Optimizing Data Movement for Accelerator Cards Using a Software Cache

Steven Wijnja
University of Twente, Enschede

Abstract—**Hardware accelerators are used to speed up computationally expensive applications in many scientific fields. However, offloading tasks to accelerator cards requires data to be transferred between the memory of the host and the external memory of the accelerator card; this data movement frequently becomes the bottleneck for increasing accelerator performance. In this work, we explore the use of a software cache to optimize communication and alleviate the data-movement bottleneck by transparently exploiting locality and data reuse. We present a generic, application-agnostic framework, dubbed SoftCache, that can be used with both GPU and FPGA accelerator cards. SoftCache exploits locality to optimize data movement in a non-intrusive manner (i.e., no changes to the algorithm are necessary) and allows the programmer to tune the cache size, cache organization, and replacement policy toward the application needs. Each cache line can store data of any size, thereby eliminating the need for separate caches for different data types. We used a phylogenetic application to showcase SoftCache. Phylogenetics study the evolutionary history and relationships among different species or groups of organisms. The phylogenetic application implements a tree-search algorithm to create and evaluate phylogenetic trees, while hardware accelerators are used to reduce the computation time of probability vectors at every tree node. Using SoftCache, we observed that the total number of bytes transferred during a complete run of the application was reduced by as much as 89%, resulting in up to 1.7x (81% of the theoretical peak) and 3.5x (75% of the theoretical peak) higher accelerator performance (as seen by the application) for a GPU and an FPGA accelerator, respectively.**

Index Terms—**Data movement, Software cache, GPU, FPGA, OpenCL, RAxML, Phylogenetic likelihood function (PLF)**

I. Introduction

Hardware accelerators such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) are used to speed up computations in many scientific fields, for example machine learning for particle physics computing [1], deep neural networks [2], Monte Carlo simulators [3], image processing [4], bioinformatics [5] and others. Generally, the computationally intensive parts of the application are offloaded to the accelerator to increase the performance of the application. Hardware accelerators have become increasingly popular in scientific computing due to their parallel processing capabilities and efficient handling of large datasets. Nowadays, an exorbitant amount of data is collected. For example, the cost of sequencing DNA has been decreasing exponentially, so fast that Moore's Law can't keep up [6]. DNA sequence analysis is critical for advancing our understanding of genetics, diagnosing diseases, studying evolution and unraveling the complexities of human existence. Accelerators offer parallelism and scalability to reduce the runtime of applications by orders of magnitudes, which is necessary because processing large amounts of data such as DNA can take days to weeks.

Offloading computational tasks to an accelerator typically require a significant amount of data transfers, as data has to be transferred to the accelerator to be processed and then transferred back to the host. In some cases (depending on the application), the output of one accelerator invocation can be reused by subsequent accelerator invocations without transferring anything back to the host. In other instances, the accelerator output has to be sent back to the host after each accelerator invocation to enable the algorithm to proceed. Both cases, however, present opportunities for data that is either computed or previously transferred to the accelerator to be reused.

Optimizing communication for hardware accelerators is extensively researched by both the industry and the scientific community. NVIDIA introduced Unified Memory in CUDA 6 SDK [7], which uses one memory address to access both CPU and GPU memory, and will migrate pages when necessary. The main goal of Unified Memory is to make the programming of GPUs easier, while also providing a high bandwidth for data transfers.

Communication overhead poses a significant bottleneck in cloud computing and data centers [8], [9]. Consequently, researchers have explored the utilization of software caches for specific applications and platforms [10]–[12]. By employing a software cache, it is possible to mitigate the impact of communication overhead and enhance overall performance.

GPU/FPGA accelerator cards have both on-chip and on-board (external) memory. Here, we focus on the communication between the external memory of the host and the external memory on the accelerator. The cost of data transfers to/from external memory is considerably more time and energy costly than the communication between on-chip memory and the processing units [13]. The on-chip memory is significantly faster than the throughput of state of the art PCI express buses. For example, the GDDR6X memory used in the GeForce RTX 3090 has a bandwidth of $936.2\,\mathrm{GB/s}$, while the PCIe 4.0 x16 used by the same GPU has a bandwidth of only $64\,\mathrm{GB/s}$. The I/O bandwidth is often the bottleneck when trying to increase the performance of hardware accelerated applications [14]–[16]. This is often caused by redundant data transfers. In essence, data that is already stored on the accelerator does not have to be sent again; such transfers are therefore redundant.

A significant performance increase can be achieved when the amount of transferred data is reduced. This is especially true for applications with a low arithmetic intensity, i.e. applications with a low computation-to-communication ratio. One can exploit locality by reusing this data, however this does require that you keep track of what data is located on the accelerator. To do this efficiently, one needs to analyze the memory access pattern of the application, which yields a time-consuming, application-specific solution without any transferable benefits for other applications/domains.

Therefore, we present a software cache, dubbed SoftCache, that is inspired by hardware caches. Hardware caches for processors are designed to make the general case fast, and we apply this same principle to efficiently manage the data transfers for applications.

The objective of this work is to design a generic framework for PCI-connected hardware accelerators. The framework should be application-agnostic, support both GPUs and FPGAs, and the cache should be customizable with regards to organisation, size and replacement policies. SoftCache relies on the OpenCL API to support both GPU and FPGA accelerators, but other APIs, e.g., CUDA, can also be used. The main goal is to exploit locality to optimise data transfers. The framework reduces the amount of data sent and thereby shorten the time spent on communication. SoftCache is customizable which allows us to test a variety of cache organisations, replacement policies and different cache sizes.

The main contributions of this work are:

- We present SoftCache, a general framework that optimizes data movement for accelerators in a non-intrusive manner, i.e., no changes to the algorithm are needed, and only minimal changes to the code are required. The framework is customizable and supports various cache organizations, replacement policies, and cache sizes. A single SoftCache instance can cache different-sized data blocks. This allows the framework to be tuned accordingly to better address the application needs. Furthermore, it supports both write-through and write-back policy, allowing the optimization of communication in both directions.
- We evaluate SoftCache with a phylogenetic application and three hardware accelerators, a GPU implementation and two special-purpose, FPGA-based architectures, for the phylogenetic likehood function. The experimental results show that SoftCache can achieve similar performance with related works that have been designed and optimized specifically for the phylogenetic likehood function. Without any changes to the hardware accelerators to improve their peak theoretical throughput, the use of SoftCache results in 1.7x and 3.5x higher accelerator performance, as perceived by the host application, for the GPU and the FPGA accelerators, respectively.

A. Research questions

The research will focus on exploring ways to optimize data movement and improve the overall performance of accelerator-based applications in a non-intrusive manner. No changes to the implementation of the algorithms should be needed and only information from the explicit data movements will be used. To properly address this problem, a research question has been formulated:

- How can we exploit locality to optimise data movement on systems that use PCI connected accelerator cards with dedicated memory for I/O bound applications?

To answer the main research question, we will explore the following sub-questions:

1) What methods are described in the literature to optimize data movement?
2) How does the framework perform on a real application?
3) How do the different cache parameters compare to each other?

In section II the necessary background is presented to get a better understanding of hardware accelerators and cache architectures. Section III explores related work and identifies the gaps and limitations of existing methods. Section IV describes the design of the framework and the technical details. In section V the method of evaluation and the results are discussed. Finally, in section VI a conclusion will be drawn and the main research question will be answered.

II. Background

A cache is a memory storage that is used to overcome memory access bottlenecks. A cache is used as a temporary storage closer to the processing unit(s), which makes it possible to retrieve data quicker. An essential feature of a cache is that it is not visible to the application whether the data is retrieved from the cache or the original source.

A distinction can be made between software and hardware caches. For example, web browsers use a software cache to store recently visited websites to load the website faster when it is accessed again [17]. When a user visits a website, the browser checks the cache for requested files and loads them from there instead of downloading them again, which reduces loading time and data transfer. Hardware caches are used in processors, graphical processing units and hard drives. Figure 1 illustrates a typical memory hierarchy in modern processing systems. Each layer communicates with the adjacent layers. The bandwidth increases and the latency decreases as we move upwards, at the cost of size. Registers and caches are utilised to temporarily store data and take advantage of temporal locality, which refers to the tendency of a program to access data that has been recently accessed or is likely to be accessed again in the near future.

The cache framework presented in this work is a software cache, but its architecture is inspired by hardware caches. In this section, background information on hardware accelerators and hardware caches is provided.

A. Hardware accelerators

Hardware accelerators are specialized hardware designed to accelerate a specific task at a faster rate than traditional CPUs. Graphic Processing Units (GPUs), Field-programmable Gate Arrays (FPGAs), Digital Signal processors and Tensor Processing Units are examples of hardware accelerators.

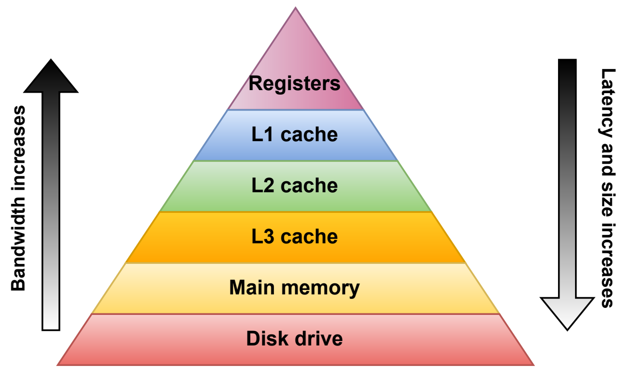


Fig. 1: Memory hierarchy of a processor

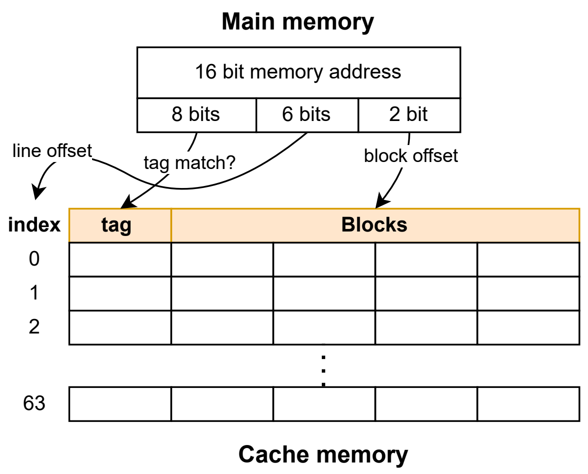


Fig. 2: Memory address and cache layout for direct mapping

To utilise the hardware accelerator, data has to be transferred to the accelerator. The data is then processed immediately in a streaming fashion or stored in dedicated memory on the accelerator board. Within this work, only GPUs and FPGAs with dedicated memory are considered, however SoftCache can also be used on other accelerators that use dedicated memory.

