IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 20, 2017, accepted August 22, 2017, date of publication September 11, 2017, date of current version October 12, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2750923

Acceleration by Inline Cache for Memory-Intensive Algorithms on FPGA via High-Level Synthesis

LIANG MA, (Student Member, IEEE), LUCIANO LAVAGNO, (Senior Member, IEEE), MIHAI TEODOR LAZARESCU, (Member, IEEE), AND ARSLAN ARIF, (Student Member, IEEE)

Department of Electronics and Telecommunications, Politecnico di Torino, I-10129 Torino, Italy

Corresponding author: Liang Ma (liang-ma@polito.it)

This work was supported in part by Xilinx Inc., and in part by the European Commission through the ECOSCALE Project under Grant H2020-ICT-671632.

ABSTRACT Using FPGA-based acceleration of high-performance computing (HPC) applications to reduce energy and power consumption is becoming an interesting option, thanks to the availability of high-level synthesis (HLS) tools that enable fast design cycles. However, obtaining good performance for memory-intensive algorithms, which often exchange large data arrays with external DRAM, still requires time-consuming optimization and good knowledge of hardware design. This article proposes a new design methodology, based on dedicated application- and data array-specific caches. These caches provide most of the benefits that can be achieved by coding optimized DMA-like transfer strategies by hand into the HPC application code, but require only limited manual tuning (basically the selection of architecture and size), are neutral to target HLS tool and technology (FPGA or ASIC), and do not require changes to application code. We show experimental results obtained on five common memory-intensive algorithms from very diverse domains, namely machine learning, data sorting, and computer vision. We test the cost and performance of our caches against both out-of-the-box code originally optimized for a GPU, and manually optimized implementations specifically targeted for FPGAs via HLS. The implementation using our caches achieved an 8X speedup and 2X energy reduction on average with respect to out-of-the-box models using only simple directive-based optimizations (e.g., pipelining). They also achieved comparable performance with much less design effort when compared with the versions that were manually optimized to achieve efficient memory transfers specifically for an FPGA.

INDEX TERMS Cache, high-level synthesis, acceleration, FPGA, optimization.

I. INTRODUCTION

High-Performance Computing and data-intensive applications, such as Machine Learning, Artificial Intelligence, and big data processing, are becoming more and more common both in large data centers and on embedded platforms. Thus, while the processing speed of, e.g., Neural Network training or database sorting, remains a primary concern, energy consumption is quickly gaining importance. In data centers, lower energy allows significant operational cost savings, while in embedded systems, such as Automated Driver Assistance Systems, lower energy implies lower cooling and manufacturing costs. These trends are witnessed by announcements recently made by companies such as Microsoft and Baidu, which use FPGAs for their search and machine learning tasks, or Amazon, which offers FPGAs on one of its AWS instance types. They are also underscored by a string of recent acquisitions performed by Intel, in particular of the second largest FPGA company, namely Altera.

This means that homogeneous hardware architectures, e.g., multi-core general purpose Xeon processors, no longer meet the heaviest computation requirements especially from the point of view of energy efficiency [1]. Thus, heterogeneous systems that cluster together different types of processors and hardware, such as CPU-GPU or CPU-FPGA, are able to achieve the best performance/cost/energy trade-offs for computationally-intensive parallel algorithms [2].

2169-3536 © 2017 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

FPGAs typically consume about an order of magnitude less power than GPUs or CPUs and provide a comparable raw computing performance, i.e., FPGAs consume an order of magnitude less energy per operation. Their adoption as data center accelerators is enabled by the recent availability of very user-friendly and extremely powerful synthesis tools, such as Stratus from Cadence, Catapult from Mentor and Vivado HLS (formerly Autopilot) from Xilinx, which significantly contribute to fulfill the FPGA promise of software-like flexibility and design ease with quasi-hardware performance and efficiency. These tools enable the designers to use high-level languages as the input to the FPGA design flow, such as C, C++ or OpenCL, which greatly eases the design and especially *verification*, and dramatically reduces the design cycle, even for FPGAs, with respect to traditional HDL-based flows.

While traditionally HLS had been regarded as a faster way to achieve worse designs than with the RTL-based flow, currently this view is changing among designers, due to several factors. First of all, the design tools are getting better and they ensure high quality results, generating automatically RTL with better or similar quality to manual design. Second, the designers are gaining experience with these tools and are thus able to steer them towards the best architectural implementations. Third, faster design cycles and significantly reduced verification times and costs imply that a much larger design space can be explored with respect to manual RTL design, which dramatically increase the chances to find the ''best architecture'' (or a broad Pareto set).

However, it is also clear that good code for a CPU or even a GPU may not be the best for an FPGA or ASIC implementation, as we will show in the result section of this article. The reason is that while HLS tools fully automate the *execution of some* micro-architectural decisions, e.g., whether a loop is pipelined or not, or whether a large C array is partitioned into several smaller on-chip memories, the choice among these options must still be made by a human designer. Moreover, tasks such as efficiently moving data from large off-chip DRAMs into on-chip memories must still be implemented by hand, by writing C or OpenCL code.

Modern FPGAs, such as the Stratix from Altera and the Virtex, UltraScale families from Xilinx, offer to the designer millions of Configurable Logic Blocks (CLBs) and Flip-Flops, megabytes of on-chip the Block RAM (BRAMs), hundreds of multiply-and-accumulate units (DSPs), and many other dedicated hardware blocks, including ARM Cortex processors [3]. Moreover, very recent design flows from both Altera/Intel and Xilinx promise software-like development for applications that are entirely written in a high-level language, like C, C++ or OpenCL, and are then compiled and synthesized for heterogeneous CPU-FPGA platforms. In particular, parallel languages that were originally developed to program GPUs, can now be used to program heterogeneous platforms such as PCs with FPGA boards, or Zynq platforms which include a multi-core CPU and a large FPGA [4].

However, the expected performance is typically not achieved by simply recompiling, via High-Level Synthesis for an FPGA target, an algorithm that was originally written for execution on a CPU or GPU. This is because the CPU or GPU architectures are fixed, hence most compiler decisions are local and relatively simple, such as intra-basic block scheduling or peephole optimizations. However, in an FPGA *the architecture is adapted to the application, rather than the application to the architecture*. While this can achieve much better optimization levels, it also implies that many more high-level decisions must be made during synthesis. HLS tools are able to automatically implement these decisions, but even their latest generations need to be directed to do so by a human or by a (very time-consuming) Design Space Exploration tool.

While the optimizations performed by a CPU or GPU compiler are considered excellent when they speed up execution by a factor of 2, the following HLS techniques can dramatically optimize the execution time of algorithms on FPGAs even by orders of magnitude. Most of them apply to loops, which are a major source of concurrency in high-level code and some languages, such as OpenCL, explicitly state that some loops can be arbitrarily parallelized, because iterations do not depend on each other:

1) *Loop pipelining* starts new iterations of a source code loop before the previous ones are completed. It is one of the best options for loop optimization in HLS, since it usually boosts the performance at a very low cost [5, p. 61]. The number of clock cycles between successive loop iteration starts (inverse of the throughput) is also called the ''Initiation Interval'' of the pipeline (in the best case, it can be one clock cycle). It is fully decoupled from the time it takes to complete one iteration, the pipeline ''latency''. Usually, memory or data dependencies between successive iterations (''loop-carried dependencies'') are the bottlenecks that increase the initiation interval. Several other synthesis techniques, e.g., array partitioning or loop interchange [6], can be applied to ameliorate this problem.
2) *Loop unrolling* creates multiple copies of the loop body to be executed fully in parallel. In some cases it can achieve even more performance than by means of pipelining, but typically at a huge resource (i.e., area) cost. A loop can be fully or partially unrolled and in both cases the maximum performance can be achieved only by means of array partitioning and may require arithmetic evaluation restructuring (e.g., adder tree balancing) [5, p. 51]. In OpenCL (similar to CUDA), the loop over work groups can be unrolled arbitrarily by definition. Thus, like on a GPU, the performance on an FPGA can be increased by instantiating multiple work groups until the computing or routing resources, or its memory bandwidth are saturated [4], [7]
3) *Exploiting on-chip memory*. Most modern FPGAs integrate thousands of independent BRAMs on chip for a

total of many MBs of storage. Accesses to these memories are both much faster in terms of latency and much more parallelizable than those to off-chip memories [8]. Many algorithms, especially the memory-intensive ones that are addressed in this article, achieve the best acceleration only by moving frequently-accessed data that reside in off-chip memories into on-chip BRAMs (or another kind of FPGA memory called LUTRAMs). As mentioned above, on-chip memories that are not carefully optimized by using partitioning directives can often become bottlenecks, because of the limited number of access ports that they offer. While on a GPU the maximum number of concurrent accesses to independent addresses (and the meaning of ''independent'') is fixed by the GPU architect, on an FPGA it must be carefully chosen by the designer, because more parallelism often implies a higher cost. Memory partitioning or memory reshaping according to user directives or to automated analysis of access patterns of a given algorithm can dramatically increase the memory bandwidth and achieve a much higher level of concurrency.

4) *Optimizing global memory interfaces*. Other methods to improve performance include instantiating multiple DRAM access ports or increasing their bit width.
On a GPU, the global memory interface subsystem receives memory read or write requests from the threads or work items that are executing on its compute units, and *coalesces* these requests whenever possible, in order to match both the available memory word size and bus burst transfer capabilities. For example, 16 accesses to adjacent properly aligned 32-bit integer array elements can be grouped *automatically at runtime* into a single 512-bit memory read, or to a burst of 4 128-bit memory reads, depending on the DRAM interface width.
On an FPGA, these groupings must be performed manually and at compile time, which requires a lot of design and tool usage expertise. Our caches simplify and automate all that.

A. MOTIVATION

As argued above, while HLS automates some low-level labor-intensive transformations from high-level code to RTL, many decisions must still be made by humans, and extensive code rewriting is sometimes needed in order to get the best performance with acceptable cost on an FPGA [7]. While automating design decisions is the domain of Design Space Exploration techniques [9], *this work focuses on totally avoiding, or, more precisely, hiding from the programmer, all the code rewritings that optimize the access to large arrays of data*. Our approach totally eliminates the significant *verification cost* of these changes, because caches are guaranteed to always deliver the right data. In the context of algorithms like those targeted by this research (which have regular access patterns), they can even ''guarantee'' good performance, where the guarantee is as good as the test cases which are used to select the cache parameters and to verify the performance post-synthesis.

The requirements that we want to satisfy in the scope of this research on accelerating memory-intensive algorithms are:

1) Enabling significant performance acceleration with respect to code that was not written specifically for FPGA, and sometimes not even very much optimized for a GPU.
2) Improving execution energy consumption by targeting an FPGA platform, by reducing off-chip memory accesses, and by reducing the execution time.
3) Supporting optimized use of external DRAM interfaces (e.g. DDR3 or DDR4) via advanced on-chip busses (e.g. AXI).
4) Enabling the use of HLS tools.
5) Keeping the standard HLS-based verification flow.
6) Requiring almost no changes to the original algorithms.
7) Not hampering the standard set of optimizations, architectural choices, etc. that are offered by the HLS tools.

B. CONTRIBUTION

Caches have been used for a long time in the domain of general-purpose CPUs. However, in that case a *single cache* is used for all the data that the processor accesses in the main memory (at most separate caches are used for code and data). This means that access conflicts between different variables (or sections of arrays) in the source code may limit the cache performance, unless sophisticated multi-way or even fully-associative architectures are used. Even in that case, the ''hot cache'' phenomenon [10] hampers several common algorithms.

In this work, we specialize caches for HLS in several directions:

- we advocate the use of *a separate cache for each source array that is mapped to DRAM*, to minimize the conflicts and to enable the efficient use of direct-mapped caches;
- we design our caches to appear as *inlined array access methods*, via the standard C++ `[]` operator, in order to require minimal source code changes;
- we support different kinds of caches, e.g., for read-only or write-only arrays, in order to best optimize their architecture;
- we automatically adapt each global memory array mapped via one of our caches to use wide memory interfaces and/or bus bursts in order to optimize transfer bandwidth with external DRAM.
- we enable verification of cache performance and correctness using the standard C++-based verification flow supported by modern HLS tools, in which a C++ testbench is used to verify the functionality of both the C++ code to be synthesized and the resulting RTL code;
- we enable the loop optimizations which can be made by the HLS tools only for the arrays mapped to on-chip BRAM, and not to off-chip DRAM.

