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 improving designer productivity thanks to convenient Software-like tools.

The problem of high deployment costs can be tackled with FPGAs, which allow to implement 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 kind of resources: registers, Block RAMs (BRAMs) and external Dynamic RAMs (DRAMs). Current HLS tools allow to exploit this memory hierarchy manually, in a scratchpad-like fashion: the objective of this thesis work is to automate the memory management by providing a 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) multi-process 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 unroll 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 **iv**

List of Tables **v**

List of Acronyms **vi**

1 Background **1**

1.1 Cache 1
1.2 Field-Programmable Gate Array 3
1.3 High-Level Synthesis 4
1.4 Previous work 5

2 Basic architecture **6**

2.1 Implementation 8

3 Optimized architectures **11**

3.1 L1 cache 11
3.2 Multiple ports 11

4 Results **13**

4.1 Matrix multiplication 13
4.2 Bitonic sorting 13
4.3 Lucas-Kanade 13

5 Conclusion **14**

Bibliography **15**

List of Figures

1.1 Cache logic structure. 2
1.2 Set associative policy address bits meaning. 3

2.1 *Single-process Basic architecture* overview. 7
2.2 *Multi-processes Basic architecture* overview. 8

3.1 Cache architecture optimized with the insertion of a L1 cache. 12

List of Tables

List of Acronyms

API	Application Programming Interface
AXI	Advanced eXtensible Interface
BRAM	Block RAM
DRAM	Dynamic RAM
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
L1	Level 1
L2	Level 2
LRU	Least Recently Used
RAM	Random Access Memory
RAW	Read After Write
RTL	Register-Transfer Level
SFINAE	Substitution Failure Is Not An Error
SW	Software

Chapter 1

Background

1.1 Cache

Memory devices are usually the performance bottleneck in the execution of memory-bound 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 setup a memory hierarchy in which fast but small memories are paired with large but slow memories, which allows to get 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.

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

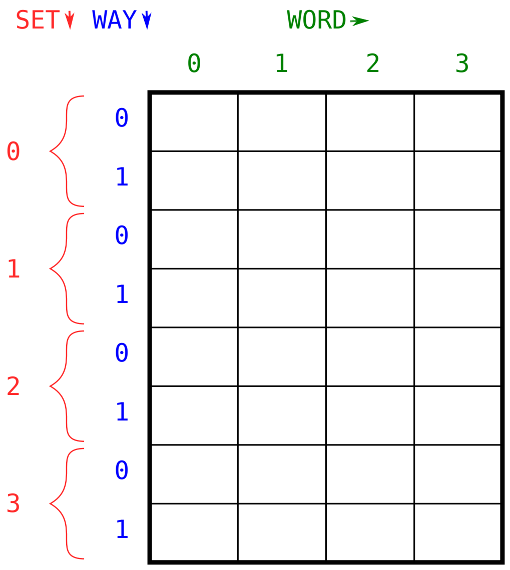

Figure 1.1: Cache logic structure.

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.

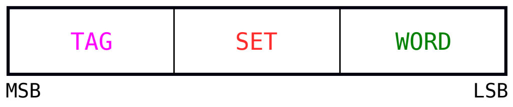

Figure 1.2: Set associative policy address bits meaning.

- *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.

1.2 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

An FPGA memory system is typically made up of:

- registers: the fastest but most expensive memories, therefore they are only a few.

- BRAMs: on chip 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 an high-level Software programming language (such as C and C++) into an HDL description.

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

Vitis™HLS [1] is one of the available HLS commercial tools.

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

Typical optimization techniques used by HLS for improving performance include:

- **Pipelining**: loops and functions logic can be pipelined so that successive iterations/calls can start while previous ones are still running. The introduced parallelism allows to increase the throughput at a limited additional area cost (only pipeline registers and a FSM are required).
- **Dataflow**: different functions composing the design are called in a pipelined fashion (similarly to pipelining, but at task level, instead of instruction level).