GPUs were originally developed to render graphics for video games and multimedia applications. Nowadays they are also used the scientific community due to their versatile and powerful parallel processors. The newest GPUs have over 10.000 processor cores, while consumer CPUs generally only have eight cores. This allows them to do many computations in concurrently, which can significantly reduce the overall execution of an application.

FPGAs on the other hand, are reprogrammable hardware that allow the users to design digital logic circuits that matches the computational requirements of an application. Moreover, FPGAs provide the opportunity to tailor hardware for particular applications, enabling parallelization and pipelining of computations for optimal performance. Pipelining within FPGAs accelerates calculations by dividing them into smaller segments and overlap the segments with consecutive iterations. For example, within a loop two numbers are multiplied together, and the result is added to a sum. Without pipelining, this would take two cycles per iteration (assuming that both addition and multiplication takes one clock cycle). By using a pipeline we can permit to start the next iteration of the loop before the previous iteration finishes, effectively overlapping the multiplication of the second iteration with the addition of the first iteration. After filling up the pipeline, this results in one clock cycle per iteration, instead of two. This enables parallel processing and can reduce the overall execution time for improved performance.

B. Cache organisations

Caches are categorized in three organisations: direct mapping, n-way set associative and fully associative. However, the cache organisations can all be classified as a n-way set associative cache. Direct mapping is essentially a m-way set associative cache, where m is the number of cache lines, and each set consists of exactly one line. A fully associative cache is in essence a cache with one set and m lines per set.

1) Direct mapping: Direct mapping requires the least amount of hardware and has a very low look-up time. The cache memory consists of a tag, and one or more blocks of data. The binary representation of the memory address is divided into tag, line number and block offset.

The line number field is used to select the line within the cache. In hardware design this is generally done by using multiplexers. A comparator is then used to check whether the tag field matches the tag of the cache line. If there is a match the data from will be used, this is called a cache hit. If there is no match, or a cache miss, the data will be transferred to the cache and the tag will be updated. The block offset is used to identify a unique byte or word in the main memory.

The number of bits used for the line number dictates the size of cache. When n bits are used, the size of the cache will be 2^n lines.

Figure 2 illustrates an example of a 16-bit address and the structure of a 64-line, 4-block direct-mapped cache. The line number field is used to select the line within the cache. In hardware design this is generally done by using multiplexers.

Note that with direct mapping a unique memory address can only be mapped onto one specific cache line with a block offset for the specific byte, as shown in Figure 3a. This results in a lower cache hit ratio compared to other cache organisations.

2) Set associative: The n-way set associative cache is a variation on the direct mapping organisation. The cache is divided into n sets where each set contains m cache lines. The host address is again divided into three sections: the tag, the set index and block offset.

The set index will determine in which set the data is stored. The tag bits are then compared with all the tags in the set. If there is a cache hit the appropriate line will be returned. If there is no match the data will be retrieved from main memory and placed in respective set. The line number within the set in

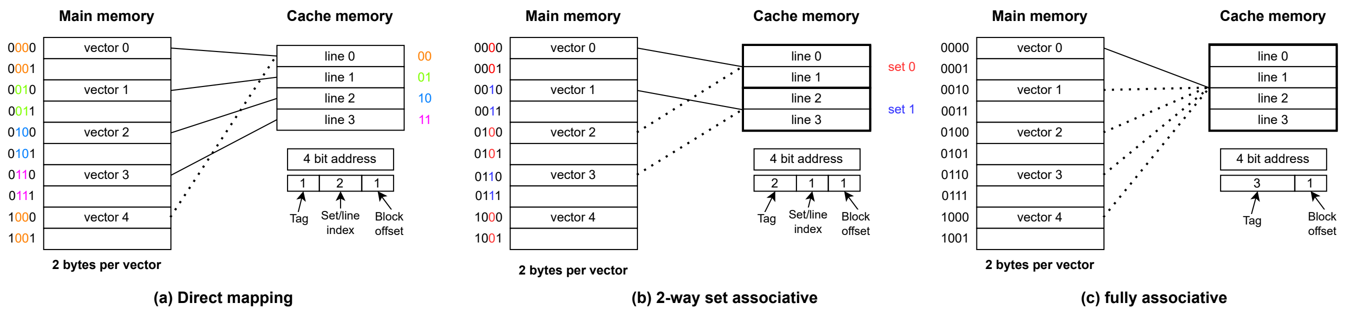


Fig. 3: Memory address to cache line assignment for direct mapped, set associative and fully associative caches

which the data is stored will be determined by the replacement policy.

Set associative caches use significantly more hardware than direct mapping. For m cache lines per set, we also need m comparators, as well as logic to combine the results of all comparators. An example of possible mappings for a 2-way set associative cache is shown in Figure 3b.

3) Fully associative: A fully associative cache is essentially a single set associative cache where a memory block can be mapped to any of the cache lines. The host address is divided in a tag and the block offset.

The tag bits are compared to all the entries of the cache. Similarly to the set associative cache, if there is no match a replacement policy will decide on which line the data will be stored.

Fully associative caches require a comparator for every cache line and therefore require the most hardware of all the cache organisations. However, it does give freedom to place the data on any cache line and therefore utilise the cache in the most optimal way

Figure 3c shows an example of the fully associative cache. No bits are reserved for the set/line index anymore, and instead three bits are used for the tag and still one bit is used for block offset. All vectors can be mapped to the all cache lines, and the cache controller will decide to which line each vector is mapped, based on the replacement policy.

C. Replacement policies

Associative caches use a replacement policy to determine which data to evict from the cache when the cache is full and new data needs to be loaded. In this work, we consider the following strategies: random, FIFO and LRU.

1) Random: The random replacement policy selects the cache line that has to be replaced by random. The benefit of random replacement is that it does not require information about the access history. The cache controller generally uses a pseudo-random number generator to select a cache line to evict when a new block of data is loaded into the cache.

2) FIFO: The first-in, first-out (FIFO) algorithm requires the cache controller to keep track of the order in which lines are loaded in the cache, and is often implemented by using a circular buffer or a queue. The data that was first added to the queue will be the first to be removed when a new block of data is loaded into the cache. When a cache line is accessed, the position in the queue does not change.

3) LRU: The least recently used (LRU) algorithm requires the cache controller to keeps track of how often a cache line is used/accessed. Usually this is done by storing the age as an integer of each cache line. Whenever a cache line is used, the age of that cache line will be reset to 0 and all other age fields in the set will be incremented by one. The algorithm will replace the oldest cache line when new data has to be stored.

An alternative method is by implementing it by using a doubly linked list, paired with a hash map for $O(1)$ accessing and updating the cache line. However, this does require two data structures with $O(n)$ space complexity.

D. Write policies

When the processor wants to write data, it first checks to see if the address it wants to write to is present in the cache. If the address is found in the cache, the write is considered a "hit". In this case, the value in the cache can be updated, avoiding the need to access slower main memory. However, this approach can result in inconsistent data, as the data in the cache may be different from that in main memory. This can cause issues in systems with multiple devices sharing the same main memory, such as in a multiprocessor system.

To address this problem, two techniques are commonly used: write-through and write-back.

1) Write-through: The write-through policy not only updates the cache, but also simultaneously writes to memory. This is the most reliable way to keep data consistent between main memory and cache.

One advantage of write-through is that it reduces the possibility of data loss or corruption. Since every write operation updates both the cache and the main memory/storage, there is no risk of losing data due to a power outage or system failure before the data is propagated to the main memory/storage. Additionally, because the main memory/storage is always consistent with the cache, it ensures that any other component that accesses the main memory/storage will see the most up-to-date version of the data.

However, a disadvantage of write-through is that it can lead to lower performance due to the increased number of memory transfers required for each kernel call. Additionally, the larger the size of the data in the cache, the longer the write-through policy can take to complete, which can have a negative impact on overall system performance. Write-through is often used in systems where data consistency is critical.

2) Write-back: The write-back policy delays the writing of data back to the main memory until it is absolutely necessary, i.e. when a cache line is replaced or if the data in main memory is used and requires to be coherent with the cache. This is done to optimize the performance of the system by reducing the number of memory write operations, which can be time-consuming and can slow down the system. In the write back policy, modified data is first written to a cache instead of immediately being written back to the main memory.

Each cache line requires a dirty bit that indicates whether the data has been modified (in which case it is marked dirty) or not. When the cache controller evicts a dirty cache line, it is will be written back to memory.

III. Related work

Several frameworks implement cache-related concepts to reduce the number of data transfers.

SemCache [10] proposed the idea of a software cache to reduce the amount of data transfers between CPU and GPU. The cache translation records who reside in the software cache contain the start address and end address of the CPU memory, the status and the start address of the memory on the GPU. This allows SemCache to have a variable granularity. Not only does SemCache reduce the number of data transfers, but it can also be extended to avoid redundant computations on the GPU. Due to the dynamic granularity, the cache overhead could become significant when there are many small records in the software cache. SemCache was benchmarked on a domain decomposition which is used for the simulation of structural dynamics problems, that rely heavily on floating point matrix multiplications. They achieved a performance increase of 30% to 40% compared to the no cache implementation, by using the write-through policy. By using the write-back policy they achieved another 4-10%.