Note that while in this work we use mostly FPGAs as our target, and hence we mention often FPGA-specific tools such as Vivado HLS, *our caches are fully generic and can be applied also to ASIC designs which access DRAM data*. We tested our designs with ASIC-oriented HLS tools, such as Mentor Catapult. Of course, in the ASIC case the cost and performance optimization requirements may be much more stringent. Yet, our caches can be used to quickly explore the memory access design space to find a good starting point for further manual implementation, converting the cache into a similarly architected "scratchpad".

Moreover, design automation support for static or simulation-based address sequence analysis to identify the best cache architecture for a given application is left to future work.

II. RELATED WORK

Modern CPUs generally include up to three levels of cache in order to reduce both data access time and energy. As the level increases, both latency and cache size (hence access power and energy) increase. These caches implement different access, replacement and coherency strategies to achieve the best *average performance* for all kinds of algorithms. Research on improving general-purpose caches is abundant. To cite just a few, Jouppi in [11] introduced an improvement to direct-mapped caches using a small fully-associative cache, the so-called victim cache or miss cache. In [12], Qureshi *et al.* presented a V-way (variable way) cache to reduce the miss conflicts existing in traditional C-way (constant way) set-associative caches. The set-balancing cache [13] and the adaptive hybrid cache [10] were introduced for similar reasons, targeting unbalanced accesses to main memory. For multi-processor systems, Matthew *et al.* [14] designed configurable L1 caches for the MicroBlaze soft processor implemented on Xilinx FPGAs and achieved up to 41% speedup by using a 32KiB 4-way cache with LRU replacement. In the same setting, Kalokerinos *et al.* [15] presented an integrated network interface and cache controller, significantly improving hardware utilization.

Latency of memory-intensive applications is particularly significant in FPGAs due to off-chip memory bandwidth limitations. Many researchers addressed this area by exploiting the highly configurable on-chip memory architecture. For example, Cheng *et al.* [16] developed a trace analysis method to detect relations among all memory accesses. Performance was greatly improved by caching independent data in separate local memories. Adler *et al.* [17] used BRAMs as statically-managed scratchpads rather than dynamically-managed caches, and described a management system for different levels of local storage. Choi *et al.* [18] implemented a multi-ported cache based on the so-called live-value table, aimed at a system architecture where both the host processor and multiple accelerators are on the same chip. In their approach, both the processor and the accelerators access the same off-chip memory via a single custom multi-port cache, which of course may become a performance bottleneck. Putnam *et al.* [19] provided a cache-based solution to simultaneously increase performance and reduce power consumption, since external DRAM accesses require much higher power than on-chip SRAM. In this design methodology, the CHiMPS HLS tool first compiles the high-level code (written in C) to an intermediate representation and then the caches are optimized according to the memory access patterns. Similarly, Winterstein *et al.* [20] also used the LLVM intermediate language to maximize the utilization of BRAMs to accelerate a specific algorithm (tree reflection).

Our approach is inspired by some of these works, in particular to reduce access conflicts by *using a separate cache, possibly with a different architecture, for each source code array* mapped to external DRAM.

III. METHODOLOGY

A. MOTIVATION

The key idea of our approach, as mentioned above, is to exploit the characteristics of many high-performance algorithms, especially those written using the OpenCL language, in which large data arrays are mapped in DRAM, and these arrays are *either only read or only written by an accelerated function mapped to hardware*. This allows us to implement *separate caches, one for each such array, without worrying about coherency*. This is essentially the use pattern of OpenCL kernels that we use as the application modeling language, and it is common to several other applications.

By using a separate cache for each such array, we can use a different cache organization and architecture (line size, cache size, associativity) *optimized specifically for the access patterns of each array*.

For example, consider a very simple algorithm, namely matrix multiplication. Although simple, it has significant practical usefulness, since it is at the root of computer vision and machine learning algorithms. In past research, it has been accelerated both for GPUs [21] and FPGAs [22]. This requires the designer to simultaneously solve two different problems:

1) creation of a pair of nested loops, going over a portion ("block") of each source matrix, and generating a block of the destination matrix;
2) transfer of the data from global DRAM to local on-chip SRAM buffers and correct use of those buffers to implement one block of multiplication.

While the first part is relatively easy and it can be even automated under control of high-level synthesis directives, the second part is much more difficult and can require a significant coding effort. Our caches can automatically implement both by *exploiting the locality* that is exposed by the designer while solving the first part.

If the source matrix blocks are rectangular (e.g., M rows of N elements, or M columns of N elements, where N is often much larger than M), then different organizations are required for the two caches. One cache needs to store M lines of N elements each, while the other

cache needs to store N lines of M elements each, for best performance.

Our approach, based on C++ template classes for the array types, allows us to keep the original source code largely unchanged and to ensure that the cache is accessed every time an element of the original array is read or written in the code. The template arguments define:

1) the data type of an element of the original array;
2) the number of dimensions and the size along each dimension;
3) the cache size, line size, associativity, and so on.

The minimal changes to the source code that are required by our approach can be evaluated by comparing the code shown in Appendix B.

A direct-mapped cache can be used for each matrix and each matrix has its own port to the external memory to achieve a good performance. Since both matrices A and B are read-only, and matrix C is write-only, special read-only and write-only caches can be used in order to avoid false loop-carried dependencies and to reduce the pipeline initiation interval. For caches with relatively small sizes (e.g., 8 or 32 words), the innermost loop can also be unrolled to further increase the performance, since the cache can be implemented as registers. Since matrix C is not the bottleneck of this algorithm, the cache for C could also be removed in order to save resources.

In this case, as will be discussed in Section V, the cache-based approach reached a performance that was within 10% of the theoretical best, namely the case in which all the matrices to be multiplied fit in the on-chip BRAM.

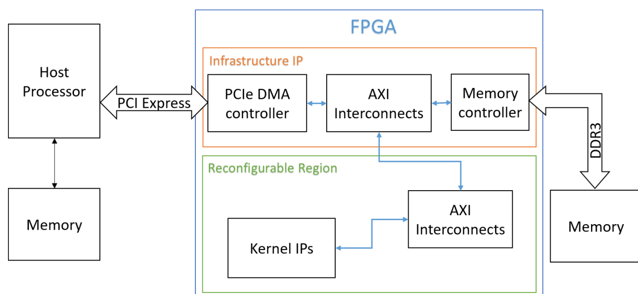


FIGURE 1. Host and accelerator system architecture.

B. HARDWARE ARCHITECTURE

Fig. 1 illustrates the hardware architecture that is considered in this research and that is supported by the Xilinx tools that we use to demonstrate it. For the sake of illustration, we use a concrete instance of a general architecture template, where the off-chip bus is PCIe, the on-chip bus is AXI, and the DRAM interface is DDR3. However, our approach is fully general and is not limited to this specific architecture. In this figure, the accelerators (called "kernel IPs" following the OpenCL terminology) are connected to the host processor by an off-chip PCIe bus and the on-chip AXI bus. The host processor downloads (via the "infrastructure IP") onto the FPGA the bitstream to configure the accelerator, and stores the data to be processed in the external memory, which is connected to the FPGA fabric via a DDR3 interface and the AXI bus. The FPGA board that we use for illustration in this article is the Xilinx ADM-PCIE-7V3, with a Virtex 7 on board. It is described in TABLE 1.

TABLE 1. Target FPGA and board.

Target Device Name	ADM-PCIE-7V3:1ddr:3.0
FPGA Part	Xilinx Virtex-7: XC7VX690T-2
Clock Frequency	200MHz
Memory Bandwidth	9.6GB/s

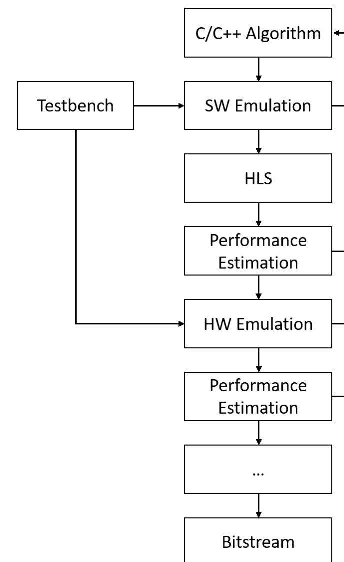


FIGURE 2. Design flow in SDAccel.

C. DESIGN FLOW

As mentioned above, we use an HLS flow to implement both the accelerated algorithm and its dedicated caches. In this research, we used Xilinx SDAccel™ v2016.2, which in turn uses VIVADO_HLS™ for HLS and VIVADO™ for logic synthesis, power estimation, etc.

As shown in Fig. 2, the SDAccel design flow starts with software (SW) emulation, which verifies the functional correctness of the algorithm using a properly designed testbench. Both the algorithm and testbench can be modeled in C, C++ or OpenCL. Then, VIVADO_HLS™ synthesizes and estimates the cost and performance for the kernel IPs. The resulting report contains information about statically analyzed latency and throughput, resource utilization and so on. The designer can use this information to further direct HLS towards the desired solution (e.g., the designer can use the pipelining iteration interval, which provides a good estimate of the final throughput). During the following hardware (HW) emulation phase (usually called RTL simulation), SDAccel™ calls VIVADO™ to connect the synthesized

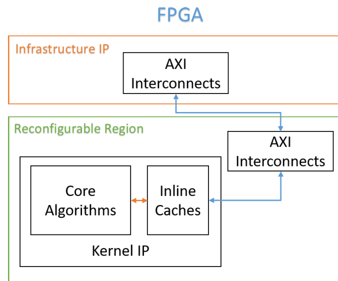


FIGURE 3. **Inline cache.**

kernel IPs with other infrastructure blocks shown in Fig. 1 and launches a co-simulation between the RTL and the high-level testbench (typically written using C/C++ or OpenCL host code). This simulation is much slower than the SW emulation, but it also generates a much more accurate report of the system performance, which in particular includes the effects of off-chip DRAM accesses.

In the literature surveyed above, the caches are usually designed as concurrent HW modules, which execute in parallel with the kernel IPs [18]. While this strategy offers some advantages, such as a better decoupling between the external memory and the IPs, it also has a significant disadvantage: it requires one to *change the accelerated kernel code* to access the caches via dedicated interfaces rather than directly access the source code arrays. This is incompatible with the strategy of providing a software-like design environment for FPGA hardware, and motivated us to create the *inline caches* that we introduce in this article.

As shown in Fig. 3, our caches are directly "inlined" in the algorithms to be accelerated. In this way, *the "golden" code that has been functionally verified by SW emulation does not need to be changed for high-performance implementation*. Only the top-level module interface (which is typically much smaller and simpler than its often intricate algorithmic code) requires some small changes, as illustrated below. In the resulting RTL, the caches are directly synthesized as part of the kernel IP.

Since the HLS tools that we currently use for synthesis do not support classes or templates in OpenCL kernel code, all our examples below are based on the C++ language. However, this is only to ease prototyping our flow. The same mechanism could be implemented also in OpenCL by slightly modifying the OpenCL HLS front-end.

As mentioned above, the design has to be modified only slightly in order to insert the inline caches in the interface of the original kernel. Further changes to the flow will be needed to analyze the array access patterns and to optimize the cache architectures. Automation of these new steps is left to future work. In this paper we perform this task by hand.

As shown in Fig. 4, some analysis of the external memory access traces is necessary to find the best cache parameters to maximize the reuse with an acceptable area cost (we will describe this in more detail below). Note that this access

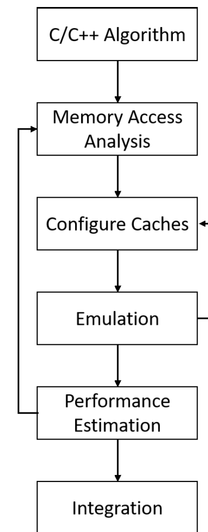


FIGURE 4. **Design flow with caches.**

analysis is needed only for arrays mapped to external ("global" in OpenCL terminology) memory, and not for the local arrays or scalars. This only requires the designers to make a few modification to the top-level function interface to replace the original data types of the global array variables with a template cache data type.

Fast SW emulation can be used to both trace the array addresses and to check the achieved hit ratio, which is automatically captured and printed by our cache models. Then can be performed the synthesis, followed by HW emulation or actual FPGA prototyping to obtain more detailed external memory performance information, which can potentially lead to further optimizations.

D. INLINE CACHES

In this work, we propose and describe several kinds of inline caches, e.g., direct-mapped and set associative, selected based on the memory-trace pattern of the applications to be optimized. Remember that in our work a separate cache is implemented for each array mapped to global memory. This means that *performance is largely independent of the global memory addresses at which each array is allocated*, and that there are no conflicts between different arrays. Since accelerated kernels typically make fairly regular accesses to each array, this means that *real-time performance of our dedicated caches is much more predictable than that of traditional shared caches*, and can be comparable to that of manually managed scratchpads.

