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, since it helps improving designer productivity thanks to high-level programming languages, software-like functional debugging, design space exploration through compiler directives...

The problem of high deployment costs can be tackled with Field-Programmable Gate Arrays, which allow to implement special-purpose hardware modules on general-purpose underlying physical architectures.

The architectures generated by HLS comply with the Von Neumann model, therefore memory-bound algorithms suffer from the memory bottleneck problem, even when synthesized to hardware.

FPGAs memory system is composed of three main kind of resources: registers, BlockRAMs and external 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 High-Level Synthesis.

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

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 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 L2 cache.

The multi-port solution provides a single L2 cache accessible from multiple FIFO 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 trade-offs, multiple test programs have been synthesized and simulated with different data accesses configurations: to AXI interface (performance lower bound), to BRAM (performance higher bound) and to caches (with multiple configurations). Each test program executes algorithms with different memory access patterns: by rows (e.g. matrix addition), by columns (e.g. matrix multiplication), by windows (e.g. 2D convolution), triangular (e.g. backward substitution).

Contents

Chapter 1

Background

1.1 Cache memory

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.

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

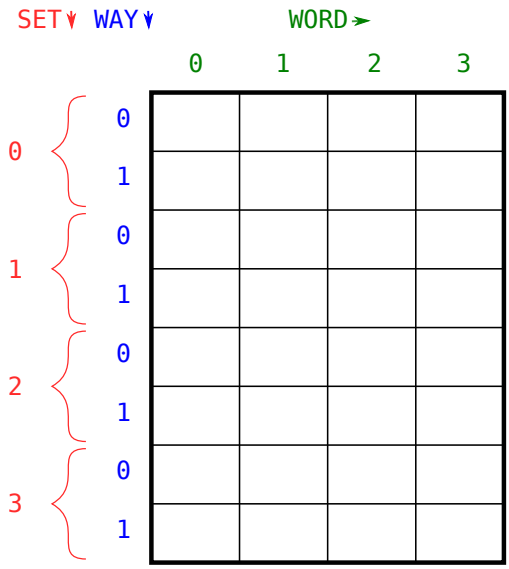

Figure 1.1: Cache logic structure.

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.

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 High-Level Synthesis

The 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 Hardware Description Language (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 expressiveness.

Workflow

The typical HLS workflow consists in:

1. software 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. software 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. hardware synthesis: the synthesizer generates an RTL description of the top level entity. It is possible to generate different architectures by setting up some parameters through dedicated directives.
4. hardware verification: the RTL description is simulated, to make sure that SW and HW outputs match.

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 an 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
  - 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.

1.3 FPGAs

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

Memory system

An FPGA memory system is typically made up of:

- registers: the fastest but most expensive memories, therefore there are only a few.
- block RAMs: on chip RAMs accessible through simple and fast interface.
- external DRAM: off chip RAMs through complex and slow interface (e.g. AXI).

1.4 Previous work

Chapter 2

Architecture

2.1 Basic architecture

The fundamental idea behind the proposed architecture is to keep application and cache logic into separate processes, in order to simplify synthesizer job and possibly obtain a better performing circuit, as shown in Figure 2.1.

When application needs to access memory:

1. application writes the request to the request FIFO.
2. cache reads the request FIFO and checks if it causes a miss
3. in case of miss, cache issues a request to the AXI interface to prepare its own memory (mapped to BRAM) for fulfilling the requested access.
4. cache performs the access to BRAM and writes the outcome to the response FIFO (in case of a read request).

Single-process solution (Read-Only)

This first proposed architecture is composed of a single pipelined process which performs all the cache functionalities.

In case of Read-Only memory accesses this process can be pipelined with an Initiation Interval of 1, while write accesses generate some dependencies on the AXI interface which cause the process II to increase, dropping cache performance.

Mapping policy	Set associative
Replacement policy	First-In First Out or Least Recently Used
Consistency policy	Write-back

Table 2.1: Basic architecture cache features

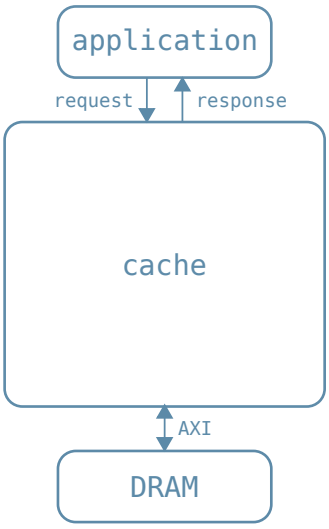

Figure 2.1: Single-process cache architecture.

Multi-processes solution (Read-Write)

In this architecture, the cache have been split into two processes (Figure 2.2):

- core: manage communication with application and keep cache data structures up to date.
- memory interface: deal with the AXI interface.

This solution allows to pipeline the *core* process with an II of one, even in case of write accesses, 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 desired performance is achieved in case of writable caches too.

2.2 Optimizations

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 Level 1 cache above the underlying cache, directly inlined in the user logic, similarly to the cache described in Section 1.4.

Since it is important not to clutter the user logic, the L1 cache is kept as simple as possible: it complies with the direct mapped mapping policy and the write-through consistency policy.

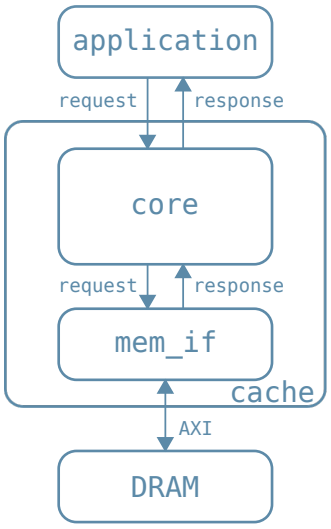

Figure 2.2: Multi-processes cache architecture.

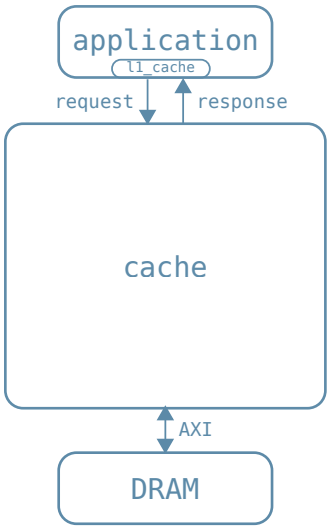

Figure 2.3: Cache architecture optimized with the insertion of a L1 cache.

The read accesses only can benefit from the L1 cache insertion, due to the write-through consistency policy: read accesses are usually more frequent than write accesses, therefore the optimization efforts has been focused on them.

Multiple ports

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

Chapter 3

Implementation

The proposed architectures have been implemented and tested using the *Vitis HLS 2021.1* design flow, which supports the synthesis of `C++14` code.

3.1 Technology

Multi-process modeling

HLS is usually intended for synthesizing sequential software code, therefore it has been necessary to develop a new mechanism for modeling multi-processes.

Each process consists of an infinite loop which communicates with other processes by means of FIFOs (`hls::stream` provided by Vitis HLS).

In order to make each process run in parallel, it was necessary to distinguish (thanks to the "`#ifdef __SYNTHESIS__`" preprocessor check) between the software simulation, which inserts each process in a `std::thread`, and the hardware synthesis, which calls each process in a dataflow region with the `disable_start_propagation` option enabled (so that each function runs in parallel, without waiting for the completion of previous ones).

Conditional code compilation

RAW dependencies

Chapter 4

Results

Chapter 5

Conclusion