DyManD [12] consists of a memory allocation system, run-time library and compiler passes. The run-time library also makes use of a software cache, containing an ordered map of the base address, size and state. It allocates memory per block, which unfortunately leads to issues with false sharing when allocation units on the same page is frequently used by both the CPU and GPU. This results in unnecessary transfers because the allocation unit may be coherent, but the page is not, and therefore the whole page is transferred. DyManD tries to fix this by using heuristics to arrange the memory in such a way that this can be partly avoided. DyManD was benchmarked on 24 different programs ranging from calculations on matrix multiplications to more complex data structures, such as linked lists, graphs and other complex data structures. DyManD achieved a geometric mean speed-up of 4.21x over the best sequential versions.

Asai et al. [18] proposed an extension to Apache Spark. Apache Spark is an open-source data processing engine developed specifically for handling large datasets. Its main aim is to offer high computational speed, scalability, and flexibility to meet the needs of Big Data. The extension implicitly avoids redundant data transfers between the CPU and GPU without any code modifications. Apache Spark uses so called resilient distributed datasets (RDD), which are immutable elements that can be distributed across the nodes of a cluster system. The general run-time behaviour of Apache Spark causes redundant data transfers because it transfers data from the CPU to the GPU, executes the kernel for the RDD and transfers back the results to the CPU, regardless of the continuation of operations. Their extension identifies GPU-isolated RDDs by analysing the dependencies. If a RDD is GPU-isolated the results will be cached and used for the next operation. To exploit the re-use of RDD the cached data is lazily deleted, using the Least Recently Used (LRU) policy. Their proposed method reduced the data transfer time by 96% on a logistic regression problem. However, one limitation of the proposed extension is its dependency on the Apache Spark framework, which restricts its applicability to specific types of data processing tasks or algorithms. Additionally, the extension is only compatible with NVIDIA GPUs, further narrowing its usability.

dlmCL [19] proposes the idea of unifying the host and device memory into one object, similar to CUDA's unified memory [20]. For devices with shared memory access it does not allocate buffers on the device, but instead maps the device-visible virtual addresses to physical addresses. These are buffers are so called zero-copy buffers and drastically reduce the number of data transfers. dlmCL is mainly an abstraction of openCL features that will implicitly make zero-copy buffers for devices with shared memory, such as integrated GPUs. dlmCL does not improve the performance for dedicated GPUs, it is actually slightly worse due to some overhead.

There are also several frameworks that reduce the number of data transfers by using compiler techniques. OmpMemOpt [21] applies a data flow analysis during compile time for partial redundancy elimination. The data transfers are optimized by using lazy code motion techniques. Invariant computations are moved out of loops and it eliminates duplicate computations in a computation path. OmpMemOpt was tested on ten different benchmarks and achieved a geometric speed-up of 2.3x and managed to reduce the number of bytes that were transferred by 50%. However, since they use compiler optimizations, the solution only works on static data transfers patterns.

Other methods such as pinned memory can also speed up the communication drastically, as explained by Rasch et al. [22]. Data in a pinned memory area does not have to be copied to a temporary buffer before it is transferred to the accelerator. It can directly use direct memory access (DMA) transfers to send and receive data. However, this does not reduce amount of data that is transferred.

The works discussed so far have primarily utilised GPUs. Related to FPGAs, Wei et al. [23] proposes Layer Conscious Memory Management framework (LCMM) for Deep Neural

Networks (DNN) that are ran on an FPGA. To reduce the required on-board memory they analyse the lifespan of different feature tensors to look for opportunities for them to share the same buffer. This is done on the computation graph by the means of graph coloring algorithms. However, the core part of the framework is their memory allocation algorithm DNNK (DNN Knapsack). DNNK allocates on-board memory for some layers while other layers access the data from off-chip memory, depending on whether the layer is memory bound or computation bound. They achieve a performance increase of 1.36x compared to previous designs on ResNet. Their work stores frequently accessed data close to the processing unit in a static manner, unlike a cache where the buffer content can be replaced. The latter has the opportunity to be more efficient with the right replacement policy.

Alachiotis et al. [16] optimized data movement for a phylogenetic application that reconstructs and evaluates phylogenetic trees on FPGAs. They cache the nodes in the tree which contain the probability vectors. By using a direct mapped software cache that stores the probability vectors, they managed to reach 75% of the maximum effective accelerator performance. They only cache data that is transferred from the main memory to the accelerator, and overlapped computation with the data transfers from accelerator to host.

Knoben [11] implemented a software cache for FPGAs, however his design can only store data of the same size, and multiple cache instances are needed to store the different sized nodes of the tree-based algorithms. Furthermore, it can not use the full size of the accelerator's on-board memory due to the fact that the cache size has to be a power of two.

There are also a number of frameworks available to make the communication with FPGAs easier [24], [25]. However, none of them reduces the number of data transfers. The frameworks reviewed in this section optimize data movement in various ways, including caching, but none of the frameworks is customizable to accommodate different applications and/or platforms. Furthermore, the caching methods related to FPGAs are only suitable for a specific application or data structure.

IV. Framework Design and Implementation

A novel framework, dubbed SoftCache, has been designed to optimize the data movement between host and accelerators. Regarding the design requirements, SoftCache should be a generic and application-agnostic framework to alleviate the data-movement bottleneck by utilizing temporal locality. SoftCache should be able to be used with minimal code changes and no changes to the algorithm should be necessary. The framework should allow the programmer to tune the cache size, cache organization, and replacement policy towards the application needs. Furthermore, it should support both GPUs and FPGAs. In this section the architecture and implementation of SoftCache will be discussed.

A. Architecture

The architectural overview is shown in Figure 4. The architecture of OpenCL applications generally consist of three layers: the algorithm, OpenCL API and the OpenCL kernels that run on the accelerator.

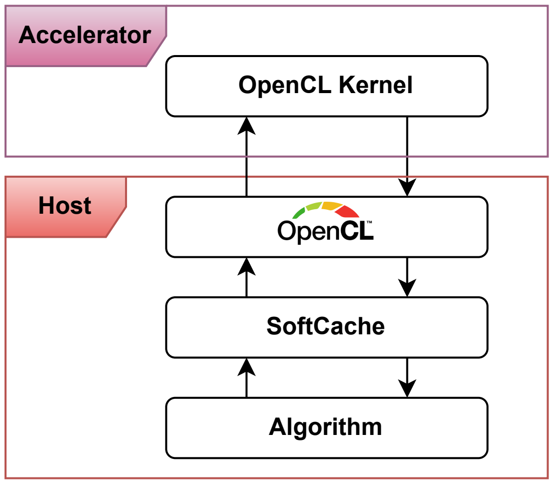


Fig. 4: High level overview of the SoftCache framework

SoftCache is a layer between the application and the OpenCL API. SoftCache uses the same function names as the OpenCL API and can be used with minimal code modifications. The framework calls the corresponding OpenCL API functions when necessary and coordinates/controls the data transfers.

B. Overview of components

Figure 5 shows an overview of the software components. An object of `Cache` class can be created by passing the command line arguments or by passing the arguments for `organisation`, `replacementPolicy`, `cacheSize` and `nrOfSets` (if the set associative cache organisation is chosen).

Multiple objects of `Cache` can be used if multiple software caches are needed. This is mainly used for FPGAs where each Super Logic Region (SLR) has its own memory bank.

The number of cache lines is determined by the constructor of the class with argument `cacheSize`. As opposed to a hardware cache, the cache line can store data of any size and is not fixed by the application. The `CacheLine` struct consists of a flag, tag, deviceAddress, size and age field:

- `flag` indicates whether the data that is referenced by the cache line is located on the CPU, Accelerator or both. This is similar to the dirty bit in a hardware cache. The enumerated flag is more verbose than the dirty bit and supports the option to write-back policy, where data is not immediately sent back to the host.
- `tag` is a pointer to the data on the host device. Unlike hardware caches, where the tag is a bit-field of the address, here the complete address is stored regardless of the cache organisation. This is required to be able to store any data size and simplifies the design. Whenever data is written to the accelerator, the pointer to the data will be compared to one or more tags in the cache, depending on

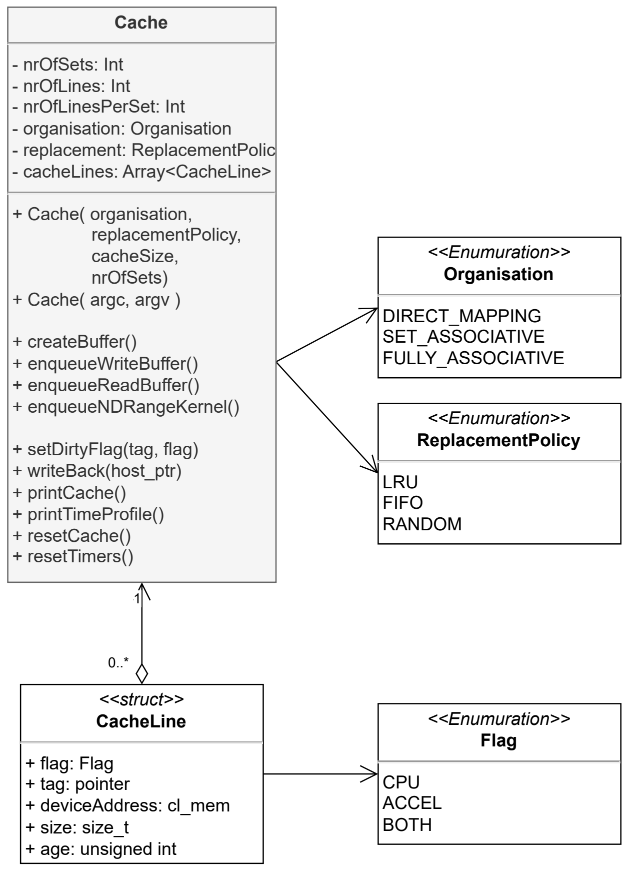


Fig. 5: Component overview of SoftCache

the cache organisation. The tag is also used as argument in the `setDirtyFlag` method. The framework does not detect changes in the data on the host, therefore when data is altered the `setDirtyFlag` method should be called to ensure coherence.