1) DIRECT-MAPPED CACHE

As its name indicates, each element of each array in the external memory has a corresponding fixed position in the cache, according to a fixed bit field of the address. The line bits in the middle of the address determine to which line in the cache it is mapped, while the word bits define the position

within the cache line. The tag bits are used to check whether a given address is contained in the corresponding line of the cache (''cache hit'') or not (''cache miss''). In the latter case, the cache fetches the correct data from external memory and updates the corresponding cache line and tag.

Each cache line is read with a single AXI bus access, possibly using a burst (depending on the line and data bus bit widths), and stored into the cache. The write policy for the caches that we implemented is write-back, i.e., only the cache is updated initially, while the external memory is updated only when the cache needs to be flushed, either due to a write miss or due to the completion of the accelerator execution. As mentioned above, in this paper we assume an execution model similar to that of OpenCL, in which global arrays cannot be read and written at the same time by the same HW-accelerated function (kernel). This avoids all kinds of coherency issues for our caches, and typically enables them to be read-only or write-only. As usual, we keep valid and dirty bits for each cache line, to indicate if it contains valid data from memory or data that needs to be written back to memory.

In this research, the direct-mapped cache was designed in C++ by using a template class as shown in Appendix A. The template arguments, as mentioned above, define the type of one element of the cache and of the corresponding off-chip memory global array, the line size and the word size. The constructor initializes the base address of the corresponding off-chip memory array (typically the value of a pointer argument of the OpenCL kernel or C++ top-level function) and other variables, like the valid and dirty bits. In HLS, the constructor is typically executed as part of the reset sequence of the HW block. A C++ namespace is used to choose among a read-only, write-only or read-write cache.

In the algorithmic code to be implemented via HLS, the external memory is usually accessed by using the `operator[]` or the `operator*` on a pointer passed from the interface. Hence, we overloaded the `operator[]` for the cache type, for uses on both the left hand side (write) and the right hand side (read) of an assignment.[1] This allows us to change only the interface of the function to be synthesized, not its code, thus dramatically reducing the design time and the likelihood of coding errors. For instance, we show the modification from the original code of the matrix multiplication algorithm in Appendix B.

The interface to external memory can be defined simply by instantiating the cache type, with the appropriate template parameters, instead of every source array that is mapped to off-chip DRAM. The constructor and destructor that we created for the cache types take care of all the bookkeeping, from initializing the cache as empty (resetting all valid and dirty bits), to flushing an output cache and printing the statistics in a simulation context, when the accelerator completes its operation.

[1] We managed to overload differently the read and write accesses to call a different cache access function, by exploiting an inner class as an agent [23].

Note that since the cache access functions (for reading and writing) are inlined into the high level kernel code, the synthesized kernel takes care of both executing the computation using the cached data, and reading/writing data from/to the main memory in case of misses. As we mentioned above, this somewhat reduces the achievable performance, but it dramatically simplifies the design flow and is consistent with OpenCL philosophy, where the work items themselves take care of moving the data from global to local memory. As we will show in Section V, the performance is excellent anyway and similar to manually optimized memory transfers between the global and local memory. In future work we are planning to experiment with the use of separate processes to handle the caches.

In order to achieve the best performance, the data width of the AXI interfaces that are used to transfer a line to and from external DRAM should have the same size as a cache line, so that a read or write can be completed in one clock cycle (plus global memory latency in case of reads, of course). If the line length is larger than the global memory read size, then burst accesses will automatically be used by our design. This is one of the key advantages that the designer gets for free by using our caches.

Algorithm 1 Read Data From Direct-Mapped Cache

Require: 32-bit *addr* **and** *Cache* with a pointer *ptr_mem* to external memory
Ensure: $data = Cache[addr]$
1: $tag, line, word \leftarrow addr$
2: $request \leftarrow request + 1$
3: **if** $tag = Cache.tags[line]$ **and** $Cache.valid[line]$ **then**
4: $\quad hit \leftarrow hit + 1$
5: **else**
6: $\quad$ **if** $Cache.dirty[line]$ **then**
7: $\quad\quad location \leftarrow Cache.tags[line], line$
8: $\quad\quad ptr_mem[location] \leftarrow Cache.array[line]$
9: $\quad\quad Cache.dirty[line] \leftarrow$ false
10: $\quad$ **end if**
11: $\quad loc \leftarrow addr >> \text{LINE_BITS}$
12: $\quad Cache.array[line] \leftarrow ptr_mem[loc]$
13: **end if**
14: $Cache.tags[line] \leftarrow tag$
15: $Cache.valid[line] \leftarrow$ true
16: **return** $data \leftarrow Cache.array[line].\text{slice}(word)$

Algorithm 1 and Algorithm 2 demonstrate how a cache reads or writes an address of global memory. The pair of variables *request* and *hit* are used as performance counters to enable cache parameter tuning also when an FPGA is used as a rapid prototyping platform, and can be accessed via FPGA-provided debugging mechanisms (e.g., via JTAG). The *valid* and *dirty* arrays have Boolean elements. The *tags* array contain unsigned integers of the appropriate length. The *array* array is used to store all the lines of data in the cache.

Algorithm 2 Write Data to Direct-Mapped Cache

Require: 32-bit *addr* **and** *data* **and** *Cache* with a pointer *ptr_mem* to external memory
Ensure: $Cache[addr] = data$
1: $tag, line, word \leftarrow addr$
2: $request \leftarrow request + 1$
3: **if** $tag = Cache.tags[line]$ **and** $Cache.valid[line]$ **then**
4: $\quad hit \leftarrow hit + 1$
5: **else**
6: $\quad$ **if** $Cache.dirty[line]$ **then**
7: $\quad\quad location \leftarrow Cache.tags[line], line$
8: $\quad\quad ptr_mem[location] \leftarrow Cache.array[line]$
9: $\quad$ **end if**
10: $\quad loc \leftarrow addr >> \text{LINE_BITS}$
11: $\quad Cache.array[line] \leftarrow ptr_mem[loc]$
12: **end if**
13: $Cache.tags[line] \leftarrow tag$
14: $Cache.valid[line] \leftarrow \text{true}$
15: $Cache.dirty[line] \leftarrow \text{true}$
16: $Cache.array[line].\text{slice}(word) \leftarrow data$

The two algorithms share a similar structure. Lines 1-4 handle cache hits. The address is split into three pieces, namely *tag*, *line* and *word*, then the value (or values, for the set-associative case) stored in *tags* is compared with the *tag* part of the address. If it is a hit, the following operation is the read from (or write to) *array*, on line 16. In both cases, the actual location of the data within the line depends on the value of *word*. If it is not a hit, then a new read from the external memory is necessary (after writing back the dirty line in case of a write or read/write cache).

In many algorithms, and in particular in the most massively parallel cases written in languages such as OpenCl, the uses of each array argument of a kernel are either read-only or write-only. Hence, we designed a special cache for these read-only and write-only memory accesses in order to speed up the synthesis, reduce the cost, and improve the performance. For instance, a read-only cache does not need to check if a line is dirty. Algorithm 1 and Algorithm 2 show the get() and set() functions for this case. The C++ code is listed in Appendix A.

2) SET-ASSOCIATIVE CACHE

In some algorithms (e.g., sorting, FFT), *data read by successive external memory accesses are not located at contiguous addresses*. In the worst case, accesses with the same stride as the line size would cause the lowest performance, since all accesses could become misses. For these applications, using a set-associative cache is the easiest solution that does not require code changes.

Fig. 5 shows an example of a 2-way set-associative cache. The data fetched from main memory can be stored in any cache set. The replace policy that we are using in our example code is Least Recent Used (LRU), but other algorithms can be implemented as well. In Fig. 5, the *LRU* field records the last time when a cache line has been read or written. In this research, we use as time stamp (i.e., *LRU* value) the *request* counter, which was also used for statistical purposes in Algorithm 1 and Algorithm 2.

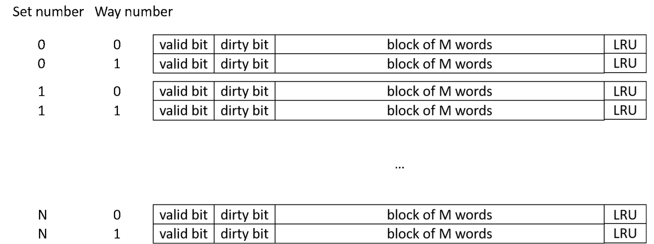


FIGURE 5. Diagram of a two-way set-associative cache.

Designers should carefully choose the number of ways of a set-associative cache when optimizing the performance, because a large number of ways causes higher resource utilization. The adaptation of traditional cache simulators to our methodology, basically by having a separate cache for each kernel argument, is left to future work.

Just like in the case of direct mapped caches, also for set-associative caches we have three variants: read-only, write-only and read-write.

In this work we did not consider fully associative caches due to the high cost of the Content Addressable Memory.

3) CACHE-DEPENDENT HLS OPTIMIZATIONS

Using the inline caches described in this work, programmers usually achieve better performance compared to original algorithms. Algorithm- and cache-dependent optimizations may be needed, though, in order to achieve the best performance. As discussed above, the designed caches are compatible with HLS optimization methods. Application-specific post-optimizations include, e.g., pipelining or unrolling a loop, and providing memory directives. For instance, a memory dependency is assumed to exist, if there is an array that is both read and written. Often, HLS may not be able to detect automatically if this memory dependency is true or not, and directives are required to optimize memory accesses by using knowledge coming from the programmer.

Another very useful optimization can be used when the designer knows that some array accesses in the code will be always hits (e.g., the access to array element $i + 1$ after accessing element i, if i is even and the line size is at least 2). When the address analysis performed by the HLS synthesis tool is not powerful enough to detect this situation due to complex address computations, this can be done manually by using two dedicated member functions provided by our cache class. The methods *retrieve()* and *modify()* can be used instead of the convenient operator "`[]`" to directly read or write respectively an element of the array by assuming that it is already in cache. These functions can dramatically improve the throughput by reducing the initiation interval of pipelined

inner loops, like a convolution operation, which accesses the same array multiple times in the innermost loops.

It is also possible, in some cases, to further optimize the accesses by using, e.g., two separate read-only and write-only caches for an array that is both read and written, and for which the programmer knows that the read and written portions by a given kernel call never overlap.

As shown in Fig. 4, a SW or HW emulation must follow every optimization that changes the source code, e.g., by using retrieve() and modify(), in order to guarantee its correctness.

IV. TEST CASES

In this work we demonstrate the usefulness of our proposed inline caches by using some memory-intensive highly parallel algorithms from various fields, for which we already had optimized manual implementations from past research.

To each algorithm, we applied also some specific optimizations that are limited to the use of synthesis directives, hence which should be correct by construction.

We classify them into three groups. The first class includes only basic optimizations (e.g., pipelining) of the innermost loop and multiple memory ports. In this case, all the input and output data to and from a kernel are stored externally.

Second, the best known optimizations of each algorithm from the literature were implemented. In this case, in order to illustrate the best theoretical performance, all data was also stored in on-chip BRAMs. Although this is not realistic for large data sizes, it provides an upper bound to the achievable performance.

Third, our proposed flow is used to include our inline caches in each design, and to perform some further code optimizations, as detailed below. In this case, all the input and output data are stored in the external memory, as well as inside the caches.

Each class may contain various options for each application. Since loop unrolling for the loops with read or write to a port is not beneficial for the first class of optimizations, we ignored this optimization in our experiments in order to provide a fair comparison. But it can be used and would result in further advantages of our approach.

We recorded for each implementation the execution time (considering the clock frequency after synthesis, placement and routing), the power consumption and resource utilization, in order to compare the effects of the acceleration provided by the inline cache.

A. K-NEAREST NEIGHBORS (KNN)

The algorithm is used to classify data in machine learning and statistics applications. As its name indicates, the algorithm aims to find the first k nearest neighbors to a given "test" point among a set of "training" points. In past work, where we manually optimized the algorithm [24], we separated the algorithm in two parts: one computes the distances between the test point to all training points, and the other searches for the k smallest among those distances. Both parts are memory intensive. Due to the fact that the first part contains only one loop which reads all the data, it can be very easily optimized simply by accessing the data in bursts (e.g., using our proposed caches in the most straightforward manner). Hence, in this work we focused only on the second part, which is shown in Algorithm 3.

