Doctoral Dissertation
Doctoral Program in Electrical Engineering (35th cycle)

Improving Quality of Results (QoR) for High-Level Synthesis (HLS) based FPGA designs

By

M. Usman Jamal

******

Supervisor(s):
Prof. Luciano Lavagno, Supervisor

Politecnico di Torino
2023

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

M. Usman Jamal
2023

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my loving parents

Acknowledgements

I extend my heartfelt and sincere, gratitude and appreciation to my advisor, Prof. Luciano Lavagno , for his exceptional guidance, unwavering support, and significant contributions throughout the completion of this thesis. His mentor-ship and expertise have been instrumental in achieving this milestone. I consider myself fortunate to have had the privilege of working under his guidance.

I would specially like to thank, Giovanni Brignone, with whom I had the pleasure to work with. I would also like to express my gratitude to my friends, Weixi, Filippo, Teodoro, and other colleagues for their support and camaraderie.

Lastly, I sincerely acknowledge the love and encouragement of my family during this challenging process.

Abstract

High-level synthesis (HLS) is an Electronic design automation (EDA methodology that work towards designing complex digital systems using high level programming languages like C, C++ or SystemC and automatically transforms them into a hardware description language (HDL) in a relatively short time. This not only increases designer productivity but also helps in exploring different designs faster and trade offs between cost and performance. One of the open issues of HLS is the memory bandwidth bottleneck which limits the performance which is extremely important for the memory bound algorithms. Thus, designs implemented on Field-programmable gate array (FPGA) via HLS suffer from this bandwidth bottleneck and off chip memory latency. Current HLS tools are incapable of automatically exploiting the memory hierarchy on FPGAs and the only way to exploit the memory hierarchy is in a scratchpad fashion, but this requires considerable design effort and therefore, time-consuming. Secondly, the existing HLS tools currently exhibit a deficiency in providing dependable estimates of final Quality of results (QoR), thereby impeding designers ability to make well-informed decisions regarding the trade-offs between cost and performance.

This thesis explores and examines both these issues and addresses them by developing solutions in order to overcome aforesaid matters in question.

The first part of this thesis addresses the issue of off-chip memory latency and bandwidth bottlenecks in FPGA designs implemented via HLS. We propose an automated FPGA memory management approach using a fully-configurable source-level cache in Xilinx *Vitis HLS*. The primary objective of our cache implementation is to minimise the amount of design effort required while enabling the designer to focus on algorithmic optimizations, specifically for memory access patterns that are data-dependent or irregular in nature. Experimental results shows that our cache implementation improve the performance of different benchmarks by up to 60 times compared to the out-of-the-box HLS solution.

The second part pertains to the enhancement of QoR estimation in HLS. For this purpose, by taking advantage of the widespread use of Machine learning (ML), we propose Graph neural network (GNN)-based model that learn and predict post-implementation QoR using pre-schedule control data flow graphs (CDFGs) and HLS optimization directives. Experimental results show that our model can estimate the timing and resource usage of a previously unseen design (i.e, a completely new CDFG) within milliseconds with high accuracy, reducing prediction errors by up to 74 % compared to the estimate generated by the HLS tool.

Contents

List of Figures

List of Tables

Acronyms

API application programming interface

AXI advanced extensible interface

BRAM block RAM

C.P. critical path

CC clock cycle

CPU central processing unit

DAGNN deep adaptive graph neural network

DDR4 double data rate 4

DRAM dynamic RAM

DSE design space exploration

DSP digital signal processing unit

EDA electronic design automation

FF flip-flop

FIFO first-in first-out

FPGA field-programmable gate array

GAT dynamic graph attention network

GCN graph convolutional network

GIN graph isomorphism network

GNN graph neural network

HBM high-bandwidth memory

HLS high-level synthesis

HW hardware

II initiation interval

IR intermediate-representation

ISA Instruction Set Architecture

L1 level 1

L2 level 2

LCS load, compute, store

LLVM low level virtual machine

LSU load-store unit

LUT lookup table

MAC multiply-acccumulate

ML machine learning

MLP multi-layer perceptron

OS operating system

PPA power, performance, and area

QoR quality of results

RAW read after write

RMSE root mean square error

RO read-only

RTL	register-transfer level
RW	read-write
SSA	static single assignment
SW	software
WO	write-only

Chapter 1

Introduction

1.1 High-level synthesis

With the rapid expansion of hardware development and the increase in design complexity, there is a growing need for more efficient and effective design techniques. Traditionally, hardware design has been an expensive and a time-consuming process requiring a high level of expertise and specialized knowledge. In recent years, HLS has emerged as an important method for hardware design, allowing high-level programming languages such as C/C++ to be automatically transformed into hardware designs [10]. The introduction of HLS has allowed designers to create hardware using high-level programming languages, allowing both faster design and faster simulation than register-transfer level (RTL). By using HLS, designers can benefit from the abstraction and modularity of high-level languages, enabling them to produce more sophisticated, powerful, and portable hardware designs in less time. In addition, designers can use HLS directives (also called pragmas) to optimize hardware implementations by tradeoffs between cost and performance. This flow allows designers to quickly and effectively experiment with different design configurations before working on the final implementation.

The primary purpose of HLS to streamline the hardware design process, enabling designers to allocate more attention to algorithmic-level design while abstracting the intricacies of low-level implementation. By raising the level of abstraction, HLS aims to enhance productivity, reduce design cycles, and enable efficient design exploration and optimization.

High-level synthesis tools utilise sophisticated algorithmic techniques to analyse and optimize high-level descriptions and help in mapping complex digital designs onto Field-programmable gate array (FPGA) or ASIC (Application-Specific Integrated Circuit) architectures in a relatively short time. These tools perform various optimizations, including scheduling, resource allocation, loop pipelining and loop unrolling etc, to generate efficient RTL designs.

Furthermore, HLS provide opportunities for hardware/software co-design. With HLS, designers can seamlessly integrate hardware accelerators or custom processing units with software components, enabling the design of heterogeneous systems that leverage the strengths of both hardware and software. This approach helps in attaining the goal of best design point while considering various limitations such as cost, performance, power-consumption and time-to-market [48].

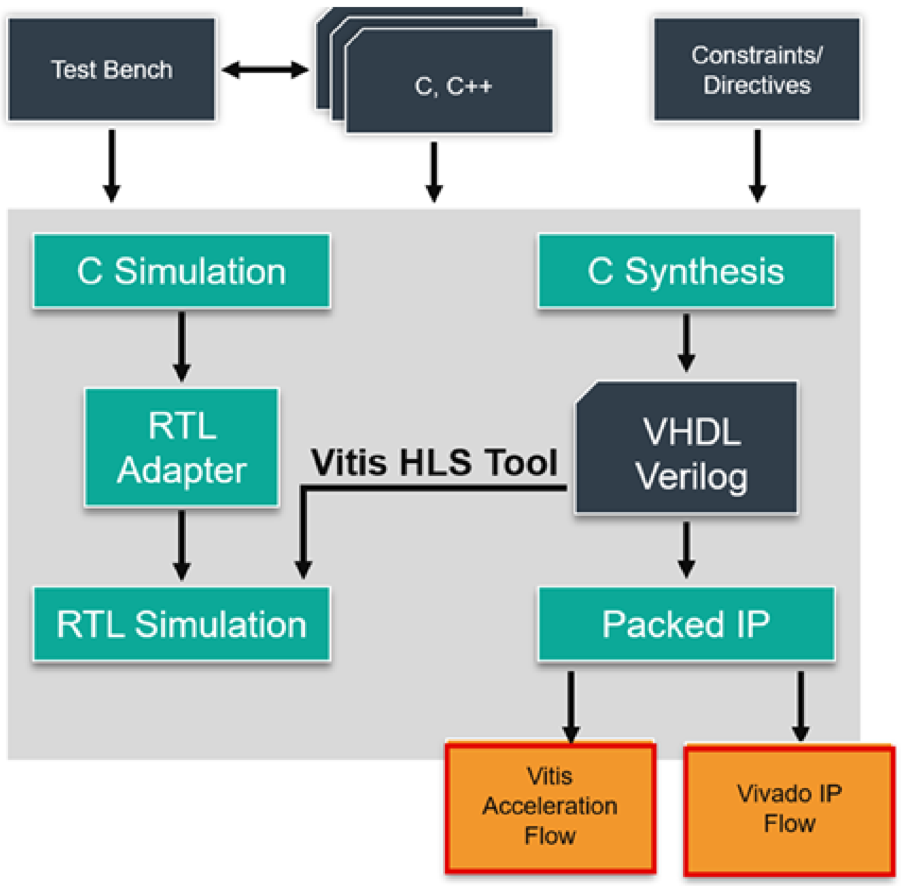


Fig. 1.1 Vitis HLS workflow [56]

Fig. 1.1 shows the *Vitis HLS* development flow which has been used in this research work.

1.2 Problem Statement

HLS can greatly decrease the amount of design effort, allowing the use of convenience software (SW)-like tools and development processes but it still suffers from some issues.

An HLS open issue is the off-chip memory latency and bandwidth bottleneck, which limits performance and is especially critical for memory-bound algorithms. The FPGA memory system is composed of two main kinds of resources: fast small on-chip memories (registers and block RAMs (BRAMs)), and slow large off-chip memories (dynamic RAMs (DRAMs)) interfaced through double data rate 4 (DDR4) or high-bandwidth memory (HBM) protocols (the latter characterized by even larger latency [51]). Current HLS tools, in particular those from the leading producer Xilinx, allow the designer to exploit this memory hierarchy only manually, in a scratchpad-like way, which often requires significant design and verification effort. *This makes harder to achieve the deployment of accelerated applications using FPGAs for a large number of applications.* Our work aims directly at filling this gap, thus *making HLS design more software-like* for use cases in which the ultimate performance need not be achieved, but *design time and effort are paramount*.

