POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

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

Supervisor
Prof. Luciano Lavagno

Candidate
Giovanni Brignone
ID: 274148

Academic year 2020-2021

Abstract

The end of the Moore's Law validity is making the performance advance of Software run on general purpose processors more challenging than ever. Since current technology cannot scale anymore it is necessary to approach the problem from a different point of view: application-specific Hardware can provide higher performance and lower power consumption, while requiring higher design efforts and higher deployment costs.

The problem of the high design efforts can be mitigated by the High-Level Synthesis (HLS), since it helps improve designer productivity thanks to convenient Software-like tools.

The problem of high deployment costs can be tackled with FPGAs, which allow implementing special-purpose Hardware modules on general-purpose underlying physical architectures.

One of the open issues of HLS is the memory bandwidth bottleneck which limits performance, especially critical in case of memory-bound algorithms.

FPGAs memory system is composed of three main kinds of resources: registers, Block RAMs (BRAMs) and external Dynamic RAMs (DRAMs). Current HLS tools allow exploiting this memory hierarchy manually, in a scratchpad-like fashion: the objective of this thesis work is to automate the memory management by providing an easily integrable and fully customizable cache system for HLS.

The proposed implementation has been developed using Vitis™HLS tool by Xilinx Inc..

The first development phase produced a single-port cache module, in the form of a C++ class configurable through templates in terms of number of sets, ways, words per line and replacement policy. The cache lines have been mapped to BRAMs. To obtain the desired performance, an unconventional (for HLS) multiprocess architecture has been developed: the cache module is a separate process with respect to the algorithm using it: the algorithm logic sends a memory access request to the cache and reads its response, communicating through FIFOs.

In the second development phase, the focus was put on performance optimization, in two dimensions: increasing the memory hierarchy depth by introducing a Level 1 (L1) cache and increasing parallelism by enabling multiple ports.

The L1 cache is composed of cache logic inlined in the user algorithm: this solution allows to cut the costs of FIFOs communications. To keep L1 cache simple it has been implemented with a write-through write policy, therefore it provides advantages for read accesses only. It is configurable in the number of lines and each line contains the same number of words of the associated Level 2 (L2) cache.

The multi-port solution provides a single L2 cache accessible from multiple FIFOs ports, each of which can be associated with a dedicated L1 cache. It is possible to specify the number of ports through a template parameter and it typically corresponds to the unrolling factor of the loop in which the cache is accessed.

In order to evaluate performance and resource usage impact of the developed cache module, multiple algorithms with different memory access patterns have been synthesized and simulated, with all data accessed to DRAM (performance lower bound), to BRAM (performance higher bound) and to cache (with multiple configurations).

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

AXI Advanced eXtensible Interface

BRAM Block RAM

CC Clock Cycle

DRAM Dynamic RAM

DSP Digital Signal Processor

FF Flip-flop

FIFO First-In First-Out

FPGA Field-Programmable Gate Array

FSM Finite-State Machine

HDL Hardware Description Language

HLS High-Level Synthesis

HW Hardware

II Initiation Interval

IPC Inter-Process Communication

L1 Level 1

L2 Level 2

LRU Least Recently Used

LSB Least Significant Bit

LUT Lookup Table

MSB Most Significant Bit

RAM Random Access Memory

RAW Read After Write

RTL Register-Transfer Level

SW Software

1 Background

The literature about cache systems, the High-Level Synthesis state of the art and an analysis of the resources available on board modern FPGAs are the fundamental background for this thesis work.

1.1 Cache

Memory devices are crucial components of computing systems as they can pose an higher bound in terms of performance, especially when executing memory-intensive algorithms. The ideal memory should be fast, large and cheap, but current technology forces the designer to choose a trade-off between the metrics.

A common solution to this problem is to set up a memory hierarchy in which fast but small memories are paired with large but slow memories, which allows getting good performance on average while containing costs.

This hierarchy can be managed by two main approaches:

- *Scratchpad*: different memories belongs to different addressing spaces: the user is in charge of manually choosing what memory to access: this approach allows to optimally exploit the hierarchy at the cost of high design effort.

- *Cache*: different memories belongs to the same addressing space: the system automatically uses the whole hierarchy, exploiting spatial locality (accessed data is likely physically close to previously accessed data) and temporal locality (accessed data has likely recently been accessed), which are typical of many algorithms.

1.1.1 Structure

A cache memory is logically split into *sets* containing *lines* (or *ways*) which are in turn made up of *words*, as shown in Figure 1.1.

Whenever a word w is requested, there are two possibilities:

- *Hit*: w is present in the cache: the request can be immediately fulfilled.

- *Miss*: w is not present in the cache: it is necessary to retrieve it from lower level memory before fulfilling the request.

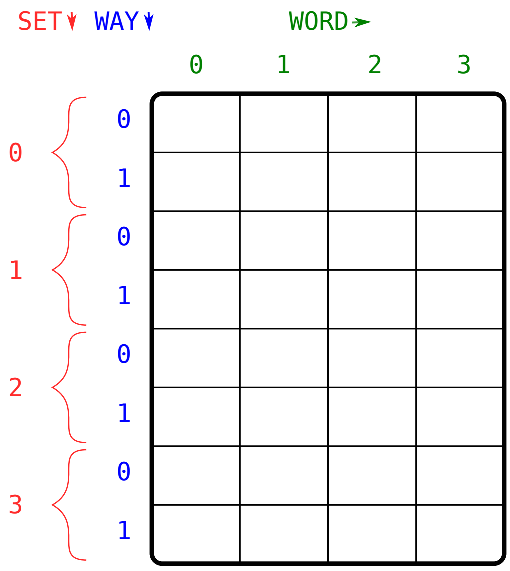


Figure 1.1: Cache logic structure.

During the data retrieving, a cache line is filled with a block of contiguous words loaded from the lower level memory, trying to exploit spatial locality of future accesses, while mapping policies and replacement policies determine which cache line to overwrite, trying to exploit temporal locality.

If the cache memory is writable, data consistency is ensured by a consistency policy.

1.1.2 Policies

Mapping policy

The mapping policy is in charge of statically associating a lower level memory line to a cache set.

The *set associative* policy is the most common mapping policy: given a cache memory with s sets of w words, the word address (referred to the lower level memory) bits are split into three parts (as shown in Figure 1.2):

1. $\log_2(w)$: offset of the word in the line.
2. $\log_2(s)$: set.
3. Remaining MSBs: tag identifying the specific line.

Special cases of this policy are:

- *Direct mapped* policy: each set is composed of a single line: the set bits identify a specific cache line, therefore there is no need for a replacement policy.

Figure 1.2: Set associative policy address bits meaning.

- *Fully associative* policy: there is only a single set, therefore the line is fully determined by the replacement policy.

Replacement policy

The replacement policy is in charge of dynamically associating a lower level memory line to a cache line of a set.

Multiple solutions of this problem have been developed, trying to maximize the temporal locality exploitation. Among the most commonly used solutions there are:

- *First-In First-Out*: the line to be replaced is the first one that has been inserted to the cache.

- *Least Recently Used*: the line to be replaced is the one that has least recently been accessed.

Consistency policy

The consistency policy is in charge of ensuring data consistency between memories belonging to different hierarchy levels.

The most common solutions to this problem are:

- *Write-back*: write accesses are performed to the highest level memory and lower level memories are updated when the cache line is replaced only.

