Article

Improving Performance of Hardware Accelerators by Optimizing Data Movement: A Bioinformatics Case Study

Peter Knoben and Nikolaos Alachiotis *

Faculty of EEMCS, University of Twente, 7522 NB Enschede, The Netherlands; p.a.h.knoben@student.utwente.nl
* Correspondence: n.alachiotis@utwente.nl

Abstract: Modern hardware accelerator cards create an accessible platform for developers to reduce execution times for computationally expensive algorithms. The most widely used systems, however, have dedicated memory spaces, resulting in the processor having to transfer data to the accelerator-card memory space before the computation can be executed. Currently, the performance increase from using an accelerator card for data-intensive algorithms is limited by the data movement. To this end, this work aims to reduce the effect of data movement and improve overall performance by systematically caching data on the accelerator card. We designed a software-controlled split cache where data are cached on the accelerator and assessed its efficacy using a data-intensive Bioinformatics application that infers the evolutionary history of a set of organisms by constructing phylogenetic trees. Our results revealed that software-controlled data caching on a datacenter-grade FPGA accelerator card reduced the overhead of data movement by 90%. This resulted in a reduction of the total execution time between 32% and 40% for the entire application when phylogenetic trees of various sizes were constructed.

Keywords: hardware accelerator; FPGA; data movement; memory bottleneck; phylogenetics

check for updates

Citation: Knoben, P.; Alachiotis, N. Improving Performance of Hardware Accelerators by Optimizing Data Movement: A Bioinformatics Case Study. *Electronics* **2023**, *12*, 586. https://doi.org/10.3390/electronics12030586

Academic Editors: Leonardo Pantoli, Egidio Ragonese, Paris Kitsos, Gaetano Palumbo and Costas Psychalinos

Received: 11 December 2022
Revised: 15 January 2023
Accepted: 19 January 2023
Published: 24 January 2023

1. Introduction

Hardware accelerators are increasingly used to reduce computation times for time-intensive algorithms. However, they do not always share memory space with the host processor, e.g., accelerator cards in data centers. When the host processor and the accelerator do not share the same memory space, as depicted in Figure 1, data has to be transferred to the accelerator's dedicated memory prior to initiating computations on the accelerator. Data transfers in memory-bound applications are time-expensive [1], which is particularly noticeable in data-intensive algorithms and can lead to a point where the use of a hardware accelerator has no positive effect on the performance.

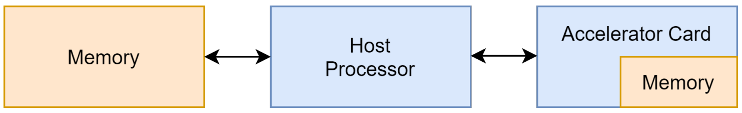

Figure 1. Block diagram of memory architecture for host processor with accelerator card.

Two possible solutions to optimize data movement include: (a) increasing the transfer speed, and (b) decreasing the number of data transfers. Assuming that applications already utilize the full bandwidth, it is very challenging to increase transfer rates without changing the hardware. Reducing the number of data transfers, on the other hand, is a solution that is highly dependent on the type of application. Efforts to reduce the number of data transfers when accelerating applications have been reported in the literature. For Deep Neural Networks (DNNs), for instance, several works have come across a memory bottleneck in DNN research [2–4]. To reduce this effect, a hierarchical memory management design is created. This memory management design is based on the different layers in DNN and shows promising results in reducing the number of data transfers [4]. A hierarchical memory can improve performance in data-intensive algorithms and is worth investigating in a more systematic and application-agnostic way.

To the best of the authors' knowledge, hierarchical memory management designs, such as caching, are not yet tested in a generic approach for an FPGA-based hardware acceleration of data-intensive algorithms. SemCache [5] explores the concept of software caching to reduce the number of data transfers between a CPU and a GPU. DyManD [6] implements a memory allocation system and a run-time library that uses a software cache for GPUs. Asai et al. [7] present an extension to IBMSparkGPU (a framework for big data processing on GPUs) that avoids redundant data transfers between the CPU and the GPU without any code modifications. dlmCL [8] unifies the host and GPU device memory into one object through address mapping. OmpMemOpt [9] relies on compiled techniques to reduce the number of data transfers by applying a data flow analysis during compile time for partial redundancy elimination. Wei et al. [2] propose a Layer Conscious Memory Management framework (LCMM) for Deep Neural Networks (DNN) implemented on an FPGA.

To reduce the negative effect of data movement on overall accelerator performance, we focus here on reducing the number of data transfers by exploiting standard caching techniques. Caching is already an established and widely used concept in processor design. Most modern processor cores have a form of cache memory implemented in the hardware. In this work, we implement this core concept in software and showcase a systematic and novel way, to the best of the authors' knowledge, to alleviate the problem of data movement between a host processor and dedicated accelerator cards. Our approach extends the existing memory hierarchy of a computing system to include, transparently to the application, the memory space of an accelerator card. This allows the memory on the accelerator to not only be used as a buffer for a compute kernel, but also as an additional memory space where data are temporarily stored, exploiting temporal and spatial locality.

Using a real-world application as a case study, we assess standard caching techniques and various related design choices, e.g., cache size, level of associativity, replacement policies, etc., which are currently used in processor design. We target the problem of accelerating phylogenetic-tree reconstruction [10] that constructs large phylogenetic trees to infer the relationship between organisms based on their genetic sequence data. Several accelerator architectures have been previously designed to boost the performance of phylogenetic tree reconstruction. Pratas et al. [11] accelerate the calculation of the PLF within MrBayes on a GPU. Zhou et al. [12] propose an improvement over the work by Pratas et al. [11]. While the work by Pratas et al. mainly focuses on the GPU-side computations, this work adopts a more hybrid approach where the CPU performs computations in parallel with the GPU. Zierke and Bakos [13] present an FPGA-accelerated solution based on MrBayes [14]. For likelihood calculations, the internal nodes of the tree are processed via a post-order traversal with minimal intervention from the host to reduce CPU-FPGA communication overheads. The solution utilizes onboard memory to cache the output vectors of the computations to minimize host-FPGA communication. Alachiotis et al. [15] accelerate the Phylogenetic Likelihood Function (PLF) using FPGAs by exploiting data dependencies between subsequent hardware calls in combination with double buffering. Malakonakis et al. [16] implement the complete RAxML algorithm on a hybrid system. The calculation of the PLF is done on an FPGA while the rest of the algorithm runs on the host CPU.