Another issue is the prediction of post-implementation Quality of results. Current commercial HLS tools do not provide reliable estimates of the final QoR [9]. As a result, designers are unable to make cost/performance trade-offs and guarantee that the design will meet the requirements because the estimation results in terms of timing and resource usage often significantly differ from the actual QoR after implementation. Graphs are a widely used model in EDA tools [40]. In recent years, EDA tools have recently begun to use machine learning approaches, especially for analysis tasks [35]. Recent research has shown that *graph-based machine learning* techniques can be successfully applied to various phases of the EDA flow, including logic synthesis [20, 50], placement and routing [37, 38, 19, 31], power estimation [33], verification [62], and testing [41]. By using the Graph neural network, in this work, we aim to predict the post-implementation QoR of an HLS design, starting from both the user C/C++ code and the user-defined optimization directives.

1.3 Contribution

In this thesis, the focus of the research is to improve Quality of results (QoR) for High-level synthesis (HLS) based Field-programmable gate array (FPGA) designs by overcoming the issues discussed in Section 1.2. One part of the research deals with the automation of FPGA memory system while the other part proposes a graph neural network-based framework to predict the post-implementation quality of results for a given HLS-based design.

The first part (Chapter 2) of the thesis proposes array-specific dataflow caches to automate FPGA memory management to overcome the off-chip memory latency and memory-bandwidth bottleneck, thus allowing the designer to focus on algorithmic optimizations. For this, a cache module is developed which works as an interface with the off-chip memory (DRAM), accessible through an advanced extensible interface (AXI) bus and sores its data to on-chip blocks RAMs (BRAM) and registers. We first developed a single port cache module in the form of a C++ class which is configurable through templates in terms of number of sets, ways, words per line and replacement policy. During the second phase, the emphasis was put on performance optimization. For this purpose, a multi-level cache is proposed which increases the memory hierarchy of the cache by adding a level 1 (L1) on the top of dataflow cache (level 2 cache). This architecture helps in further reducing the memory access latency and provides a basis for building a cache architecture with multiple concurrent accesses. Dataflow and Multi level caches provide a maximum throughput of one access per iteration. This is not efficient for cases where the array is accessed multiple times in a single iteration. Therefore, a multi-port cache is developed that enables multiple concurrent read accesses to the same array. This is implemented as an extension of the multi-level cache. The number of ports can be configured through a template parameter. This also allows to overcome *Vitis HLS* limitation of a single reader per AXI interface. To adhere to the HLS high-productivity philosophy, we paid a special attention to the HLS user-friendliness in terms of (a) *configurability* (the cache characteristics can be set through parameters), (b) *ease of use* (the cache can be inserted into existing designs with just a few lines of boilerplate code), (c) *observability* (cache information critical for parameter tuning, e.g., hit ratio, can be profiled during SW simulation).

The second part (Chapter 3) proposes a graph neural network (GNN) based framework is developed to predict the post-implementation QoR from the pre-schedule control data flow graph (CDFG) representation of an HLS design targeting FPGA implementation, also considering the user HLS optimization directives. For this purpose, a dataset is built from a variety of designs covering a wide range of different applications from the well known HLS benchmark suites. To create multiple hardware implementations for each design, different HLS synthesis directive and various clock periods are used. Each design point is synthesized and implemented with *Vitis HLS* and Vivado respectively. A method is proposed that exploits a graph based representation of an HLS design that includes both program semantics and synthesis directive information but not scheduling and binding information. This graph is extracted from the low level virtual machine (LLVM) model generated by the open source front end of *Vitis HLS*. QoR prediction problem is formulated as a multi-objective regression task to estimate post-implementation resource usage and timing without invoking the back end of the HLS tool. A multi-objective GNN based learning model is trained to learn the underlying heuristics and optimizations techniques to predict the desired objectives, namely lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path timing (C.P.).

1.4 Thesis Structure

The main objective of this research is to improve Quality of results (QoR) for High-level synthesis (HLS) based Field-programmable gate array (FPGA) designs by studying the two important problems in HLS which are related to memory bandwidth bottleneck and estimation of final QoR. In this regard, the thesis is divided into two major parts and structured in different chapters as follows:

- Chapter 1 presents a thesis statement and offers an overview of the underlying motivation that motivates the research.
- Chapter 2 proposes an automated FPGA memory management flow using a user-friendly cache in order to improve the performance of memory-bounded algorithms.
- Chapter 3 proposes a GNN-based predictive model to predict the post-implementation QoR.

- Chapter 4 summarizes the thesis work and offers insights into potential avenues for future research.

Chapter 2

Array-Specific Dataflow Caches for HLS of Memory-Intensive Algorithms on FPGAs

Part of the work described in this chapter has been previously published in [5].

According to the best design practice from Xilinx [56], efficient HLS kernels should comply with the load, compute, store (LCS) paradigm to mitigate the off-chip memory bottleneck, i.e., access external DRAM only by load and store dataflow tasks, which are then responsible for buffering on the on-chip memory the data consumed and produced by the compute task(s). The main drawback of the LCS approach is the significant design effort needed for converting a generic algorithm into LCS form, which often requires full rewriting and redesigning of the source code.

A cache is in general helpful to implement well-performing designs in a short time. Moreover, techniques such as manual buffering or polyhedral transformations [11, 45] cannot be applied to programs with irregular or input-dependent memory access patterns, and are only partially implemented in commercial design tools such as *Vitis HLS*. Therefore, a cache could be the only solution for quickly optimizing the performance of such designs using commercial tool flows.

The aim of the work in this chapter is to automate efficient off-chip memory accesses through an easy to use and fully customizable cache system for HLS, which works as an interface with the off-chip DRAM, accessible through an AXI bus, and

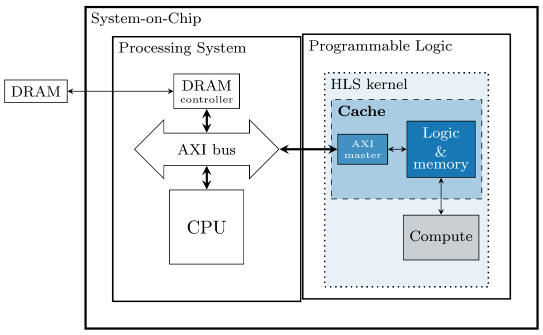


Fig. 2.1 Our cache embedded in a hardware (HW) setup.

stores its data to on-chip BRAMs and registers. Figure 2.1 shows the resulting system when our cache is used to accelerate an HLS kernel. Our cache is placed within the HLS kernel. The computation logic of the kernel accesses the cache, rather than the AXI bus directly.

From a high-level point of view, the cache has the objective of isolating the off-chip memory accesses into a dataflow task, in accordance with the LCS pattern.

From a low-level point of view, the cache has the dual purpose of (a) reducing the number of DRAM accesses, i.e., the data stored in the cache is reused as long as it hits, and only the misses need to access the DRAM, and (b) optimizing DRAM accesses, i.e., the DRAM is accessed in lines (aligned groups of consecutive words), which allows taking advantage of AXI bursts and interface widening, even with hard to analyze or totally irregular access patterns.

2.1 Related Work

The need for automated memory management for FPGAs is attested by the multiple works on this topic.

Matthews et al. [44], Choi et al. [7] designed FPGA-based caches. These works differ from ours as they are aimed to accelerate specific soft-processors implementations instead of generic HLS designs.

Jo et al. [27] developed an OpenCL framework whose memory subsystem inserts direct-mapped, single-level, and single-port caches in between the kernel accessing the memory, and the external memory. They implemented at RTL both the kernels (which consist of a predefined set of intellectual property blocks) and the cache. Our work therefore differs both in terms of cache architecture complexity (our caches provide set associativity, two levels, and multiple ports), and in terms of technology (our cache is compatible with any HLS design).

Several works focused on optimizing the memory accesses through RTL cache modules inserted between the kernel accessing the off-chip memory and the off-chip memory interface. These modules can be either inserted manually or through a dedicated framework, such as the one proposed by Adler et al. [3] which virtualizes the FPGA memory hierarchy and includes some caching capabilities. Winterstein et al. [52] improved this framework specifically for HLS by allocating the unused BRAMs to maximize the cache sizes. However, an RTL cache module fails to provide significant speedup when coupled with an HLS kernel. For example, the *Vitis HLS* scheduler, unaware of the external cache module, inserts a minimum latency between a memory request operation and its corresponding response based on the architecture of the memory adapter, thus preventing the exploitation of the cache acceleration. Our cache is instead implemented at the source level, and it is specifically designed to avoid scheduling based on the worst case (cache miss). This allows the HLS tool to optimize the circuit accordingly.

We ran some experiments adding an RTL cache module (specifically the Xilinx System Cache [60]) to the interface of an HLS kernel. The results show that the RTL cache did not provide any advantage. It simply introduced an overhead, as discussed in Section 3.8.

Cong et al. [11], Pouchet et al. [45] designed a workflow for improving data locality of HLS programs through compiler-level loop transformations, taking advantage of the polyhedral representation. Moreover, they exploited this locality by automatically inserting on-chip buffers. These techniques are limited to programs with affine loop bounds and memory accesses, while a cache can be used with any program, including those with irregular or data-dependent memory accesses. A cache could benefit from their improved locality by achieving higher hit ratios with simpler cache configurations.