- `deviceAddress` points to a memory location on the accelerator. When `createBuffer` is called, memory will be allocated on the accelerator and the address to this buffer will be returned. When data is written to the accelerator, it must be written to one of the buffers.
- `size` stores the size of the data object. The size is used to keep track how much data is allocated on the accelerator, and in the future it could be used for more sophisticated replacement policies.
- `age` is used to store the age of the cache lines. The age is used for replacement policy algorithms, see subsection IV-D.

C. Cache indexing

As described in subsection II-A, hardware caches use a bit-field of the memory address to compute the index of the cache line, in direct mapped caches, or the set in set associative caches. However, this comes with some limitations. The block size should be a power of two, and equal for all cache lines, and the number of cache lines also needs to be a power of two.

These limitations have a big impact on the flexibility and efficiency of the cache. For example, when processing trees, the leaves and nodes of the tree can be of different sizes. To be able to store both, a naive approach would require the block size to be equal to the largest data block. This would leave a significant amount of memory unused.

Similarly, we would like to utilise the full memory space of the accelerator. For example, when using a GPU that has 6GB of RAM, only 4GB can be utilised with this approach, since 2^{32} = 4GB, and the next step would be 2^{33}, which is 8GB. Overprovisioning does not work in this case, because memory addresses would be mapped to accelerator memory that may not exist.

Instead of using a bit-field to compute the index, it'd be more efficient to use all the bits of the memory address to compute the index. Therefore a hash table approach is used for direct mapping and set associative cache organisations. To the best of our knowledge, there are no alternatives for bit-fields/hash tables to directly map the memory addresses to a line/set index. However, since the main goal of direct mapping and set associative caches is to reduce the overhead for adding and removing entries, alternative data structures such as balanced search trees could also be used as an alternative to direct mapping/set associative caching. The hash table approach is used as it is closely related to the hardware architecture of caches, and it has the lowest time complexity of $O(1)$ for search, insert, and delete.

The hash function in Equation 1 is used to compute the index:

$$i = p \mod n \tag{1}$$

where i is the index of the respective cache set, p is unsigned integer representation of the pointer to the host memory location and n is the number of cache sets. The hash function functionally works the same as bit-fields when the number of lines in a cache is a power of two, with the same limitations. Therefore the cache size should be carefully chosen.

To ensure an equal distribution of the keys, the cache size should be a prime number. One additional requirement is needed for the cache size. When taking the modulo of the number of possible memory addresses (on a 64-bit system this is 2^{64}) with a prime number, and the result is 1, this essentially means that $p \mod n$ is simply the sum of the binary representation of the address, which will result in many collisions. Therefore the cache size should be a prime number and $p \mod n \neq 1$. Additionally, the size of the data that is stored in the cache should not be a multiple of n. While this imposes limitations on the cache size, it does allow the use of almost the entire memory space since there are many prime numbers to choose from. Given the cache size, SoftCache will find the closest prime number which satisfies the constraints.

1) Direct mapping: In direct mapped caches the number of sets is equal to the number of cache lines. On a cache look-up, Equation 1 is used to find the respective set. Since the host memory is generally significantly larger than the accelerator memory, it is possible that multiple host memory addresses are mapped to the same location on the accelerator. To prevent this, cache lines that are transferred to the accelerator, will be locked until the kernel is executed. If two (or more) host memory addresses are mapped to the same location the accelerator, the second address will be mapped to a random cache line that is not yet locked. This ensures that data on the accelerator is not overwritten until the kernel is executed. After kernel execution the lines will be unlocked.

2) Set associativity: similarly to the direct mapped organisation, the number of sets should be a prime number and $p \bmod n \neq 1$, where p is again the unsigned integer representation of the host memory address, and n is the number of sets in the cache. Given the number of sets, SoftCache will automatically find the closest prime number that satisfies the constraints.

The cache is divided into N sets, each set containing an equal number of cache lines. On a cache look-up, Equation 1 is used to find the respective set. Within the set, the host memory address is compared to the tags in the cache iteratively.

The replacement policies described in subsection IV-D is used to decide which cache line to evict within a set, if no match is found.

3) Fully associativity: To find the matching cache line in fully associative caches, the tag of each cache line has to be compared with the host address. Generally, in hardware caches this is done with n comparators for n cache lines, however in software we can simply iterate through all the cache lines until a matching tag is found. If no match is found, the replacement policies described in subsubsection IV-D1 will be used to decide which cache line should be evicted. Note that, unlike in direct mapping and set associative configurations, a fully associative cache does not impose any limitations to the cache size, but the cache control overhead is significantly higher.

D. Replacement policies

Set associative and fully associative caches use replacement policies to determine which cache line should be replaced. For simplicity, only three replacement policies have been implemented. Random replacement (RR), first-in first-out (FIFO) and least recently used (LRU) are the most commonly used replacement algorithms. They are effective and have low overhead cost.

There is one potential issue that can occur when using a software cache for host-accelerator communication. Generally, for every kernel call multiple data transfers are required. Therefore it is possible that the data sent in an earlier data transfer is overwritten before the kernel is executed. To prevent this from happening, cache lines that were sent will be locked until the kernel is executed. If the replacement policy selects a locked cache line, it will pick a random non-locked cache line to replace.

1) Random replacement: When a cache line needs to be replaced, a random line is selected from the set of available lines. For fully associative caches this can be any line. In n-way set associative caches a random line is selected from the set that matches the memory address. A pseudo random number generator is used to select a random line. This means that any line in the set could be replaced, regardless of how frequently or recently it was accessed. The random replacement policy has a low execution overhead. The update time complexity is $O(1)$.

2) First-in first-out: The FIFO policy is implemented as a pointer to the last cache line in a set. This results in an update time complexity of $O(1)$. When a new block of data is added to cache, the pointer is incremented by one, and the line the pointer points to is replaced. The pointer wraps around to first element of the cache ones it exceeds the number of lines of the set. This ensures that the first element that was added to the cache, will also be the first element that will be evicted.

3) Least-recently used: To support the Least-recently used policy additional information about the cache line has to be stored. A field that stores the age of the cache line is added. When the cache is used, the age of all cache lines is increased. If there is a cache hit, the age of that specific cache line is reset to zero. The cache line with the oldest age will be evicted on replacement.

The complexity of updating a cache line is $O(n)$ where n is the number of lines in a set. This is due to the fact that the line with the highest age counter has to be found, and the age of each line has to be updated.

E. Write policies

SoftCache supports write-through and write-back, which apply to data transfers from the device to the host. Write-through writes data back to the host after each kernel invocation. Write-back will write back data to the host memory on a cache line replacement and the data can be transferred back explicitly by calling the `writeBack` method with the respective host pointer. If no argument is given, all cache lines with `ACCEL` flag will be transferred back to the host. The dirty bit is implemented as an enumerated type with the states `CPU`, `ACCEL` or `BOTH`.

V. Evaluation

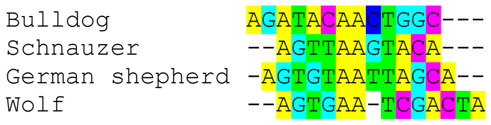
In this section we are going to evaluate the software cache on a phylogenetic application. In section V-A the phylogenetic application will be described. Section V-B describes the experimental setup that is used to perform the experiments. Section V-C describes the results of using various replacement policies. Section V-D compares direct mapping, 3-way set associative and 5-way associative caching, using the best replacement policy. Section V-E describes the effect of different cache size. In section V-F the performance of the application is evaluated using the best configurations.

A. Phylogenetic analysis

We initially describe the phylogenetic application that is used to showcase SoftCache. The phylogenetic analysis starts with the acquisition of DNA sequences, as depicted in Figure 6a. The DNA sequences are then aligned using computational tools to create a multiple sequence alignment as depicted in Figure 6b. The MSA allows for comparison of nucleotide

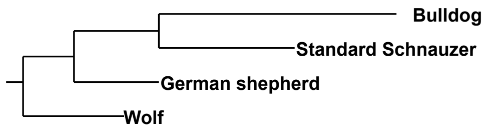
```
Bulldog          AGATACAACTGGC
Schnauzer        AGTTAAGTACA
German shepherd  AGTGTAATTAGCA
Wolf             AGTGAATCGACTA
```

(a) Dog breeds and their DNA sequences



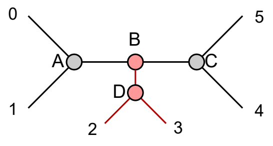
```
Bulldog          AGATACAACTGGC---
Schnauzer        --AGTTAAGTACA---
German shepherd  -AGTGTAATTAGCA--
Wolf             --AGTGAA-TCGACTA
```

(b) Multiple sequence alignment (MSA)

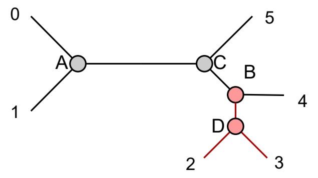


(c) Phylogenetic tree

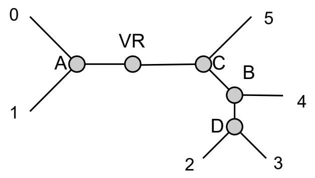
Fig. 6: The DNA sequences are aligned to create the multiple sequence alignment, which is then used to create a phylogenetic tree. Adapted from [16].



