Introduction
Architecture
Implementation
Results
Future work
Summary

Cache for HLS

A multi-process architecture

Brignone Giovanni

June 7, 2021

Politecnico
di Torino

Introduction
Architecture
Implementation
Results
Future work
Summary

Motivation

Introduction
Architecture
Implementation
Results
Future work
Summary

Motivation

Motivation

- **Problem**:
  big arrays mapped to DRAM $\Rightarrow$ performance bottlenecks

- **Proposed solution**:
  cache module designed to:
    - exploit BRAMs
    - easily integrable into any *Vitis HLS* design
    - have an high hit ratio

Introduction
Architecture
Implementation
Results
Future work
Summary

Inlined architecture
Multi-process architecture

Introduction
Architecture
Implementation
Results
Future work
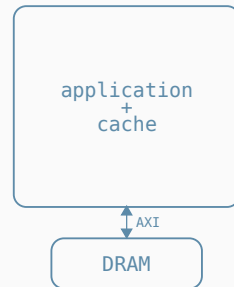Summary

Inlined architecture
Multi-process architecture

Inlined architecture (Ma Liang)

Array access $\rightarrow$ Inlined cache logic

- cache logic mixed with application logic

- one cache per array

- single-port only

Introduction
Architecture
Implementation
Results
Future work
Summary

Inlined architecture
Multi-process architecture

Multi-process architecture

Array access $\rightarrow$ Request to separate module

- decoupling between application and cache logic
  (communication through FIFOs)
- one cache per array
- may support multiple ports (work in progress)

Introduction
Architecture
Implementation
Results
Future work
Summary

Tools
Internal architecture
Problems and solutions
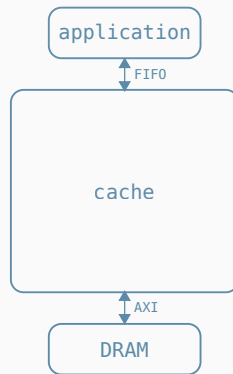
1 Introduction

2 Architecture

3 Implementation
  Tools
  Internal architecture
  Problems and solutions

4 Results

5 Future work

Introduction
Architecture
Implementation
Results
Future work
Summary

Tools
Internal architecture
Problems and solutions
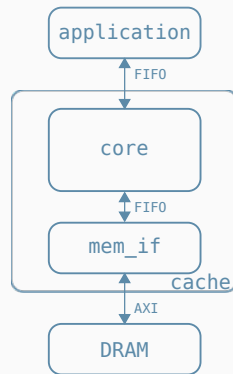
Tools

Vitis HLS (2020.2) workflow (C++):

- **Multiple processes modeling**:
  infinite loops parallelized by:
  - <u>SW simulation:</u> `std::thread`
  - <u>Synthesis:</u> `DATAFLOW` with *start propagation* disabled
- **Inter-process communication**: `hls::stream` FIFOs
- **Performance optimization**:
  - loop pipelining: new request each cycle
  - automatic port widening: access one line at a time in DRAM
- **Customization:** cache parameters set through `template`
- **Limitations**: cannot override "`[] operator`" for set due to automatic class
  disaggregation

Introduction
Architecture
Implementation
Results
Future work
Summary

Tools
Internal architecture
Problems and solutions

Internal architecture

- **Problem**: persuade HLS to schedule cache HIT response early in the pipeline
- **Proposed solution**: split the cache into two processes:
  1. `core`:
     - manage requests from `application`
     - keep cache data structures up-to-date
  2. `mem_if`:
     - manage requests from `core`
     - access DRAM

     $\Rightarrow$ synthesizer job is simplified:
     HIT response scheduled earlier

application
↕ FIFO
core
↕ FIFO
mem_if
cache
↕ AXI
DRAM

Introduction
Architecture
Implementation
Results
Future work
Summary
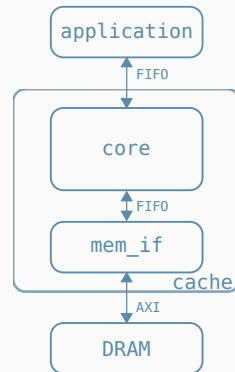
Tools
Internal architecture
Problems and solutions

mem_if implementation
Functionality

Manage DRAM accesses:

1. read request from core
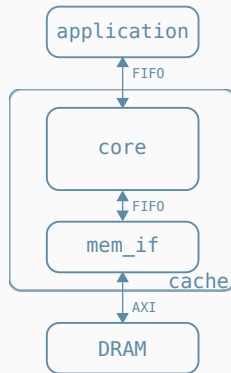2. access DRAM
3. write response to core (if read request)