- *Write-through*: each write access is propagated along the whole hierarchy.

1.1.3 Benefits

A two-level memory hierarchy is composed of a L1 cache memory (access time: t_{L1}; access energy: E_{L1}) and a L2 memory (access time: t_{L2}; access energy: E_{L2}), with $t_{L1} << t_{L2}$ and $E_{L1} << E_{L2}$.

This memory hierarchy is accessed n_{tot} times and n_{hit} of these accesses are cache hits.

The *hit ratio* is defined as:

$$H := \frac{n_{\text{hit}}}{n_{\text{tot}}} \tag{1.1}$$

The *average access time* and *energy* are defined as:

$$\begin{cases} \bar{t}(H) := H t_{\mathrm{L1}} + (1-H) t_{\mathrm{L2}} \\ \overline{E}(H) := H E_{\mathrm{L1}} + (1-H) E_{\mathrm{L2}} \end{cases} \tag{1.2}$$

Equation 1.2 shows the criticality of the *hit ratio*: the performance and power consumption advantages provided by the cache are significant if and only if H is sufficiently near to 1.

1.2 Field-Programmable Gate Array

Field-Programmable Gate Arrays are integrated circuits able to implement special purpose circuits described in Hardware Description Language (HDL), thanks to their programmable logic blocks and interconnections.

1.2.1 Memory system

A FPGA memory system is typically made up of:

- Registers: the fastest but most expensive memories, therefore they are only a few.
- BRAMs: on chip Random Access Memories (RAMs) accessible through simple and fast interface.
- External DRAMs: off chip DRAMs accessible through complex and slow interface (e.g. AXI).

1.3 High-Level Synthesis

High-Level Synthesis (HLS) is an Electronic Design Automation technique aimed at translating an algorithm description in a high-level Software programming language (such as C and C++) into a HDL description.

HLS allows designing more complex systems in less time, compared to HDL design, moreover makes the Hardware and Software co-design easier, at the cost of limited low-level control.

This Section is mainly referred to *Vitis™ HLS 2020.2* [1] and *2021.1* [2], but most currently available HLS commercial tools provide equivalent features.

1.3.1 Workflow

The typical HLS workflow consists of:

1. *SW implementation*: the top-level entity is a C function: the function arguments are the entity ports and the functionality is implemented in SW; in order to guarantee synthesizability some constraints should be respected (e.g. no dynamic memory allocation).

2. *SW verification*: the testbench can be developed as a simple main function which calls the top-level entity function, therefore the functionality is verified like any SW: it is possible to exploit traditional tools (e.g. debuggers, print statements...).

3. *HW synthesis*: the synthesizer generates a Register-Transfer Level (RTL) description of the top-level entity. It is possible to generate different architectures by setting up some parameters through dedicated directives.

4. *HW verification*: the RTL description is simulated, to make sure that SW and HW outputs match.

1.3.2 Optimization techniques

HLS tools provide different optimization techniques which can be set up by means of compiler directives.

Pipelining

Given a set of sequential stages (e.g. A, B and C of Figure 1.3) which compose an operation (e.g. A + B + C of Figure 1.3) which has to be executed multiple times, the pipelining technique inserts pipeline registers at the output of each stage, so that each stage can run in parallel on different input data (e.g. at the third clock cycle, while C is processing first input, B is processing second input and A is processing third input). The introduced parallelism allows to increase the throughput at a limited additional area cost (only pipeline registers and a FSM are required).

The throughput is determined by the interval (expressed in number of clock cycles) between the beginning of two consecutive executions of the operation, which is called Initiation Interval (II). The optimal pipeline has an II equal to one: at the steady state, one output per clock cycle is produced.

The pipelining can be performed at instruction level, within a loop or a function, or at function level (in HLS terminology this particular kind of pipelining is called *Dataflow*).

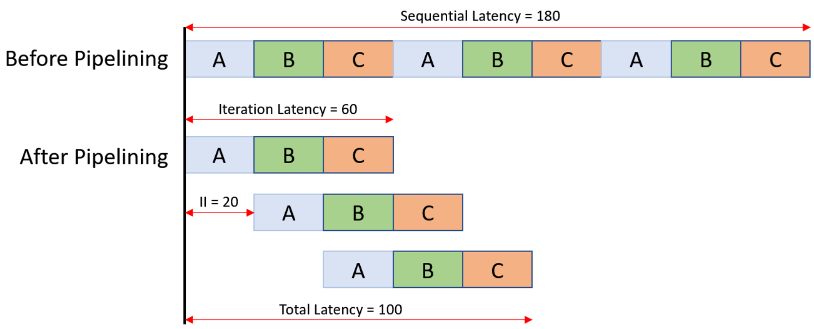


Figure 1.3: Pipelining example.

Loop unrolling

The logic of a rolled loop allows the execution of one iteration at a time: if the loop iterates N times and each iteration has a latency L_{it}, the total loop latency is equal to $L_{loop,rolled} := N \cdot L_{it}$.

The loop unrolling technique instantiates the logic for executing f iterations at a time (where f is the unrolling factor). If there are no dependencies between different iterations, the latency of the unrolled loop is: $L_{loop,unrolled}(f) := \frac{N}{f} \cdot L_{it}$.

Loop unrolling can improve both latency and throughput, but it is expensive in terms of resource usage, since they are multiplied by f.

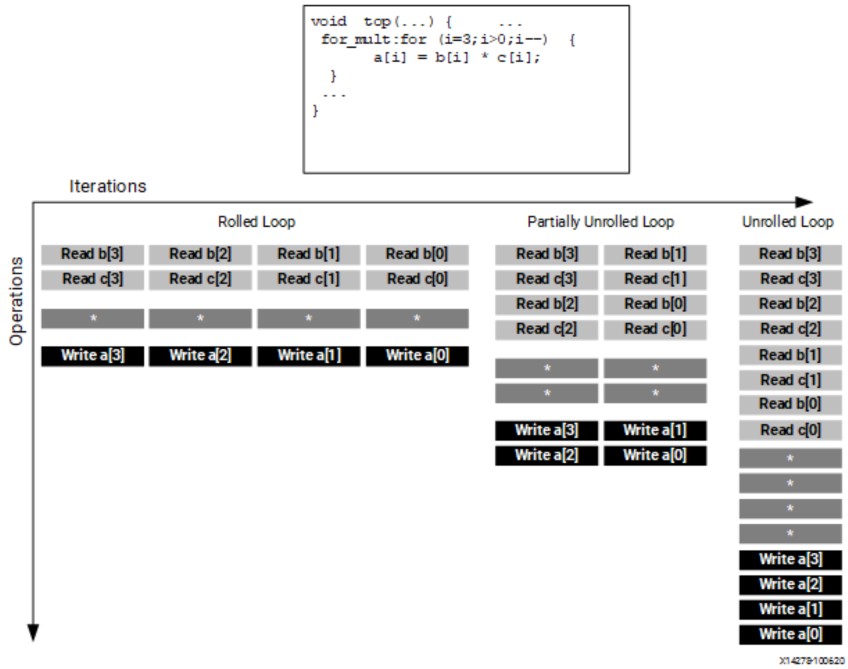


Figure 1.4: Loop unrolling example.

