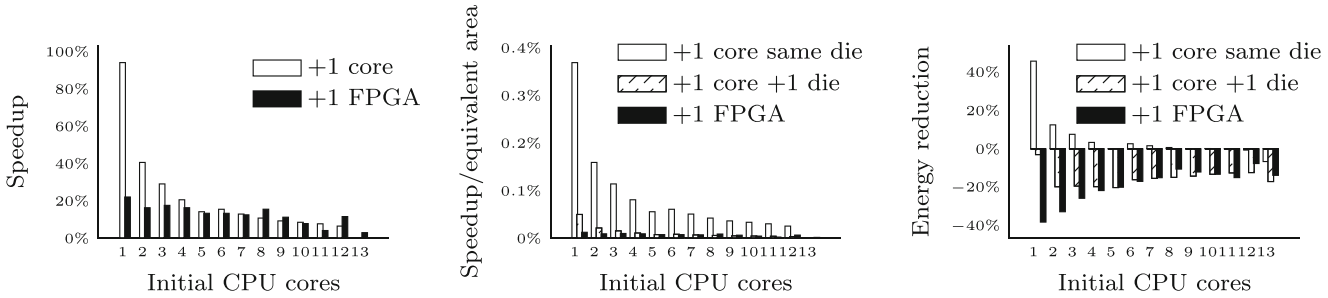
	CPU (fixed/per core)	FPGA
Silicon area (mm^2)	306 (96/15) [25]	144 [25, 26]
Equivalent area	5,200 (1,600/250) [27]	1,800 [27]
Power (W)	120 (15/7.5) [28]	21.8

frequency, microarchitecture and caches size. As we did not have the permissions to read the actual CPU power consumption during the execution of pS^5, we used the TDP to estimate the power consumption when all cores are used. We compared the die size and TDP of the 14-core Xeon E5-2680 v4 to those of the 10-core Xeon E5-2640 v4 belonging to the same processor family to estimate area and power fixed and per core.

As for the FPGA, we estimated its total area by comparison to that of the CPU die as shown in a press article about an upcoming Intel CPU-FPGA product [26]. To make a fair comparison between chips manufactured with different processes, we normalize the physical area to the area of one million 6T SRAM cells in the respective process (*unit of equivalent area*), which is 5,880 μm^2 and 8,100 μm^2 for the Intel 14 nm and the TSMC 20 nm process respectively. The actual FPGA power on the HARPv2 system during the execution of our pHS^5 was measured to be 21.8 W, of which 13.5 W are due to the transceivers and is constant irrespective of whether the AFU was running or not, and 8.3 W are attributed to the core rail that powers the reprogrammable logic.

From the data in Fig. 8, we computed the relative speedup when adding one CPU thread comparing to that of adding an FPGA with our 6 PEs. When computing the FPGA gain, we considered the best between the pHS^5-block_sequential and pHS^5-no_block at each thread count, assuming an ideal scheduler that can each time select the best policy. We took the average across the three benchmarks. The results are shown in Fig. 9a. Adding a CPU core provides the highest speedup up to four cores; on higher core counts, the advantage of a CPU core is comparable to that of the FPGA. Once the results are normalized by the equivalent area (Fig. 9b), adding a CPU core on the same die is clearly the best solution, providing up to an order of magnitude higher speedup per equivalent area than the FPGA. The gap is even larger when looking at the energy cost of sorting the dataset (Fig. 9c): the speedup provided by the FPGA is never enough to compensate for the additional power consumption it involves, whereas an additional CPU core can in some instances provide positive energy savings. The FPGA indeed provides the same speedup of an additional CPU core but at 7 times the area and 3 times the power cost.

However, this is not really a fair comparison because, in adding a core, we assumed that the overall fixed infrastructure for a CPU is already there whereas, to add our FPGA accelerator, the complete FPGA infrastructure needs to be added; this accounts for example for the fixed power consumed by the transceivers which, based on our measurements, accounts for 60% of the total FPGA power during the execution of pHS^5. Therefore, it seems useful to contrast the addition of the FPGA to the addition of a core which cannot fit in the same die, or in general to replace a CPU die with n cores with two dies with $n+1$ cores overall. In this case, another CPU core or an FPGA have similar area and energy cost, at least when the first CPU already has five or more cores. From another perspective, if the FPGA was on the same chip and could share the fixed CPU costs just like an additional CPU core, it could also perform similarly in terms of area and energy efficiency. Moreover, considering that in pS^5 the fraction of parallelizable runtime for the CPU is close to 100% (any core can independently process any kinds of available job) and only 30% to 40% for the FPGA, the FPGA could potentially outperform the CPU



(a) Speedup (b) Speedup per equivalent unit area (one million 6T SRAM cells) (c) Energy saving

Figure 9 Impact of adding an additional CPU core or an FPGA (average of the three benchmarks). Higher is better on all charts. As the FPGA is on a separate die, we distinguish the case of an additional CPU core that fits on the same die or that requires introducing an additional die. In terms of pure performance, with five or more cores, adding the FPGA provides the same speedup as an additional CPU core. The additional CPU core provides better performance per area and energy savings but only as long as the core fits in the same die and it is not necessary to parallelize across multiple dies or even sockets.

on applications where the FPGA could process kernels that take a larger fraction of total runtime. Finally, the mix of FPGA resources that our application needs is biased towards on-chip memory and only 24% of the available ALMs are used. If the blend of resources provided by the FPGA better matches that of the application—for example, if the FPGA also offered high-density on-chip memories such as eSRAM or UltraRAM—we can expect the FPGA to provide better performance per unit area, although it is an open question whether there is a blend that would be more universally beneficial for a broad class of accelerators.