The Intel HLS [25] tool provides load-store units (LSUs) that can cache DRAM data in BRAM in case of read-only (RO) memories. Our experiments described in Section 3.8 suggest that the tool fails to determine the optimal cache configuration and the user has limited control to improve it.

The work by Ma et al. [39] is closest to ours. They proposed an open-source array-specific HLS cache module as a set of C++ classes, compatible with *Vivado HLS 2016.2*. Different from our work, the cache logic is inlined in the application. While this helps keeping the hit latency low in simple cases, it violates the LCS pattern. Moreover, their architecture increases the pipelining complexity. To mitigate this problem, they mapped the whole cache data to registers. However, in the experiments discussed in Section 3.8, we verified that the pipelines embedding their cache require higher initiation intervals (IIs), or are not pipelineable at all. Moreover, mapping all the data to registers limits strongly the cache size scalability due to HW resources constraints. Instead, our architecture completely hides the cache logic and the memory interface from the main computations performed by the kernel. This allows the HLS to synthesize pipelines with low II while mapping cache data to cheaper BRAMs. Finally, their cache automatically handles only one access port thus providing only one read or write per clock cycle (CC). The only way to perform multiple accesses per CC is to guarantee that other accesses, beyond the first one in a given CC, will always be hits, and make it explicit through the `retrieve` and `modify` functions. This is both difficult and error-prone to analyze manually in complex cases.

2.2 Dataflow Cache

The *Dataflow cache* architecture (Fig. 2.3a) is isolated into a dedicated dataflow process. An HLS kernel that is configured to use the cache for one of its top-level DRAM-mapped arrays is split into two dataflow tasks: (i) the compute task, which includes all the application logic except for the external memory interface, which is replaced with the simpler cache interface, and (ii) the cache task (or, in general, one cache task per array that uses the cache), which buffers data and interfaces with the external DRAM. Thus, the kernel automatically complies with the LCS architecture without any manual code change.

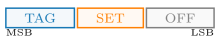
This architecture is characterized by information flow from the compute task to the cache (the address to be accessed and the data to be written), and from the cache to the compute task (the read data).

Algorithm 1 describes the *Dataflow cache* functionality. The cache task waits for a request and executes the standard cache operations: it checks if the request is a hit

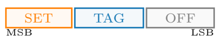
Algorithm 1 *Dataflow cache* functionality.

Require: *Compute* needs to access an array associated with *Cache* at address *addr* in read mode ($op = R$) or write mode ($op = W$, $data =$ element to be written).
Ensure: The operation requested by *Compute* is fulfilled by *Cache*.

```
procedure COMPUTE
    ...
    Send $op$ to $Cache$
    Send $addr$ to $Cache$
    if $op = W$ then
        Send $data$ to $Cache$
    else
        Wait for $Cache$ response
        Receive $data$ from $Cache$
    end if
    ...
end procedure

procedure CACHE
    Wait for $Compute$ request
    Receive $op$ from $Compute$
    Receive $addr$ from $Compute$
    if $op = W$ then
        Receive $data$ from $Compute$
    end if
    $line : addr \in line$
    if $line \Rightarrow MISS$ then
        if $line_{\text{old}} \Rightarrow DIRTY$ then
            $DRAM(line_{\text{old}}) \leftarrow BRAM(line_{\text{old}})$
        end if
        $BRAM(line) \leftarrow DRAM(line)$
    end if
    if $op = W$ then
        $BRAM(addr) \leftarrow data$
    else
        $data \leftarrow BRAM(addr)$
        Send $data$ to $Compute$
    end if
end procedure
```



(a) Standard mapping.



(b) Swapped mapping.

Fig. 2.2 Configurable address bit mapping.

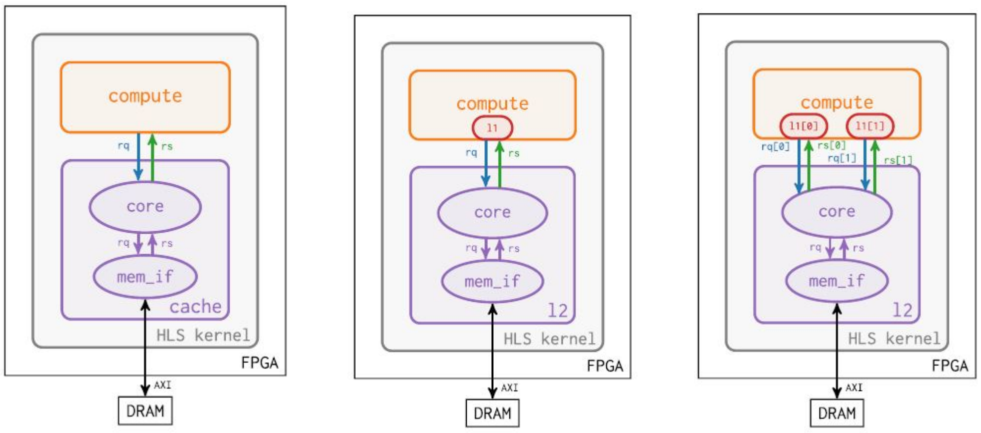
LISTING 2.1 Source code modifications for accelerating the `compute` function with our cache.

```
+#include "cache.h"
+
+typedef cache<DATA_TYPE, RD_ENABLED, WR_ENABLED,
+   MAIN_SIZE, N_SETS, N_WAYS, N_WORDS_PER_LINE, LRU,
+   SWAP_TAG_SET, LATENCY> cache_type;

 template <typename T>
 void compute(T &a) {
    for (auto i = 0; i < (N - 1); i++) {
 #pragma HLS pipeline
       a[i] = a[i + 1];
    }
 }

 extern "C" void top(DATA_TYPE *a) {
 #pragma HLS interface m_axi port=a bundle=gmem0
+#pragma HLS dataflow
+   cache_type a_cache(a);
-   compute(a);
+   cache_wrapper(compute<cache_type>, a_cache);
 }
```

or a miss, it updates the cache data structures (valid bits, dirty bits, tag bits, . . .), and it performs the DRAM read or write operation. For reads, it also sends back the data. The compute task sends the read or write request to the cache. For reads, it waits for the response containing the read data.

The *Dataflow cache* uses the set associative mapping and the write-back consistency policy. It is configurable in terms of (a) word size, (b) number of words per line, sets, and ways, (c) replacement policy, least recently used or first-in first-out (FIFO), (d) address bit mapping, standard (Fig. 2.2a) or swapped (Fig. 2.2b, convenient in use cases like the one discussed in Section 2.5.2, i.e., a matrix accessed by columns). It can implement a fully associative policy if the number of sets is one, or a direct mapped policy if the number of ways is one.

Listing 2.1 highlights the modifications needed for accelerating the `compute` function with our cache. Users simply need to (1) set the cache parameters through the `cache` class template arguments, and (2) instantiate the cache and call the `compute` function through the `cache_wrapper` function in a dataflow region.



(a) *Dataflow cache*. (b) *Multi-level cache*. (c) *Multi-port cache*.

Fig. 2.3 Baseline *Dataflow cache* architecture, and its extensions.

It is worth noting that the `compute` function is unchanged, since we overloaded the `operator[]`, like Ma et al. [39], to allow using a cache object as if it were a traditional array.

In the presence of feedback between different tasks, the HLS-generated HW circuit would deadlock. Moreover, we have to carefully specify cycle by cycle the scheduling of the operations to avoid losing performance or causing unexpected deadlocks.

To avoid the deadlock, the *write request* and *read response* must be scheduled into separate pipeline stages by:

1. Explicitly declaring a dependency between *write request* and *read response* using the `write_dep` and `read_dep` FIFO access functions provided by *Vitis HLS* to define a partial ordering between accesses to different streams.

2. Setting the dependency distance to 1 CC by delaying it with the `reg` function, also provided by *Vitis HLS*.

While this solution guarantees the functionality of the generated HW, it fails to achieve high throughput. In fact, assuming that both the *Core* and the *Memory Interface* are pipelined with an II of 1 CC (i.e., the most performance-critical case) and the *Memory Interface* pipeline depth is $D > 1$, the HLS scheduler schedules the *read response* in the CC following the *write request* because it is unaware of the latency between *write request* and *read response*. This reduces the overall performance.

However, if we set the dependency distance between *write request* and *read response* to D CCs, the scheduler inserts $D-1$ pipeline stages between them. In each CC, the *Core* writes one request and receives one response. Our solution allows optimally exploiting the pipeline with an II of 1 CC, without incurring stalls.

2.2.1 Dataflow cache implementation

The *Dataflow cache* is implemented as a C++ class, compatible with *Vitis HLS*. All the configurable parameters are set using class template arguments.

The cache task provides high throughput in steady-state (it serves a hitting request in one CC), since it is optimally pipelined with $II = 1$ CC.

AXI interface

The *Memory interface* task accesses the AXI bus at every request from the *Core* task. To save resources, it is not pipelined. Pipelining would rarely help, because a well-configured cache should never get multiple sequential misses, especially considering that there is one dedicated cache per source code array.

All DRAM accesses handle whole cache lines, which are sequential and aligned to the line size. To enable the HLS tool to infer that accesses are aligned, we explicitly zeroed the least significant bits of the address. This enables automated port widening and burst inference. If the line size is at most the maximum AXI interface width, it is accessed in a single request, else (more commonly) it is accessed in a burst request.