Introduction
Architecture
Implementation
Results
Future work
Summary

Tools
Internal architecture
Problems and solutions

core implementation

Functionality

❶ read request from `application`

❷ check if it is an HIT or a MISS

❸ if MISS:
- issue a `write` request of the line to be replaced to `mem_if` (if write-back is necessary)
- issue a `read` request of the line to be loaded to `mem_if`
- read `mem_if` response and update BRAM

❹ access BRAM

❺ write response to `application` (if read)

```
application
    ↕ FIFO
  ┌─────────────┐
  │    core     │
  │      ↕ FIFO │
  │   mem_if    │
  │      cache  │
  └─────────────┘
    ↕ AXI
   DRAM
```

Introduction
Architecture
Implementation
Results
Future work
Summary

Tools
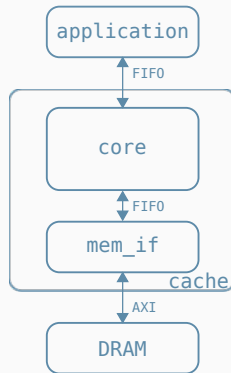Internal architecture
Problems and solutions
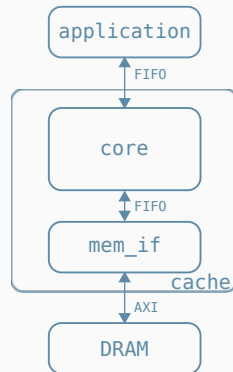
BRAM dependencies - Line loading

- **Problem**: reading BRAM line immediately after it has been loaded from DRAM: *Read-After-Write* dependency
- **Proposed solution**:
  1. store the mem_if response in a **buffer**
  2. perform the request on the buffer
  3. update BRAM

Introduction
Architecture
Implementation
Results
Future work
Summary
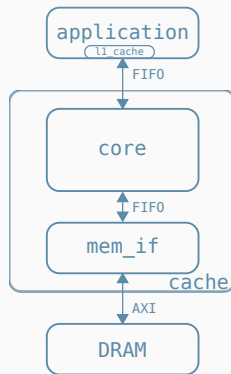
Tools
Internal architecture
Problems and solutions

BRAM dependencies - RAW request

- **Problem**: write request immediately followed by a
  read request to the same element: *Read-After-Write*
  dependency on BRAM
- **Proposed solution**: raw_cache inside cache:
  - write request: store the written line to a buffer
  - subsequent read request to the same line: **access the
    buffer instead of BRAM**

Introduction
Architecture
Implementation
Results
Future work
Summary

Tools
Internal architecture
Problems and solutions

Request optimization

- **Problem**:
  - each FIFO access costs 1 cycle
  - accesses to arrays are often sequential
- **Proposed solution**: `l1_cache` in the interface (`application` side):
  - `read` request: get the whole cache line, store it in a buffer and return the requested element
  - subsequent `read` request to the same line: access the buffer
  - ⇒ avoid passing through FIFOs

Introduction
Architecture
Implementation
Results
Future work
Summary

Performance
Matrix Multiplication

1 Introduction

2 Architecture

3 Implementation

4 Results
    Performance
    Matrix Multiplication

5 Future work

Introduction
Architecture
Implementation
Results
Future work
Summary

**Performance**
Matrix Multiplication

Performance

- **Loop pipelining**:
  - core: $II = 1$
  - mem_if: $II = \left\lceil \frac{line\_size}{AXI\_if\_size} \right\rceil$
- **L1 HIT**: latency of 1 cycle
- **HIT**: latency of 6 cycles
- **MISS**: latency of ~ 80 cycles

Introduction
Architecture
Implementation
Results
Future work
Summary

Performance
Matrix Multiplication

Matrix Multiplication

- A matrix: each row accessed n times:
  $hit\_ratio = \frac{(line\_size \cdot x) - 1}{line\_size \cdot x}, x = \begin{cases} 1, & line\_size < m \\ n, & otherwise \end{cases}$

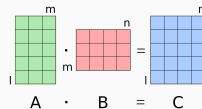- C matrix accessed by rows: $hit\_ratio = \frac{line\_size - 1}{line\_size}$

A · B = C

Introduction
Architecture
Implementation
Results
Future work
Summary

Performance
Matrix Multiplication
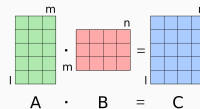
Matrix Multiplication

- B matrix accessed by columns:
  modified *direct mapped* address mapping:

  | tag | line | offset |

  ⬇

  | line | tag | offset |

$$\Rightarrow \begin{cases} hit\_ratio = \frac{(line\_size \cdot x) - 1}{line\_size \cdot x} \\ x = \begin{cases} 0, & n\_lines < m \\ 1, & n\_lines \geq m \wedge line\_size < n \\ l, & n\_lines \geq m \wedge line\_size \geq n \end{cases} \end{cases}$$
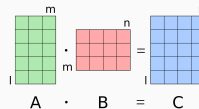
Introduction
Architecture
Implementation
Results
Future work
Summary