5 Related Work

5.1 Parallel String Sorting and Sorting on FPGAs

Besides pS^5 [13], 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 [14]. 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 and that there is no single best string sorting algorithm for all use cases: for example, parallel multikey quicksort (pMKQS) and parallel radix sort outperformed pS^5 on very uniform datasets such as Random. 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. They obtain a 6× to 18× speedup with an NVIDIA Tesla K40C compared to Bingmann's pMKQS running on a desktop machine. Deshpande et al. [8] 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. They obtained one order of magnitude of speedup with an nVidia GTX 580 compared to a dual core Core 2 Duo CPU. 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. [9] 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 and multiple channels to external memory can further increase the maximum dataset size, it is still bounded by a function of the total FPGA on-chip memory. Chen et al. [29] proposed an architecture based on sample sort to efficiently sort datasets that do not fit in the FPGA on-chip memory using a heterogeneous CPU-FPGA system; however, it can only handle fixed-length keys. At the other end of the spectrum, Jun et al. [30] proposed an FPGA accelerator to sort terabyte-sized datasets of fixed-length keys limited by the flash storage bandwidth. Both Matai et al. [10] and Chen et al. [31] proposed frameworks that generate sorting architectures optimized according to different metrics (area, speed, power). The works focus on design automation and on simplifying design space exploration under complex constraints; some of the generated architectures have been evaluated on datasets that are at most on the order of hundreds of thousands of keys.

5.2 Heterogeneous CPU-FPGA Platforms

Zhang et al. [18] presented a hybrid CPU-FPGA sorter for HARPv1. The dataset is split into blocks that are sorted by a merge sorter on the FPGA; blocks are eventually merged by the CPU. Computation on the two platforms can partially overlap as the first blocks can be merged by the CPU while the FPGA is sorting the next ones. They obtained 1.9× and a 2.9× speedups compared to FPGA- and CPU-only implementations. Using the CPU to merge the blocks effectively 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. [19] on a Zynq CPU-FPGA platform, obtaining a similar conclusions that the CPU becomes the bottleneck for problems that are large compared to the amount of FPGA on-chip memory. One possible reason is that the CPU merge algorithms that have been proposed are sequential, which heavily underutilizes modern multi-core CPUs, especially those used in datacenters. 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. [16] proposed a hybrid breadth-first search (BFS) implementation on a Zynq CPU-FPGA system. By offloading the BFS steps with the largest frontier to the FPGA and processing the remaining ones on the CPU, they obtained 7.8× and 2× speedups compared to CPU- and FPGA-only implementations. Weisz et al. [17] 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. [32] 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. Sixteen PEs achieve a 4× speedup on the SMEM kernel compared to a single CPU core; on the largest CPU thread count (12 threads), they achieve a 26% speedup on the SMEM kernel and a 8% speedup on the entire algorithm. 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: we use atomic global variables on the CPU side to ensure that the CPU does not send more jobs than the FPGA can handle and we rely on a dispatcher on the FPGA to allocate the workload to the 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.

6 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 shared memory 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.

As FPGAs reach the datacenters, customers with RTL experience but with a limited design time budget may wonder whether it is more convenient to offload part of the computations to the tightly integrated FPGA or to rent more CPU cores. If we consider pS^5 as an example of a real world high-performance parallel application involving a mix of different algorithms, our analysis suggests that an FPGA on a different chip essentially performs like an additional CPU core on a separate chip both in terms of performance per unit area and energy consumption. Arguably, the conclusion could be different on applications whose runtime has a larger fraction of FPGA-accelerable kernels (30% to 40% in pS^5), if the CPU offloads to the FPGA not only parallel computation but also random memory accesses, if the cost of the FPGA is lowered by sharing the same CPU fixed infrastructure as an additional CPU core on the same die would do, or if more resources can be allocated to further optimize the hardware. At present, and when developers code in RTL and have moderate time to develop accelerators, FPGAs are not yet a clear winner when competing with complex highly optimized parallel software running on high-performance CPUs.

Funding Open Access funding provided by EPFL Lausanne.