Algorithm 3 K-Nearest Neighbors (KNN) Algorithm

Require: $dist$ the array of distances **and** number of records num **and** k

Ensure: $d[0] = dist[v_0] \leq d[1] = dist[v_1] \cdots \leq d[k-1] = dist[v_{k-1}] \leq dist[i]\ \forall i \notin \{v_0, v_1, \ldots v_{k-1}\}$

1: $d_{\text{last}} \leftarrow 0$
2: **for** $i = 1$ to k **do**
3: $\quad d_{\min} \leftarrow \infty$
4: $\quad$ **for** $s = 0$ to $num - 1$ **do**
5: $\quad\quad dis \leftarrow dist[s]$
6: $\quad\quad$ **if** $dis < d_{\min}$ **and** $dis > d_{\text{last}}$ **then**
7: $\quad\quad\quad d_{\min} \leftarrow dis$
8: $\quad\quad$ **end if**
9: $\quad$ **end for**
10: $\quad d[i] \leftarrow d_{\min}$
11: $\quad d_{\text{last}} \leftarrow d_{\min}$
12: **end for**

Algorithm 3 contains two loops, with one memory access and a few operations in each iteration. The first implementation pipelines the innermost loop. The second implementation also assumes that all distances are already stored in on-chip memories, and thus provides the best achievable speedup. The third implementation uses our read-only and write-only direct-mapped inline caches, with various configurations, to accelerate the algorithm. Due to algorithm simplicity, we did not use post-optimizations.

B. BITONIC SORTING

Sorting algorithms are among the most essential and fundamental algorithms in computer science. Various sorting schemes have been implemented in software or hardware for a large variety of applications. Bitonic sorting offers an excellent level of parallelism and it can be modified, as discussed in [25], into several phases, each of which using read-only and write-only arrays. Hence, it has also been accelerated both on FPGAs [7] and on GPUs [25].

Algorithm 4 contains three nested loops. Each iteration in the outermost loop sorts blocks of size 2^b into the bitonic sequences (i.e., sequences that are first increasing, then decreasing, then possibly increasing once more). The middle loop is over stride sizes s and is used to merge two adjacent bitonic sequences into a large sequence. The innermost loop has a constant number of iterations, and swaps the values of two data items at a distance of 2^s if they are not in the correct order. This algorithm, like KNN, is very memory intensive.

Also in this case, the first implementation pipelines the innermost loop. Since that loop performs two read operations and two write operations to the same array in each iteration,

Algorithm 4 Bitonic Sorting Algorithm

Require: a the array to be sorted **and** array size $N = 2^n$ **and** sorting direction dir

Ensure: $a_i \geq a_j \forall i \geq j$ for dir = true **or** $a_i \leq a_j \forall i \geq j$ for dir = false

```
1:  for b = 1 to n do
2:    for s = i − 1 to 0 do
3:      for i = 0 to N/2 − 1 do
4:        dir_0 ← (i/2^(b−1))&1
5:        dir_0 ← dir_0 or dir
6:        step ← 2^s
7:        pos ← 2 × i − (i&(s − 1))
8:        a[pos], a[pos + step] ← order(a[pos], a[pos +
          step], dir_0) {swap two values if they are not in
          correct order}
9:      end for
10:   end for
11: end for
```

a loop-carried dependency causes a large initiation interval, i.e., a slow pipeline throughput.

The second implementation, which is discussed, for example, in [25], divides the algorithm into two parts. The first one splits the global array into multiple arrays, each with the size equal to the on-chip memory size, and then it uses Algorithm 4 to sort these small arrays into bitonic sequences. The second part merges these bitonic sequences into the fully sorted array.

The third implementation assumes that the array to be sorted can fit in local memory, and then uses Algorithm 4 to sort it. Of course this is unrealistic for large arrays, but it has been included to show the best achievable performance.

Our cache-based implementations, due to the read and write stride accesses to external memory shown in Algorithm 4, require 2-way set-associative caches to achieve the best performance. Note that if the stride size is relatively small (smaller than the cache line size), one can easily prove[2] that the two values are stored in the same cache line after one fetching. Even if the stride size is large, the two values will be mapped to different cache lines in the same set. This guarantees the two write operations to hit.

The two read and two write operations in the innermost loop would still create a loop-carried dependency, as discussed above, and require a large pipeline initiation interval. However, one can easily note that the two write operations can never be misses because they access the same array addresses as the read operations. Thus, in this case we can use the modify() method to significantly reduce the initiation interval and dramatically improve the performance.

In order to further remove the dependency created when the two accesses conflict with each other, we can consider one more optimization. We exploit the fact that the iterations in the innermost loop are independent, hence the loop can be unrolled. Memory traces showed that once the 2-way set associative cache fetched the new data into the cache line, a number of following iterations would never miss. The number depends on the cache line size, but if this number of iterations is grouped together via partial loop unrolling, then only one initial access would need to go through the miss check, while the following unrolled iterations can just use the retrieve() and modify() methods to improve performance.

[2] Considering that the sequence starts from position 0 and that both the stride size and cache line size are the powers of 2.

C. SMITH-WATERMAN ALGORITHM

Was originally proposed to align two sequences with partial matches [26], as is performed by the UNIX `diff` command. A similar algorithm, called Needleman-Wunsch, implements a global searching technique [27]. Due to their efficiency, both algorithms have been widely used in the field of bioinformatics, e.g., to compare gene sequences [28]. Hence, many research works have focused on their acceleration, e.g., on FPGAs [27] and on GPUs [29].

The Smith-Waterman algorithm contains two parts. The first part constructs a score matrix and is the most expensive. The second part, called traceback, traverses the matrix to align the two sequences. In this research, we considered only the first part, shown in Algorithm 5, and we can see that the score matrix is accessed multiple times in one iteration.

Algorithm 5 Score Matrix Construction Part of the Smith-Waterman Algorithm

Require: two sequences seq_0 and seq_1 with length N **and** $N > 0$

Ensure: a score matrix $M = \{m_{ij}\}_{(N+1)\times(N+1)}$ **and** a direction matrix $D = \{d_{ij}\}_{(N+1)\times(N+1)}$

```
1:  gap ← −1
2:  m_ij ← 0, ∀i, j
3:  for i = 1 to N do
4:    {Loop along the rows.}
5:    for j = 1 to N do
6:      {Loop along the columns.}
7:      if seq_0[i] ≠ seq_1[j] then
8:        match ← 2
9:      else
10:       match ← −1
11:     end if
12:     val_0 ← m[i − 1][j − 1] + match {Up left entry.}
13:     val_1 ← m[i − 1][j] + gap {Up entry.}
14:     val_2 ← m[i][j − 1] + gap {Left entry.}
15:     m[i][j] ← max(val_0, val_1, val_2)
16:     d[i][j] ← 0, 1, 2 {Correspond index of maximum
        value.}
17:   end for
18: end for
```

The first implementation of this algorithm again pipelines the loop, and uses four separate external memory ports to access the four arrays.

The second implementation is based on the example code that is distributed with SDAccel™ from Xilinx Inc. Its first optimization was to use burst reads and burst writes to transfer both input sequences and both matrices (score and direction) between the external memory and the on-chip memory. As can be seen in Algorithm 5, the computed score in each iteration is the left value for the next iteration, which implies a loop-carried dependency. Similarly, the up value for each iteration is the up left value for the next iteration. Hence, the second optimization is to replace memory accesses to the left entry and up left entry by two local variables in order to eliminate some loop-carried dependencies involving slow memory accesses. These optimizations achieved a very good acceleration of the original algorithm.

In the third implementation uses again our inline caches, e.g., direct-mapped ones and read-only direct-mapped ones. The read-only direct-mapped caches were used for the two input sequences. The score matrix is both read and written, hence this optimization could not be applied, and a read/write cache had to be used. We noted that the synthesis tool detected many false loop-carried dependencies. Hence, in a post-cache manual optimization, we used directives to instruct it to ignore those dependencies in order to reduce the initiation interval.

Note that several such dependencies were also automatically eliminated by using both a read-only and a write-only direct-mapped cache to access non-overlapping sections of the score matrix. In addition, special cache methods that access the cache more efficiently if successive addresses are guaranteed to be contiguous could also be used in this case.

D. LUCAS-KANADE ALGORITHM

Was first introduced in [30] and has been widely adopted in the computer vision domain, especially for optical flow estimation. In the optical flow application, two images taken close in time are analyzed to find small (thanks to time proximity) pixel displacements due to movements of various objects. J.Y. Bouguet [31] also used it to solve the feature tracking problem. This algorithm is used to compute partial derivatives of images as shown in (1) and (2):

$$I_x(x,y) = \frac{\partial Im(x,y)}{\partial x} = \frac{Im(x+1,y) - Im(x-1,y)}{2}, \quad (1)$$

$$I_y(x,y) = \frac{\partial Im(x,y)}{\partial y} = \frac{Im(x,y+1) - Im(x,y-1)}{2}. \quad (2)$$

Algorithm 6 illustrates a simplified version of the Lucas-Kanade algorithm. The operations omitted have almost no impact on performance. The implementation that we use in this article is mainly based on (3), (4) and (5),

$$G \doteq \Sigma_{x=p_x-w_x}^{p_x+w_x} \Sigma_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \quad (3)$$

$$b \doteq \Sigma_{x=p_x-w_x}^{p_x+w_x} \Sigma_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} \delta I I_x \\ \delta I I_y \end{bmatrix}, \quad (4)$$

$$v_{opt} = G^{-1} b, \quad (5)$$

Algorithm 6 Lucas-Kanade Algorithm

Require: two frames of images $image_0$ and $image_1$ **and** other coefficients
Ensure: v_{opt}
1: **for** $j = 0$ to $HEIGHT - 1$ **do**
2: **for** $i = 0$ to $WIDTH - 1$ **do**
3: $G_{2\times2} \leftarrow 0$
4: $b_{2\times1} \leftarrow 0$
5: **for** $w_j = -w_y$ to w_y **do**
6: **for** $w_i = -w_x$ to w_x **do**
7: $center \leftarrow \mathrm{Pos}(i+w_i, j+w_j)$
8: $left \leftarrow \mathrm{Pos}(i+w_i-1, j+w_j)$
9: $right \leftarrow \mathrm{Pos}(i+w_i+1, j+w_j)$
10: $up \leftarrow \mathrm{Pos}(i+w_i, j+w_j-1)$
11: $down \leftarrow \mathrm{Pos}(i+w_i, j+w_j+1)$
12: $im^0_{val} \leftarrow image_0[center]$
13: $im^1_{val} \leftarrow image_1[center]$
14: $\delta I \leftarrow d(im^0_{val}, im^1_{val})$
15: $im^0_{left} \leftarrow image_0[left]$
16: $im^0_{right} \leftarrow image_0[right]$
17: $I_x \leftarrow (im^0_{right} - im^0_{left})/2$
18: $im^0_{up} \leftarrow image_0[up]$
19: $im^0_{down} \leftarrow image_0[down]$
20: $I_y \leftarrow (im^0_{down} - im^0_{up})/2$
21: $G \leftarrow G + g_{2\times2}(I_x, I_y)$
22: $b \leftarrow b + f_{2\times1}(\delta I, I_x, I_y)$
23: **end for**
24: **end for**
25: $G \leftarrow \mathrm{inverse}(G)$
26: $v_{opt}[j][i] \leftarrow G \times b$
27: **end for**
28: **end for**

which compute the optimum optical flow vector [31]. The function Pos() is used to ensure that a pixel is located in the image frame.

The algorithm contains four loops. The first two are over all the pixels of the images and the last two are over the computation window. The bottlenecks are located in the innermost loop and are due to the five accesses to external memory.

As usual, the "External memory" implementation simply pipelines the innermost loop and uses separate memory ports for different input and output arrays.

In the Algorithm 6, the five pixels of $image_0$ accessed by the innermost loop include the center pixel (defined by i, j, w_i, w_j) and four other pixels around the center pixel. When focusing only on the innermost loop, the center pixel and the right pixel can easily be reused in the following iteration. In this case, the number of accesses to external memory reduces to three instead of five.