(a) Pruning



(b) Regrafting



(c) Traversal

Fig. 7: Rearranging using a sub-tree-pruning-and-regrafting (SPR) move. In the first step, the red sub-tree is pruned. A new branch is created between the nodes A and C and the old one removed. In the second step the sub-tree is reattached to branch between C and 4. During this process, the branch between C and 4 is removed, and two new branches are created. In order to evaluate the tree, the PLF in Equation 3 is used to traverse the tree in post-order: $A = PLF(0, 1)$, $D = PLF(2, 3)$, $B = PLF(D, 4)$, $C = PLF(B, 5)$ and finally $VR = PLF(A, C)$. VR is then used to calculate the likelihood of the tree with Equation 4 and 5. Adapted from [16].

or amino acid sequences between different organisms and highlights regions of conservation and divergence. Once the MSA is generated, it will be used to construct a phylogenetic tree, as shown in Figure 6c. The phylogenetic tree shows the evolutionary relationships between the organisms based on the sequence data. The length of branches indicate the evolutionary distance and the genetic divergence between the organisms.

There are many different tree topologies possible based on the same sequence data. Subtree pruning and regrafting (SPR) is used to explore alternative tree topologies. A subtree is pruned from the tree, and is regrafted to another part of the tree, as shown in Figure 7a and 7b. This creates a new topology that may be more optimal. The phylogenetic likelihood function is than used to give a score to each tree. The phylogenetic likelihood function (PLF) is recursively invoked to evaluate the given phylogenetic tree. The phylogenetic tree is represented as binary tree, and the PLF is invoked by traversing the tree in post-order. Starting at the tips and working its way to the root of the tree. This process is depicted in Figure 7c.

The phylogenetic likelihood function is a fundamental concept in the field of phylogenetics, which seeks to infer evolutionary relationships between species based on molecular data, which can be either DNA or protein sequences. The evolutionary relationships are represented by a phylogenetic tree. The inner nodes represents common ancestors and the leaves of the tree represent the species that are being investigated. However, there are many possible tree topologies which have to be evaluated to find the most likely topology. The number of possible tree topologies also increase exponentially with the number of species under investigation. For example, for 100 taxa the number of unrooted trees is $1.70 * 10^{182}$, i.e. more trees than the number of atoms in the known universe. To understand the phylogenetic likelihood function, it is important to first consider the transition probability for a given nucleotide to change from from one state to another, i.e. for a nucleotide A to transform to a nucleotide A, C, G or T, which can be thought of as a continuous-time Markov process which relies on a transition rate matrix Q. Matrix Q in this context describes the transition probability of a nucleotide. The Q matrix is a 4×4 matrix for DNA data, and a 20×20 matrix for amino acids. Matrix Q can then be used to calculate nucleotide substitution probability matrix P for a given branch length t:

$$P(t) = e^{Qt} \tag{2}$$

which gives the probability for all 16 possible transitions for DNA.

When analyzing aligned sequences it is generally assumed that every column represents an independent realisation of the Markov process under a fixed tree. Which means that the likelihood of the alignment is simply the product of the likelihoods of each nucleotide. The calculation is done in post-order, starting at the tip nodes and working up to the root of a binary tree. The partial likelihood vectors are stored at the internal nodes. The partial likelihood $\vec{L}_I^u(c)$ is the probability vector for site c of the input alignment at the inner node I, given the left and right child nodes. The vector contains the four probabilities for $u \in N$ and $N = \{A, C, G, T\}$. The partial likelihood vector is defined by the recursive equation:

$$\vec{L_I^u}(c) = \left(\sum_{s \in N} P_{u \to s}(t_l) \vec{L_X^s}(c) \right) \left(\sum_{s \in N} P_{u \to s}(t_r) \vec{L_Y^s}(c) \right) \tag{3}$$

where $P_{u \to s}(t)$ is the probability for a nucleotide to go from state u to state s, given a branch length t. $\vec{L_X^s}$ and $\vec{L_Y^s}$ are the partial likelihood vectors of the left and right child nodes, respectively.

When the root of the tree is reached, the probability vector $\vec{L_{vr}^s}(c)$ is used to calculate the final likelihood LH. First, the likelihood of site c is calculated with

$$l(c) = \sum_{s \in N} \pi_s \vec{L_{vr}^s}(c) \tag{4}$$

where $\pi_s, s \in N$ and $N = \{A, C, G, T\}$ are the prior probabilities observing nucleotides A, C, G, and T at the virtual root. The final likelihood score is the sum of logarithmic likelihood scores per site $l(c)$:

$$LH = \sum_{c=1}^{m} \log(l(c)) \tag{5}$$

The process is repeated for different tree topologies. After the likelihood is found, the tree topology is rearranged using a sub-tree-pruning-and-regrafting (SPR) move. First a subtree is selected and detached from the tree, which is called pruning. The subtree is then attached to a different branch, which is the regrafting step. This results in a new tree topology to evaluate.

There are many phylogenetic tools that can be used for evaluation, however in this work we use RAxML [26]. RAxML is a popular software tool used for phylogenetic analysis, which employs advanced algorithms to construct accurate and well-supported trees from molecular sequence data. The PLF in Equation 3 takes up a substantial amount of the execution time, accounting for up to 95% of the total time required for tree construction.

During the traversal of the tree, parent nodes become the child nodes in subsequent PLF calls.The accelerator takes the two child nodes and uses the PLF to compute the probability vector of the parent node. By using a software cache, nodes that have already been located on the accelerator are not sent again, reducing the amount of data transfer required.

There is a lot of opportunity to accelerate the PLF. However, previous attempts were limited by data movement between the host and accelerator, and not the computational capabilities [14], [16], [27]. The leaves of the tree consists of DNA or protein sequences, each sequence contains m alignment sites. The inner nodes with the probability vectors. The inner nodes consist $m * S * C$ double precision values, where m is the number of alignment sites, S is the number of states (4 for DNA and 20 for protein), and C is the factor used to account for rate heterogeneity among alignment sites. This results in a vast amount of data per node.

The results of the PLF are needed by the host for branch length optimization. There are two functions in RAxML that uses the resulting probability vectors, `sumGAMMA` and `evaluate`. `sumGAMMA` pre-computes the element-wise product of the probability vectors and is used for the Newton-Raphson procedure to optimize the branch lengths of the tree. Before optimizing the branch length, the PLF function is called to ensure that the probability vectors are up to date. `evaluate` is called on the virtual root using the equations from 4 and 5. Therefore, it is important that the corresponding nodes are transferred back to the host for processing. This is important to consider when optimizing the data movement from accelerator back to host.

B. Experimental setup

The experiments are initially conducted on a NVIDIA GeForce GTX 1060 graphics card on a personal computer, with a AMD Ryzen 5 1600X processor running at 3,6GHz.

For the experimental setup memory access traces are extracted from the RAxML application for multiple number of sequences and alignment patterns. The memory trace files contain the indices of the `xVector` or `yVector`, as well as the case number. The `xVector` and `yVector` contain the inner and tip nodes, respectively. There are three different cases to consider: tip-tip, tip-inner and inner-inner. Tip-tip takes as input two tip nodes and computes the inner parent node. The tip-inner case takes one tip node and one inner node to compute the inner parent node. And finally the inner-inner case takes two inner nodes as input as computes the parent inner node.

In the experimental setup two vectors are created that replicate the `xVector` and `yVector` of the RAxML application, using the same size and memory layout as the original application. A basic kernel has been developed to mimic the functionality of the `newviewIterative` function, which is responsible for most of the program's execution time and can benefit greatly from acceleration. Additionally, since the kernel is just a basic kernel, it is possible to replace the kernel execution time with results from previous work to get an estimate of the performance increase by utilizing caching.

For evaluation we use the byte hit ratio, a standard metric that is often used in evaluating web caching methods [28]. The byte hit ratio is the ratio between the total amount of bytes that are served by the cache, divided by the total amount of bytes that would've been sent without caching. This includes transfers from host to device, as well as transfers from device to host. Since SoftCache is used to cache data of different sizes per cache line (the tips of a phylogenetic tree are strings whereas the inner nodes are arrays of double-precision floating-point numbers), the cache hit ratio is not a good indication of performance. The byte hit ratio is directly related to the time spent on data transfers, and therefore an important metric to consider.

1) Parameters: For the phylogenetic likelihood function there are three parameters that have a large impact on the execution time and on the performance of the cache: the number of taxa, the number of distinct alignment patterns and whether the data is DNA or protein.

The number of taxa has a direct influence on the size of the tree. Since a binary tree is used to represent the species, there will $n-2$ inner nodes and n tip nodes, where n is the number of taxa. Since SoftCache stores the nodes in the cache, it will have an impact on the hit ratio. If the number of lines in the cache exceeds the number of nodes in tree, you would get the theoretical maximum hit ratio.

However, for large data it is not possible to store all the data on the accelerator, and therefore the cache size will be smaller than the number of nodes in the tree, which will have an impact on the cache hit ratio. The size of the data depends on the number of taxa, the number of distinct alignment patterns and the type of data.

For the evaluation of hit ratio, 1000 alignment patterns is used with 100, 250, 500 and 1000 taxa. The cache size will be set on 16 and 32 lines. For the evaluation of expected speed-ups 15 taxa is used with a distinct number of alignment patterns ranging from 3590 to 250.000. The number of taxa and the number of distinct alignment patterns are chosen based on the values used by Izquierdo-Carrasco et al. [27] to facilicate comparisons with related work.