Memory optimizations

- <u>On-chip memory:</u>
  - **Array partitioning**: given a partitioning factor f, an array is split into f portions, each one mapped to a dedicated memory element.

    This allows multiple concurrent accesses to the same array, at the cost of higher memory elements usage.

    Figure 1.5 shows different partitioning modes.
- <u>Off-chip memory:</u>

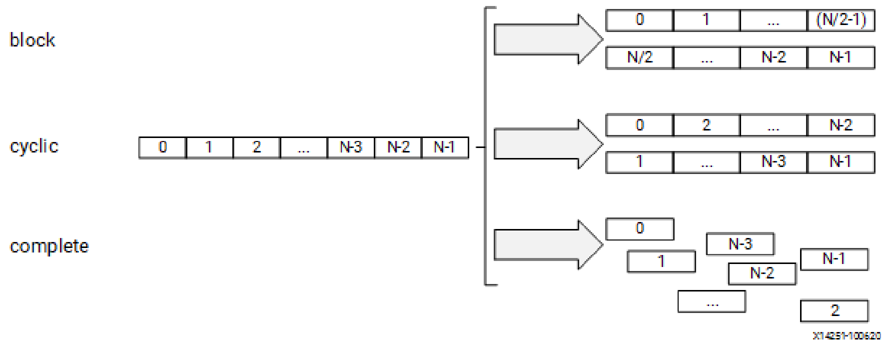


Figure 1.5: Array partitioning examples.

- **Interface widening**: multiple data elements are packed into a single bigger word, to perform multiple accesses at the same time.
- **Burst accesses**: multiple memory accesses are aggregated into AXI bursts to reduce overall latency and improving throughput.

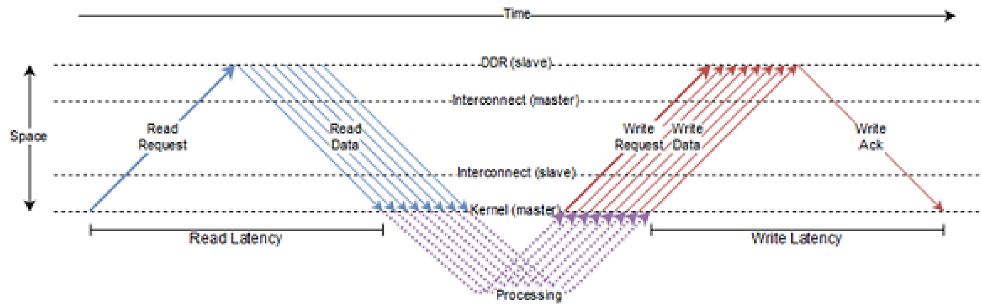


Figure 1.6: Burst read and write example.

2 Motivation

HLS tools are currently unable to automatically exploit the memory hierarchy present on FPGAs: the only way to take advantage of them is the manual management in a *scratchpad*-like manner, which requires additional design and verification efforts.

The proposed solution **automates the low-level memory management** through a cache module for HLS, which works as an interface with the off-chip DRAM (accessible through an AXI bus) and stores its data to on-chip BRAMs and registers.

The proposed cache module has the **dual purpose** of:

- *Reducing the number of DRAM accesses*: misses only needs to access DRAM.
- *Optimizing DRAM accesses*: lines are accessed in bursts through a widened memory interface.

FPGAs provide multiple DRAM ports and HLS can assign each array to a different port: this allows implementing **array-specific** caches, which in general can be easily tuned to reach high hit ratios, since access patterns to a single array are usually regular and there is no interference between accesses to different arrays.

A special attention has been put on **user-friendliness**:

- *Configurability*: cache characteristics can be set through parameters.
- *Integrability*: cache can be inserted into existing designs without requiring many changes.
- *Observability*: critical cache data (e.g. hit ratio) can be profiled during SW simulation for easing the cache parameters tuning.

2.1 Ma's cache

Liang Ma et al. proposed a `C++` cache implementation [5] compatible with the *SDAccel™* HLS tool.

It is an array-specific cache module in the form of different `C++` classes: each of them implements an access type (read only/write only and read write) and a mapping policy (direct mapped and set associative).

To improve the *integrability* the `operator[]` has been overloaded so that the cache object can be accessed in the same way as array variables, minimizing the required changes to the code which integrates the cache.

This architecture is **inlined**: the cache logic is directly inserted in the user algorithm logic. This is the major limitation of this solution, since the additional logic inserted in the algorithm may make it too complex and worsen the generated circuit performance.

2.2 Proposed solution

The primary goal of this thesis work is to develop the *Basic cache*, a cache architecture which runs in a separate process with respect to the application using it, trying to solve the main limitation of *Ma's cache*: the application logic cluttering due to the inlining.

This architecture has been then optimized in two dimensions:

- *Multi-levels cache*: a L1 cache are added to the cache hierarchy, with the objective of further reducing memory access latency.
- *Multi-ports cache*: multiple cache access points are added to the cache, each one with a dedicated L1 cache, so that multiple requests can be served in parallel.

3 Basic cache

The *Basic cache* is aimed at solving the main limitation of *Ma's cache*: application logic cluttering due to inlining.

3.1 Architecture

The fundamental idea behind the *Basic cache* is that the cache logic is inserted in a separate process with respect to the application logic accessing it (Figure 3.1): this isolation makes the cache always perform in the same manner, independently of the algorithm accessing it, while keeping the application logic as clean as possible, since application only has to write requests to cache and read responses, instead of integrating the whole cache logic.

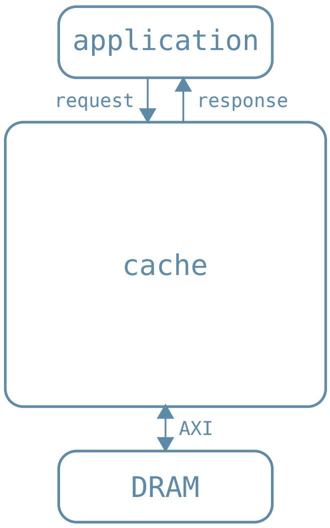


Figure 3.1: *Single-process Basic cache* architecture.

3.1.1 Functionality

If application A needs to access the array associated with the cache C:

1. *A* sends the access request to *C*: operation (i.e. read or write), address and (in case of write access) data.
2. *C* receives the request and checks if the requested address causes a miss.
3. (in case of miss) *C* prepares its BRAM memory for fulfilling the requested access:
   - (if needed) writes back to DRAM the BRAM line to be replaced.
   - reads from DRAM the requested line and store it to BRAM.
4. *C* performs the requested access to BRAM and (in case of read request) sends requested data to *A*.

3.1.2 Characteristics

The *Basic cache* is compliant with the set associative mapping policy and the write-back consistency policy. It is configurable in terms of:

- Word type and number of words per line.
- Number of sets and ways (therefore, it is possible to obtain a fully associative policy by setting the number of sets to 1 or a direct mapped policy by setting the number of ways to 1).
- Replacement policy (Least Recently Used or First-In First-Out).

3.1.3 Single-process Basic cache

The *Single-process Basic cache* is composed of a single pipelined process which performs all the cache functionalities.

This process can be pipelined with an II equal to 1 when:

- Memory accesses are Read-Only.
- A cache line can fit a single AXI transaction (i.e. line is not bigger than the maximum AXI interface width: 512 or 1024 bits typically, depending on the specific device).