If the next outer loop is also considered, then one or two lines can be reused by exploiting a structure known as a "line buffer", which contains two rows of the current image. Loop unrolling could also be used in this case to further improve

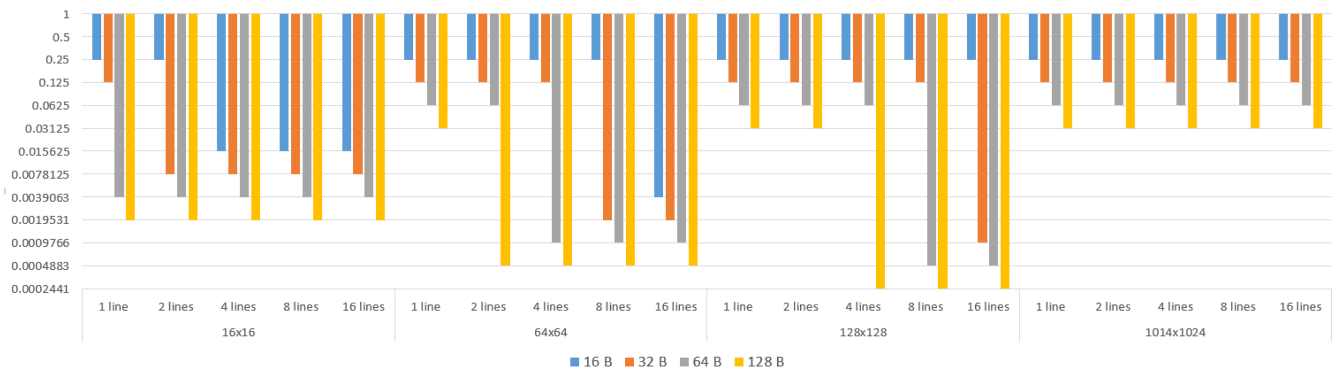


FIGURE 6. **Miss ratios for different numbers of lines, data sizes and line sizes for matrix *A* of matrix multiplication (log scale).**

speed. If the next outer loop is also considered, a large buffer can be exploited to store even more lines of the image in the on-chip memory.

For simplicity, we adopted two combined optimizations in the "On-chip memory" (unrealistically fast) implementation. The first one was to copy all the pixel data to the on-chip BRAMs in order to maximize reuse. The second one was to use two variables to store the center and right pixels as described above, in order to reduce the initiation interval of the innermost loop.

In this case, our cache implementation was based on read-only caches, which were very helpful to accelerate the algorithm.

Further acceleration could also be obtained by manual post-optimizations to improve the innermost loop initiation interval, below the initial value of 5 selected by the synthesis tool. It required moving a prefetching operation before the innermost loop, and then using the direct retrieve() method to access the data inside the loop. Then, the initiation interval could be reduced to 1. In this case, a large enough cache behaves pretty much like a line buffer.

V. RESULTS

In this section we discuss the performance and cost of our caches using the test cases described in the previous section. Throughout the section, performance was estimated using the so-called "hardware emulation" capability of SDAccel™ v2016.2, which in fact is RTL simulation. The DDDR3 DRAM interface is also simulated at the cycle-accurate level. In addition, both device power and resource utilization are estimated after logic synthesis, placement and routing by Vivado™ v2016.3. The codes of the various types of caches and the implementations of these applications are available in the repository at this link, https://github.com/HLSpolito/Cache_Application.

A. MATRIX MULTIPLICATION

As mentioned above, we report three classes of implementations for each algorithm, namely one with all data in external memory (lower bound on performance), one with all data initially transferred to on-chip memory (upper bound), and one with the best architecture that we found for our caches. We used two direct-mapped read-only caches for input matrices A and B, and a direct-mapped write-only cache for output matrix C.

TABLE 2. **Performance and resource utilization for various implementations of matrix multiplication (16 × 16 matrices).**

Implementation	Ext. Mem.	On-chip Mem.	Cache	
Loop flatten	Yes	Yes	No	Yes
Exec. time (ms)	0.241	**0.027**	0.058	**0.031**
Power (W)	**0.507**	**0.471**	1.345	1.201
Energy (mJ)	0.122	**0.013**	0.078	**0.037**
BRAM	3	2	38	31
DSP	3	3	3	3
LUT	1792	1462	6588	5699
FF	3051	2237	16186	17794

Note that due to a limitation of the HLS tool that we used (namely Vivado_HLS™) we had to slightly modify the code of the "On-chip mem." and "With caches" implementations, in order to make the loop nest perfect – we incorporated the output matrix assignment into the last iteration of the innermost loop. This manual code change almost doubles the overall performance. Note that the change is required regardless of our caches, and we applied it to all the implementations for a fair comparison.

TABLE 2 compares performance, power consumption and resource utilization of all implementations using 16 × 16 matrices. The caches for matrices A and C contained one 16-word line each (i.e., one row of the matrix). The cache for matrix B contained 16 16-word lines, which was also the size of matrix B.

Note how the cache-based implementation with loop flattening achieves essentially the same performance as the "ideal" implementation, where all data fits in the on-chip memory. Of course, the caches have a significant resource cost, which becomes particularly noticeable for computationally-simple algorithms like matrix multiplication. Moreover, the energy consumption of the best cache implementation is only 30% of that of the external memory implementation. This is without considering the energy con-

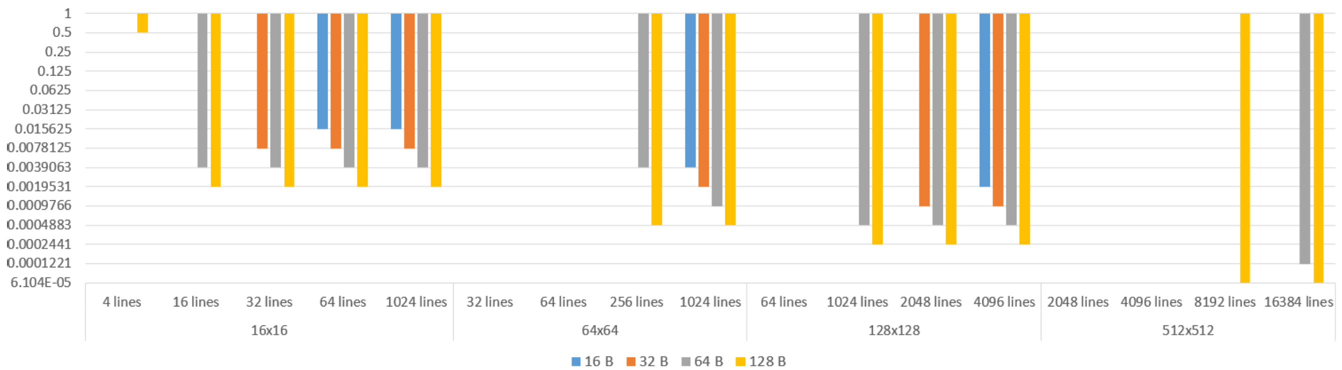


FIGURE 7. Miss ratios for different numbers of lines, data sizes and line sizes for matrix *B* of matrix multiplication (log scale).

sumed by the external memory itself, which would make the cache-based implementations even more efficient, due to the low miss ratio.

More complete results, for a broad range of matrix and numbers of lines, are reported in Fig. 6 and Fig. 7. As shown in Fig. 6, the hit ratio of the caches applied to matrix A is highly dependent on the line size and is not affected by the number of lines in the cache until the cache can hold all the data in the matrix, when the miss ratio can be reduced to 0.02%.

The caches applied to matrix B have a different behavior, as discussed above. The miss ratio can be small only when the cache size is the same the matrix size as shown in Fig. 7, thus making caches useful for matrix B only in order to automatically perform burst accesses to global memory.

B. K-NEAREST NEIGHBORS

We tested the KNN algorithm using a set of data containing 2048 locations of a series of hurricanes [7], each represented as a pair of floating point numbers for latitude and longitude. The host code of this OpenCL design computes all the distances from a given reference point to all the points in the data set, then sends the distances to an OpenCL kernel that finds the k smallest distances, where $k = 5$.

For each value of k, we tested four kernel implementations belonging to the usual three classes. The "Ext. mem." one kept all computed distances in external memory. The "On-chip mem." one copied all data to the on-chip memories at the beginning of the program, to maximize reuse. As usual, this is realistic only for small matrices. The "With caches" one used a read-only direct-mapped cache to store the input array *dist* (see Algorithm 3), with two cache configurations. The first configuration used a very small cache, with one line of 256 bytes. The second configuration used a larger cache, with 32 lines of 256 bytes (the size of the entire *dist* array).

For KNN with $k = 5$, the performance and resource utilization are listed in TABLE 3. We can see that there is no performance advantage from our caches in this case, because each distance is used exactly once by the kernel in the inner most loop, and because the HLS tool managed

TABLE 3. Performance and resource utilization for various implementations of KNN with $k = 5$.

Implementation	Ext. Mem.	On-chip Mem.	Cache	
Cache size (byte)	—	—	128	8K
Exec. time (ms)	**0.107**	**0.116**	0.162	**0.117**
Power (W)	**0.7**	**0.53**	0.777	1.114
Energy (mJ)	**0.075**	**0.061**	0.126	0.13
BRAM	2	8	16	16
DSP	4	4	4	4
LUT	8015	2454	4271	25596
FF	9620	3766	10026	77979

to understand the very simple access pattern and created a burst access to the external memory the same way as our cache does it. Note that this design is *a worst-case example for our cache methodology*, since input data reuse is trivial except for caches that are at least as large as the datasets. However, it demonstrates that the performance overhead of our caches (8KB) is minimal (about 9%).

The execution time of the external memory implementation is about 8% faster than both the on-chip memory and the best cache implementation. Note that an overly small cache (128B) has a significant performance penalty (about 50%) in this case. This shows that cache type and size must be carefully selected for each target application. The miss ratio for various data sizes and cache configurations, obtained via functional simulation in C++, are reported in Fig. 8. As shown, the number of lines in the cache has no effect on the miss ratio till the cache can hold the entire dataset. The miss ratio is inversely proportional to the line size.

C. BITONIC SORTING ALGORITHM

Like KNN, also this algorithm is memory-dominated and with limited data reuse. Nevertheless, without requiring almost any source code change our caches improved the performance, mostly by accessing the memory in bursts.

We performed RTL simulation of six total implementations, each sorting arrays with 128, 1024 and 4096 words

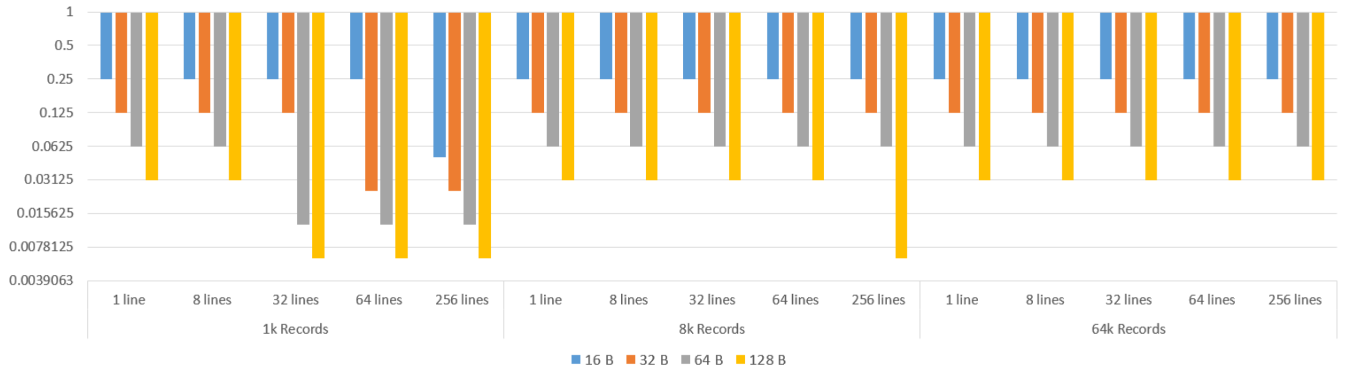


FIGURE 8. Miss ratios for different numbers of lines, data sizes and line sizes for KNN (log scale).

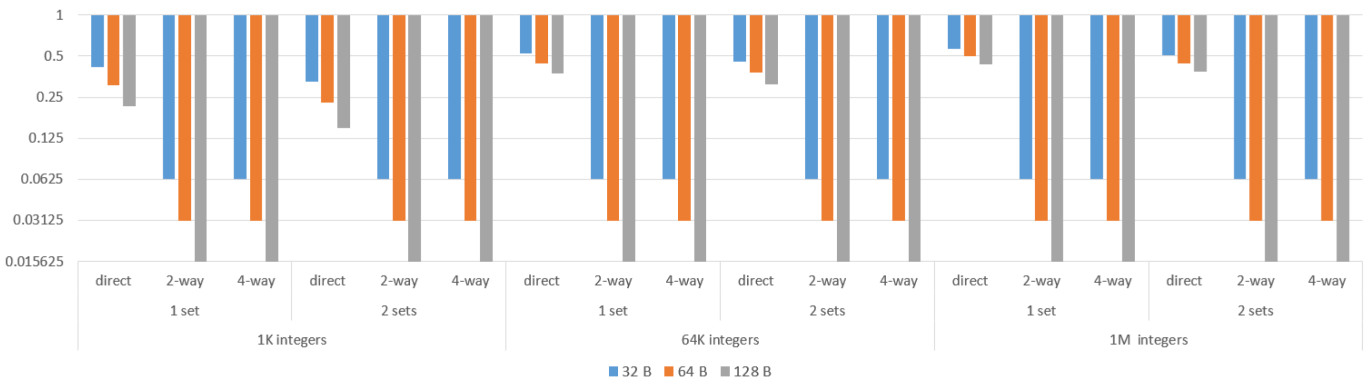


FIGURE 9. Miss ratios for different numbers of lines, data sizes and line sizes for bitonic sorting (log scale).