In this work, we targeted an AWS EC2 F1 hardware instance with an FPGA accelerator card. We observed that the transfer time of host-to-accelerator data transfers was reduced by 6×. In our setup, we only cached the accelerator input data, which resulted in up to a 40% improvement in the overall accelerator throughput perceived by the host application.

2. Materials and Methods

2.1. System Overview

In our proposed approach for systematic software caching to improve the performance of hardware accelerators, the cache controller and cache memory are separated. The host processor is the master for all communication in this system. It initiates all data transfers both to and from the accelerator card. For this reason, the software cache controller runs on the host processor. Unlike hardware caches, however, the data is stored on the accelerator memory and is separated from the rest of the cache controller. Our separated-cache design is depicted in Figure 2. The host processor keeps track of the different entries in the cache memory and has an additional field that stores a pointer to the location of the (cached) data in the accelerator card's memory space. Figure 3 shows a block diagram of the system with the software cache. It is similar to the block diagram previously shown in Figure 1, but the memory space on the accelerator card holds the cache memory whereas the host processor performs the cache control.

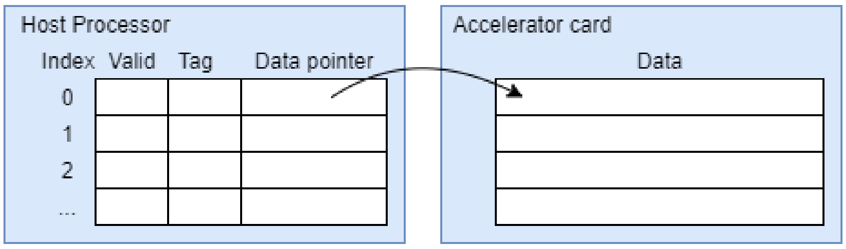

Figure 2. Cache separation between the host and the accelerator. All cache-control logic is implemented in software on the host processor whereas the actual caching of data happens on the main memory on the accelerator card.

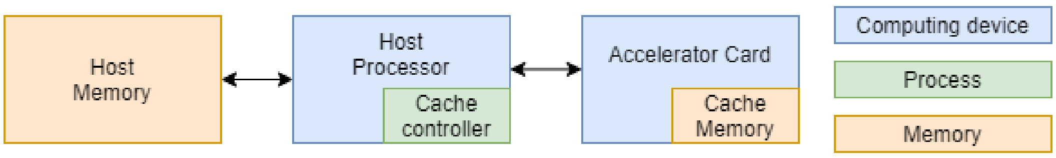

Figure 3. Block diagram of the architecture for host and accelerator with caching.

2.2. Generality of the Design

To ensure that our design supports different algorithms, there must be some level of control for the user's application to exploit. There are multiple different configurations and replacement policies that define the functionality of a cache. For each application, which combination of cache configuration and replacement policy will perform best will differ. For this reason, our implementation supports multiple cache configurations and replacement policies to allow the developer to choose the most suitable one for the application at hand. Our design also allows the application to define the block size for the data to be stored. For tree-based algorithms, like the phylogenetics application we used as a case study in this work, the block size is determined by the data size for each node of the phylogenetic tree. Since this caching method is aimed at data-intensive algorithms, this most likely will result in large block sizes, on the order of MB or even GB.

Depending on the target platform, the amount of onboard memory differs. Therefore, our cache design accommodates different memory sizes by defining the number of cache lines the cache will have. One more setting that the user can control is the tag size. In practice, the memory space required for the tag can be several orders of magnitude smaller than the block size (application-specific cache blocks can be several GB in size, as can be the case in our Bioinformatics case study when complex organisms like humans are investigated) and is negligible. Table 1 provides an overview list of all settings that can be set by the user. The supported replacement policies used to replace a cache line with a new one when the cache is full are: Random, FIFO (First In First Out), LRU (Least

Recently Used), MRU (Most Recently Used), LFU (Least Frequently Used), and MFU (Most Frequently Used). For example, LRU will replace the least recently used cache line whereas LFU will replace the least frequently used cache line.

Table 1. Overview of software-cache configuration settings.

Setting	Value
Cache configuration	Direct-mapped, 2-way set/4-way set/fully associative
Replacement policy	Random, FIFO, LRU, MRU, LFU, MFU
Block size	Number of bytes
Cache size	Number of cachelines
Tag size	Number of bits

2.3. Implementation and OpenCL Interface

Our software-cache implementation is technology-independent and can thus support different accelerator cards, e.g., FPGAs and GPUs. There already exist many implementations of cross-platform applications. With Open Computing Language (OpenCL), development times are significantly reduced compared to traditional low-level languages resulting in the widespread use of OpenCL [17]. OpenCL is a framework for writing code that is executed on multiple platforms. It is supported by CPUs, GPUs, FPGAs, and other hardware accelerators, making it an ideal tool for the front-end interface of the software cache. Our implementation of the software-cache control is done in C to allow for performance optimizations that minimize the overhead of the cache controller. The C implementation and the interaction with standard OpenCL routines for data movement are hidden behind an OpenCL-like interface to control cross-platform communication. This way, it can be instantiated on any OpenCL-supported device while all the complexity of deploying the software cache in a source code is hidden behind standard OpenCL routines for data transfers, e.g., `clEnqueueReadBuffer`. In the following, we present the basic functions of our software-cache implementation.

2.3.1. `CreateCache`

The `CreateCache` function is a constructor, described in Listing 1.