Write accesses generate some dependencies on the AXI interface, while large cache lines require multiple AXI transactions: both of them cause an increase of the cache process II, reducing cache performance.

3.1.4 Multi-processes Basic cache

The *Multi-processes Basic cache* splits cache into two processes (Figure 3.2):

- *Core* process: manages communication with application and keeps cache data structures up to date.
- *Memory interface* process: deals with the AXI interface.

This architecture is aimed at solving the performance limitations of the *Single-process Basic cache*: it manages to pipeline the *core* process with an II equal to 1, even in case of write-only accesses or long lines, since the AXI interfacing resides in the separate *memory interface* process.

The latency of the response to a hitting request depends on the *core* process only, therefore with this solution the best performance is achieved in case of write-only caches too.

In the case of caches which are accessed both in read and in write mode, it has not been possible to achieve an II of 1, due to dependencies on the cache memory. Given that a read-write cache implies at least one read access and one write access,

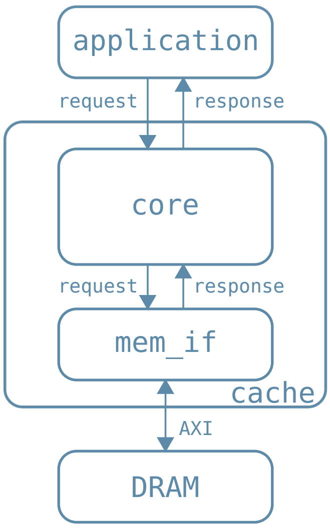


Figure 3.2: *Multi-processes Basic cache* architecture.

3.2 Implementation

The *Basic cache* is implemented in the form of a `C++14` [3] class, compatible with *Vitis™ HLS 2021.1*. All the configurable parameters are set through class template arguments.

The cache class is logically split into two parts:

- *Internals*: cache functionalities.
- *Interface*: APIs for managing requests and responses from application side.

Internals and *Interface* communicate with each other through a *Port* (Table 3.1), in a *Master/Slave* fashion:

- *Interface* sends to *Internals* a *request* (operation, address and write data).
- *Internals* sends to *Interface* a *response* (read data), after executing the requested operation.

Content	Description	Direction
Operation	Read/Write	*Internals* $\rightarrow$ *Interface*
Address	Index to be accessed	*Internals* $\rightarrow$ *Interface*
Write data	Data to be written to memory	*Internals* $\rightarrow$ *Interface*
Read data	Data read from memory	*Internals* $\leftarrow$ *Interface*

Table 3.1: Data exchanged through *Port*.

Process modeling HLS is intended for synthesizing sequential Software code, therefore it has been necessary to develop a novel technique for modeling multiprocess designs.

The proposed model follows the *Master/Slave* paradigm:

1. *Master* sends a request to *Slave*.
2. *Slave* executes the requested operation and optionally sends a response to *Master*.

Slave must be modeled as an infinite loop which waits for requests from *Master* before executing its functionality, while *Master* can be modeled as standard sequential code (or it can be in turn a *Slave* of another *Master*).

The parallelism between *Master* and *Slave* is modeled differently depending on the compilation target:

- *SW simulation*: each process is mapped to a `std::thread`.
- *HW synthesis*: each process is a dataflow function, in a dataflow region with the `disable_start_propagation` option disabled (which allow each function to run in parallel, without waiting for the completion of previous ones).

The distinction between simulation and synthesis code can be performed through the “`#ifdef __SYNTHESIS__`” preprocessor directive.

The communication between the two processes is performed through a *port*, which contains data flowing from *Master* to *Slave* (request) and from *Slave* to *Master* (response). Request and response are mapped to one or more FIFOs which are written from the transmitter and read from the receiver. `hls::stream` class by *Vitis™ HLS* can be used as FIFO implementation.

3.2.1 Internals

The *Internals* implementation differs between the *Single-process* and the *Multi-processes* implementations:

- *Single-process Basic cache*: single process which implements all the cache functionalities.
- *Multi-processes Basic cache*:
  - *Core* process: same as *Single-process Basic cache* process, but it does not directly access the AXI bus: it issues requests to the *memory interface* process through FIFOs.
  - *Memory interface* process: it accesses the AXI bus as requested by the *core* process.

Single-process Basic cache, with respect to the *Multi-processes* one, requires lower resource usage and better performance, when it is possible to schedule its process with an II equal to 1 (read-only accesses with line not larger than the maximum AXI interface bitwidth): therefore it is automatically instantiated whenever it is convenient.

Dataflow checking

Alternatively executing the *Multi-processes* or the *Single-process* code with traditional `if` statements would generate errors during the synthesis, particularly in the *Dataflow check* step (which checks if each `hls::stream` has a single reader and a single writer): the compiler builds both branches of the `if` statements, independently of the fact that one of them is never executed.

The problem has been solved through a wrapper class, which conditionally includes a `hls::stream` object, exploiting the template specialization mechanism.

Arrays partitioning

Cache memory (which stores the actual data) must be accessed one line per clock cycle: it is mapped to a BRAM array cyclically partitioned with a factor equal to the number of lines.

Helper data (e.g. `tag`, `valid`, `dirty`...) is stored to completely partitioned arrays, mapped to registers, in order to avoid dependencies as much as possible and get the best performance.

AXI optimizations

To exploit the *Vitis™ HLS* support to automatic port widening and burst accesses to AXI interface, every access to external DRAM accesses a whole cache line. The accessed addresses Least Significant Bits (LSBs) are explicitly set to 0 so that synthesizer can infer that they are aligned to the line size.

If the cache line is at most equal to the maximum AXI interface width, it is accessed in a single request, otherwise it is accessed in multiple burst requests.

Read After Write dependencies

In case of read-write caches, the *Core* process II increases to 3 due to RAW dependencies on the cache BRAM.

To mitigate this issue the *RAW cache* has been developed: it is a single-line cache which provides the functions:

- `get_line`: in case of hit, read the *RAW cache* line; in case of miss, read the cache line.
- `set_line`: write both the *RAW cache* line and the cache line.

Cache memory is always accessed through the *RAW cache* and the `set_line` function is called once per iteration at most: if a cache line has been written, it is impossible that it is read in the next iteration, since the RAW cache would hit and return its line. This allows to falsify the RAW dependency with distance 1 on the cache memory (by setting to `false` the RAW inter-iteration dependencies and to `true` the RAW inter-iteration dependencies with distance 1).

This solution allows to schedule the cache process with an II equal to 2. The *RAW cache* could be extended to a fully-associative cache complying with the FIFO replacement policy, allowing to falsify the RAW dependency with distance 2 and achieving an II of 1.

A read-write cache implies that it is accessed at least two times per iteration (once in read mode, once in write mode), therefore, due to the issues discussed in Subsection 3.2.2 it is not possible to fully exploit the cache pipelining. In this case the cache II does not have a relevant impact on effective performance: *RAW cache* could not provide real advantages and it has not been included in the final design, to keep it simpler.

3.2.2 Interface

Interface provides APIs for managing requests and responses between application and cache:

- `get`: send a read request and read the response.
- `set`: send a write request.

To improve user-friendliness, similarly to *Ma's cache*, the `operator[]` has been overloaded so that a cache object can be used as a traditional array (e.g. `val = cache[i]` calls `val = cache.get(i)` and `cache[i] = val` calls `cache.set(i, val)`).