By default, *Vitis HLS* assumes *AXI latency* 64 CCs. This is useful to send pipelined requests on the AXI interface. However, our *Memory interface* is not pipelined. Thus, we set the AXI latency to zero, which makes the *Memory interface* stall after issuing an AXI request and resume right after the response, saving resources without losing performance.

Cache interface

To interface with the cache, we exposed the user-callable application programming interfaces (APIs) for managing requests and responses between the compute task and the cache.

- The `get` function accepts as input the address to read from cache and returns the read data. Internally it sends a read request (writing the address to the request FIFO), waits for the request-response latency (discussed later) in case of a hit or for longer in case of a miss, reads the data from the response FIFO, and returns the received data.
- The `set` function accepts as input the address and the data to write to the cache. Internally, it sends a write request (writing the address and the data to the request FIFO).

We overloaded the `operator[]` to automatically call the `get` and `set` functions, e.g., in Listing 2.1, `a[i] = a[i + 1]` is automatically compiled to `a.set(a.get(i + 1), i)`.

As discussed in Section 2.2, the request-response distance should match the cache latency. The request-response distance value has an importance in terms of achieving a steady state without stalling. If the correct distance value is not set, it prevents exploiting the cache pipelining. In Section 2.5.2, the impact of this value on the execution time is studied. However, cache latency varies at runtime, as hits and misses (which have different latencies) are interleaved, depending on the access pattern and the cache configuration. Moreover, we need to distinguish between the different memory access types.

- For RO caches, the optimal distance value is typically around the average memory access latency $\overline{lat}$

  $$\overline{lat} = lat_{\text{cache}} \cdot hit\,ratio + lat_{\text{DRAM}} \cdot miss\,ratio. \tag{2.1}$$

  lat_{cache} varies from 3 CCs to 5 CCs based on cache configuration and timing constraints, lat_{DRAM} depends on the target FPGA board, and *hit ratio* and *miss ratio* depend on the application and cache configuration.

- Read-write (RW) caches are affected by data dependencies with distances corresponding to the request-response distance. The latter should therefore balance cache performance and computation task performance (task II depends on the dependency distance). Experimental results (Section 2.5.4) show that a 2 CCs distance typically gives the best overall performance.

- For write-only (WO) caches, the request-response distance has no meaning because there is no response.

A template parameter is available to the users willing to fine-tune the distance of the caches in their designs.

2.3 Multi-Level Cache

The *Multi-level cache* extends the memory hierarchy of the cache by adding a level 1 (L1) cache on top of the *Dataflow cache*, i.e., the level 2 (L2) cache, as shown in Fig. 2.3b. This architecture is aimed at reducing the latency between a read access request and the corresponding response.

We are not interested in further accelerating the writes. Write latency has a negligible impact on performance, considering that they never stall the compute task (there is no response from the cache to the main computation), provided that the request FIFO is deep enough to accommodate all the pending writes. Moreover, write accesses are usually less frequent than reads.

Finally, the *Multi-level cache* is the starting point for enabling multiple concurrent accesses in the *Multi-port cache* described in Section 2.4.

Similarly to the cache by Ma et al. [39], the L1 cache is inlined in the compute logic. This reduces the latency of the memory accesses by avoiding the inter-task communication. Even if the L1 cache is inlined, the compute task pipeline II is preserved, unlike the cache by Ma et al. [39]. This is because (i) in case of miss the L1 cache interacts with the L2 cache instead of with the external DRAM. Furthermore, (ii) the L1 cache complies with the write-through policy (the L1 cache aims at accelerating only the reads), introducing fewer dependencies compared with the write-back policy.

To implement the *Multi-level cache* architecture, we extended the *Dataflow cache* source code. In the *Dataflow cache*, the response flows from the L2 to the compute task and contains a single word. In the *Multi-level cache* architecture, the response flows from the L2 to the L1 cache, and holds a whole cache line.

The *Dataflow cache* APIs were updated to support the L1 cache by adding the `get_line` function. Moreover, we upgraded the implementation of the `get` and `set` functions, while keeping their signature unchanged.

- The `get_line` function receives as input the address to read from cache and returns the line to which the address belongs. In particular, if the address hits the L1 cache, the line is read from the L1 cache. Otherwise, the request is issued to the L2 cache, as with the `get` function of the *Dataflow cache*.
- The `get` function calls the `get_line` function and returns the requested word only.
- The `set` function marks the L1 cache line as dirty, if it hits, according to the write-through policy. Additionally it forwards the write request to the L2 cache as with the *Dataflow cache*.

The L1 cache supports the set-associative mapping policy. The number of sets and ways of the L1 cache are configurable through template parameters. Note that when the L1 cache parameters are set to zero, the resulting architecture is equivalent to the *Dataflow cache*.

Similarly to the L2 cache, the L1 cache memory is bound to BRAMs and the helper data is bound to registers. Both the L1 and the L2 caches use the same memory technologies, therefore the L1 cache could have comparable or even bigger size than the L2 cache. In experiment sections, Section 2.5.2 and Section 2.5.3, there are cases where L1 cache is larger than L2 cache. In these cases, L2 cache works as a memory arbiter. The results show the advantage in terms of performance.

According to our experimental results (Section 2.5.2), when an L1 cache is included on top of the L2 cache a convenient default value for the L2 request-response distance is 3 CCs for RO accesses, and 2 CCs for RW accesses.

Note that the default RW distance is lower than the RO one because higher distance values would make the read after write (RAW) dependencies distance longer and reduce the overall performance, as discussed in Section 2.2.1.

2.4 Multi-Port Cache

The *Dataflow* and *Multi-level* cache architectures provide a maximum throughput of one access per CC. This is efficient for pipelined algorithms, which access each cached array at most a single time per iteration. To efficiently implement algorithms which access the same array multiple times per iteration (either due to the user code or after a loop unrolling), we designed the *Multi-port cache* that enables multiple concurrent read accesses to the same array.

With the *Multi-port cache* architecture, a shared L2 cache exposes an arbitrary number of ports, each with a private L1 cache, as shown in Fig. 2.3c. The private L1 caches enable scheduling multiple memory accesses at the same time, without increasing the II of the compute task.

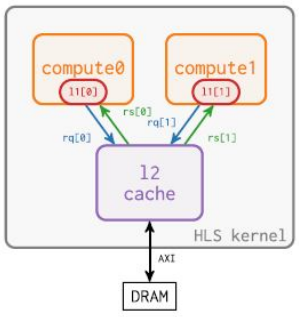
Hence, unlike Ma et al. [39], the L1 cache does not use directly the single DRAM interface, but goes through the shared L2 cache. Thus, we do not require users to manually mark explicitly some accesses as “always hit” (through the `retrieve` and `modify` functions), which would require extensive manual analysis and code changes and may lead to incorrect behavior.

To keep the cache logic simple and to avoid negatively affecting the compute task II, we did not implement any coherency mechanism. To guarantee the correct functionality, the *Multi-port cache* only supports read accesses. The extension to concurrent write or RW accesses is left to future work.

The *Multi-port cache* is implemented as an extension of the *Multi-level cache*. The number of ports P can be configured through a template parameter. When it is set to one, the architecture is equivalent to the *Multi-level cache*.

The *Core* task of the L2 cache was updated to cycle over each port, i.e., it sequentially serves the requests from the first to the last port, before restarting from the first one. Any port that did not send any request is skipped. This code pattern (hidden from the user behind the cache `operator[]`) can be optimized by the HLS tool to statically schedule P array accesses with $II = 1\,\mathrm{CC}$ in most cases.

For each port, we allocate a private L1 cache, and the related pair of request and response FIFOs (to communicate with the shared L2 cache).



(a) Dataflow graph.

```
void compute0(cache_type &c, ...) {
    ...
    c.get(addr, 0);
    ...
}

void compute1(cache_type &c, ...) {
    ...
    c.get(addr, 1);
    ...
}

void top(data_type *arr, ...) {
#pragma HLS interface m_axi port=arr
#pragma HLS dataflow
    cache_type c;
    c.run(arr);
    compute0(c, ...);
    compute1(c, ...);
}
```

(b) Source code.

Fig. 2.4 Multiple-reader DRAM-mapped array, associated with our cache.

The access port can be selected either automatically or manually, when the user-friendly automatic port selection does not lead to the desired II for the algorithm pipeline.

- With the automatic selection, each call to `get_line` (which is in turn called by `get`) is automatically associated to a specific port by means of a member variable holding the port index, which is updated after each access. This is implemented directly in the `get` function, that keeps track of the last accessed port and uses this information to bind a specific request to a specific port.
- The manual port selection allows one to explicitly inform the tool that each access uses different address and data streams, and that the dependencies are false. It is implemented by adding the `port` parameter (which identifies the number of the port to be accessed) to the `get` function (in this case the `operator[]` cannot be used).

In addition to the performance advantage, our *Multi-port cache* allows overcoming the *Vitis HLS* limitation of a single reader per AXI interface. Indeed, each L2 cache (associated with a single AXI interface) can expose multiple ports in the form of pairs of request/response FIFOs. These ports can connect the L2 cache to one or more `compute` dataflow tasks. Since the L2 cache ignores the ports with no pending requests, the `compute` tasks can seamlessly issue requests to the L2 cache at different rates. Figure 2.4 shows the dataflow graph of a kernel with a DRAM-mapped array that is read from two `compute` dataflow tasks, through a single L2 cache. Additionally, each `compute` task has its own private L1 cache. In *Vitis HLS*, if designers need to access the same DRAM array from different dataflow tasks, they must instantiate multiple AXI bundles, associated to the same underlying buffer in DRAM. Note that, due to the loose synchronization between dataflow tasks in *Vitis HLS*, both a dual-ported cache and a pair of bundles can be used meaningfully only for read-only arrays. Otherwise enforcing cache coherency or preserving data dependencies in a shared array between two processes would be very difficult.