Listing 1. CreateCache function header.

```
struct Cache_t * CreateCache(
    int numberOfCacheLines,
    int dataSize,
    int tagSize,
    enum CacheConfiguration_t config,
    enum ReplacementPolicy_t policy);
```

This creates the cache controller. This function allocates memory space based on the arguments given and returns a pointer of type `Cache_t`. The main `Cache_t` structure for the software cache is described in Listing 2.

Listing 2. Cache_t struct declaration.

```
typedef struct Cache_t {
    int tagSize;
    int dataSize;
    int numberOfLinesPerSet;
    int numberOfSets;
    enum CacheConfiguration_t config;
    CacheLine_t ** cacheLine;
    enum ReplacementPolicy_t policy;
} Cache_t;
```

It contains all the parameters of the cache given by the user. When `CreateCache` is called, the allocation of the software cache happens on both the accelerator card, for the actual data, and on the host memory for all control data. The pointer returned by `CreateCache` is used as an input argument for the other functions. The benefit of this approach is that multiple cache instances can exist and function at the same time within the same application, each being uniquely identified by a pointer of type `Cache_t`.

2.3.2. `FreeCache`

The `FreeCache` function is a destructor, described in Listing 3.

Listing 3. FreeCache function header.

```
void FreeCache(
    struct Cache_t * cachePtr);
```

When `CreateCache` is called, memory space is allocated for the different fields of the cache. This memory space must be freed up again in order to prevent memory leaks. To achieve this, the `FreeCache` function is provided.

2.3.3. `clCreateCacheBuffer`

When using OpenCL, the `clCreateBuffer` function is used to allocate memory on the accelerator by creating a `cl_mem` object in the accelerator memory that is initialized with the content of an associated memory space in the host memory. To use the software cache, the custom `clCreateCacheBuffer` can be used in a very similar fashion. This function first checks if the data is already in the cache memory (on the accelerator card), and if so, it returns the `cl_mem` object pointing to that location. If the data is not in the cache, it will internally invoke the `clCreateBuffer` function to allocate memory space on the accelerator and transfer the data to the accelerator before returning the `cl_mem` object pointing to the new data location.

This function sets/updates all the relevant attributes in the cache controller, i.e., the tag, the valid bit, and the access order for the replacement policies. The `clCreateCacheBuffer` function is described in Listing 4.

Listing 4. clCreateCacheBuffer function header.

```
cl_mem clCreateCacheBuffer (
    cl_context context,
    cl_mem_flags flags,
    size_t size,
    void *host_ptr,
    cl_int *errcode_ret,
    struct Cache_t* cachePtr)
```

The only addition in the syntax with respect to the regular OpenCL `clCreateBuffer` function is the addition of the pointer to the software cache. Depending on the flags given as arguments, the function can slightly change. By default, the flags `CL_MEM_READ_WRITE` and `CL_MEM_COPY_HOST_PTR` are assumed as arguments for this function. This tells the cache that the data is only to be transferred when not in the cache already. There are, however, instances when an entry is already in the cache but the data is outdated and needs to be overwritten and thus transferred to the cache. To do this, the `CL_MEM_COPY_HOST_PTR` flag must be omitted. This way, data is always written to the cache.

2.3.4. `clEnqueueReadCacheBuffer`

To transfer data from the accelerator to the host, the `clEnqueueReadBuffer` OpenCL is used. When a software cache is used, the custom `clEnqueueReadCacheBuffer` is used instead. This function is an adaptation of the standard `clEnqueueReadBuffer` function. The difference is that no `cl_mem` object is given as an argument since the cache determines where the data to be read back is stored. When a cache entry is requested that is not in the cache the function will return `1` and no data is retrieved from the accelerator. The `clEnqueueReadCacheBuffer` is described in Listing 5.

Listing 5. clEnqueueReadCacheBuffer function header.

```
int clEnqueueReadCacheBuffer (
    cl_command_queue command_queue,
    cl_bool blocking_read,
    size_t offset,
    size_t size,
    void *host_ptr,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event,
    struct Cache_t* cachePtr)
```

2.4. Demonstration

To show the ease of use of a software case via the OpenCL-like API, we provide a series of code snippets based on a simple vector-addition example application in OpenCL [18]. To demonstrate the additional cache-related code, snippets are given with and without the cache. Most of the code remains the same and is not shown in this section, only the differences are highlighted. To use a software cache, an instance needs to be created first. Thereafter, data transfers from the host to the accelerator and backward, as dictated by the application needs, will use the cache and benefit from avoiding redundant transfers when possible. Upon completion of the hardware-accelerated part of the application, the software cache is freed.

2.4.1. Cache Instantiation

At the beginning of the code, the cache needs to be instantiated. This is achieved by calling the `CreateCache` function and can be seen in Listing 6. No code needs to be removed at this stage and no OpenCL function has to be replaced.

Listing 6. Code snippet of cache instantiation.

```
struct Cache_t* myCache = CreateCache(8, dataSize, 32, direct_mapped
    , fifo_RP);
```

2.4.2. Data from Host to Accelerator

For the data transfer from host to accelerator the example uses the code shown in Listing 7 to create `cl_mem` objects and assign the correct data to it before providing it as an argument to the kernel.

Listing 7. Code snippet of data transfers without cache.

```
//Create the input arrays in device memory and copy the host
    pointers
input1 = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, dataSize, h_input1, &err);
checkError(err, ''Creating buffer input1'');
input2 = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, dataSize, h_input2, &err);
checkError(err, ''Creating buffer input2'');
output = clCreateBuffer(context, CL_MEM_READ_WRITE, dataSize, NULL,
    &err);
checkError(err, ''Creating buffer output'');
```

```
// Set the arguments to the compute kernel
err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &input1);
err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &input2);
err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &output);
checkError(err, ''Setting kernel arguments'');
```

To implement the cache, the code has to be changed to the code shown in Listing 8. The code is very similar but there are two key differences: (a) it is important for defining the output that the `CL_MEM_COPY_HOST_PTR` argument is not passed to the function. This will make sure that the output will always be written to the cache even if an outdated version of that entry is already present, and (b) for the output, instead of a `NULL` pointer now the pointer that points to the data in the host memory space is given. This way the cache controller can determine the location of the output in the cache memory and can keep track of the entries in the cache. The only other difference from the original OpenCL function arguments is the additional argument that provides the pointer to the cache struct.