- **Loop unrolling**: the loop logic is instantiated multiple times, to execute multiple loop iterations in parallel, reducing latency and improving throughput.
- **Memory optimizations**:
  - On-chip memory:
    - *Array partitioning*: given a factor f, an array is split into f portions, each one mapped to a dedicated memory element, allowing more accesses to the same array at the same time, at the cost of higher memory elements usage. It can be block: each portion contains contiguous elements; cyclic: each portion contains interleaved elements.
  - Off-chip memory:
    - *Bursting*: multiple memory accesses are aggregated to reduce overall latency and improving throughput.
    - *Interface widening*: multiple data elements are packed into a single bigger word, to perform multiple accesses at the same time.

1.4 Previous work

Liang Ma et al. proposed an inlined cache implementation [3] in the form of `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).

Each cache is associated with a specific array stored to off-chip DRAM and stores its data to on-chip BRAMs and registers. Since the cache is dedicated to a specific array and the accesses to a single array are usually regular it is in general easy to tune the different parameters to get high hit ratios.

The `operator[]` has been overloaded such that the cache can be used in the same way of arrays, allowing to reduce the coding efforts when integrating the cache in an existing algorithm.

During the synthesis the cache is inlined in the user algorithm: this may clutter the logic and limit the maximum achievable performance.

Chapter 2

Basic architecture

The *Basic architecture* consists in a cache whose logic is separate from application logic since it runs in a distinct process (Figure 2.1). It starts from same assumptions of Ma's architecture (introduced in Section 1.4): accelerate accesses to off-chip DRAM arrays associating them with dedicated caches which store their data to on-chip BRAM and registers.

The acceleration is achieved by:

- optimizing AXI accesses: the cache aggregates DRAM accesses in lines, allowing to exploit automatic port widening and burst accesses.
- reducing AXI accesses: the AXI interface is accessed in case of cache miss only.

The *Basic architecture* is aimed at solving the main limitation of Ma's one: inlining the cache into application makes the logic more complex and the synthesis may not be able to generate a circuit with optimal performance.

Since the cache process is isolated from the application logic, it should always perform in the same manner, independently from the algorithm in which it is used, while the algorithm may get better performance, since it would only contain FIFO accesses instead of the whole cache logic.

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

1. A writes the request to the request FIFO.
2. C reads the request FIFO and checks if the requested address causes a miss.
3. (in case of miss) C issues a request to the AXI interface to prepare its own memory (mapped to BRAM) for fulfilling the requested access.
4. C performs the access to BRAM and (in case of read request) writes requested data to the response FIFO.
5. (in case of read request) A reads requested data from the response FIFO.

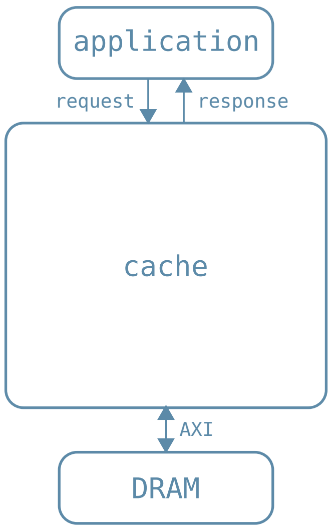

Figure 2.1: *Single-process Basic architecture* overview.

Characteristics The *Basic architecture* 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).

Single-process and Multi-processes Basic architectures The *Single-process Basic architecture* is composed of a single pipelined process which performs all the cache functionalities.

This process can be pipelined with an Initiation Interval (II) of 1 when memory accesses are Read-Only and a cache 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 since they do not fit a single one: both of these increase the process II, worsening cache performance.

The *Multi-processes Basic architecture* splits cache into two processes (Figure 2.2):

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

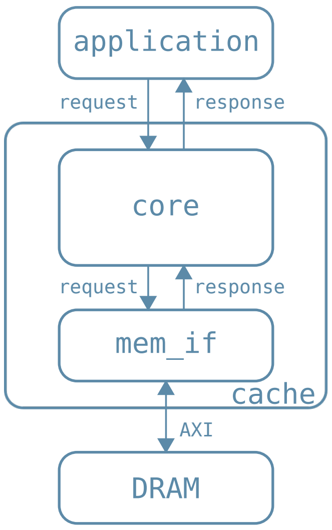

Figure 2.2: *Multi-processes Basic architecture* overview.

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

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

2.1 Implementation

The *Basic architecture* is implemented in the form of a `C++14` [2] 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.