2.5 Experiments

To evaluate the impact of the proposed cache architecture in terms of power, performance, and area (PPA), we used the cache in some memory-intensive benchmarks with very different access patterns. *We selected two "classical", frequently used algorithms (matrix multiplication and convolution), since they are widely known and provide good and easy to understand examples. In practice, our caches should be used either (i) for seldom used algorithms, for which a manual optimization effort would not be justified, or (ii) for those that do not exhibit regular access patterns, such as bitonic sorting, which is our third benchmark.*

We synthesized the benchmarks as *Vitis HLS* kernels and deployed them on a physical FPGA board to measure the resulting PPA. The experimental workflow consists of: (1) SW simulation, (2) HLS synthesis, (3) logic synthesis, place and route, and bitstream generation, and (4) execution and measurements.

Steps (1) and (2) were performed in *Vitis HLS* 2021.2 (using *Vitis* flow defaults), and step (3) in *Vivado* 2021.2 [57] (using *Vivado* defaults for synthesis and implementation). All steps targeted the *Avnet Ultra96v1* [4] board, hosting a *Xilinx Zynq UltraScale+* FPGA. Figure 2.5 shows the block design for implementing an HLS kernel with three DRAM-mapped arrays (such as the matrix multiplication and convolution test cases). Given an algorithm (which determines the number of inputs and outputs, and by consequence of the AXI interfaces), the HLS kernel exposes the same interface, even when it is optimized with our cache, since the cache is fully implemented with HLS inside the kernel itself.

The board runs the *PYNQ Linux 2.7.0* [59] operating system (OS), whose *PYNQ* library is exploited in step (4).

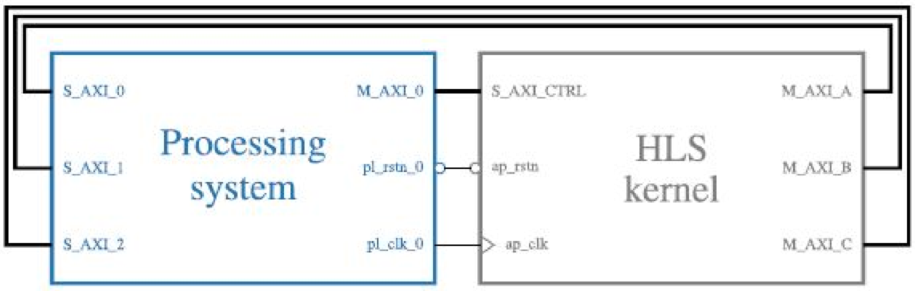


Fig. 2.5 Block design with three DRAM arrays.

We collected the data from different sources:

- *SW simulation reports*
  - *Hit ratio*: ratio between the number of requests that hit data in cache and the number of all requests for a specific cache memory.
- *Post place and route reports*
  - *Area*: number of LUTs, FFs, BRAMs and DSPs required to implement the whole design.
  - *Maximum clock frequency*: the maximum frequency at which timing was met by the implementation flow, achieved by gradually increasing the clock frequency constraint. The frequency higher bound is 333 MHz, that is the maximum supported frequency for the AXI adapter (330 MHz in practice, due to the clock generation logic limited precision).
- *Runtime measurements*
  - *Performance* (t_{ex}): execution time, measured between the assertions of the start and the end signals of the kernel.
  - *Power* (P): average power, measured by the sensor on the system power rail during kernel execution. Note that the selected board does not allow measuring the power of the FPGA only, therefore P is the power consumed by the whole board, including the CPU.

The measured quantities are not fully deterministic. The timer measuring t_{ex} may not be stopped at the exact time when the kernel asserts its end signal, since it checks this condition through polling and the CPU might be busy running other tasks of the OS. Also, power consumption is affected by different factors, such as the CPU load or the temperature. Thus, each runtime measurement was taken five times and is collected as the average and the standard deviation of these measurements. The energy consumption (E) is computed as the average energy, $E := P t_{\text{ex}}$.

To limit the design space, in all the cache configurations we used a default L2 cache request-response latency. For single-level RO configurations, we computed the default distance value as 7 CCs, according to (2.1), where the $\text{lat}_{\text{cache}}$ was set to the worst-case, i.e., 5 CCs, lat_{DRAM} was set to 40 CCs according to the measurements by Marjanovic [43] of the read latency of the high-performance coherent ports of the target board, and hit ratio was assumed to be 95 % (these values were only used to set the cache parameters, while the runtime results reflect the real latencies and hit ratios). The experiments show that these approximations achieve good pipeline performance. A significant performance degradation is observed only if one assumes very low (1 CC), or very high (more than lat_{DRAM}) distance values. We used a default distance of 3 CCs for multi-level, and 2 CCs for RW cache configurations. WO cache configurations are unaffected by the distance parameter.

In order to compare directly the cycle count performance of the various designs, we constrained the clock frequency to 100 MHz in all experiments, except for those that are related to the timing impact of the cache (Sections 2.5.2 to 2.5.4).

We manually chose the cache parameters, such as the line size, number of lines, and so on, based on the array access patterns. However, there are multiple methods to automate the selection of these parameters, as attested by a large amount of past work, for example those analyzed by Upadhyay and Sudarshan [49]. Integration of those approaches with our cache is left to future work.

2.5.1 Reference designs

We compared the collected results with:

1. *Baseline*: the kernel generated by default by the HLS tool, whose computational core directly accesses the external DRAM through the AXI interface.

2. *RTLCache*: the *Baseline* HLS kernel, with the Xilinx System Cache RTL module inserted in between the HLS kernel and the AXI DRAM interface (when the cache module configurability allows a setup with non-zero hit ratios).

3. *Manual*: the kernel manually optimized for buffering the data using the on-chip memories (when the memory access patterns allow it).

Ma et al. cache reference

Ma et al. [39] reported results collected from unreliable sources. They collected the area figures from post-HLS-synthesis reports, which are estimations known to be affected by significant errors. Moreover, they estimated performance and power data using RTL simulation, which is based on simplified models (especially for the AXI model, the DRAM controller, and the DRAM itself), which are crucial in this context. Additionally, due to the long execution time of the RTL simulations, their input sizes were limited to small values.

Nevertheless, since their code is open source, we tried to generate results comparable to ours by applying our implementation flow to their cache. We first adapted their cache, (designed for *Vivado HLS*) to *Vitis HLS*. The changes involved only their APIs, not the HW. However, using *Vitis HLS* for the kernels embedding their cache generates very poorly performing HW, e.g., the matrix multiplication innermost loop achieved $II = 141\,\mathrm{CC}$ instead of 1 CC in their tests using *Vivado HLS*, and the bitonic sorting loop was not pipelined at all in *Vitis HLS*. Therefore, we stopped the implementation flow at the HLS synthesis step, since their cache would perform even worse than the *Baseline* that achieves better pipelining, and we avoided any further comparison.

Intel cache reference

To evaluate the caching capabilities of the Intel LSUs [24], we used the *Intel DevCloud* environment, which provides the Intel HLS tool and enables remote access to an *Intel programmable acceleration card* hosting an *Arria 10 GX* FPGA. The tool automatically allocates an LSU for each off-chip array, and each RO LSU can include a cache. The cache characteristics (number of words per line, number of sets,

Algorithm 2 *Standard Matrix Multiplication*

```
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 StdMatMult($A, B, C$)
    LOOP_I: for ($i \leftarrow 0; i < N; i \leftarrow i+1$) do
      for ($j \leftarrow 0; j < P; j \leftarrow j+1$) do
        $acc \leftarrow 0$
        LOOP_K: for ($k \leftarrow 0; k < M; k \leftarrow k+1$) do
          $acc \leftarrow acc + A[i][k] \cdot B[k][j]$
        end for
        $C[i][j] \leftarrow acc$
      end for
    end for
  end procedure
```

number of ways, ...) are determined automatically and are not reported to the user, who can optionally control only the total cache size.

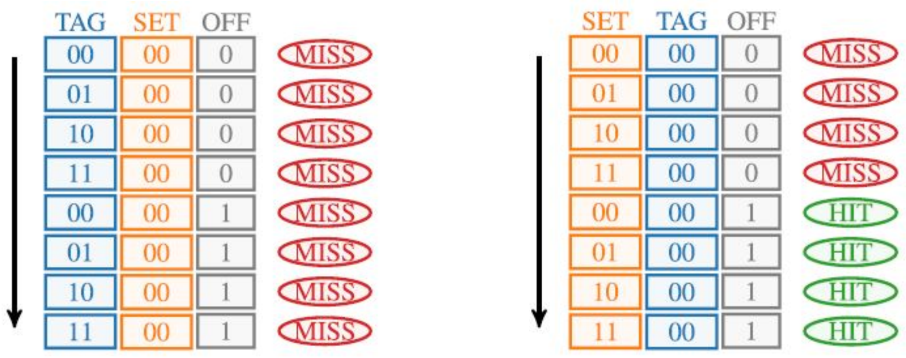
We analyzed the PPA impact of the LSUs by running some experiments using a standard matrix multiplication (Algorithm 2). The tested configurations include (a) the automatic test case, in which we did not set the cache sizes, (b) the lower-bound test case, in which we set all the cache sizes to 0, and (c) the upper-bound test case, in which we set the caches to fit the whole matrices. Compared with the lower-bound test case, the automatic case is 8 % faster and the upper-bound is 80 % faster. The automatic cache parameters selection is therefore suboptimal. Most probably because one matrix is accessed by columns, hence with limited locality. Moreover, the performance advantages are quite limited even in the upper-bound case, when the matrices are entirely stored to cache. This is because the *Intel Arria 10* has a low off-chip memory latency, from 3 CCs to 23 CCs [23].