TABLE 4. Performance for various implementations of bitonic sorting applied to arrays with different sizes N, and using a cache line size of 64 bytes. L_{max}, in bytes, is the maximum on-chip memory used (when limited).

Array size	$N = 2^7$	$N = 2^{10}$	$N = 2^{12}$
External Mem.	0.702	11.03	62.98
Limited on-chip Mem.	0.333 ($L_{max} = 128$)	6.353 ($L_{max} = 256$)	46.13 ($L_{max} = 256$)
Full on-chip Mem.	**0.04**	**0.577**	**3.241**
Set-assoc. cache	0.494	7.571	42.93
1st opt. set-assoc. cache	0.287	4.473	25.36
2nd opt. set-assoc. cache	**0.0815**	**1.388**	**7.865**

filled with random integers. The three implementations without caches were discussed in Section IV-B. For the "Limited on-chip mem." implementation, which uses the limited on-chip memories to sort sub-arrays, we considered the maximum on-chip RAM sizes to be $L_{max} = 128$ bytes and 256 bytes. Note that we have to use these small sizes, because the RTL simulation is very slow. As usual, we also report results on miss ratios for larger arrays and caches in Figure 9. The last three "With caches" implementations were accelerated using various cache types and configurations. The first cache implementation was 2-way set-associative, with 128 total bytes and a line of 64 bytes. The second cache implementation used the same configuration but with a post-cache manual optimization, namely we replaced some write accesses with calls to the member function that assumes that the data to be written are already in cache and does not cause a flush. The third cache implementation added a manual prefetch loop before the array access code in the original implementation, thus avoiding the external memory loop latency in the main pipelined loop.

TABLE 5. Performance and resource utilization of various optimizations on bitonic sorting applied to arrays with size $N = 2^{10}$.

Implementation	Ext. Mem.	On-chip Mem.		Set-associative cache		
		$L_{\max} = 256$	Full	Orig.	1st opt.	2nd opt.
Initiation interval	20	4		50	28	3
Exec. time (ms)	11.03	6.353	**0.577**	7.571	4.473	**1.388**
Number of transfers	84136	488414	**128**	7040	7040	**3520**
Average size per transfer (byte)	4	4.16	64	64	64	**128**
Power (W)	**0.451**	0.683	**0.465**	1.534	1.179	2.155
Energy (mJ)	4.975	4.339	**0.268**	11.61	5.274	**2.991**
BRAM	1	1	2	16	16	31
LUT	1575	11633	1441	12546	8843	22142
FF	2045	10585	1971	22669	16882	31101
DSP	0					

Two configurations of the 2-way set-associative caches were implemented in order to test the effects of cache sizes on performance. One implemented a 128-byte 2-way set associative cache with a line size of 64 bytes, and the other one implemented a cache with a size of 256 bytes and a line size of 128 bytes.

TABLE 4 shows the performance of the implementations discussed above, sorting arrays with different lengths. As expected, the implementation with all data stored in external memory is the slowest. Transferring all data to very large on-chip memories has the best performance, about 20× faster. The other local memory implementation, with a limited maximum size ($L_{\max}$ = 128 and 256 bytes) is much less effective and achieves a speedup of about 2×. The speedup achieved by a 2-way set-associative cache without any post optimization is about 1.5×. With the first optimization scheme, the speedup can reach 2.5×. Finally, prefetching achieves 8× speedup and saves about 40% energy consumption.

Power, resource utilization and data transfer statistics for the array with size $N = 2^{10}$ are shown in TABLE 5.

The "Ext. mem." implementation keeps all data in external memory. It consumes the least power due to its simple architecture. It performed 85k data transfers, each reading or writing only 4 bytes, since in this case the HLS tool was not able to automatically infer burst accesses.

The "On-chip mem." implementation is much faster and achieves most of its performance gains by making only 128 data transfers of 64 bytes each, in burst mode.

The caches are also able to similarly reduce the total number of transfers and increase the burst size of each access. As mentioned above, the bitonic sorting kernels from which we started had no data reuse, so the caches help only by coalescing accesses in bursts. The implementation with the 2-way set-associative cache only required 7k memory transfers, each containing 128 bytes. The bottleneck for this implementation is the initiation interval of the innermost loop, which is 2.5× larger than in the "Ext. mem." implementation and 12.5× larger than in the "On-chip mem." and the best "With caches" implementations. There are two main reasons for this long initiation interval. First, there are two read operations and two write operations in each iteration. Even though the write operations never miss, the synthesis tool is not able to ignore the false dependencies between the writeback of a dirty line and the read which updates the line, in case of a read miss. Hence, the first optimization decreases the initiation interval by around 2×, while keeping the number of transfers essentially identical, thus improving performance by about 2×.

TABLE 6. Effect of cache sizes on the performance of bitonic sorting.

Array size	$N = 2^7$		$N = 2^{12}$	
Cache line size (byte)	64	128	64	128
Cache size (byte)	128	256	128	256
2nd opt. set-assoc.	**Exec. time (ms)**			
	0.1288	**0.0815**	11.28	**7.865**
	Device power (W)			
	1.263	2.041	1.253	2.146
	Energy consumption (mJ)			
	0.163	0.166	14.13	16.88

The second optimization used twice the total cache size, halved the number of transfers and managed to achieve an initiation interval of 3 by prefetching the data, and hence preventing the false memory access dependencies in the main loop.

TABLE 6 shows the execution time and device power required by the implementations with different line sizes for the two arrays respectively. Doubling the number of lines improves performance by about 1.5×, but also increases device power by the same factor. I.e., it improves performance and increases resource cost, but keeps total energy consumption essentially the same. As Fig. 9 shown, the miss ratio is dramatically reduced with a 2-way associative cache instead of a direct-mapped one. More than 2 ways or more than 1 set have no effect on the miss ratio.

TABLE 7. Performance and resource utilization of various optimizations applied to the Smith-Waterman algorithm with $N = 84$.

Implementation	Ext. Mem.	Full on-chip Mem.	Direct-mapped cache			R/W-only cache	
			Original	Opt. directives	Opt. func.	Original	Opt.
Init. interval	7	**2**	46	36	37	20	**3**
Exec. time (ms)	1.759	**0.1556**	1.714	1.353	1.389	0.8558	**0.1976**
Power (W)	**0.58**	**0.513**	4.297	2.507	3.754	2.364	1.598
Energy (mJ)	1.02	**0.0798**	7.365	3.392	5.214	2.023	**0.3158**
BRAM	3	6	60	60	60	60	60
DSP	0	1	0	0	0	0	1
LUT	3347	1900	90434	47283	75243	17288	10800
FF	4783	2672	121936	47993	121776	41471	28096

D. SMITH-WATERMAN ALGORITHM

The source code for the Smith-Waterman algorithm was based on the example code that is distributed with Xilinx SDAccel™. The host code generates two genome sequences with a length $N = 84$, then it sends them to the kernel and retrieves the score matrix and the direction matrix. Using these two matrices, the host code produces and verifies the alignment of the two genomes. We tested a total of seven implementations, belonging to the usual three classes. The "Ext. mem." and "On-chip mem." implementation, as usual, kept all the arrays and matrices in external and on-chip memory respectively. The "With caches" one used two 128-byte read-only direct-mapped caches for the two sequences and a 1 KiB read/write direct-mapped cache for the score matrix and direction matrix. The cache for the score matrix is the bottleneck of this algorithm, because it can be both read and written. The memory access dependencies discussed in the bitonic sorting case also affect the performance of this algorithm, due to the read and write operations in one loop iteration, with the resulting miss read after a dirty line write-back. In this case, some dependencies are true and some are false. As a first optimization, we used a directive to instruct the synthesis tool to ignore a false loop-carried dependency among memory accesses within the innermost loop.

As a second optimization, we replaced the writeback operation to the score matrix with a cache member function which can be used when the designer knows (or a data access analysis tool can infer) that the write operations access consecutive addresses, in order to boost performance.

As discussed, a write-only cache for the write operation at the current position of the score matrix and a read-only cache for the two reads from the previous line of the score matrix can reduce dramatically the initiation interval. Hence, the third implementation used a read-only direct-mapped cache and a write-only direct-mapped cache for the score matrix, instead of the unified cache used in the previous optimization. Of course, it can be used only for long sequences (i.e. when the sequence size is larger than the cache size) with a large score matrix, where the contents of the two caches can never overlap.

The last one used again the member function that assumes consecutive addresses when writing the 128-byte write-only cache (as in the second optimization).

TABLE 7 presents the performance and resource utilization of the seven implementations. As usual, keeping all the data on-chip achieves the best performance.

The use of the direct-mapped caches significantly reduced the external memory accesses. However, we see that the performance did not improve much due to the false loop-carried dependencies that were not ignored by the Vivado HLS synthesis tool, and that added up to a large initiation interval.

However, we eliminated these dependencies either using synthesis tool directives, or by separating the read/write array (and its cache) into one read-only and one write-only. Ignoring the dependencies using a directive reduced the initiation interval by $1.3\times$ and improved the performance by the same factor, while splitting the arrays and caches improved the performance by $2\times$.

Finally, the last implementation reduced the initiation interval to 3 (and increased the performance accordingly), which is essentially the same as the "ideal" implementation, which uses only on-chip memories. Caches have a higher resource utilization, of course, but it is never as high as storing all data on-chip. Note also that energy consumption of the best cache implementation is only 1/3 the energy needed to access all data in off-chip memory, again not considering the energy required by the off-chip memory itself, which would be much smaller in the cache case.

The miss ratios for larger sequences are reported in Fig. 10 and Fig. 11. As shown in Fig. 10, the performance of the direct-mapped cache is acceptable only when the cache size is twice the sequence size (i.e. it contains two rows of the score matrix). The miss ratios of the split read and write caches only depend on the line size rather than on the number of lines, as shown in Fig. 11.

E. LUCAS-KANADE ALGORITHM

While real-life algorithm applications compute the optical flow on relatively large images (up to several megapixels), in this section we report RTL simulation results for small images, of 64×36 pixels, each pixel represented on 8 bits. We also report miss ratios for more realistic image and cache sizes, from functional simulation in C++.

As before, the "Ext. mem." and the "On-chip mem." implementations used only off-chip and on-chip memories.

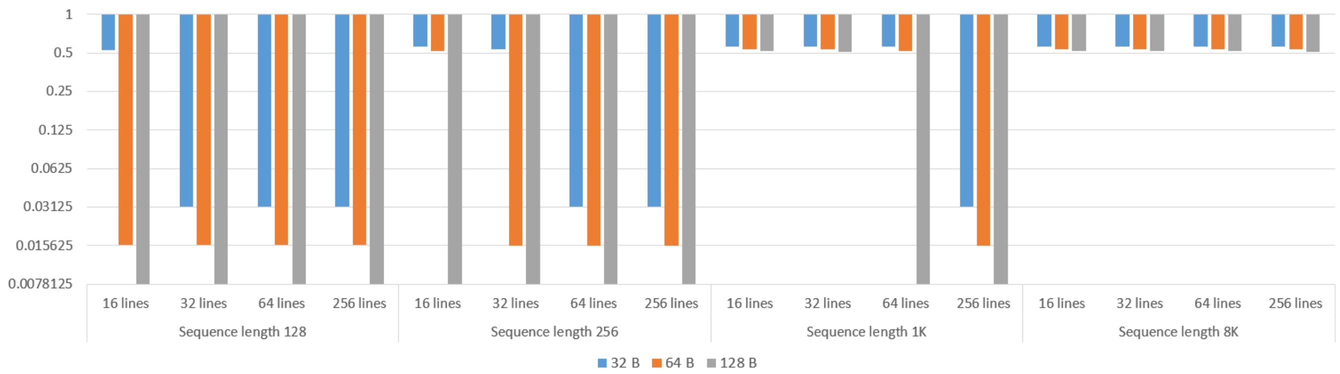


FIGURE 10. Miss ratios for different unified cache numbers of lines, data sizes and line sizes for the score matrix of Smith-Waterman (log scale).