Listing 8. Code snippet of data transfers with cache.

```
//Write to the cache on the device memory
input1 = clCreateCacheBuffer(context, CL_MEM_READ_WRITE |
    CL_MEM_COPY_HOST_PTR, dataSize, h_input1, &err, myCache);
checkError(err, ''Creating buffer input1'');
input2 = clCreateCacheBuffer(context, CL_MEM_READ_WRITE |
    CL_MEM_COPY_HOST_PTR, dataSize, h_input2, &err, myCache);
checkError(err, ''Creating buffer input2'');
output = clCreateCacheBuffer(context, CL_MEM_READ_WRITE, dataSize,
    h_output, &err, myCache);
checkError(err, ''Creating buffer output'');

// Set the arguments to the compute kernel
err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &input1);
err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &input2);
err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &output);
checkError(err, ''Setting kernel arguments'');
```

Note that the error-checking functionality built-in OpenCL is still functional with the added cache functionality. In both examples, the OpenCL-defined error type is provided as an argument and can be used to check correctness.

2.4.3. Data from Accelerator to Host

After the kernel is executed, the output data from the accelerator can be transferred back to the host. The original code uses a `clEnqueueReadBuffer` function to achieve this, which can be seen in Listing 9.

Listing 9. Code snippet of retrieving data from accelerator.

```
// Read back the result from the compute device
err = clEnqueueReadBuffer(commands, output, CL_TRUE, 0, dataSize,
    h_c, 0, NULL, NULL);
checkError(err, ''Reading back h_c'');
```

With the cache implementation, this method can still be used. This, however, does not update the access order in the cache for replacement tracking, so performance might drop due to improper tracking of usage for replacement, but correctness is still ensured. The caching software does, however, also offer some extra functionality in the form of the `clEnqueueReadCacheBuffer` function. The use of this function can be seen in Listing 10.

Listing 10. Code snippet of retrieving data from cache.

```
// Read back the result from the compute device
err = clEnqueueReadCacheBuffer(commands, NULL, CL_TRUE, 0, dataSize,
    h_c, 0, NULL, NULL, myCache);
checkError(err, ''Reading back h_c'');
```

This function allows for any data to be retrieved from the cache memory. For this function, no location on the device memory has to be given as an argument, since the cache controller contains that information. Only the location in the host memory where the data is to be stored is required. The controller will find the location in the cache, and if the requested data is not in the cache the function will return `1`, which can be used with the error functionality to identify such an occurrence. Therefore, it is possible to retrieve cached data computed with previous accelerator calls, if, of course, the cache size is big enough and the data is not yet overwritten with new data.

2.4.4. Cache Destruction

The last piece of code that is added to the example is the cache destructor function `FreeCache`, shown in Listing 11.

Listing 11. Code snippet of cache destruction.

```
FreeCache(myCache);
```

3. Results and Discussion

3.1. Use Case

To evaluate the performance of the software cache, a real-world algorithm, the Phylogenetic Likelihood Function (PLF) [19], from the field of Bioinformatics is used. The PLF uses matrix multiplications to determine the likelihood of a part of a tree structure. In a phylogenetic tree, each leaf node, henceforth referred to as a tip node, represents a different DNA sequence (different organism). The inner nodes of the tree, however, correspond to extinct common ancestors for which no DNA sequences are available. Therefore, each inner node is described by a series of probability vectors. This differentiation between the tip and inner nodes is actually important since there is a different amount of data stored for each type of node. All cache lines within a cache have a fixed-size data field. It is possible to store smaller amounts of data in a larger cache line but that is not efficient. In the case of the PLF, every inner node is 128× larger in terms of memory size than a tip node, making it inefficient to store the data of both nodes in the same-sized data field. In this case, we create two different cache instances with different sizes, one for the tip nodes and one for the inner nodes. DNA sequences used in phylogenetics can easily contain over 100,000 characters, and with 16 double-precision values per character in every inner node, it becomes a highly data-intensive application.

3.2. Experimental Setup

Two tools were used to create simulated sequence data, `ms` and `seq-gen`. `ms` is a tool that can generate a sample of a neutral evolutionary model [20]. Next, the `seq-gen` tool can create multiple DNA sequences based on that generated sample and some given parameters [21]. For our tests, the sample data is generated with the command in Listing 12.

Listing 12. ms command for generating simulated data.

```
ms X 1 -T | tail +4 | grep -v // >treefile_name
```

where `X` is the number of different sequences (number of tip nodes in the tree), `-T` specifies the output type, `tail +4` and `grep -v` remove comments and other information from the output file, and, finally, `>treefile_name` represents the output file. `X` in this case is a variable and will change for different tests. Next, the DNA sequences are created with the `seq-gen` tool using the command in Listing 13.

Listing 13. seq-gen command for converting ms data to DNA sequences.

```
seq-gen -mHKY -l 1000 -s .2 <treefile_name >seqfile_name.phy
```

Here, `-mHKY` specifies the used model, `-l 1000` is the number of alignment sites for the sequences, `-s .2` sets the variation θ to `.2` per base pair and then the names of the in- and output files are set.

For phylogenetic tree reconstruction, we used RAxML [22], an open-source tool that determines the likelihood of different phylogenetic trees for a given set of DNA sequences. This tool would be used as a base for testing and evaluating our software-cache implementation. RAxML tool was called with the following parameters, providing as input the output file generated by `seq-gen` as shown in Listing 14.

Listing 14. RAxML command.

```
   raxmlHPC -m GTRGAMMA -s seqfile_name.phy -n test1
```

Here `-m GTRGAMMA` sets the DNA substitution model, `-s seqfile_name.phy` defines the input file and `-n test1` defines a name for the run.