2.1.1 Internals

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

- *Single-process Basic architecture*: single process which implements all the cache functionalities.

- *Multi-processes Basic architecture*:
  - *core* process: same as *Single-process Basic architecture* process, but it does not directly accesses 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.

Array 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 best performance.

Process modeling

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

A process is modeled as an infinite loop and their parallelism is modeled differently depending on the compilation target:

- *SW simulation*: each process is mapped to an `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.

Different processes can communicate by means of FIFOs (`hls::stream` provided by Vitis™HLS), which allow unidirectional point-to-point communication between two processes. It is possible to insert multiple FIFOs between each process, in both directions, therefore allowing to setup duplex communication.

Since `hls::stream` provides blocking operations, these FIFOs can be also used for synchronization purposes.

Read After Write dependencies

The cache process II is limited by the RAW dependencies on the cache lines, therefore 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 of distance of 1 on the cache memory, making it possible to schedule the cache process with an II of 1.

Conditional code compilation

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

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 that each `hls::stream` has a single reader and a single writer): the compiler builds both branches of the `if` statements, independently from the fact that one of them is never executed.

The problem can be solved through the Substitution Failure Is Not An Error (SFINAE) pattern, which allows to make the compiler ignore parts of code when the template substitution is invalid, instead of throwing an error. This technique can be easily implemented by using the `std::enable_if` structure which conditionally returns an invalid type, triggering the SFINAE pattern.

2.1.2 Interface

Provide APIs for managing requests and responses from application to cache:

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

To improve user friendliness, similarly to Ma's implementation, 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)`).

Chapter 3

Optimized architectures

3.1 L1 cache

Each FIFO access costs one clock cycle, which has to be paid for each memory access. To improve performance each read request to the cache does not return a single data element, but a whole cache line. This allows to insert a L1 cache above the underlying cache, directly inlined in the user logic (Figure 3.1), similarly to Ma's architecture.

An equivalent approach could have been applied to write requests, but given that read accesses are usually more frequent than write accesses, therefore more sensitive to optimizations and that it is crucial not to clutter the user logic, the L1 cache is kept as simple as possible: it complies with the write-through consistency policy, which is simpler but does not provide any advantage to write accesses.

To keep up with simplicity the L1 cache is direct-mapped.

3.1.1 Implementation

The L1 cache has been added to the *Basic architecture* through a few modifications:

- the template argument determining the number of L1 cache lines has been added.
- the data FIFO from cache to algorithm has been extended to fit a whole cache line.
- the `get` function, instead of directly issuing the read request to the L2 cache, it checks if the L1 cache hits: if so it reads the data from the L1 cache, otherwise it issues the request to the L2 cache.
- the `set` function sets the L1 cache line to dirty.

3.2 Multiple ports

The vast majority of algorithms accesses memory inside loops, which in HLS can be optimized in two main ways: pipelining, which perfectly fits the single-port cache archi-

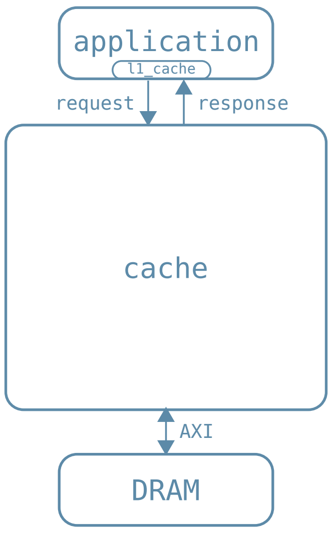

Figure 3.1: Cache architecture optimized with the insertion of a L1 cache.

tecture, and unrolling, with which the II would increase, since each unrolled iteration should access the same FIFO at the same time.

To solve this problem a multi-port architecture is proposed: each port has dedicated FIFOs and the cache process serves each request in order. Each port has also a dedicated L1 cache, which can be used without any coherency problem, since they follow the write-through consistency policy.

3.2.1 Implementation

Port binding

Chapter 4

Results

4.1 Matrix multiplication

4.2 Bitonic sorting

4.3 Lucas-Kanade

Chapter 5

Conclusion

Bibliography

[1] 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.

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

[3] 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.