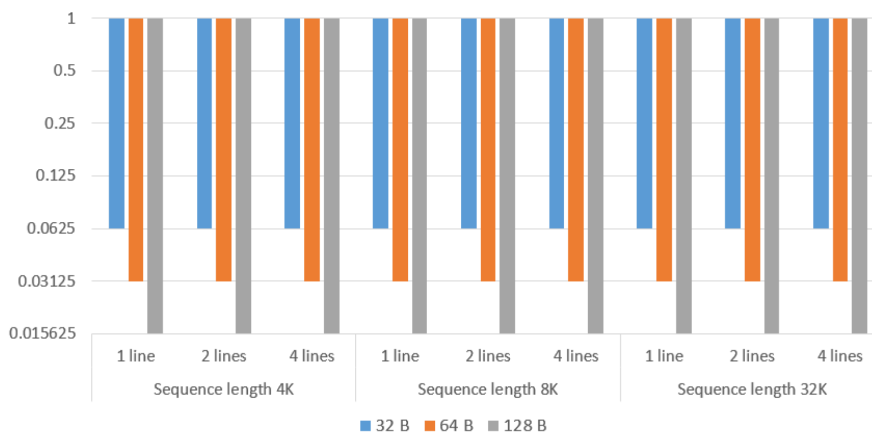


FIGURE 11. Miss ratios for different split read and write cache numbers of lines, data sizes and line sizes for the score matrix of Smith-Waterman (log scale).

TABLE 8. Performance and resource utilization for various implementations of the Lucas-Kanade algorithm.

Frame size	**64x36**, window size = 5				
Implementation	**Ext. Mem.**	**Full on-chip Mem.**	**Small caches**	**Large caches**	**Opt. caches**
Hit ratio (%)	—	—	99.98	99.99	99.7
Execution time (ms)	43.78	**5.636**	12.01	11.7	**9.2**
Initiation interval	5	3	5	5	1
Number of transfers	1677312	**4680**	51136	23990	**23702**
Average size per transfer (byte)	4	4.9	64	126.5	**127**
Power (W)	**0.689**	**0.693**	**1.588**	1.759	1.737
Energy (mJ)	30.16	**3.906**	18.58	20.58	**15.98**
BRAM	2	4	31	45	37
DSP	21	21	21	21	21
LUT	5888	5669	27631	45366	35254
FF	7846	7604	35376	56140	46383

We then developed several optimized "With caches" implementations. The first one used a 64-byte one-line write-only direct-mapped cache for the output vector, a 64-byte one-line read-only direct-mapped cache for the second image (which is read once in each innermost loop iteration), and a 256-byte four-line read-only direct-mapped cache for the first image (which is read five times in each iteration). The size of the read-only direct-mapped cache used for the first image is sufficient to store three lines of the image. Hence, it acts essentially as a line buffer, but without, as usual, requiring any manual code change.

The second one doubled the line size of the two read-only direct-mapped caches with respect to the first one, thus doubling both the burst size and the cache size.

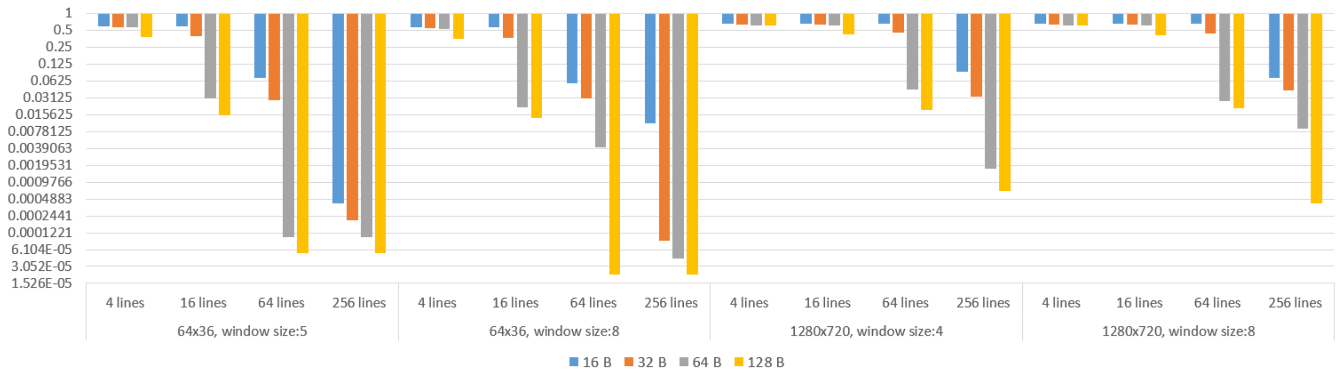


FIGURE 12. Miss ratios for different numbers of lines, data sizes and line sizes for the input image of Lucas-Kanade (log scale).

TABLE 9. Summary of the performance of the five algorithms with and without caches.

	Matrix multiplication	K-Nearest Neighbors	Bitonic sorting	Smith-Waterman	Lucas-Kanade
Exec. time of Ext. Mem. (ms)	0.241	0.107	11.03	1.759	43.78
Energy of Ext. Mem. (mJ)	0.122	0.075	4.975	1.02	30.16
Exec. time of the cache (ms)	0.031	0.117	1.388	0.1976	9.2
Energy of the cache (mJ)	0.037	0.13	2.991	0.3158	15.98
Exec. time ratio of the cache	**0.129**	1.093	**0.126**	**0.112**	**0.21**
Energy ratio of the cache	**0.303**	1.733	**0.601**	**0.310**	**0.53**

The third one used a post optimization that assumes access to consecutive addresses, as described in the previous section, with the goal to reduce the initiation interval. Note that its effectiveness, as before, is reduced by a limitation of the Vivado HLS tool, which is unable to flatten a loop inside a pipelined loop (as in the matrix multiplication case).

The performance and resource utilization of the four implementations are listed in TABLE 8. The "Ext. mem." implementation, which keeps all the images in the external memory, has a very long execution time because it accesses the external memory 1.6M times. However, this algorithm (like most computer vision, machine learning and artificial intelligence algorithms) exhibits very high levels of data reuse. In particular, each pixel of the first image is accessed many times by this algorithm. Hence, the "On-chip mem." implementation that stores all data in the on-chip memory maximizes data reuse and requires only 4.7k transfers from/to the external memory. In addition, the on-chip memory can be accessed using two ports, so the initiation interval is reduced from 5 to 3. In summary, this implementation improves performance by about 8.5×.

Two factors improve significantly the performance of the optimizations using our caches. First of all, as in the case of bitonic sorting, the caches use bursts to increase the data size of each transfer. Second, the caches exploit the very significant amount of data reuse of this algorithm. As shown in the table, even a very small cache (comparable in size to a line buffer, which is a standard implementation for this kind of algorithms) speeds up kernel execution by 3.6× while consuming only 60% of the energy.

The miss ratio of the most frequently accessed array is only 0.02% and it required only 50k data transfers of about 64 bytes each (the ideal lower bound is about 5k transfers). The larger cache doubles the transfer size and halves the miss ratio to 0.01%. The initiation interval is reduced to 1 clock cycle for the last implementation.

The last optimization accelerates the algorithm by 4.8× and reduces energy consumption by 2× compared to the "Ext. mem." implementation. Although both cache sizes are Pareto-optimal, the smaller cache probably offers the most effective cost-performance trade-off. The miss ratios for various frame sizes, window sizes and cache configurations are reported in Fig. 12. The lowest miss ratio can be 0.000015, leading to excellent data reuse. Even for large frames and large windows, relatively small caches can obtain a low miss ratio (around 0.1%).

F. SUMMARY

For all five algorithms considered, each drawn from a very different application domain, we summarize in TABLE 9 and Fig. 13 the execution time and energy consumption for the best implementations, with and without caching. Except for KNN, which has no data reuse and has very simple addressing patterns that allow Vivado HLS to infer external memory burst accesses, the best implementations with caches improve execution speed by up to 8× and reduce energy by about 2×.

Fig. 14 shows the performance of our caches with respect to the *ideal lower bound* represented by the transfer of all data to a hypothetical very large on-chip memory. Except for KNN, for the reason that was already mentioned, our

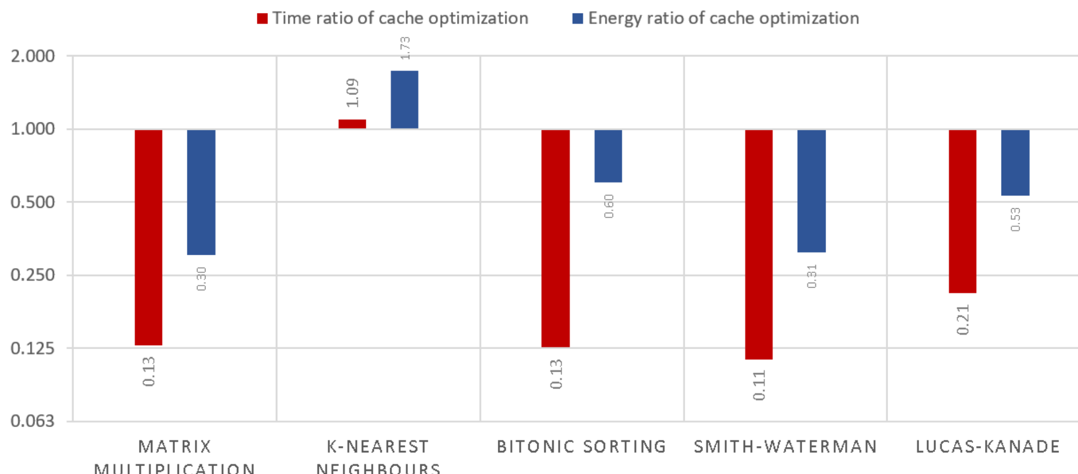


FIGURE 13. Summary of the performance improvements achieved by the caches (log scale).

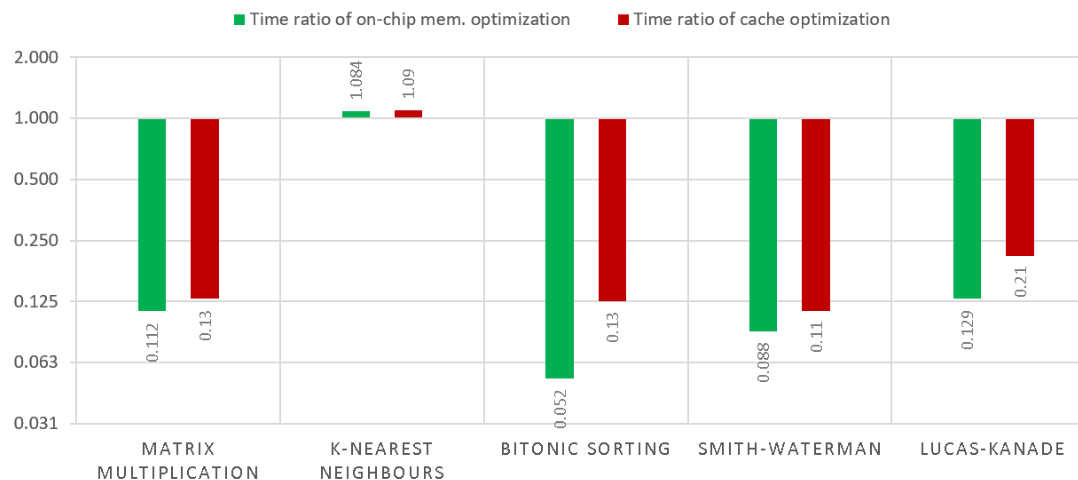


FIGURE 14. Ratios of execution time of two kinds of optimizations. (log scale).

caches obtained very similar performance with a realistic memory occupation. Manual optimization of off-chip and on-chip memory, which typically requires extensive code restructuring, can achieve comparable or better performance using the same amount of local memory as our caches, but it requires a large amount of manual optimization work.

VI. CONCLUSION

This article introduces a new methodology for optimizing the memory-intensive algorithms by using inline caches that are synthesized from a C++ model onto an FPGA. These caches are designed using a synthesizable style supported by most high-level synthesis tools, not just the Vivado HLS tool that was used to implement several design examples in this work. These caches can be easily used by designers, since they follow traditional cache design concepts and categories, e.g., direct-mapped or set-associative. We provide several variants that can be adapted to different use contexts (e.g., read-only, write-only, etc.). They also include design aids (e.g., memory access tracing capabilities, miss ratio reporting) that can be used to ease cache size and architecture optimization. Note that since the caches are modeled in C++, both their functionality and any manual optimizations aimed at further improving performance *can be fully verified in C/C++*, without requiring synthesis or RTL simulation.

The new methodology was then applied to five algorithms from very different application areas such as machine learning, data sorting, genomics and computer vision. The original algorithms with a few basic optimizations, such as loop pipelining, were used as a performance, resource usage and energy consumption reference. We also considered an ideal "best case" implementation, in which all data could fit on-chip. We then showed how using our caches, with different parameters and some further optimizations, could

significantly improve performance without requiring the extensive code changes that are typically required to manually optimize on-chip memory usage. In order to fairly compare these implementations, all of them keep the same computation architecture (loop pipelining, unrolling, etc.), only changing the memory architecture.