As expected, RAxML is highly computationally intensive. For this reason, memory-access traces were extracted from various runs, which were subsequently used in cache-performance evaluation tests. The access pattern is stored in a trace file. An example of the first few rows of a trace is provided in Listing 15.

Listing 15. The beginning of a memory trace.

```
width 156
0x2039d94      0x2039e30      0x21f4940      tt
0x203a0a0      0x203a13c      0x21f9750      tt
0x203a004      0x21f9750      0x21fe560      ti
0x2039f68      0x21fe560      0x2203370      ii
...
```

The `width` given at the top represents the length of the genomic sequences to be processed. Each row in the trace file corresponds to one accelerator call. In every row, the first two addresses point to the data of the child nodes while the third address points to the data of the parent node. This is where the result of a single accelerator invocation will be stored. As can be seen, each line also contains a pair of letters, which can be `tt`, `ti`, or `ii`. Each letter indicates whether the respective child node is a tip node or an inner node in the tree. This information will be used later for accessing the correct software-cache instance.

The number of sequences directly translates to the number of tip nodes in the tree and thus the size of the tree. For T tips, there are $T - 1$ inner nodes. By increasing the number of sequences, the number of different data blocks that need to be transferred to the accelerator increases. We evaluated application performance by constructing phylogenetic trees with 100, 250, 500, and 1000 DNA sequences. In all runs, the DNA sequence length was 1000 characters.

3.3. Hardware Accelerator and Platform

To assess performance improvements with the software cache and without it, we reproduced the PLF hardware accelerator for FPGAs that was recently presented by Malakonakis et al. [16]. The authors described a pipelined hardware accelerator for the AWS cloud and evaluated its performance on an AWS EC2 F1 instance. To yield comparable results with this study, we also target the Xilinx Alveo family of accelerator cards [23]. The Alveo cards offer Gen3x16 PCIe interfacing and onboard memory. We used Xilinx Vitis 2020.1 [24] for the accelerated application and Vivado HLS 2020.1 for the design and development of the accelerator hardware.

3.4. Performance Evaluation

3.4.1. Access Pattern

This section provides a detailed performance evaluation of our software-controlled caching mechanism. As with every cache implementation, performance depends on the access pattern; thus, it can vary from application to application as well as across different runs of the same application when execution is not deterministic. The Bioinformatics application we used in this work as a case study (phylogenetic tree reconstruction) implemented heuristics to search the phylogenetic tree space, and as such, the effect of caching was expected to vary from execution to execution, even when the same phylogenetic dataset was processed. An analysis of the access pattern of RAxML revealed that accelerator calls were not completely random over the nodes of a phylogenetic tree. Phylogenetic tree rearrangements proceeded in iterative stages, with each stage first altering the tree structure and then refining this change by invoking the hardware accelerator on a small subtree several times. We analyzed the access pattern of all four datasets (100, 250, 500, and 1000 DNA sequences) and observed that over 90% of the tree rearrangement steps (over all runs) resulted in between 5 and 9 localized hardware accelerator calls on the same subtree. Therefore, the effect of caching was not expected to change considerably across different runs for RAxML since misses would occur 5 to 9 times less frequently than hits.

3.4.2. Replacement Policies

In Figure 4, the performance of the different replacement policies for the caches can be seen in terms of the hit ratio. For the inner cache, it can be observed that the Random, FIFO, and LRU replacement policies outperform the other three. This effect increases with higher associativity. The LFU and MFU replacement policies do not perform well for small cache sizes (8 cache lines in our tests) because the options for determining the best cache line to replace are limited. The low hit ratios for the MRU can be explained by the type of memory access pattern that RAxML uses. MRU performs best in a cyclical access pattern, which RAxML does not perform when accessing the individual nodes of a phylogenetic tree. For the tip cache, once again Random, FIFO, and LRU perform well, but this time MFU also performs well. For a larger number of sequences (tree size) or a higher degree of cache associativity, however, the performance drops. The performance of the MFU is greatly improved by a smaller number of nodes per cache line. In Figure 5, the performance for a cache with 32 cache lines is shown, where the MFU appears to perform best for the smaller tree sizes. In realistic scenarios, however, the number of nodes will most likely be larger than 1000, and the number of cache lines will be very small. Hence, despite the fact that the MFU scores well in some instances, it most likely will not perform that well in a real application.