Deadlock prevention

The HLS scheduler is not able to infer the dependency between the request writing (W) and the response reading (R) in the `get` function (i.e. it is not aware that first the request has to be written, then it is necessary to wait for the cache latency and finally the response has to be read).

For that reason the scheduler optimizes the logic by inserting both W and R into the same pipeline stage. This leads to a deadlock: R is blocked since it reads from an empty FIFO (it cannot contain the response yet) and it blocks the whole stage, including W, making R wait for the response to a request which cannot be sent.

The deadlock has been fixed by inserting a clock operation between W and R (calling `ap_wait`), which forces W and R to separate pipeline stages.

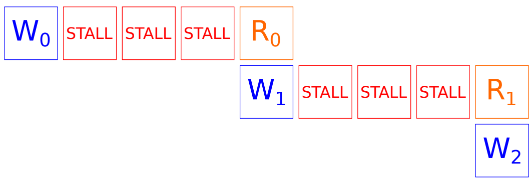
Cache pipeline exploiting

At the steady state, in case of hit, the cache can process one request per cycle, thanks to its optimal pipelining (i.e. II equal to 1).

HLS is not aware of the dependency and latency between request writing (W) and response read (R), so it schedules R just after W (Figure 3.3a): at runtime R_i, which should be executed in the cycle following W_i, stalls, since the cache response has a latency (and W_{i+1} stalls too, by consequence).

W_{i+1} is executed after waiting for the full latency of the cache (Figure 3.3b) and the final result is that cache never receives multiple requests in consecutive cycles, it never reaches the steady state and its throughput is the same as if it were not pipelined.

(a) Static schedule.



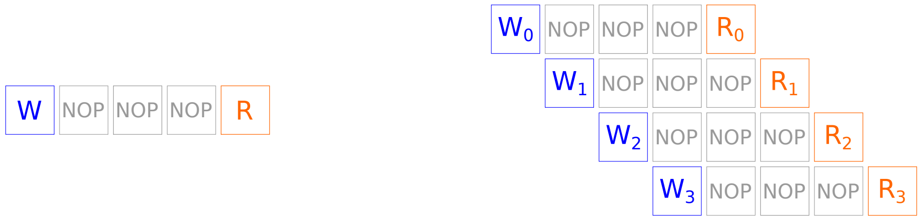
(b) Runtime.

Figure 3.3: Stalling schedule of request writing and response reading.

To mitigate this issue the `ap_wait` between request write and response read has been replaced with `ap_wait_n(LATENCY)`, where `LATENCY` is an integer value set through a template parameter. This forces the scheduler to insert `LATENCY` clock cycles between W and R (Figure 3.4a), so that at runtime stalls are avoided (in case of hit) and one request per cycle is sent to cache (Figure 3.4b).

`LATENCY` is not set to a constant because its optimal value highly depends on memory access pattern and cache configuration, and can be determined by means of design exploration.

This is a partial solution: the `ap_wait` forces all the subsequent operations to wait: when there are multiple calls to `get` per iteration (e.g. A and B), W_B has to wait



(a) Static schedule. (b) Runtime.

Figure 3.4: Optimal schedule of request writing and response reading.

`LATENCY` cycles after W_A before being scheduled (Figure 3.5a). This situation makes the application loop II to increase, since it must guarantee the order of accesses to FIFOs (i.e. $W_{A,i+1}$ cannot be executed before $W_{B,i}$).

To actually fix this problem (with the schedule shown in Figure 3.5b), a mechanism for informing the scheduler about dependencies and latency between specific operations is probably needed, but this is not available in *Vitis™ HLS 2021.1*.

(a) Achieved static schedule. (b) Optimal static schedule.

Figure 3.5: Static schedules in case of multiple accesses per iteration.

4 Multi-levels cache

The *Multi-levels cache* is aimed at improving performance by making the memory hierarchy deeper, adding a faster L1 cache memory on top of it. This alternative approach has been proposed to overcome the difficulties, to fully exploit the optimal pipeline of the *Basic cache*, due to the scheduler unawareness about the latency between request writing and response reading (as explained in Section 3.2).

4.1 Architecture

The *Multi-levels cache* introduces a L1 cache inlined in the application logic (Figure 4.1): the scheduler exactly knows the latency of each L1 cache operation and can build an application pipeline which stalls in case of L1 miss only.

In order not to fall into the same cluttering issues of *Ma's cache*, the L1 cache is kept as simple as possible:

- Mapping policy: direct-mapped.
- Consistency policy: write-through.

The write-through consistency policy discards any advantage for write accesses, but given that simplicity is a priority and read accesses are usually more frequent than writes, and they suffer the most from the scheduling issues which lead to the introduction of the L1 cache, this has been considered the best trade-off.

4.2 Implementation

The *Multi-level cache* has been implemented adding the L1 cache to the *Basic cache*. It is possible to configure the number of L1 cache lines through the `L1_CACHE_LINES` template parameter. When it is set to 0, the resulting architecture is equivalent to the *Basic cache*.

4.2.1 Internals

The only difference with respect to the *Basic cache* implementation is that the response to a read request does not send a single word, but a whole cache line (therefore the data FIFO flowing from cache to application has been widened accordingly).

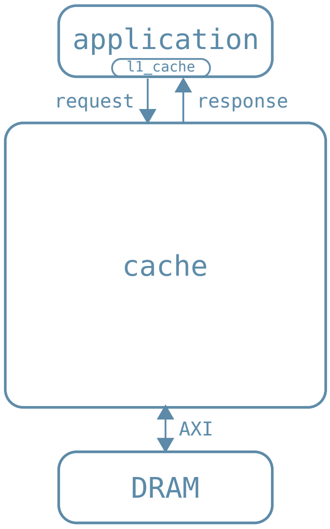


Figure 4.1: *Multi-levels cache* architecture.

4.2.2 Interface

The L1 cache is contained in the *Interface*: the newly introduced `get_line` function receives an address A in input and it returns the line to which A belongs. In particular, it first checks if A hits in the L1 cache: if so it reads the data from the L1 cache, otherwise it issues the request to the L2 cache.

It is still possible to use the same *Basic cache* APIs, which have been updated to support the L1 cache:

- `get`: it calls the `get_line` function and then returns the requested word.
- `set`: it sets L1 cache line to dirty, if it hits, and it forwards the request to the L2 cache.

5 Multi-ports cache

The computational core of many algorithms consists in a loop, which HLS can optimize with two techniques: *Pipelining* and *Unrolling*.

The *Basic* and *Multi-levels* caches are suitable for *Pipelining* since they complete one access per clock cycle, at the steady state, in case of hit, however they are not suitable for *Unrolling*, since they do not support concurrent accesses.

The *Multi-ports cache* has been specifically designed for adding support to multiple **concurrent accesses** to the same cache memory, allowing to efficiently **unroll** application loops.

5.1 Architecture

The *Multi-ports cache* is characterized by multiple ports accessed in parallel (Figure 5.1).

Each port has dedicated logic for communicating with the shared L2 cache and an independent L1 cache.

Multiple independent ports allow **removing dependencies** between different accesses to the cache. This brings the advantage of achieving better performance, making it possible to schedule multiple requests at the same time, without increasing the application loop II, but it also brings the disadvantage of not guaranteeing the expected ordering between different accesses. To guarantee the correct functionality the *Multi-ports* architecture is compatible with **read-only** accesses.