From the performance comparison results, we can conclude that the use of our inline caches can accelerate most memory-intensive algorithms, except for those which do not exhibit a significant amount of data reuse, and for which high-level synthesis tools cannot automatically infer memory access bursts. In summary, our cache implementations improved performance by up to 8× energy by about 2×, achieving comparable results to the best available manual optimizations of the on-chip memory architecture, while requiring a much shorter design time.

Currently a designer needs to manually choose the types and the sizes of the caches and the specific member functions used to access the cache, in order to achieve the best performances for their applications. Our future work will focus on automating both application profiling, in order to select the best cache architecture for each DRAM array, and static address analysis, in order to infer which accesses are always "hits".

APPENDIX A
CODE OF THE INLINE DIRECT-MAPPED CACHE

This appendix shows the most significant fragments of the code of the template class of the inline direct-mapped cache. The `operator[]` method is overloaded and an inner class is used to differentiate between the methods to be called when the operator is used in a left-hand-side or right-hand-side context.

```
template<typename T,int SET_BITS,int LINE_BITS>
class Cache {
private:

static const int CACHE_SETS = 1 << SET_BITS;
static const int LINE_SIZE = 1 <<LINE_BITS;
static const int DATA_BITS = sizeof(T) * 8;
typedef ap_uint<DATA_BITS> LocalType;

class inner {
  public:
  inner(Cache *cache,  const int addr):
    cache(cache),addr(addr) {}
  operator T() const{
    return cache->get(addr);
  }
  void operator= (T data){
    cache->set(addr, data);
  }
  private:
  Cache *cache;
  const int addr;
};

public:
typedef ap_uint<DATA_BITS*LINE_SIZE> DataType;
Cache(DataType * mem):ptr_mem(mem){...}
inner operator[](const int addr) {
  return inner(this, addr);
}
~Cache(){/* writeback code*/ ...}

private:

int requests, hits;
DataType * const ptr_mem;
DataType array[CACHE_SETS];
ap_uint<32-SET_BITS-LINE_BITS>
tags[CACHE_SETS];
bool valid[CACHE_SETS], dirty[CACHE_SETS];

T get(const int addr) {
  const ap_uint<32 - SET_BITS-LINE_BITS> tag
= addr >> (SET_BITS+LINE_BITS);
  const ap_uint<SET_BITS> set_i
= (addr >> LINE_BITS);
  const ap_uint<LINE_BITS> block = addr;
  requests++;
  bool match = tags[set_i] == tag;

  DataType dt;
  if(valid[set_i] && match) {
    hits++;
    dt = array[set_i];
  } else {
    dt = ptr_mem[addr >> LINE_BITS];
    array[set_i] = dt;
  }
  tags[set_i] = tag;

  valid[set_i] = true;
  LocalType data = lm_data::GetData<DATA_BITS,
   DATA_BITS * LINE_SIZE,
   LINE_BITS>::get(dt, block);
  return *(T*)&data;
}

void set(const int addr, const T& data) {
  const ap_uint<32 - SET_BITS-LINE_BITS> tag =
   addr >> (SET_BITS+LINE_BITS);
  const ap_uint<SET_BITS> set_i =
   (addr >> LINE_BITS);
  const ap_uint<LINE_BITS> block = addr;

  requests++;

  bool match = tags[set_i] == tag;
  if(valid[set_i] && match) {
    hits++;
  } else {
    if(dirty[set_i]) {
      ap_uint<32> paddr = tags[set_i];
      ptr_mem[paddr<<SET_BITS | set_i] =
      array[set_i];
    }
    array[set_i] = ptr_mem[addr >> LINE_BITS];
  }
```

```
  LocalType ldata = *(LocalType*)&data;

  array[set_i] = lm_data::SetData<DATA_BITS,
   DATA_BITS * LINE_SIZE,
   LINE_BITS>::
   set(array[set_i], ldata, block);
  tags[set_i] = tag;
  valid[set_i] = true;
  dirty[set_i] = true;
}
};
```

APPENDIX B
ORIGINAL AND MODIFIED CODE OF MATRIX MULTIPLICATION

The basic matrix multiplication code contains three nested loops over rows, columns and inner product iteration. The innermost loop can be pipelined or unrolled as desired, by setting tool-specific directives.

```
void mat_mult(int *a, int *b, int *c) {
  for (int row=0;row<rank;row++){
    for (int col=0;col<rank;col++){
      int tmp=0;
      for (int index=0;index<rank;index++) {
#pragma HLS pipeline
        int aIndex = row*rank + index;
        int bIndex = index*rank + col;
        tmp += a[aIndex] * b[bIndex];
      }
      c[row*rank + col] = tmp;
    }
  }
}


typedef Cache<int, 0, a0> CacheTypeA;
typedef Cache<int, b0, b1> CacheTypeB;
typedef Cache<int, 0, c0> CacheTypeC;

void mat_mult(CacheTypeA::DataType *a_orig,
  CacheTypeB::DataType *b_orig,
  CacheTypeC::DataType *c_orig) {
  CacheTypeA a(a_orig);
  CacheTypeB b(b_orig);
  CacheTypeC c(c_orig);

  for (int row=0;row<rank;row++){
    for (int col=0;col<rank;col++){
       int tmp = 0;
       for (int index=0;index<rank;index++) {
#pragma HLS PIPELINE
         int aIndex = row*rank + index;
         int bIndex = index*rank + col;
         tmp += a[aIndex] * b[bIndex];
       }
       c[row*rank+col] = tmp;
    }
  }
}
```

REFERENCES

[1] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2014, pp. 10–14.

[2] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[3] D. Xie, J. Lai, and J. Tong, "A high utilization rate routing algorithm for modern FPGA," in *Proc. 9th Int. Conf. Solid-State Integr.-Circuit Technol. (ICSICT)*, Oct. 2008, pp. 2333–2336.

[4] *SDAccel Environment Optimization Guide*, Xilinx Inc, San Jose CA, USA, Mar. 2017.

[5] M. Fingeroff, *High-Level Synthesis: Blue Book*. Bloomington, IN, USA: Xlibris Corporation, 2010.

[6] L. Ma, F. B. Muslim, and L. Lavagno, "High performance and low power Monte Carlo methods to option pricing models via high level design and synthesis," in *Proc. Eur. Modelling Symp. (EMS)*, Pisa, Italy, Nov. 2016, pp. 157–162.

[7] F. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017.

[8] F. Winterstein, K. Fleming, H.-J. Yang, S. Bayliss, and G. Constantinides, "MATCHUP: Memory abstractions for heap manipulating programs," in *Proc. ACM/SIGDA Int. Symp. Field-Programm. Gate Arrays*, 2015, pp. 136–145.

[9] J. Keinert *et al.*, "SystemCoDesigner—An automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, 2009, Art. no. 1.

[10] J. Cong, K. Gururaj, H. Huang, C. Liu, G. Reinman, and Y. Zou, "An energy-efficient adaptive hybrid cache," in *Proc. IEEE Int. Symp. Low Power Electron. Design (ISLPED)*, Aug. 2011, pp. 67–72.

[11] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. 17th Annu. Int. Symp. Comput. Archit.*, 1990, pp. 364–373.

[12] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The V-Way cache: Demand-based associativity via global replacement," in *Proc. 32nd Int. Symp. Comput. Archit. (ISCA)*, 2005, pp. 544–555.

[13] D. Rolán, B. B. Fraguela, and R. Doallo, "Adaptive line placement with the *set balancing cache*," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2009, pp. 529–540.

[14] E. Matthews, N. C. Doyle, and L. Shannon, "Design space exploration of L1 data caches for FPGA-based multiprocessor systems," in *Proc. ACM/SIGDA Int. Symp. Field-Programm. Gate Arrays (FPGA)*, New York, NY, USA, 2015, pp. 156–159. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689083

[15] G. Kalokerinos *et al.*, "FPGA implementation of a configurable cache/scratchpad memory with virtualized user-level RDMA capability," in *Proc. Int. Symp. Syst., Architectures, Modeling, Simulation (SAMOS)*, Jul. 2009, pp. 149–156.

[16] S. Cheng, M. Lin, H. J. Liu, S. Scott, and J. Wawrzynek, "Exploiting memory-level parallelism in reconfigurable accelerators," in *Proc. IEEE 20th Annu. Int. Symp. Field-Programm. Custom Comput. Mach. (FCCM)*, Apr. 2012, pp. 157–160.

[17] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, "Leap scratchpads: Automatic memory and cache management for reconfigurable logic," in *Proc. 19th ACM/SIGDA Int. Symp. Field Programm. Gate Arrays*, 2011, pp. 25–28.

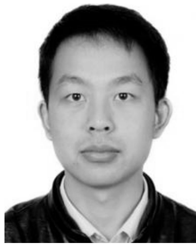
[18] J. Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski, "Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems," in *Proc. IEEE 20th Annu. Int. Symp. Field-Programm. Custom Comput. Mach. (FCCM)*, Apr. 2012, pp. 17–24.

[19] A. Putnam *et al.*, "Performance and power of cache-based reconfigurable computing," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 395–405, 2009.

[20] F. Winterstein, K. Fleming, H.-J. Yang, J. Wickerson, and G. Constantinides, "Custom-sized caches in application-specific memory hierarchies," in *Proc. Int. Conf. Field Programm. Technol. (FPT)*, Dec. 2015, pp. 144–151.

[21] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graph. Hardw.*, 2004, pp. 133–137.

[22] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *Proc. 2005 ACM/SIGDA 13th Int. Symp. Field-programm. Gate Arrays*, 2005, pp. 86–95.
[23] Yu. (Nov. 2014). *Overload the Brackets Operator to Perform Complex Operations*. [Online]. Available: https://argcv.com/articles/3228.c
[24] F. B. Muslim, A. Demian, L. Ma, L. Lavagno, and A. Qamar, "Energy-efficient FPGA implementation of the k-nearest neighbors algorithm using OpenCL," *Ann. Comput. Sci. Inf. Syst.*, vol. 9, pp. 141–145, Oct. 2016.
[25] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, May 2009, pp. 1–10.
[26] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Molecular Biol.*, vol. 147, no. 1, pp. 195–197, 1981.
[27] A. G. Seliem, W. A. El-Wafa, A. Galal, and H. F. Hamed, "Parallel smith-waterman algorithm hardware implementation for ancestors and offspring gene tracer," in *Proc. World Symp. Comput. Appl. Res. (WSCAR)*, Mar. 2016, pp. 116–121.
[28] B. Harris, A. C. Jacob, J. M. Lancaster, J. Buhler, and R. D. Chamberlain, "A banded smith-waterman FPGA accelerator for mercury BLASTP," in *Proc. Int. Conf. Field Programm. Logic Appl. (FPL)*, Aug. 2007, pp. 765–769.
[29] L. Ligowski and W. Rudnicki, "An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, May 2009, pp. 1–8.
[30] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proc. 7th Int. Joint Conf. Artif. Intell.*, 1981, pp. 674–679.
[31] J.-Y. Bouguet, "Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm," *Intel Corp.*, vol. 5, nos. 1–10, p. 4, 2001.



LIANG MA (S'17) received the M.S. degree (Hons.) from the Politecnico di Torino, Italy, in 2014, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunications under the supervision of Prof. L. Lavagno. His research interests focus on the high-level synthesis, the electronic system level design and the low-power high-performance computing.

LUCIANO LAVAGNO (SM'89) received the Ph.D. degree in electrical engineering and computer science from U.C. Berkeley in 1992. He was an Architect with the POLIS HW/SW co-design tool. From 2003 to 2014, he was an Architect with the Cadence CtoSilicon high-level synthesis tool. Since 1993, he has been a Professor with the Politecnico di Torino, Italy. He co-authored four books and over 200 scientific papers. His research interests include synthesis of asynchronous circuits, HW/SW co-design, high-level synthesis, and design tools for wireless sensor networks.

MIHAI TEODOR LAZARESCU (M'98) received the Ph.D. degree from Politecnico di Torino, Italy, in 1998. He was a Senior Engineer with Cadence Design Systems, founded several startups and currently serves as an Assistant Professor with the Politecnico di Torino. He co-authored over 40 scientific publications and several books. His research interests include sensors for indoor localization, reusable WSN platforms, high-level hardware/software co-design and high-level synthesis of WSN applications.

ARSLAN ARIF (S'17) received the master's degree from the National University of Sciences and Technology, Pakistan. He is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunication, Politecnico Di Torino, Italy. His current research interests include high-level synthesis, computation accelerators (FPGA and GPU) and Internet of Things.

• • •