References

1. Knuth, D. E. (1998). *The art of computer programming: sorting and searching* Vol. 3. London: Pearson Education.
2. Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107–113.
3. Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., & Manocha, D. (2009). Fast BVH construction on GPUs. In *Computer Graphics Forum*, (Vol. 28 pp. 375–384). Wiley Online Library.
4. Burrows, M., & Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm: Systems Research Center.
5. Casper, J., & Olukotun, K. (2014). Hardware acceleration of database operations. In *Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (pp. 151–160). ACM.
6. Chhugani, J., Nguyen, A. D., Lee, V. W., Macy, W., Hagog, M., Chen, Y.-K., Baransi, A., Kumar, S., & Dubey, P. (2008). Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, *1*(2), 1313–1324.
7. Satish, N., Harris, M., & Garland, M. (2009). Designing efficient sorting algorithms for manycore GPUs. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (pp. 1–10). IEEE.
8. Deshpande, A., & Narayanan, P. J. (2013). Can GPUs sort strings efficiently? In *20th International Conference on High Performance Computing (HiPC)* (pp. 305–313). IEEE.
9. Koch, D., & Torresen, J. (2011). 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* (pp. 45–54). ACM.
10. Matai, J., Richmond, D., Lee, D., Blair, Z., Wu, Q., Abazari, A., & Kastner, R. (2016). 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* (pp. 195–204). ACM.
11. Sklyarov, V., Skliarova, I., & Sudnitson, A. (2014). FPGA-based accelerators for parallel data sort. *Applied Computer Systems*, *16*(1), 53–63.
12. Chen, R., Siriyal, S., & Prasanna, V. (2015). Energy and memory efficient mapping of bitonic sorting on FPGA. In *Proceedings of the 23rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (pp. 240–249). ACM.
13. Bingmann, T., & Sanders, P. (2013). Parallel string sample sort. In *European Symposium on Algorithms* (pp. 169–180). Springer.
14. Bingmann, T., Eberle, A., & Sanders, Peter (2017). Engineering parallel string sorting. *Algorithmica*, *77*(1), 235–286.
15. Neelima, B., Shamsundar, B., Narayan, A., Prabhu, R., & Gomes, C. (2017). Kepler GPU accelerated recursive sorting using dynamic parallelism. *Concurrency and Computation: Practice and Experience, 29*(4).
16. Umuroglu, Y., Morrison, D., & Jahre, M. (2015). 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* (pp. 1–8). IEEE.
17. Weisz, G., Melber, J., Wang, Y., Fleming, K., Nurvitadhi, E., & Hoe, J. C. (2016). 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* (pp. 264–273). ACM.
18. Zhang, C., Chen, R., & Prasanna, V. (2016). High throughput large scale sorting on a CPU-FPGA heterogeneous platform. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 148–155).
19. Chen, R., & Prasanna, V. K. (2016). Accelerating equi-join on a CPU-FPGA heterogeneous platform. In *Proceedings of the 24th IEEE Symposium on Field-Programmable Custom Computing Machines* (pp. 212–219). IEEE.
20. Gupta, P. K. (2016). Accelerating Datacenter Workloads. Keynote presented at FPL 2016.
21. Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., & M., Haselman (2014). A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*.
22. Frazer, W. D., & McKellar, A. C. (1970). Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)*, *17*(3), 496–507.
23. Rantala, T. (2015). A collection of string sorting algorithm implementations. https://github.com/rantala/string-sorting.
24. Bingmann, T. (2014). Engineering parallel string sorting for multi-core systems. https://panthema.net/2013/parallel-string-sorting/.
25. De Gelas, J. (2016). The Intel Xeon E5 v4 review: Testing Broadwell-EP with demanding server workloads. https://www.anandtech.com/show/10158/the-intel-xeon-e5-v4-review.
26. Williams, C. (2016). Here's what an Intel Broadwell Xeon with a built-in FPGA looks like. https://www.theregister.co.uk/2016/03/14/intel_xeon_fpga/.
27. Jones, S. (2016). 10 nm SRAM projections - who will lead. https://www.semiwiki.com/forum/content/5620-10nm-sram-projections-who-will-lead.html.
28. Intel Corp. (2017). Product specifications. https://ark.intel.com/.
29. Chen, H., Madaminov, S., Ferdman, M., & Milder, P. (2020). FPGA-accelerated samplesort for large data sets. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (pp. 222–232).
30. Jun, S.-W., Xu, S., & et al. (2017). Terabyte sort on fpga-accelerated flash storage. In *Proceedings of the 25th IEEE Symposium on Field-Programmable Custom Computing Machines* (pp. 17–24). IEEE.
31. Chen, R., & Prasanna, V. K. (2017). Computer generation of high throughput and memory efficient sorting designs on FPGA. *IEEE Transactions on Parallel and Distributed Systems*, *28*(11), 3100–3113.
32. Chang, M.-C. F., Chen, Y.-T., Cong, J., Huang, P.-T., Kuo, C.-L., & Yu, C. H. (2016). The SMEM seeding acceleration for DNA sequence alignment. In *Proceedings of the 24th IEEE Symposium on Field-Programmable Custom Computing Machines* (pp. 32–39). IEEE.