5.2 Implementation

The *Multi-ports cache* has been implemented extending the *Multi-levels cache*.

It is possible to configure the number of ports through the `PORTS` template parameter. When it is set to 1, the resulting architecture is equivalent to the *Multi-levels cache*.

5.2.1 Internals

To avoid dependencies issues, whenever `PORTS` is greater than 1, the *Multi-process Internals* architecture is generated.

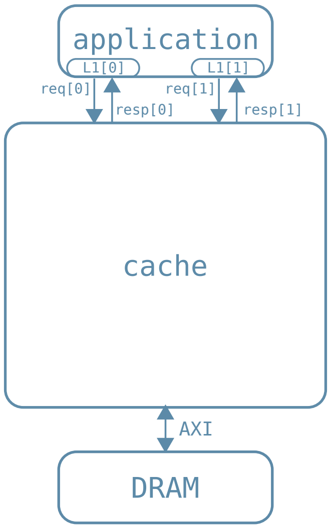


Figure 5.1: *Multi-ports cache* architecture.

The *Core* process has been modified to serve requests coming from all the ports by inserting an unrolled loop which iterates over all the ports. HLS guarantees all the dependencies on cache data structures, and the resulting II of the *Core* process is equal to `PORTS`.

5.2.2 Interface

FIFOs between *Core* and application and L1 cache have been replaced with arrays of FIFOs and L1 caches, completely partitioned, so that they are independent.

Each call to `get_line` (which is in turn called by `get`) is automatically associated with a specific port by means of a member variable holding the port index and is updated after each access.

FIFOs accesses scheduling

Ideally the request write (W) and the response read (R) should be scheduled in parallel in the same cycle (Figure 5.2b). Due to the scheduler limitations (described in Subsection 3.2) it is not possible to achieve such a schedule, since there is a forced clock cycle between W and R, which delays all the subsequent operations.

The resulting schedule (Figure 5.2a) is almost equivalent to the one achieved with the *Basic cache* in case of multiple accesses per iteration (Figure 3.5a), with the difference that request and response FIFOs are distinct, since they belong to separate ports, therefore the scheduler does not have to ensure dependencies between subsequent reads and writes and application loop II does not increase.

At the steady state, in case of hit, one W and R are executed per cycle, allowing to fully exploit the L2 cache pipeline.

(a) Achieved static schedule.

(b) Parallel static schedule.

Figure 5.2: Static schedules in case of 2-ports cache.

5.3 Limitations

In some particular situations (e.g. when cache is explicitly accessed multiple times per iteration) the simulation of the generated circuit enters a deadlock. The source of this problem can be probably found in the port indexing and to be fixed may require more control over the operations scheduling, which is not provided by *Vitis™ HLS 2021.1*.

6 Results

The proposed cache architecture has been embedded in multiple *Vitis HLS* kernels implementing different algorithms, to evaluate both the performance gain and the resource usage of different cache configurations.

Each algorithm has been selected for its memory intensiveness and for its specific memory access patterns.

6.1 Simulation environment

Kernels have been synthesized by the C Synthesis in *Vitis™ HLS 2021.1*, targeting the `xcvu9p-flgb2104-2-e` part, running at a clock frequency of $250MHz$.

Vitis™ 2021.1 provides two main kind of simulation:

- Hardware Emulation: accurate, but slow.
- C/RTL Co-Simulation: fast, but not very accurate (especially for what concerns the AXI interface model).

HW Emulation has been used for determining the delay of the AXI interface (which is around 4 clock cycles). The AXI latency has been accordingly set to 3, so that the synthesizer can better optimize the circuit and Co-Simulation results match HW emulation as much as possible.

Synthesizer	C Synthesis in *Vitis™ HLS 2021.1*
Simulator	C/RTL Co-Simulation in *Vitis™ HLS 2021.1*
Flow target	`vitis`
Part codename	`xcvu9p-flgb2104-2-e`
Clock period	$4ns$
AXI latency	3

Table 6.1: Simulation environment configuration.

6.1.1 Reference memory models

The results have been compared with the output of synthesis and simulation of same algorithms implemented with different data access mechanisms: *global memory* (performance lower bound), *local memory* (performance higher bound) and *Ma's cache*.

Global memory

The algorithms access data directly from external DRAM through AXI interface, without any optimization: this is the straightforward but slowest solution, therefore it determines the performance lower bound.

Local memory

All the data required by algorithms is stored to local BRAMs: it determines the performance higher bound, but it is unfeasible in general, due to the limited amount of BRAMs.

With this solution the kernel:

1. Moves all the input data from DRAM to BRAMs.
2. Performs all the computations accessing data to and from BRAMs.
3. Moves all the output data from BRAMs to DRAM.

The execution time of DRAM accesses is not of interest, therefore it has been subtracted from reported results.

Ma's cache

Ma's cache was designed for *Vivado™ HLS 2016.2*: with some minor changes it is possible to synthesize it with *Vitis HLS 2021.1*, but it would need some more optimizations to achieve the original performance in the new environment.

The most fair comparison should have been done by using results generated in the exact same environment, but since this was not possible, all the results about Ma's cache have been collected from Ma's paper "Acceleration by Inline Cache for Memory-Intensive Algorithms on FPGA via High-Level Synthesis" and PhD thesis "Low power and high performance heterogeneous computing on FPGAs".

From the comparison of the results about global and local memory reported by Ma with the ones obtained in the current environment it is clear that the resource usage figures are roughly equivalent, while the execution times differ up to one order of magnitude (most probably due to different AXI latency values), therefore they are not comparable, but they have been reported for completeness.

6.1.2 Configurations

The reported results match the problems sizes and cache configurations proposed by Ma, so that it is possible to make direct comparisons.

The additional degrees of freedom provided by the proposed cache (`get` latency, multiple cache levels, multiple ports) have been exploited to further explore the design space.

6.2 Matrix multiplication

The standard row-by-column *Matrix multiplication* algorithm (Algorithm 1) includes two memory access patterns: by rows (A and C) and by columns (B).

Each row of A matrix is accessed P times and then it is not accessed anymore: the most convenient A cache is composed of a single line which fits a matrix row, which is filled each time a new row is accessed and it hits until the next row is accessed.

Each column of B matrix is accessed P times: the B cache, to get an hit ratio greater than 0 needs to contain at least M lines and comply with the fully-associative mapping policy. The results reported by Ma used a direct-mapped cache with M lines each one containing P elements (so that it is as big as the B matrix).

C elements are accessed sequentially and only once: any single-line cache with n words per line would have an hit ratio of $\frac{n-1}{n}$.

The implementation used during the tests applies both pipelining and unrolling (with factor equal to the number of ports) to the innermost loop.

Algorithm 1 *Matrix multiplication* algorithm.

Require: $A \in \mathbb{R}^{N \times M}, B \in \mathbb{R}^{M \times P}, C \in \mathbb{R}^{N \times P}$
Ensure: $C = A \times B$
  procedure MULTIPLY(A, B, C)
    for $i = 0, \ldots, N-1$ **do**
      for $j = 0, \ldots, P-1$ **do**
        $tmp \leftarrow 0$
        for $k = 0, \ldots, M-1$ **do**
          $tmp \leftarrow tmp + A[i][k] \cdot B[k][j]$
        end for
        $C[i][j] \leftarrow tmp$
      end for
    end for
  end procedure

6.2.1 16x16 matrices

In the case of *Matrix multiplication 16x16*, matrices A, B and C are sized 16×16 ($N = 16, M = 16, P = 16$).