C. Replacement policies

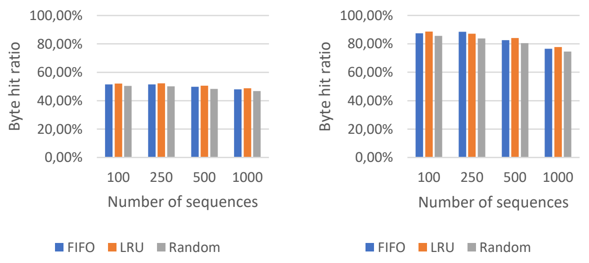
The replacement policy has a large impact on the byte hit ratio. The most optimal replacement policy should discard data that are not used for the longest time in the future. This experiment is conducted to find the best replacement policy for the phylogenetic application. The experiment is performed on a 3-way, 5-way and fully associative cache with 32 cache lines, to see if the best replacement policy varies between different cache organisations. For the same reason, the experiment is performed on various number of sequences.

The effect of the cache replacement policy is heavily dependent on memory access pattern of the application. Since the phylogenetic application does a post-order tree traversal, it is expected that recency-based policies, such as LRU and FIFO, will perform well. The random replacement policy is expected to perform mediocrely.

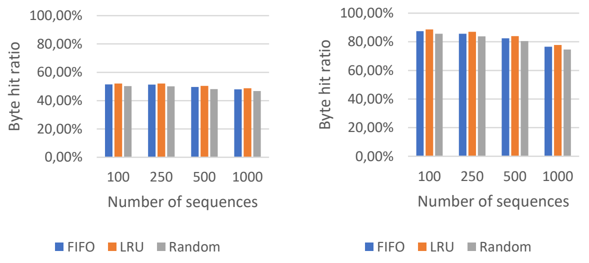
Figure 8 shows the byte hit ratio for a 3-way, 5-way and fully associative cache, respectively, for various replacement policies. It is noticeable that the pattern of all three figures is the same. There are no significant differences between the different cache organisations. LRU performs the best in all three cases, FIFO is slightly worse, and the random replacement policy scores the worst. Since the RAxML application traverses the phylogenetic trees in post-order traversal, it makes sense to evict old nodes first. However, it is interesting that a generic replacement policy that randomly replaces lines still performs quite well.

As expected, the byte hit ratio decreases when the number of taxa increases. This is due to the fact that the ratio between cache lines and the number of taxa becomes smaller.

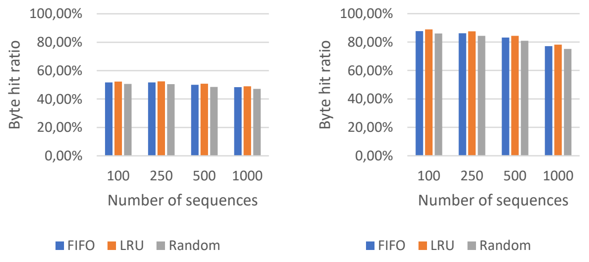
We can reduce the number of bytes transferred by up to 52.25% using write-through and 88.96% by using a write-back policy. It should be noted that with the write-through policy we can only reduce the number of bytes transferred



(a) Write-through and write-back for 3-way set associative cache



(b) Write-through and write-back for 5-way set associative cache



(c) Write-through and write-back for fully associative cache

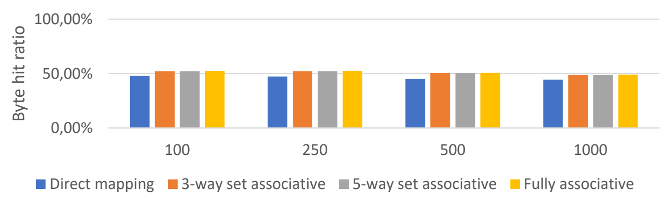
Fig. 8: Byte hit ratio for LRU, FIFI and Random replacement policies

by roughly two-thirds, since every kernel call requires two transfers from host-to-device and one transfer from device-to-host, and by using write-through only the number of host-to-device transfers can be reduced. In the general case, all three transfers are of the same size, with the exception when tip nodes are transferred.

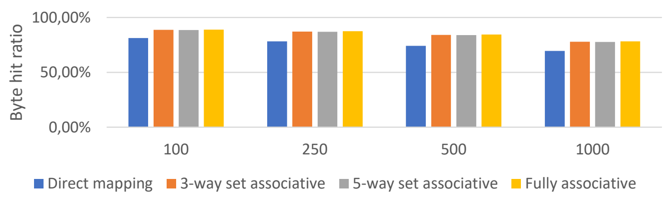
D. Cache organisations

In subsection V-C we found that the best replacement policy for this application is LRU. In order to find the best cache organisation, we compare the different cache organisations with using the best replacement policy. The experiment is performed on different number of sequences to see if there are any variances between them. The number of cache lines is set to 32. Traditionally, fully associative caches are known to perform the best, and that is also expected in this case. Fully associative caches offer the most flexibility in terms of where a specific memory block can be placed in the cache.

In Figure 9 shows the results of this experiment. For the set and fully associative caches, the LRU replacement policy is chosen since it is the replacement policy with highest byte hit

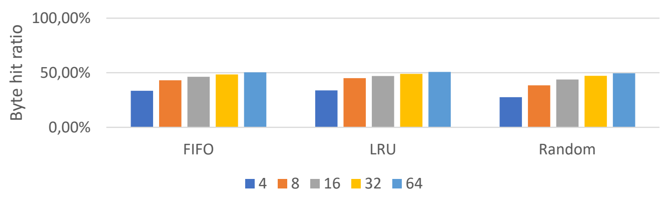


(a) Write through

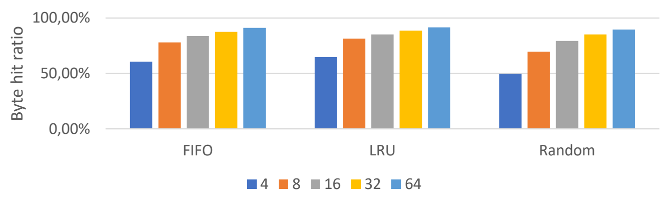


(b) Write back

Fig. 9: Byte hit ratio for different cache organizations using a cache with 32 lines and LRU replacement policy.



(a) Write through



(b) Write back

Fig. 10: Comparison on the data set with 1,000 sequences for 4, 8, 16, 32 and 64 cache lines.

ratio. Surprisingly, both set-associative caches perform only slightly worse compared to the fully associative cache, with a difference of less than one percentage point.

However, it is evident that direct mapping performs poorly. Direct mapping has the lowest overhead, however a cache miss is way more costly than the additional overhead. The overhead for set associative and fully associative caches will increase with the number of cache lines. In that case, set associative caches would still be way more beneficial because you can reduce the search space for the replacement algorithms with N times for N-way associative caches, while still benefiting from the high hit ratio of replacement policies.

E. The impact of different cache sizes

This experiment explores different cache sizes to see what the impact is. Ideally, you should configure the cache size to utilise most of the accelerator's memory. If the cache can perform well with a low number of cache lines, it will also perform well on accelerators with little dedicated memory and/or the data size per cache line can be increased, which allows for scalability. It is expected that the hit ratio will increase when the number of cache lines is increased.

In Figure 10 cache sizes between 4 and 64 are compared on a data set with 1000 sequences. It is interesting to note that halving the cache size only has a minor impact on the performance for this application. For example, when using FIFO with write through, a cache size of 64 lines has a byte hit ratio of 50.70%, while a cache size of 32 lines has a byte hit ratio of 49.03%. Halving the cache size only leads to a decrease of 2 to 3 percentage points. This is also visible in the Figure 10. When looking at the memory trace files it became evident that most nodes were used again within 8 kernel calls. A cache with 4 cache lines is too small to benefit from the memory access pattern of the application. A significant decline in performance of approximately 10 to 20 percentage points, is observed when the number of cache lines is reduced from 8 to 4. RAxML traverses the phylogenetic tree is post-order, and therefore only four cache lines is too small to benefit from cached child nodes. However, increasing the size past eight is only beneficial when the tree is regrafted, since a higher cache size increases the chance that nodes can be reused. However, this observation is very specific to the application. Since the phylogenetic application performs tree traversals, we can exploit locality between parent and child nodes and therefore achieve a significant performance improvement even with a small cache.

F. Application speed-up evaluation

For the last experiment, we will evaluate SoftCache on a phylogenetic application. The total speed-up of the application depends the performance of the accelerator, and the time spend on transferring data. It is expected that SoftCache has to most impact on applications that spend relatively little time on computations, and a large amount of time on transferring data.

The speed-up reported in this section is relative to the CPU implementation of the Newview-AVX function in RAxML. This is mainly done because related work also reported their speed-ups relative to the Newview-AVX implementation. This enables a convenient and meaningful comparison between different approaches.

A performance model was employed to assess the effect of SoftCache on accelerator performance and the overall application considering previous highly optimized GPU [27] and FPGA [14], [16] accelerators. The performance model takes into account the actual RAxML traversal order per tree and estimates the performance of the accelerator when called through SoftCache. Our performance model has the following format:

$$P = \sum_{i \in n} \Big[DT_i(A,B) + KT_i + DT_i(C) \Big] \tag{6}$$

where $i \in n$ and n is the set of kernel invocations needed by the application, $DT_i(A, B)$ is the data transfer time of moving data from the host to accelerator, $DT_i(C)$ is the data transfer time for moving data from accelerator back to host, KT_i is the kernel execution time required to perform the computations on the data. This assumes that the kernel that takes two nodes as input, A (left child node) and B (right child node), and uses it to compute C (common ancestor). $DT_i(A, B)$ is heavily dependent on the cache hit ratio. If A and B are both cached, $D_i(A, B)$ will essentially be zero. $DT_i(C)$ is influenced by the write-back and write-through policy.

The main benefit of using a performance model is that we can use the results of previous acceleration efforts to evaluate SoftCache; by considering the performance of previously published, highly optimized accelerators, we can more accurately assess the benefits that come from optimizing the data movement. The OpenCL profiler is used to measure the time it takes to transfer data from and to the GPU. TFor the FPGA platforms, the data transfer times from the respective papers are used.