We did not have access to an Intel FPGA with a higher off-chip memory latency, which would make the cache impact more significant. Thus, the low-latency of off-chip memory coupled with the limited control over the LSU cache parameters prevented us from performing a more thorough comparison with our cache.

2.5.2 Matrix Multiplication

The *Matrix Multiplication* (*MatMult*) standard implementation (*StdMatMult*) is shown in Algorithm 2. It accesses each matrix according to a specific pattern:

- A is accessed by rows, and each row is accessed P times, for a total of $N \times M \times P$ memory accesses. Its cache should fit a matrix row at a time.



(a) Standard address bit map. (b) Swapped address bit map.

Fig. 2.6 *MatMult*: sequence of addresses of B accessed during the first 8 iterations, where $B \in \mathbb{R}^{4\times 8}$ has a 4-set direct-mapped cache.

- B is accessed by columns, and each column is accessed P times, for a total of $N \times M \times P$ memory accesses. Since the matrix is stored in row-major order, the spatial locality is very poor. To get a non-zero hit ratio, we need either an expensive M-way fully associative cache, or a more efficient M-set direct-mapped cache exploiting the swapped address bit mapping (Fig. 2.2b).

  With an M-set direct-mapped cache, the standard address bit mapping (Fig. 2.6a) results in subsequent accesses to the same set with new tags leading to continuous cache line overwriting and misses. Our custom address bit mapping (Fig. 2.6b) enables instead subsequent reads to access distinct sets with the same tag and yields a high hit ratio.

- C is accessed sequentially, once. A single-line n-word cache provides $n-1$ hits every n accesses, making write burst inference easier.

The *StdMatMult* algorithm requires the B cache to have M lines. While this is feasible with relatively small matrices, it cannot scale up with matrix sizes.

To make the cache configuration independent of M, and ensure scalability, we also implemented a blocked matrix multiplication (*BlkMatMult*) algorithm (Algorithm 3). It is a commonly used efficient implementation of *MatMult*, which accesses all matrices by blocks, instead of columns, to improve the spatial locality of accesses to the B matrix:

- A is accessed by sub-rows, within a block. Each sub-row, of BLK elements, is accessed BLK times. Therefore, the A cache should fit a block row at a time.

Algorithm 3 *Block Matrix Multiplication*

```
Require: $A \in \mathbb{R}^{N \times M}, B \in \mathbb{R}^{M \times P}, C \in \mathbb{R}^{N \times P}, BLK \in \mathbb{N}$
Ensure: $C = A \times B$
  procedure BLKMATMULT($A, B, C$)
    for ($jj \leftarrow 0; jj < P; jj \leftarrow jj + BLK$) do
      for ($kk \leftarrow 0; kk < M; kk \leftarrow kk + BLK$) do
        LOOP_I: for ($i \leftarrow 0; i < N; i \leftarrow i + 1$) do
          for ($j \leftarrow jj; j < jj + BLK; j \leftarrow j + 1$) do
            $acc \leftarrow 0$
            LOOP_K: for ($k \leftarrow kk; k < kk + BLK; k \leftarrow k + 1$) do
              $acc \leftarrow acc + A[i][k] \cdot B[k][j]$
            end for
            $C[i][j] \leftarrow C[i][j] + acc$
          end for
        end for
      end for
    end for
  end procedure
```

- B is accessed by sub-columns, within blocks. Each block is accessed N times, therefore its cache should fit one block at a time.

- C has the same access pattern as A, but its cache requires up to BLK ways to provide non-zero hit ratio when the partial unrolling (discussed later) is applied to the innermost loop.

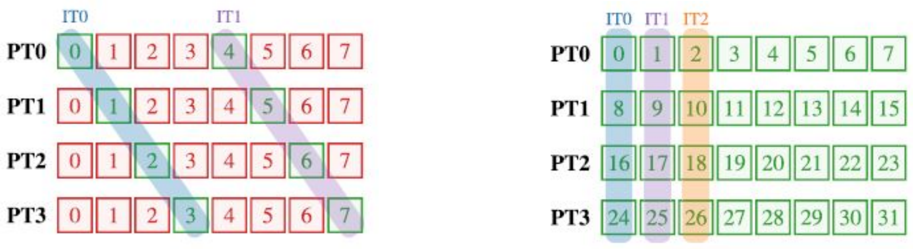
In all implementations, the algorithm innermost loop (`LOOP_K`) was pipelined with $II = 1$. The implementation was further optimized through loop unrolling by a factor UF.

For *StdMatMult*, we considered two kinds of unrolling:

- *Horizontal*: unrolls the innermost loop (`LOOP_K`). To keep $II = 1$ for `LOOP_K`, each iteration of the unrolled loop is assigned to one of the UF A and B cache ports.

  Figure 2.7a highlights a fundamental limitation of this unrolling approach when combined with multi-port caches. The data is replicated in each cache, but only one every UF elements is actually used, thus leading to significant resource and performance waste.

- *Tiled*: divides the `LOOP_I` iteration count by UF and adds a fully unrolled inner loop [15]. All iterations of that new loop use the same element of B and a different one of A. Therefore, the B cache is single-port, while the A cache has UF ports. With this approach, each A port contains different data (Fig. 2.7b).



(a) *Horizontal* unrolling. (b) *Tiled* unrolling.

Fig. 2.7 *MatMult*: content of L1 caches of A during the first iterations, where $A \in \mathbb{R}^{4\times 8}$ is associated with a four-port single-line cache with eight words. *PTn* identifies the n-th port. The green boxes represent elements that read during execution, red boxes are elements loaded in cache but never accessed. The numbers inside the boxes are the addresses of the elements of the A matrix. *ITi* highlight the elements accessed in parallel at the i-th iteration.

The hit ratio is preserved as the unrolling factor scales up and no resources are wasted. All the elements loaded into the cache are actually used, allowing the algorithm to run at full speed for as many iterations as the words per cache line, significantly improving the performance with the same resource usage as *Horizontal*.

In *BlkMatMult* we exploited the *Tiled* unrolling only, for similar reasons to *Tiled StdMatMult*. To maximize the performance, we doubled *UF* until we used all the resources of our (small) FPGA.

All the *MatMult* tests use the same matrix sizes, $N = P = 1024$, $M = 128$, and data type of 32-bit integers.

Table 2.1 shows the cache configurations tested with *StdMatMult*, while Table 2.2 summarizes the *BlkMatMult* ones. We tested block sizes of 16, 32, and 64.

As a reference, we implemented the *Baseline* test case. The unrolling, applied to the *Baseline* test case, would be detrimental, since the II of `LOOP_K` would dramatically increase due to structural dependencies on the AXI interface (which exposes one port only), resulting in performance degradation. Therefore, our cache enabled us to conveniently unroll the algorithm loop, without any change to the algorithm itself.

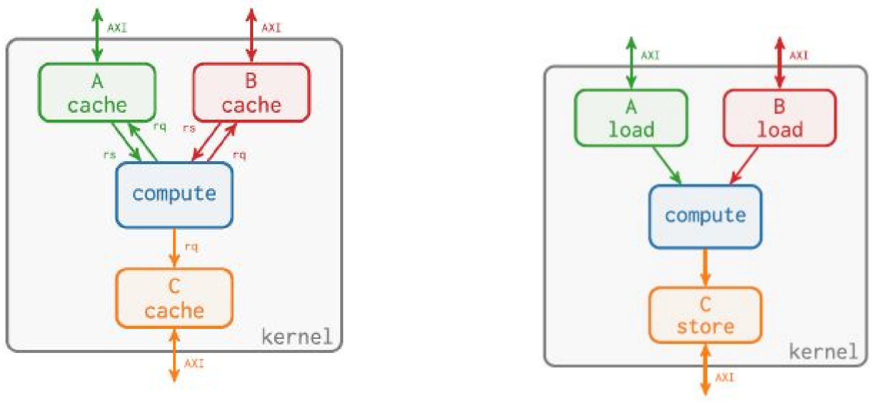
The *Manual* test case optimizes the design according to the LCS pattern. All the off-chip memory accesses use the maximum AXI interface bitwidth of the board (128 bits, or four 32-bit elements per transaction). The B load task reads the B matrix

Table 2.1 *StdMatMult*: tested cache configurations.

Implementation	**Array**	**Words**	**L2 sets**	**L2 ways**	**L1 sets**	**L1 ways**
Single-level (L2)	A	$M/2$	2	1	0	0
	B	32	M	1	0	0
		64	M	1	0	0
	C	32	1	1	0	0
		64	1	1	0	0
Horizontal (L1)	A	$M/2$	1	1	2	1
	B	32	1	1	M/UF	1
		64	1	1	M/UF	1
	C	32	1	1	0	0
		64	1	1	0	0
Tiled (L1tld)	A	$M/2$	1	1	2	1
	B	32	1	1	M	1
		64	1	1	M	1
	C	32	1	UF	0	0
		64	1	UF	0	0

Table 2.2 *BlkMatMult*: tested cache configurations.