This problem has been explored first with L2 caches only, and then with the multi-level caches.

Single-level cache configuration

The cache sizes have been fixed with the values shown in Table 6.2. The `get` latency and the number of ports have been determined through design space exploration.

Matrix	Sets	Ways	Words per line	L1 lines
A	1	1	16	0
B	16	1	16	0
C	1	1	16	0

Table 6.2: Single-level cache configuration for *Matrix multiplication 16x16*.

Design space exploration Figure 6.1 shows the execution time with respect to the `get` latency, for different numbers of ports.

It is worth noting that the `get` latency has a big impact on effective performance, especially in the single-port case (one order of magnitude). This makes clear that the cache process itself can run at high speed and the bottleneck is the scheduling of the FIFOs accesses.

Increasing the number of ports can provide significant advantages when the `get` latency is not optimal, because multi-port allow to schedule some cache requests in consecutive clock cycles.

The best performance is achieved by the single-port, since in this case the caches *core* process has an II of 1: with a `get` latency of 1 it is not possible to take full advantage of the `core` pipelining (as explained in Subsection 3.2.1), therefore the design keeps stalling even at the steady state (Figure 6.2a: a new request is written every multiple cycles) but the optimal `get` latency allows to fully exploit the pipelining and at every cycle one request is written and a new response is read (Figure 6.2b).

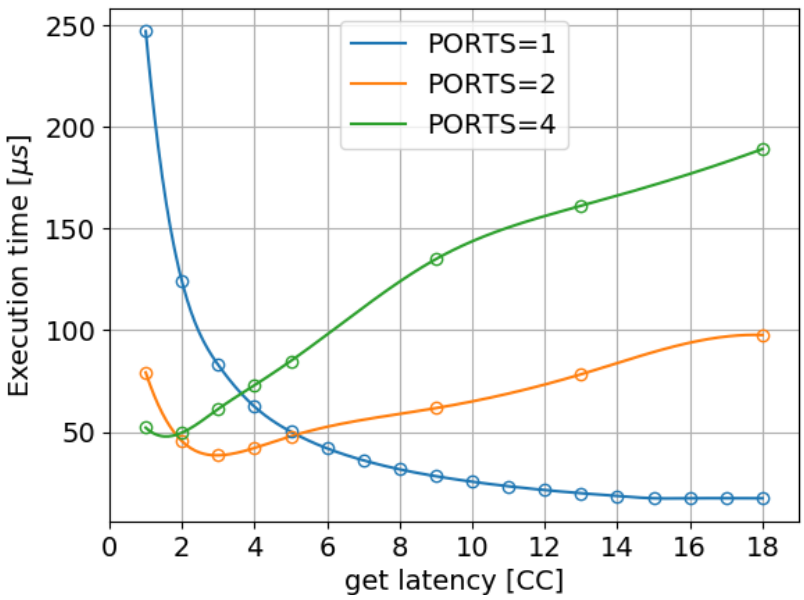
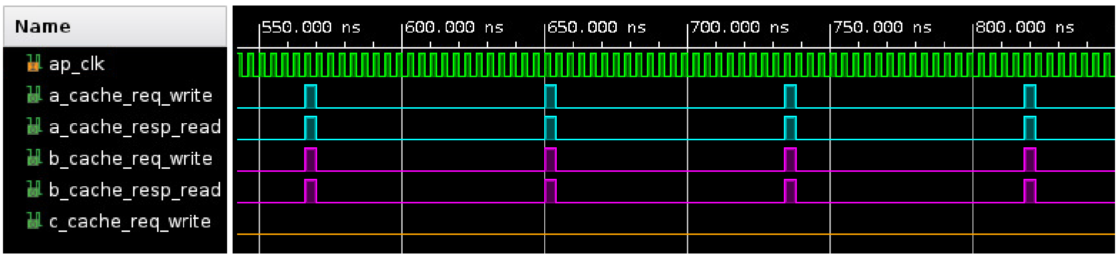
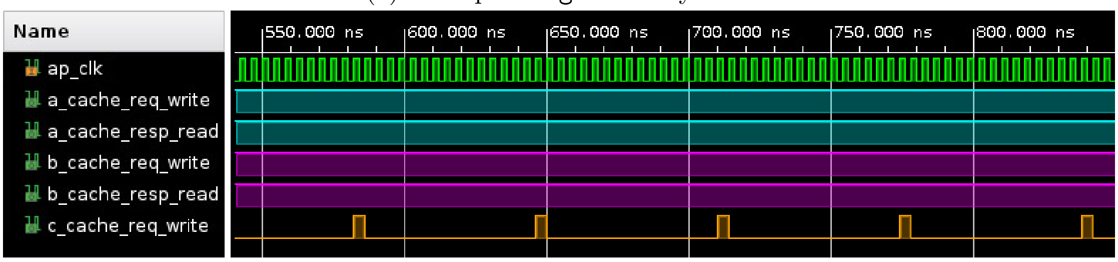


Figure 6.1: Design space of *Matrix multiplication 16x16* (single-level).



(a) Sub-optimal `get` latency of 1.



(b) Optimal `get` latency of 15.

Figure 6.2: Request and response waveforms for *Matrix multiplication 16x16* single-level and single-port.

	1-port	2-ports	4-ports
Execution time [μs]	17	39	47
BRAM	90	165	90
DSP	3	6	12
LUT	57653	87437	118434
FF	26597	37686	39352

Table 6.3: Performance and resource usage of *Matrix multiplication 16x16* (single-level).

Comparison with other memory models Table 6.4 reports most relevant figures about performance and resource usage of *Matrix multiplication 16x16*.

The proposed cache requires significantly more resources than Ma's cache (especially in terms of LUTs), but allows to get performance which are equivalent to the performance higher bound.

	Global memory	Local memory	Ma's cache [1]	Proposed cache
Execution time [μs]	30	17	31	17
BRAM	34	93	31	90
DSP	3	3	3	3
LUT	4421	7120	5699	57653
FF	4736	7841	17794	26597

Table 6.4: Performance and resource usage of *Matrix multiplication 16x16*.

Multi-levels cache configuration

The cache sizes have been fixed with the values shown in Table 6.5. The `get` latency and the number of ports have been determined through design space exploration.

Matrix	Sets	Ways	Words per line	L1 lines
A	1	1	16	1
B	1	1	16	16
C	1	1	16	0

Table 6.5: Multi-levels cache configuration for *Matrix multiplication 16x16*.

Design space exploration From Figure 6.3 it is clear that the `get` latency is not relevant is this case, since all the cache hits are on the L1 cache.

[1] Data collected from a different environment, not fully comparable.

Increasing the number of ports allows to significantly improve performance, since the multiple L1 caches can run effectively in parallel, but the higher is the number of ports, the lower is the hit ratio of each L1 cache: 4 ports is the optimal configuration for what concerns performance.