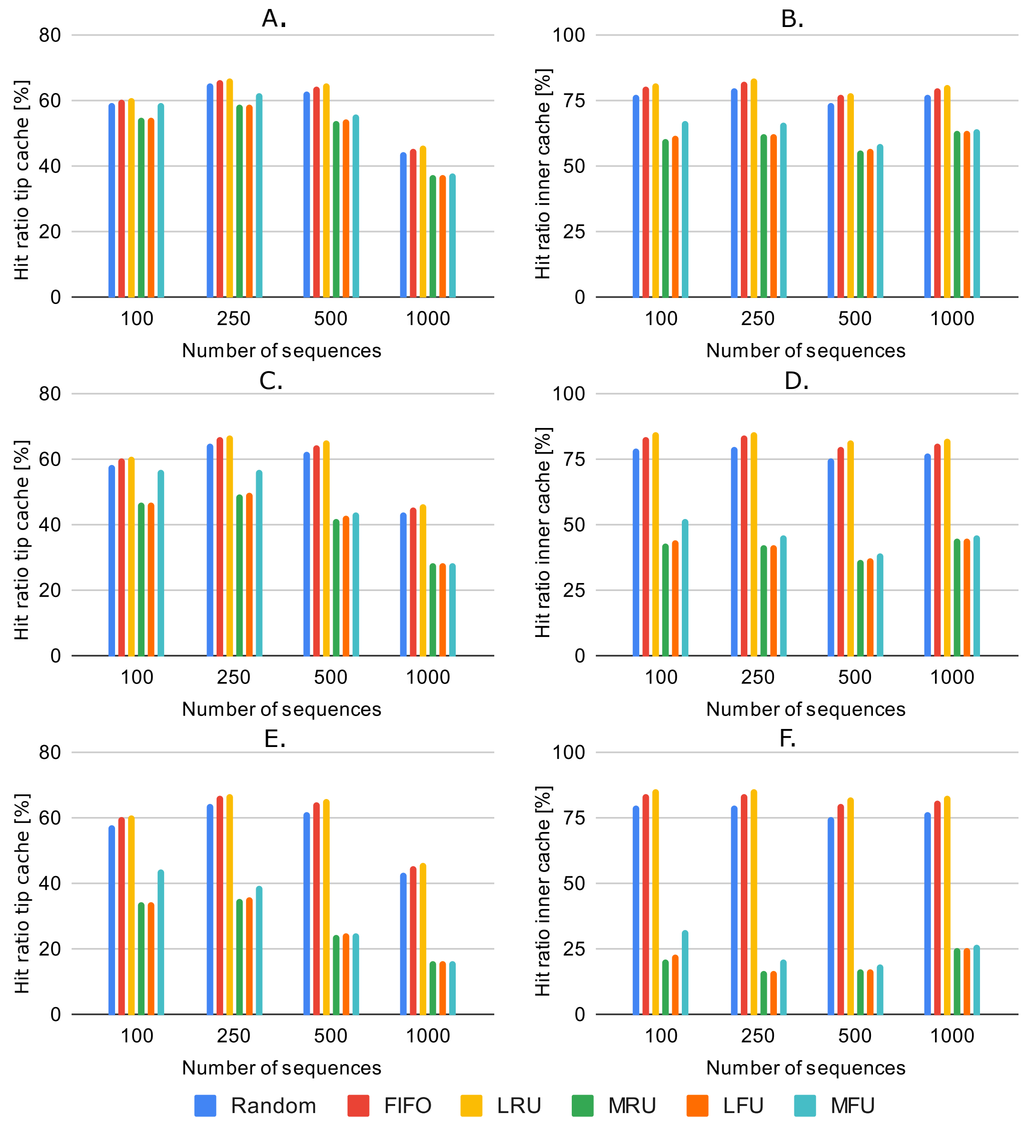

Figure 4. Hit/miss ratio comparison of replacement policies for caches configured with 8 cachelines. (**A**): 2-Way set-associative tip cache, (**B**): 2-Way set-associative inner cache, (**C**): 4-way set-associative tip cache, (**D**): 4-Way set-associative inner cache, (**E**): Fully associative tip cache, (**F**): Fully associative inner cache.

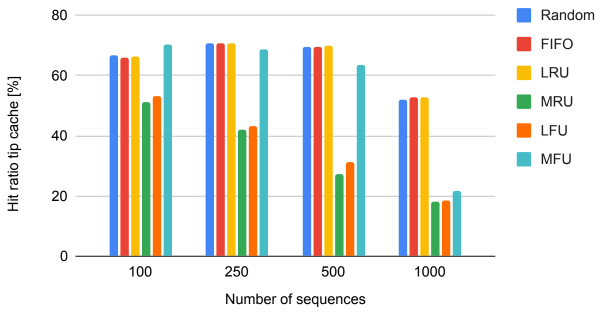

Figure 5. Hit ratio for a fully associative tip cache with 32 cache lines.

3.4.3. Level of Associativity

Figure 6 provides a comparison of the different levels of associativity. All results refer to the LRU replacement policy. Tests for all replacement policies have been performed (results not shown) and exhibit similar behavior with the presented LRU replacement policy. The tip cache shows that the level of associativity has very little influence on the hit ratio. Only the direct-mapped cache slightly underperforms in comparison with the others. For the inner cache, the performance also increases with the level of associativity but the differences among the associative caches are small.

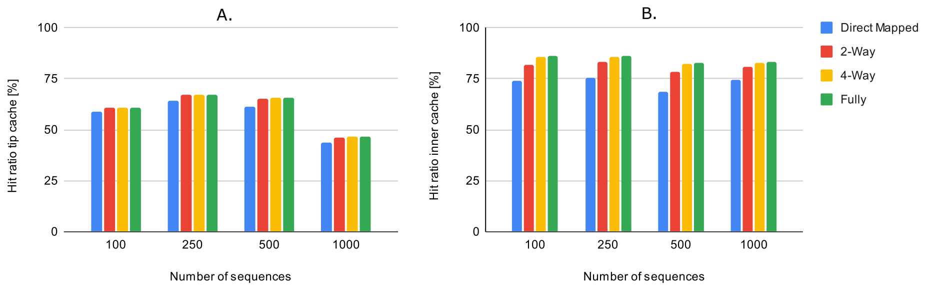

Figure 6. Comparison of tip (**A**) and inner (**B**) cache configurations with varying associativity based on the hit ratio. All caches are of the same size (8 cache lines) and implement the LRU replacement policy.

Furthermore, we performed a worst-case timing evaluation for the different levels of associativity. While comparisons based on the hit ratio are more realistic, the frequency of hits affects performance. For the worst-case timing comparison, no data is written to the cache. Therefore, every memory access is a miss and requires the worst-case timing overhead for determining it. Figure 7 illustrates the results for the tip and inner caches. For the tip cache, these differences are small and the direct-mapped cache shows the highest time overhead overall. For the inner case, there is quite a significant difference in overhead timing over the associativity levels, with higher levels of associativity having considerably higher time overhead. All measured time overheads, however, are negligible when compared with the total transfer time required per tree node.

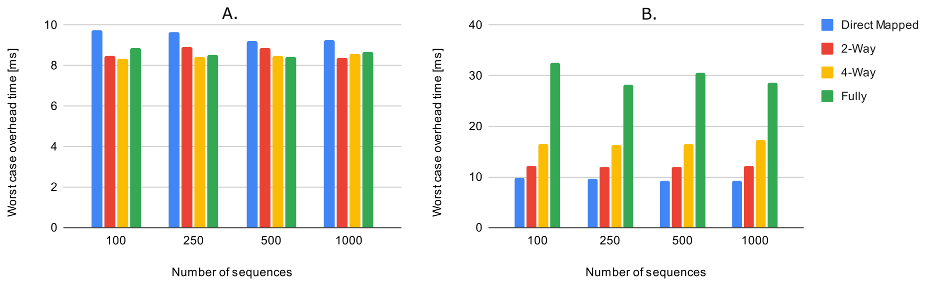

Figure 7. Worst-case time comparison for tip (**A**) and inner (**B**) caches when all memory accesses are misses. All caches are of the same size (8 cache lines) and implement the LRU replacement policy.