Implementation	**Array**	**Words**	**L2 sets**	**L2 ways**	**L1 sets**	**L1 ways**
Single-level (L2blk)	A	BLK	1	1	0	0
	B	BLK	1	BLK	0	0
	C	BLK	1	BLK	0	0
Multi-level (L1blk)	A	BLK	1	1	1	1
	B	BLK	1	1	1	BLK
	C	BLK	1	BLK	0	0



(a) Our cache. (b) LCS.

Fig. 2.8 *MatMult*: tested dataflow architectures.

once, four columns at a time. The *A* load task reads the *A* matrix multiple times, in bursts. The compute task computes 16 multiply-acccumulate (MAC) operations per CC. The store task stores four elements of *C* at a time.

Figure 2.8 compares the dataflow architecture generated with our caches with the LCS one. The similarity between the two architectures is very strong: the only major difference is the absence of the request FIFO from the compute to the load tasks, in case of the LCS architecture. This is because the input data address computation must be factored out of the compute task and moved into the load and store tasks to implement the LCS paradigm. This refactoring is the major design cost that our cache alleviates.

For the *RTLcache*, due to the limited configuration options of the Xilinx System Cache (it provides only two or four ways, and it does not support our custom address bit mapping), the best performing configuration in that case is the *BlkMatMult* algorithm, with block size equal to four.

The cache configurations selected for the test cases reach high hit ratios, above 96 % for *StdMatMult* and 99 % for *BlkMatMult*. Figure 2.9 shows the performance gain, i.e., $t_{\mathrm{ex},\mathit{Baseline}}/t_{\mathrm{ex}}$, with respect to the area cost, i.e., $(\mathrm{LUT}/\mathrm{LUT}_{\mathit{Baseline}} + \mathrm{FF}/\mathrm{FF}_{\mathit{Baseline}} + \mathrm{BRAM}/\mathrm{BRAM}_{\mathit{Baseline}} + \mathrm{DSP}/\mathrm{DSP}_{\mathit{Baseline}})/4$, of the test cases embedding our caches. Most of the points are in the "green" area, where t_{ex} speedup is larger than the resource overhead.

Figure 2.10 shows the detailed data for some significant test cases, including (a) the reference test cases, i.e., *Baseline* and *Manual*, (b) the least resource-demand-

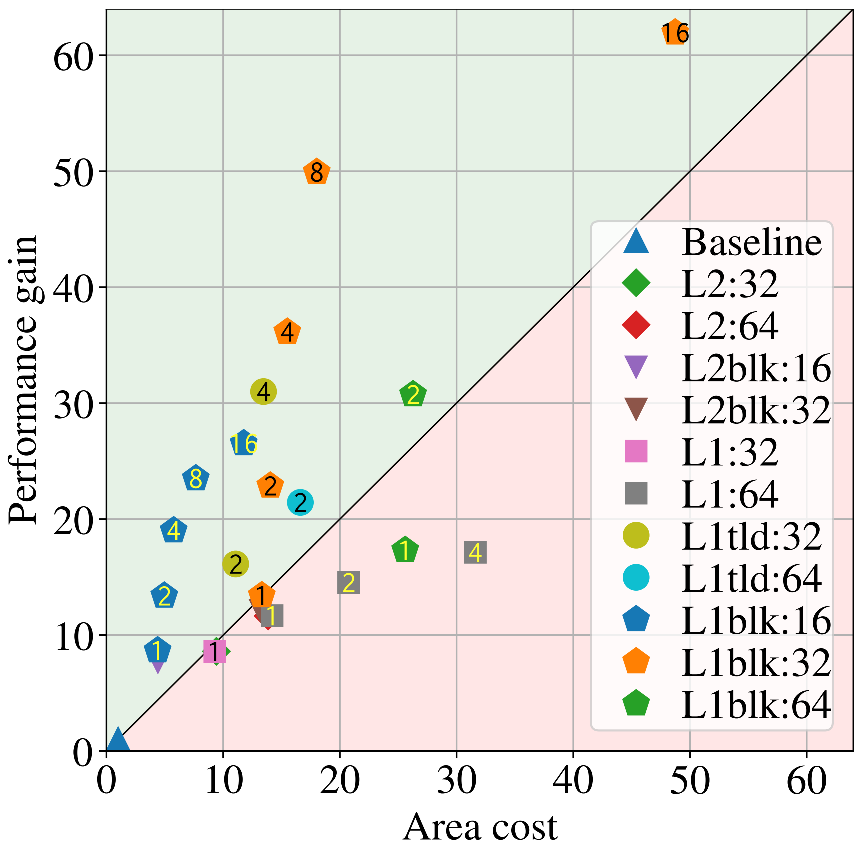
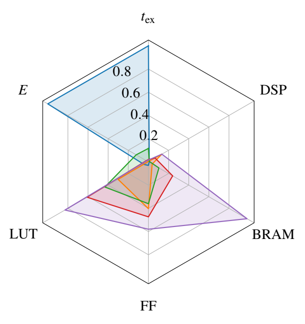


Fig. 2.9 *MatMult*: performance gain (t_{ex} relative to *Baseline*) with respect to area cost (average of LUTs, FFs, BRAMs, and DSPs usage relative to *Baseline*). *StdMatMult Single-level* is labelled *L2:WORDS*, *Horizontal* is *L1:WORDS*, and *Tiled* is *L1tld:WORDS* (*WORDS* are the number of words per line of *B* and *C* caches). *BlkMatMult Single-level* is labelled *L2blk:BLK*, and *Multi-level* is *L1blk:BLK* (*BLK* are the block sizes). The numbers over the markers are the unrolling factors.



	Baseline	**Manual**	**L2:32**	**L1blk:32 (8 ports)**	**L1blk:32 (16 ports)**
t_{ex} (s)	31.98	0.13	3.72	0.64	0.52
P (W)	4.35	4.83	4.56	4.67	4.68
E (J)	139.3	0.62	16.96	2.99	2.41
LUT	3104	21259	30534	42954	58515
FF	4292	56905	50866	66810	81863
BRAM	1.5	8	22.5	52	211.5
DSP	3	48	3	24	48
Perf. gain	1.0	246.0	8.6	50.0	62.7
Area cost	1.0	10.4	9.4	18.0	48.7

Fig. 2.10 *MatMult*: PPA of some significant test cases. t_{ex} and *E* are relative to the *Baseline*. The resource usages are relative to the total resources provided by the target FPGA.

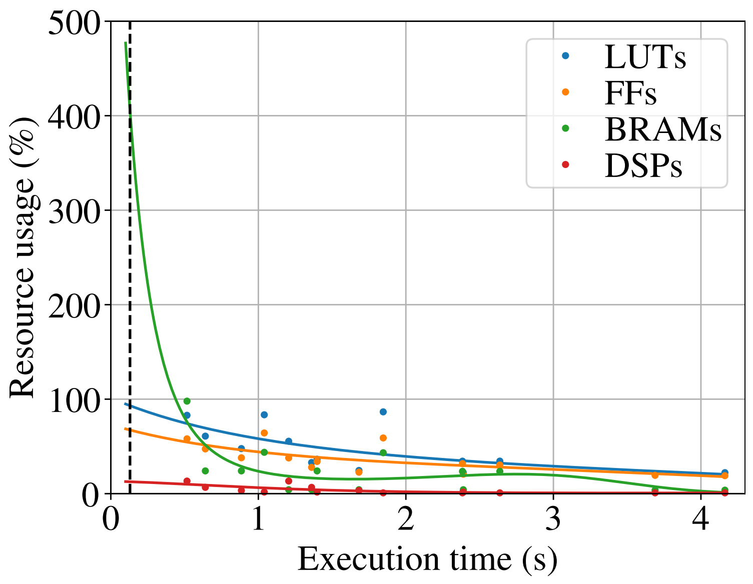


Fig. 2.11 *BlkMatMult*: regression estimating the resource usage with respect to the execution time of the test cases with our caches. The dashed vertical line highlights the execution time of the *Manual* test case. The dots are the real data, the lines are the regression predictions.

ing cache configuration with the *StdMatMult* algorithm, i.e., *L2:32*, (c) the most convenient cache configuration in terms of performance gain and area cost ratio, i.e., *L1blk:32 (8 ports)*, and (d) the fastest cache configuration, i.e., *L1blk:32 (16 ports)*. Compared with the test cases with caches, the *Manual* design provides better overall QoR. The *Manual* solution is a tailored solution not only to optimize memory accesses but also to attain higher parallelism. This requires extensive rewriting of the algorithmic code which is time-consuming. In the case of cache, the algorithm is not modified but only the loops are unrolled. The cache can be combined with other optimizations to achieve manual-like performance. In case of irregular memory accesses, caches are the solution as demonstrated in Section 2.5.4. However, the aim of our work is not to achieve better PPA than manual optimizations, but rather to get significantly better QoR (with respect to the *Baseline*), while greatly reducing the design effort.