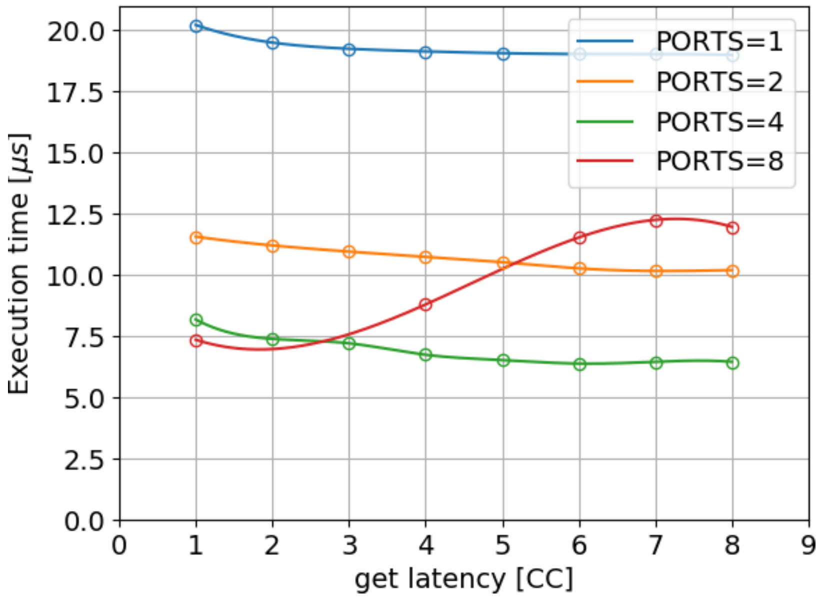


Figure 6.3: Design space of *Matrix multiplication 16x16* (multi-levels).

	1-port	**2-ports**	**4-ports**	**8-ports**
Execution time $[ns]$	19022	10274	6458	
BRAM	129	165	237	
DSP	3	6	12	
LUT	58138	81779	118794	
FF	41961	101315	238374	

Table 6.6: Performance and resource usage of *Matrix multiplication 16x16* (multi-levels).

6.2.2 32x32 matrices

Single-level configuration

Design space exploration

Matrix	Rows	Columns	Sets	Ways	Words per line	get latency
A	32	32	1	1	32	7
B	32	32	32	1	32	7
C	32	32	1	1	32	-

Table 6.7: *Matrix multiplication 32x32*, cache configuration A.

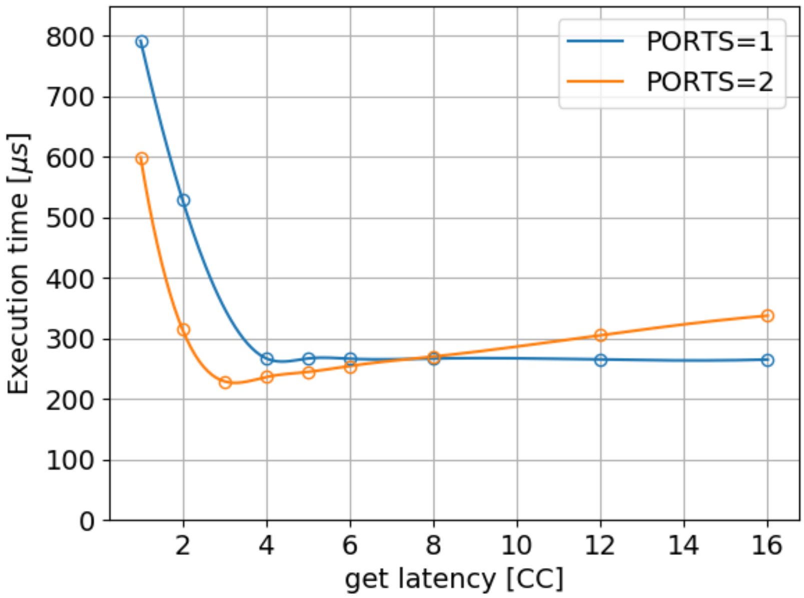


Figure 6.4: Design space of *Matrix multiplication 32x32* (single-level).

	Global memory	Local memory	Proposed cache	Ma's cache
Execution time [ms]				0.031
BRAM				31
DSP				3
LUT				5699
FF				17794

Table 6.8: Performance and resource usage of *Matrix multiplication* with setup B.

6.3 Bitonic sorting

Bitonic sorting (shown in Algorithm 2) is a sorting algorithm which provides an high degree of parallelism, therefore it is suitable for Hardware implementations.

Algorithm 2 *Bitonic sorting* algorithm.

Require: $a \in \mathbb{R}^N, N = 2^n$; dir: sorting direction
Ensure: $a[i] \geq a[j], \forall i \geq j \wedge dir = true \vee a[i] \leq a[j], \forall i \geq j \wedge dir = false$

```
procedure SORT(a, dir)
    for b = 1, ..., n do
        for s = i − 1, ..., 0 do
            for i = 0, ..., N/2 − 1 do
                dir_0 ← (i/2^(b−1))&1
                dir_0 ← dir_0|dir
                step ← 2^s
                pos ← 2i − (i&(s − 1))
                if a_pos > a[pos + step] ≠ dir_0 then
                    tmp ← a[pos]
                    a[pos] ← a[pos + step]
                    a[pos + step] ← tmp
                end if
            end for
        end for
    end for
end procedure
```

6.3.1 Comparisons

Setup A

n	Sets	Ways	Words per line	`get` latency
10	1	2	16	2

Table 6.9: Performance and resource usage of *Bitonic sorting* with setup A.

6.4 Lucas-Kanade

	Global memory	**Local memory**	**Proposed cache**	**Ma's cache**
Execution time $[ms]$	3.385	0.347	2.827	1.388
BRAM	4	60	58	31
DSP	0	0	0	0
FF	1150	3445	15397	31101
LUT	1710	3508	44590	22142

Table 6.10: Performance and resource usage of *Bitonic sorting* with setup A.

Algorithm 3 *Lucas-Kanade* algorithm.

Require: $A \in \mathbb{R}^{N \times M}, B \in \mathbb{R}^{M \times P}, C \in \mathbb{R}^{N \times P}$
Ensure: $C = A \times B$
  procedure MULTIPLY(A, B, C)
    for $i = 0, \ldots, N$ **do**
      for $j = 0, \ldots, P$ **do**
        $C[i][k] \leftarrow 0$
        for $k = 0, \ldots, M$ **do**
          $C[i][j] \leftarrow C[i][j] + A[i][k] \cdot B[k][j]$
        end for
      end for
    end for
  end procedure

7 Conclusions

A Cache integration

Bibliography

[1] Xilinx Inc. *Vitis High-Level Synthesis User Guide*. Mar. 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf.

[2] Xilinx Inc. *Vitis High-Level Synthesis User Guide*. Aug. 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug1399-vitis-hls.pdf.

[3] ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. Fourth. Dec. 2014.

[4] Liang Ma. "Low power and high performance heterogeneous computing on FPGAs". PhD thesis. Politecnico di Torino, Feb. 2019. DOI: http://dx.doi.org/10.6092%2Fpolito%2Fporto%2F2727228.

[5] Liang Ma, Luciano Lavagno, Mihai Lazarescu, and Arslan Arif. "Acceleration by Inline Cache for Memory-Intensive Algorithms on FPGA via High-Level Synthesis". In: *IEEE Access* PP (Sept. 2017), pp. 1–1. DOI: 10.1109/ACCESS.2017.2750923.