3.4.4. Cache Size

The cache size most likely will be dictated by the available memory on the accelerator card. Since the previous tests show that the overhead timing adds negligible costs compared to the hit ratio, it is expected that a larger number of cache lines will result in improving performance further. To verify this, we performed a test by increasing the number of cache lines. In Figure 8, the hit ratio for the tip and inner caches are shown. Both caches are fully-associative and implement the LRU replacement policy. As expected, larger cache sizes achieve higher hit ratios and perform better.

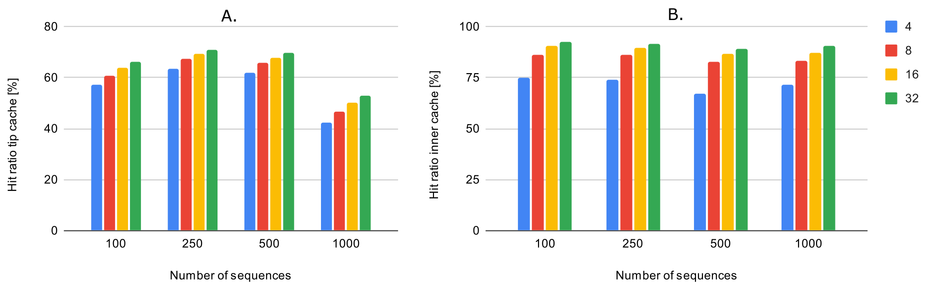

Figure 8. Comparison of hit-ratio performance of a fully-associative tip cache (**A**) and a fully-associative inner cache (**B**) as the cache size increases from 4 cache lines to 32 cache lines. The LRU replacement policy is implemented.

3.4.5. Application Time Breakdown and Overall Performance

To evaluate the overall performance of the phylogenetic application, we employed the best-performing cache configurations, all with the LRU replacement policy. An overview of the different configurations is provided in Table 2.

Table 2. Different software-cache configurations used for the evaluation of the phylogenetics application.

	Inner Cache Configuration	**Inner Cache Replacement Policy**	**Tip Cache Configuration**	**Tip Cache Replacement Policy**
Conf. 1	Direct mapped	n/a	Direct mapped	n/a
Conf. 2	2-Way associative	LRU	2-Way associative	LRU
Conf. 3	4-Way associative	LRU	4-Way associative	LRU
Conf. 4	Fully associative	LRU	Fully associative	LRU

Figure 9A provides a time-breakdown of the application (based on the trace file from the 1000-sequence phylogenetic analysis) for each of the different software-cache configurations of Table 2 and the reference case where no software cache is used. Accelerator processing time and output time (data transfer from the accelerator to the host) are the same overall configurations. A significant performance improvement is observed for the input-data transfer times, where the transfer times are reduced from 12 s to 2 s in the best scenario, thereby achieving a 6× reduction. Note also the negligible impact that the tip input has on the total execution time.

Figure 9B provides the total execution time for each memory trace. Note that all previous results focused on caching of application data (tip- and inner-node data) that exhibited the locality of reference. We now evaluate overall application performance by including all data transfers required per accelerator call, i.e., accounting for application data with no locality of reference. RAxML requires a total of 144 double-precision values per PLF invocation, i.e., per accelerator call. As can be observed in the figure, all evaluated software-cache configurations improve overall application performance and the difference between the configurations are not significant, but the larger phylogenetic analyses are expected to benefit more from software-caching.

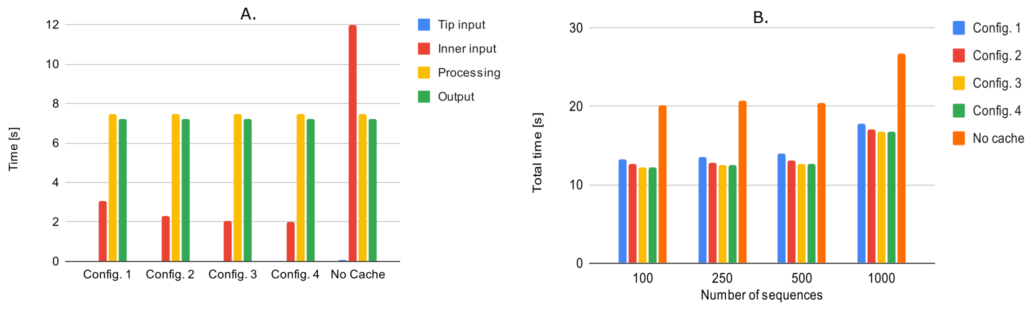

Figure 9. (**A**). Time-breakdown per cache configuration (Table 2) for 1000 accelerator calls. ("tip input" is the tip data-transfer time, "inner input" is the inner vectors data-transfer time, "processing" is the hardware-accelerator execution time, and "output" is the accelerator-to-host data transfer time) (**B**). Overall application time comparison for analyses with 100, 250, 500, and 1000 organisms.

4. Conclusions

In this work, we described and evaluated a generic implementation of a software cache that could be used to improve the performance of hardware accelerators in data centers by optimizing data transfers. By systematically caching data on the accelerator's external memory, overall accelerator performance (as perceived by the host processor) was improved without any changes to the actual hardware. Our software-cache implementation included an OpenCL-like interface and required only minimal changes to an application to exploit caching. We evaluated performance using a real-world Bioinformatics application that reconstructs the evolutionary relationship of organisms. Performing tests on an AWS EC2 F1 instance on the cloud, we observed that software caching reduced the data transfer time from the host to the accelerator by up to 6×, which resulted in an up to 40% performance improvement of the whole application (with the same hardware accelerator and no caching exploited for the data transfers from the accelerator back to the host).