Performance
Matrix Multiplication

Matrix Multiplication
Practical example

- **Problem size**:
$$\begin{cases} l & = 16 \\ m & = 32 \\ n & = 64 \end{cases}$$

- **Caches size**:
$$\left.\begin{array}{ll} n\_lines_A = 2, & line\_size_A = 32 \\ n\_lines_B = m, & line\_size_B = 32 \\ n\_lines_C = 2, & line\_size_C = 32 \end{array}\right\} \Rightarrow \begin{cases} hit\_ratio_A \simeq 100\% \\ hit\_ratio_B = 96.9\% \\ hit\_ratio_C = 96.9\% \end{cases}$$
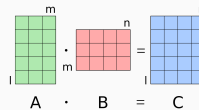

A · B = C

Introduction
Architecture
Implementation
Results
Future work
Summary

Performance
Matrix Multiplication

Matrix Multiplication
Practical example

- **Execution time without caches**:
  $2,428,745\mu s$
- **Execution time with caches**:
  $1,160,105\mu s$

  $\Rightarrow$ execution is 2.1x faster

Introduction
Architecture
Implementation
Results
Future work
Summary

Performance
Matrix Multiplication
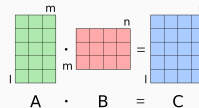
Matrix Multiplication

Reference example

- **Problem size**:
  $$\begin{cases} l & = 16 \\ m & = 32 \\ n & = 64 \end{cases}$$

- **Caches size**:
  $$\left.\begin{array}{ll} n\_lines_A = l, & line\_size_A = m \\ n\_lines_B = m, & line\_size_B = n \\ n\_lines_C = l, & line\_size_C = n \end{array}\right\} \Rightarrow \begin{cases} hit\_ratio_A \simeq 100\% \\ hit\_ratio_B = 99.9\% \\ hit\_ratio_C = 98.4\% \end{cases}$$
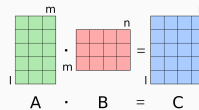

$$A \quad \cdot \quad B \quad = \quad C$$

Introduction
Architecture
Implementation
Results
Future work
Summary

Performance
Matrix Multiplication

Matrix Multiplication
Reference example

- **Execution time without caches**:
  $2,428,745\mu s$

- **Execution time with caches**:
  $369,045\mu s$

  $\Rightarrow$ execution is 6.6x faster

Introduction
Architecture
Implementation
Results
Future work
Summary

Multi-port cache

1 Introduction

2 Architecture

3 Implementation

4 Results

5 Future work
Multi-port cache

Introduction
Architecture
Implementation
Results
Future work
Summary

Multi-port cache

Multi-port cache

- **Problem**: cache manages one request per cycle $\Rightarrow$ **unrolling** a loop which uses the cache is not trivial
- **Possible workarounds**: access multiple data elements for each request and do all the unroll in `application`; two possible ways:
  - `get_line`: return a whole cache line
  - pack multiple data elements in a single cache element (e.g. `hls::vector`)
- **Proposed solution**: provide N **ports**, where N is the unroll factor

Introduction
Architecture
Implementation
Results
Future work
Summary

Multi-port cache

Multi-port cache
Implementation

- **Straightforward implementation**:
  - provide N FIFOs on the interface
  - unroll `core` loop by a factor N

- **Problem**: all the N requests may refer to the same line
  $\rightarrow$ each iteration must wait completion of previous one

Introduction
Architecture
Implementation
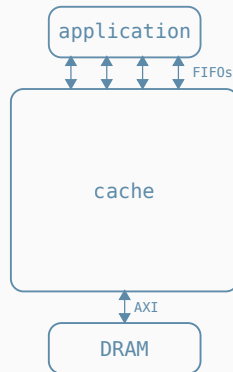Results
Future work
Summary
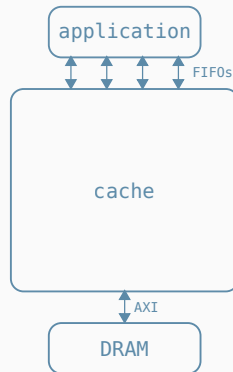
Multi-port cache

Multi-port cache
Implementation

- **Proposed solution**:
  1. read all the N requests
  2. check if they are all HIT
  3. if at least one MISS issue all the requests to mem_if
  4. access BRAM
  5. write all the responses

  - cache addressing mode converted to an associative one
  - possibly implement core at RTL to have full control on scheduling

Introduction
Architecture
Implementation
Results
Future work
Summary

Summary

- **Architecture**: *Divide et Impera*
- **Results**: good single-port pipeline performance
- **Future work**: ease application unroll