1) Speed-up on GPU platform: Figure 11 shows the modelled performance using Equation 6, in terms of speed-up based on the performance of the GPU kernel described by Izquierdo-Carrasco et al. [27]. The kernel execution times from Figure 4 of the paper were used in the performance model. The time spent on data transfers were measured by running the application with a basic kernel. The paper assumes no memory transfers for the kernel execution times, and therefore serve as an upper limit of the speed-up that can be achieved. The kernel was invoked roughly 54.000 to 64.000 times, depending on the number of distinct alignment patterns.

The speed-up is shown for three different policies: no cache, write-through and write-back. The speed-up shown is relative to the CPU implementation of the Newview-AVX. The speed-up scales when the number of distinct alignment patterns increases, however it is clear that the no cache implementation is worse compared to the CPU version regardless of the number of distinct alignment patterns. A speed-up of 1.25x was achieved with write-through, compared to the no cache implementation, and 1.6x with write-back. The maximum possible speed-up with this kernel was 1.97x. This means that 63.4% and 81% of the maximum theoretical speed-up was achieved with caching.

It is noticeable that with and without caching the performance is worse than the RAxML-AVX implementation for a low number of alignment patterns. The kernel execution times are in that case a significant portion of the total execution time, as seen in the first section of Figure 14b. No speed-up was achieved without caching, which shows that optimizing the data movement is critical to achieve a decent speed-up on the GPU.

2) Speed-up on FPGA platforms: Figure 12 shows the speed-up that can be achieved based on the performance of the optimized hardware architecture described by Malakonakis et al. [14]. The FPGA instance is based on the dual core execution times of the Xilinx ZCU102 development board, which contains reconfigurable logic combined with an ARM processor. The ZCU102 has a shared-memory architecture

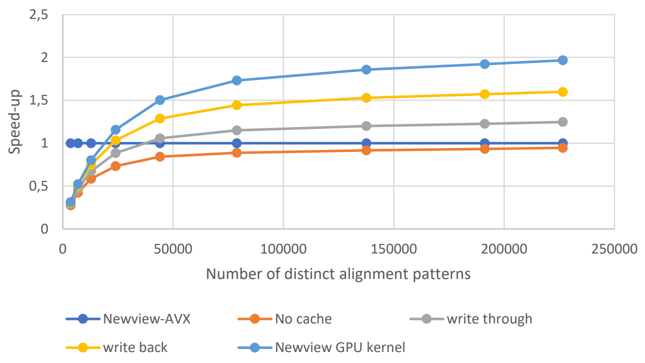


Fig. 11: GPU [27] performance improvement using a fully associative 8-block cache with LRU. Newview-AVX is the reference CPU implementation. The Newview GPU kernel is the theoretical peak performance (no data movement).

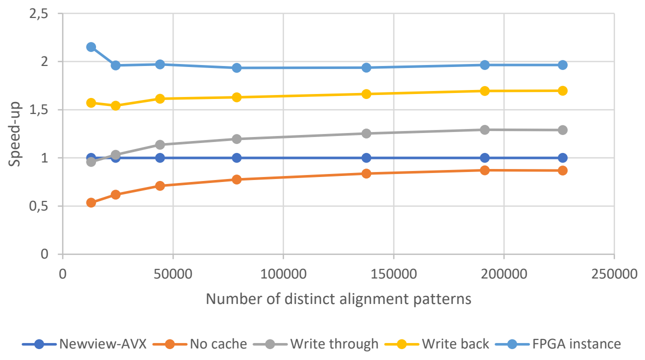


Fig. 12: Performance improvement of the FPGA accelerator by Malakonakis et al. [14] using a fully associative 8-block cache with LRU.

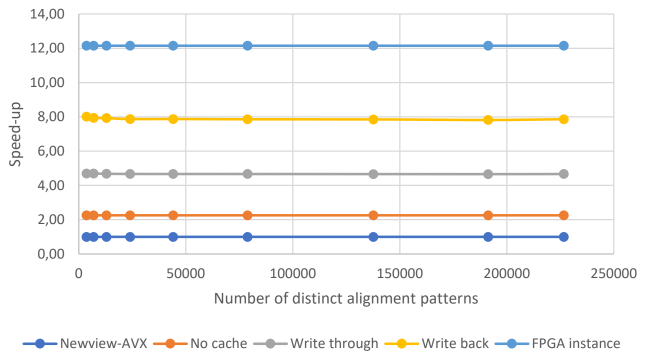


Fig. 13: Performance improvement of the FPGA accelerator by Alachiotis et al. [16] using a fully associative 8-block cache with LRU.

which means that the time spent on data transfers is negligible and serve as an upper limit of what is achievable with caching. The paper reports the execution time per phylogenetic likelihood function (PLF) call, while the performance of the GPU was reported for the entire run of the program. To present comparable results with the GPU, the execution time of a single PLF call is multiplied with the number of calls of the entire RAxML run.

Interestingly, the speed-up is linearly related to the number of distinct number of alignment patterns. Due to the nature of the application, both the GPU and FPGA kernel seem to perform comparably well. The PLF is a memory-bound operation, and the GPU has a higher memory bandwidth, and benefits more from memory coalescing. On the other hand, FPGAs parse the data sequentially, and is mainly benefiting from a deeper pipeline. Since the FPGA parses the data sequentially, the speed-up is also linear to the number of alignment patterns. This means that the FPGA instance is faster on a low number of distinct number of alignment patterns, as the GPU cannot benefit from memory coalescing that much. However, as the number of alignment patterns increases, the gap between the FPGA instance and GPU kernel decreases.

The performance of the no cache implementation is still worse compared to the CPU implementation. For 226637 distinct alignment patterns the speed-up is 1.29x and 1.70x for write-through and write-back, respectively. The maximum theoretical speed-up is 1.96x for the FPGA instance without any data movement.

Alachiotis et al. [16] achieve higher speed-ups for the kernel execution. In their work they executed the PLF on a Amazon AWS EC2 F1 server, which contains a datacenter grade FPGA accelerator cards. Here it becomes even more apparent that communication is the bottleneck. Figure 13 shows the speed-ups for a complete run with various alignment patterns using the FPGA instance and data transfer times of their paper. A speed-up of 7.85x over the CPU version was achieved with write-back method, which is comparable to the speed-up of 7.8x that they were getting with caching and double buffering. They only cached the data from host to accelerator. With the double buffering approach they managed to hide most of the data transfer cost from accelerator to host.

This shows that our general solution achieves equally good results as the application-specific solution that was proposed by Alachiotis et al. [16]. Since double buffering overlaps computation with data movement, it will also lose its effectiveness when the computations are further optimized.

3) Time breakdown: Figure 14 illustrates a time breakdown of the complete application for different SoftCache configurations. The figure shows that the time spent on data transfers was considerably reduced with caching. The first section shows the breakdown for the GPU-accelerated execution of RAxML [27]. For 226,637 distinct alignment patterns on the GPU platform, the time spent on communication without caching exceeds the time spent on kernel execution. The time spent on communication is decreased by up to 47% and 79% with write-through and write-back, respectively. The accelerator performance was improved by 1.3x and 1.7x, resulting in an application speed-up of 1.3x and 1.6x for write through and write back, respectively.

The second section of the figure shows the breakdown for the FPGA-accelerated execution of RAxML based on the implementation of Malakonakis et al. [14]. Due to the faster kernel times and slower data movement, it becomes more apparent that data movement causes significant bottlenecks for the application. However, with caching, the time spent on communication was drastically reduced, by up to 48% and 82% for write-through and write-back, respectively. The accelerator performance was improved by 1.5x and 2.0x, resulting in an application speed-up of 1.3x and 1.7x for write through and write back, respectively.

The third section shows the results for the FPGA-accelerated execution of RAxML based on the work of Alachiotis et al. [16]. Without caching, the processing on the FPGA occupies less than 20% of the total execution time, with the remainder used for data transfers. The speed-up without caching is only 2.26x over the CPU execution. With caching, the time spent on communication is reduced by up to 63% and 88% for the write-through and the write-back policies, respectively. The accelerator performance was improved by 2.1x and 3.5x, resulting in up to 4.7x and 7.9x speedup over the CPU implementation, using write back and write through.

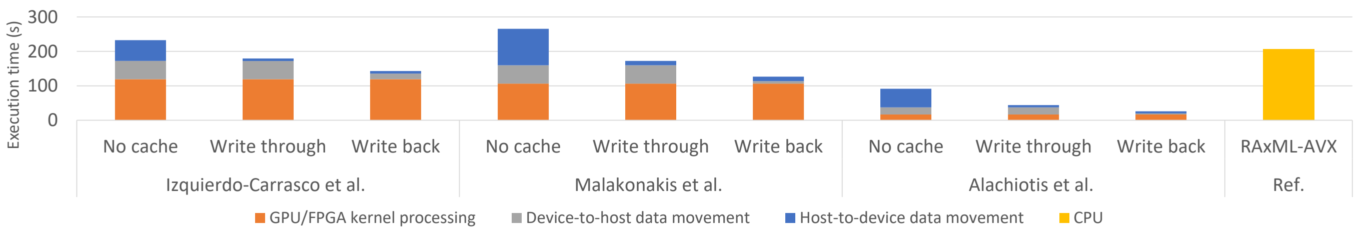
The time breakdown also shows that the platform with the highest communication-to-computation ratio (FPGA [16]), benefits the most from caching.