Author Contributions: Conceptualization and methodology, P.K. and N.A.; software, validation, investigation, resources, data curation, P.K.; writing—original draft preparation, P.K.; writing—review and editing, N.A.; visualization, P.K.; supervision and project administration, N.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: All code is openly available at: https://github.com/pephco/SCILA (accessed on 7 September 2022).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kim, M.A.; Edwards, S.A. Computation vs. memory systems: Pinning down accelerator bottlenecks. In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Springer: Berlin/Heidelberg, Germany, 2012; Volume 6161, pp. 86–98. [CrossRef]
2. Wei, X.; Liang, Y.; Cong, J. Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management. In Proceedings of the 56th Annual Design Automation Conference, Las Vegas, NV, USA, 2–6 June 2019; pp. 1–6. [CrossRef]
3. Zhang, X.; Wang, J.; Zhu, C.; Lin, Y.; Xiong, J.; Hwu, W.M.; Chen, D. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD, San Diego, CA, USA, 5–8 November 2018; pp. 2–9. [CrossRef]
4. Xiao, Q.; Liang, Y.; Lu, L.; Yan, S.; Tai, Y.W. Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. In Proceedings of the Design Automation Conference, Austin, TX, USA, 18–22 June 2017; pp. 1–6, Part 12828. [CrossRef]
5. AlSaber, N.; Kulkarni, M. Semcache: Semantics-aware caching for efficient gpu offloading. In Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, Eugene, OR, USA, 10–14 June 2013; pp. 421–432.

6. Jablin, T.B.; Jablin, J.A.; Prabhu, P.; Liu, F.; August, D.I. Dynamically managed data for CPU-GPU architectures. In Proceedings of the Tenth International Symposium on Code Generation and Optimization, San Jose, CA, USA, 31 March–4 April 2012; pp. 165–174.
7. Asai, R.; Okita, M.; Ino, F.; Hagihara, K. Transparent avoidance of redundant data transfer on GPU-enabled apache spark. In Proceedings of the 11th Workshop on General Purpose GPUs, New York, NY, USA, 25 February 2018; pp. 22–30.
8. Begunkov, P. dlmCl: Optimization of CPU-GPU memory transfers for OpenCL devices with HSA. In Proceedings of the 5th International Workshop on OpenCL, Toronto, ON, Canada, 16–18 May 2017; pp. 1–2.
9. Barua, P.; Zhao, J.; Sarkar, V. OmpMemOpt: Optimized Memory Movement for Heterogeneous Computing. In Proceedings of the European Conference on Parallel Processing, Warsaw, Poland, 24–28 August 2020; Springer: Berlin/Heidelberg, Germany, 2020; pp. 200–216.
10. Berger, S.A.; Alachiotis, N.; Stamatakis, A. An optimized reconfigurable system for computing the phylogenetic likelihood function on dna data. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Shanghai, China, 21–25 May 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 352–359.
11. Pratas, F.; Trancoso, P.; Stamatakis, A.; Sousa, L. Fine-grain parallelism using multi-core, cell/BE, and GPU systems: Accelerating the phylogenetic likelihood function. In Proceedings of the 2009 International Conference on Parallel Processing, Vienna, Austria, 22–25 September 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 9–17.
12. Zhou, J.; Liu, X.; Stones, D.S.; Xie, Q.; Wang, G. MrBayes on a graphics processing unit. *Bioinformatics* **2011**, *27*, 1255–1261. [CrossRef] [PubMed]
13. Zierke, S.; Bakos, J.D. FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods. *BMC Bioinform.* **2010**, *11*, 184. [CrossRef] [PubMed]
14. Ronquist, F.; Huelsenbeck, J.P. MrBayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics* **2003**, *19*, 1572–1574. [CrossRef] [PubMed]
15. Alachiotis, N.; Brokalakis, A.; Amourgianos, V.; Ioannidis, S.; Malakonakis, P.; Bokalidis, T. Accelerating Phylogenetics Using FPGAs in the Cloud. *IEEE Micro.* **2021**, *41*, 24–30. [CrossRef]
16. Malakonakis, P.; Brokalakis, A.; Alachiotis, N.; Sotiriades, E.; Dollas, A. Exploring Modern FPGA Platforms for Faster Phylogeny Reconstruction with RAxML. In Proceedings of the IEEE 20th International Conference on Bioinformatics and Bioengineering, BIBE, Cincinnati, OH, USA, 26–28 October 2020; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2020; pp. 97–104. [CrossRef]
17. Shata, K.; Elteir, M.K.; EL-Zoghabi, A.A. Optimized implementation of OpenCL kernels on FPGAs. *J. Syst. Archit.* **2019**, *97*, 491–505. [CrossRef]
18. McIntosh-Smith, S.; Deakin, T. HandsOnOpenCL. Github Repository. 2019. Available online: https://github.com/HandsOnOpenCL (accessed on 7 September 2022).
19. Felsenstein, J. Evolutionary trees from DNA sequences: A maximum likelihood approach. *J. Mol. Evol.* **1981**, *17*, 368–376. [CrossRef]
20. Hudson, R.R. Generating samples under a Wright-Fisher neutral model of genetic variation. *Bioinformatics* **2002**, *18*, 337–338. [CrossRef] [PubMed]
21. Rambaut, A.; Grassly, N.C. Seq-gen: An application for the monte carlo simulation of dna sequence evolution along phylogenetic trees. *Bioinformatics* **1997**, *13*, 235–238. [CrossRef]
22. Stamatakis, A. RAxML version 8: A tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics* **2014**, *30*, 1312–1313. [CrossRef]
23. Xilinx. Adaptable Accelerator Cards for Data Centre Workloads. 2020. Available online: https://www.xilinx.com/products/boards-and-kits/alveo.html (accessed on 7 September 2022).
24. Xilinx. Vitis Software. 2020. Available online: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html (accessed on 7 September 2022).