VI. Conclusion and future work

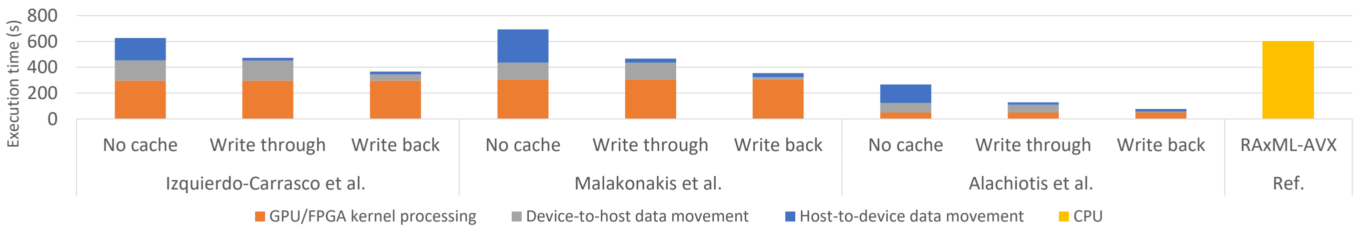
In this paper, we explored the use of a software cache to optimize data movement and improve the overall performance of accelerator-based applications. The main research question states ”How can we exploit locality to optimise data movement on systems that use PCI connected accelerator cards with dedicated memory for I/O bound applications?”. The results show that SoftCache is able optimise the data movement by exploiting locality on both GPUs and FPGAs.

In order to make SoftCache applicable to any application, the framework supports different cache organisations and replacement policies, that can be tuned towards the application. Each cache line can store data of any size, and therefore it is not needed to have separate caches for different types of data, i.e. it can store nodes, vectors or matrices of different sizes in the same cache. The write policies allow SoftCache to optimize data movement in both directions. The size of the cache is also configurable to accommodate different memory sizes. However, the number of cache lines/sets are limited to prime numbers of direct mapping and set associative caches, which means that SoftCache does utilise the entire memory of the accelerator effectively using these cache organisations.

The framework has been evaluated on a phylogenetic application. The results show that SoftCache can drastically decrease the amount of data transferred in the phylogenetic application. Up to 81% of the maximum theoretical speed-up was reached on the GPU implementation, and on the FPGA



(a) 78,873 distinct alignment patterns



(b) 226,637 distinct alignment patterns

Fig. 14: Time breakdown of full application execution times using GPU/FPGA accelerators without and with SoftCache, including transfer and kernel execution times for 78,873 and 226,637 alignment patterns. The first section shows the time breakdown of the GPU implementation by Izquierdo-Carrasco et al. [27], the second section shows the FPGA implementation by Malakonakis et al. [14] and the last section shows the cloud FPGA implementation by Alachiotis et al. [16].

platform the cache performs just as well as the application-specific optimisations of related work, getting speed-ups of up to 7.9x on the overall application. The write-back method can significantly improve the performance on this application. The results show that regarding the different cache organisations, that direct mapping scores the worst. Set associative and fully associative caches show similar results, with fully associative caching being slightly better. A fully associative cache with the LRU replacement policy can get up to 89% byte hit ratio.

The results show that by using SoftCache we can optimize data movement and reduce the time spend on moving data by nearly 89%. The speed-ups achieved as a result of this, means that we can significantly decrease the runtime of an application. Overall, we managed to improve accelerator performance by up to 1.7x and 3.5x, resulting in a speed-up of 1.6x and 7.9x over the fastest CPU version, for GPU and FPGA, respectively. Evaluating large phylogenetic trees can easily take more than a week on a CPU, a speed-up of 7.9x means that it can now be completed within a day. It is also clear that SoftCache is most effective on applications and platforms where the communication-to-computation ratio is high, as seen with server-grade FPGA.

For future work, SoftCache can be improved in several ways. Firstly, we can extend its functionality to support multiple memory banks. Although SoftCache currently supports a separate cache instance for each memory bank, introducing features that facilitate data movement between different cache instances would allow applications to transfer data between banks without the need to transfer it back to the host memory. This could be achieved, for example, by utilizing DMA (Direct Memory Access). This would benefit FPGAs with multiple super logic regions, where individual memory banks can be dedicated to specific logic regions This would also allow for caching data on the accelerator while maintaining the option to update the cache table on the host, without having to move the data to the host, in scenarios where data is shifted from one memory bank to another. Separate cache instances can also be used to utilise multiple accelerators that are connected to the same host.

Secondly, SoftCache does not utilise the entire memory of the accelerator effectively. To address this, we can explore ways to better utilise the available memory. An improved hash function that does not require prime numbers as cache size would increase the flexibility of direct mapped and set associative caches. Allowing users to overprovision the cache would also optimize the utilization of the accelerator memory. Currently, the cache size should be selected to support the worst-case scenario where each cache line contains the largest data size in the application. Instead, we could overprovision the cache and if there is not enough room in the cache to add data on a cache miss, SoftCache could remove cache lines based on the replacement policy until there is enough space to add the data. Note that this does not work for direct mapped caches, since it would cause memory addresses to be mapped to cache memory that does not exist and a replacement policy is required to decide which lines to remove.

Finally, the replacement policies could take into account the size of the data when deciding which cache line to replace. Large data takes longer to transfer, and should therefore have a higher priority to stay in the cache. These improvements would enhance SoftCache's performance and increase its versatility for various applications.

References

[1] J. Duarte, P. Harris, S. Hauck, B. Holzman, S.-C. Hsu, S. Jindariani, S. Khan, B. Kreis, B. Lee, M. Liu *et al.*, "Fpga-accelerated machine learning inference as a service for particle physics computing," *Computing and Software for Big Science*, vol. 3, no. 1, pp. 1–15, 2019.

[2] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran *et al.*, "Fast inference of deep neural networks in fpgas for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.

[3] S. Okada, K. Murakami, S. Incerti, K. Amako, and T. Sasaki, "Mpexs-dna, a new gpu-based monte carlo simulator for track structures and radiation chemistry at subcellular scale," *Medical Physics*, vol. 46, no. 3, pp. 1483–1500, 2019.

[4] J. Li, Y. Chi, and J. Cong, "Heterohalide: From image processing dsl to efficient fpga acceleration," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 51–57.

[5] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "Genax: A genome sequencing accelerator," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 69–82.

[6] K. Wetterstrand. (2021) Dna sequencing costs: Data. [Online]. Available: https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data

[7] "Maximizing unified memory performance in cuda," https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/, accessed: 16-10-2022.

[8] M. Chen, Y. Hao, L. Hu, M. S. Hossain, and A. Ghoneim, "Edge-cocaco: Toward joint optimization of computation, caching, and communication on edge cloud," *IEEE Wireless Communications*, vol. 25, no. 3, pp. 21–27, 2018.

[9] P. Fent, A. van Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper, "Low-latency communication for fast dbms using rdma and shared memory," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1477–1488.

[10] N. AlSaber and M. Kulkarni, "Semcache: Semantics-aware caching for efficient gpu offloading," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 421–432.

[11] P. Knoben, "Software caching for tree-based algorithms on accelerator cards," Master's thesis, University of Twente, 2021.

[12] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for cpu-gpu architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 165–174.

[13] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving gpu energy efficiency," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, pp. 1–23, 2014.

[14] P. Malakonakis, A. Brokalakis, N. Alachiotis, E. Sotiriades, and A. Dollas, "Exploring modern fpga platforms for faster phylogeny reconstruction with raxml," in *2020 IEEE 20th International Conference on Bioinformatics and Bioengineering (BIBE)*. IEEE, 2020, pp. 97–104.

[15] M. S. B. Altaf and D. A. Wood, "Logca: A high-level performance model for hardware accelerators," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 375–388, 2017.

[16] N. Alachiotis, A. Brokalakis, V. Amourgianos, S. Ioannidis, P. Malakonakis, and T. Bokalidis, "Accelerating phylogenetics using fpgas in the cloud," *IEEE micro*, vol. 41, no. 4, pp. 24–30, 2021.

[17] Y. Fountis. (2023) How does the browser cache work? [Online]. Available: https://pressidium.com/blog/browser-cache-work/

[18] R. Asai, M. Okita, F. Ino, and K. Hagihara, "Transparent avoidance of redundant data transfer on gpu-enabled apache spark," in *Proceedings of the 11th Workshop on General Purpose GPUs*, 2018, pp. 22–30.

[19] P. Begunkov, "dlmcl: Optimization of cpu-gpu memory transfers for opencl devices with hsa," in *Proceedings of the 5th International Workshop on OpenCL*, 2017, pp. 1–2.

[20] D. Negrut, R. Serban, A. Li, and A. Seidl, "Unified memory in cuda 6.0. a brief overview of related data access and transfer issues," *SBEL, Madison, WI, USA, Tech. Rep. TR-2014-09*, 2014.

[21] P. Barua, J. Zhao, and V. Sarkar, "Ompmemopt: Optimized memory movement for heterogeneous computing," in *European Conference on Parallel Processing*. Springer, 2020, pp. 200–216.

[22] A. Rasch, J. Bigge, M. Wrodarczyk, R. Schulze, and S. Gorlatch, "docal: high-level distributed programming with opencl and cuda," *The Journal of Supercomputing*, vol. 76, no. 7, pp. 5117–5138, 2020.

[23] X. Wei, Y. Liang, and J. Cong, "Overcoming data transfer bottlenecks in fpga-based dnn accelerators via layer conscious memory management," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[24] D. de la Chevallerie, J. Korinth, and A. Koch, "fflink: A lightweight high-performance open-source pci express gen3 interface for reconfigurable accelerators," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 4, pp. 34–39, 2016.

[25] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "Riffa 2.1: A reusable integration framework for fpga accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 4, pp. 1–23, 2015.

[26] A. Stamatakis, "Raxml version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies," *Bioinformatics*, vol. 30, no. 9, pp. 1312–1313, 2014.

[27] F. Izquierdo-Carrasco, N. Alachiotis, S. Berger, T. Flouri, S. P. Pissis, and A. Stamatakis, "A generic vectorization scheme and a gpu kernel for the phylogenetic likelihood library," in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, 2013, pp. 530–538.

[28] K. Baskaran and C. Kalaiarasan, "Improving hit ratio and byte hit ratio using combined pre-fetching and web caching," *International Review on Computers and Software*, vol. 9, no. 8, pp. 1426–1433, 2014.