Note that increasing the number of ports of the caches, and hence their resources, uniformly increases performance. Figure 2.11 shows the results of using regression to predict the resource usage to achieve a given execution time with our cache. According to this model, to achieve performance on par with the *Manual* reference

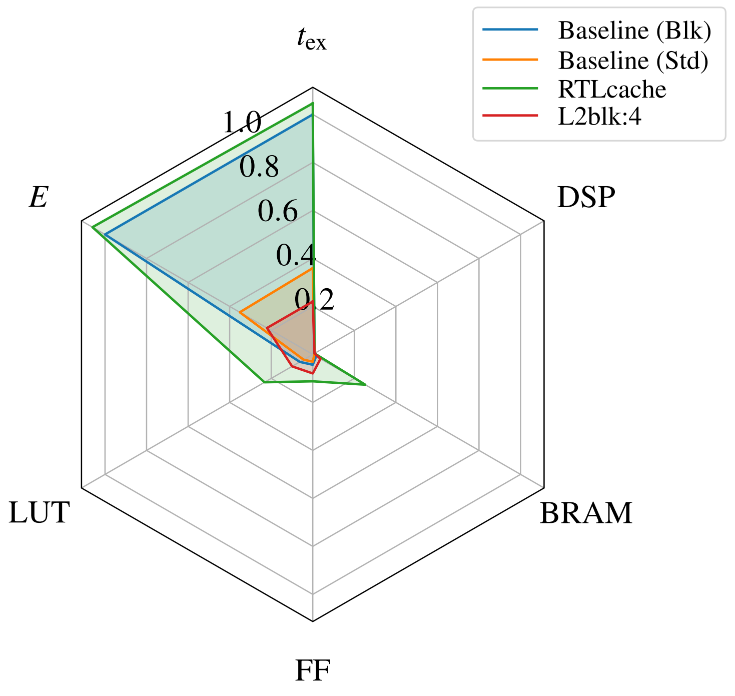


Fig. 2.12 *MatMult*: PPA of some test cases related to the *RTLcache* case. t_{ex} and E are relative to the *Baseline (Blk)*.

design, our caches would require 4 times the available BRAMs, while the other kinds of resources would be sufficient.

Matrix Multiplication RTLcache test case

The Xilinx System Cache supports only two or four ways. Therefore, the theoretically most performant setup is with *BlkMatMult* with block size four (which is still too small to provide large performance gains). The caches associated with A and C should be single-line, while the cache associated with B should provide four ways, each of four words. However, the Xilinx System Cache minimum size is 32 kB, with at least two ways and 64 B per line, therefore the caches of the *RTLcache* test case are dramatically oversized. On the contrary, the test case with our cache (*L2blk:4*), thanks to its fine-grained configurability, was set up to allocate only the resources that are actually needed. Figure 2.12 summarizes the results of these tests. For reference, besides the usual *Baseline (Std)* test case, that implements an unoptimized version of *StdMatMult* algorithm, we also included the *Baseline (Blk)* test case, which implements the unoptimized *BlkMatMult* algorithm with block size four. We included it to quantify the impact of the Xilinx System Cache on the very same kernel, directly connected to the AXI interface.

Both the *Baseline (Blk)* and the *RTLcache* designs are significantly slower than the *Baseline (Std)*. For the *Baseline (Blk)*, this is because the *BlkMatMult* is not meant for running without a cache. For the *RTLcache*, this is because the RTL cache module is inserted a posteriori (after HLS), thus the kernel is synthesized assuming that all the memory references access the off-chip memory. Therefore, it is scheduled to wait for the expected latency of the AXI master controller that is used to access DRAM, which has a minimum latency, hardcoded into the HLS scheduler, of *at least* 7 CCs. Thus for cache hits it waits for much longer than needed, while for misses it waits for shorter than needed (the cache introduces an additional latency when missing), and then it stalls until the memory request is fulfilled. On the other hand, our dataflow protocol hides from the computation process schedule the fact that it is accessing DRAM, thus allowing it to achieve the best throughput in case of cache hits.

The result is that the RTL cache is not only unable to provide any advantage, but it also slightly worsens the performance and the energy consumption. Moreover, it also introduces a large area overhead, due to the oversized caches.

The *L2blk:4* test case is significantly faster than the *Baseline (Blk)*, proving the effectiveness of our HLS cache implementation with respect to the System Cache. However, it is not much faster than the *Baseline (Std)*, since the small block size limits the performance advantage.

Matrix Multiplication timing analysis

To evaluate the impact of our cache on the critical path, we measured the maximum clock frequency of some test cases. Table 2.3 reports the results of the experiments. The *Baseline* design is very simple: it consists of a loop that computes a MAC operation per iteration using a DSP (which is one of the fastest resources on the FPGA). Therefore, it is able to achieve the maximum clock frequency of 330 MHz. With the *StdMatMult*, all the instantiated caches are direct-mapped (including the *B* one, thanks to our custom address bit mapping). The resulting design can still run at 330 MHz, even in the *Multi-level* configuration. The *BlkMatMult* test cases require 32-way fully-associative caches. The high number of ways makes these caches inher-

Table 2.3 *MatMult*: maximum achievable clock frequency of some test cases. The relative maximum clock frequency is normalized over the maximum clock frequency of the AXI adapter (330 MHz).

Test case	**Maximum clock frequency** (MHz)	**Relative maximum clock frequency** (%)
Baseline	330	100
Manual	330	100
L2:32	330	100
L1:32	330	100
L2blk:32	260	79
L1blk:32 (1 p)	250	76
L1blk:32 (16 p)	150	45

Table 2.4 *MatMult*: Performance achieved for some test cases.

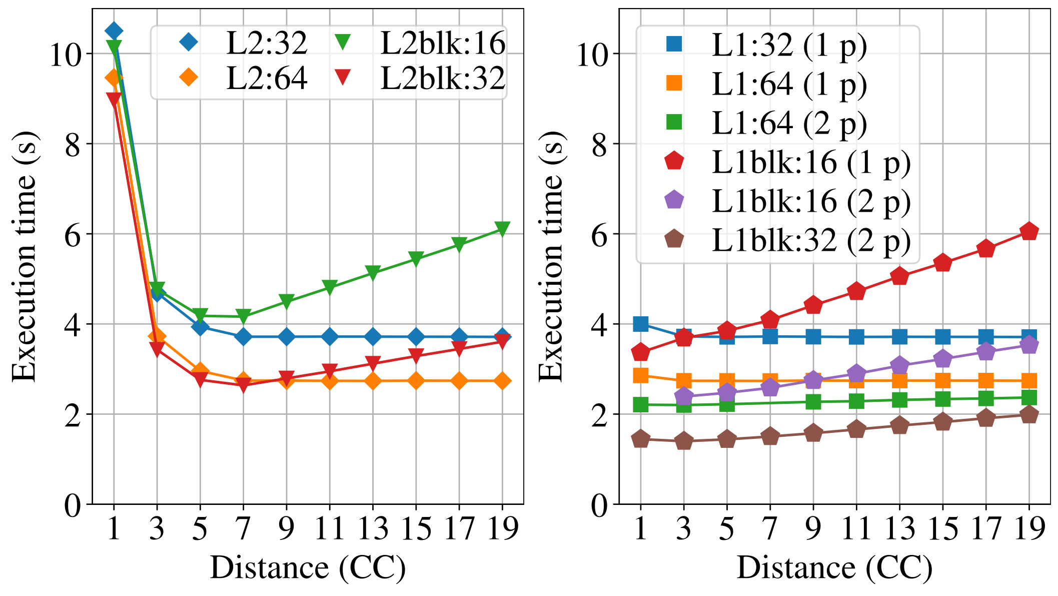
Test case	**Maximum clock frequency** (MHz)	t_{ex} (s)
L2:32	330	1.95
L1:32	330	1.94
L2blk:32	260	1.13
L1blk:32 (1 p)	250	1.05
L1blk:32 (16 p)	150	0.36

ently more complex than the direct-mapped ones, therefore they introduce a critical path which limits the maximum clock frequency. The *Single-level* configuration can run at a clock frequency up to 260 MHz. For the *Multi-level* configurations, the single-port test case can reach a clock frequency of 250 MHz. The extreme case with 16 ports can only reach a maximum clock frequency of 150 MHz. This is not only due to the more complex cache architecture, but also because of the algorithm unrolling, and the high resource utilization.

The performance i.e. execution time, achieved for different cache test cases at maximum clock frequencies achieved is shown in Table 2.4. On the other hand, due to memory bandwidth limitation, the performance remains the same even at higher clock frequencies for both Baseline and Manual.

Matrix Multiplication request-response distance

To check the efficiency of the approximations for the default L2 cache request-response distance of RO cache configurations, we characterized the t_{ex} with respect to the distance in some test cases. For the *Single-level* configurations, Fig. 2.13a shows that in all test cases a distance of 1 CC results in a very high t_{ex} since it prevents



(a) *Single-level MatMult*.

(b) *Multi-level MatMult*.

Fig. 2.13 *MatMult*: execution time with respect to L2 cache request-response distance.

exploiting the cache pipelining, as discussed in Section 2.2. The t_{ex} significantly decreases with the distance up to 5 CCs to 7 CCs. For higher distances, the t_{ex} of the *StdMatMult* test cases is approximately constant, while the one for *BlkMatMult* increases again. These results suggest that our choice of a default distance of 7 CCs is effective.

For the *Multi-level* configurations, Fig. 2.13b shows that the t_{ex} of *StdMatMult* is roughly constant with the distance, apart from the distance of 1 CC which is slightly slower. The *BlkMatMult* t_{ex} is instead directly proportional (by a small factor) to the distance. Any distance value between 1 3CCs should be a balanced choice. Our default value of 3 CCs is therefore well suited.

2.5.3 2D Convolution

Algorithm 4 implements the *2D Convolution* (*Conv2D*). Each matrix is characterized by a specific memory access pattern.

- A is accessed according to a window pattern with size $P \times Q$ and stride one.

Algorithm 4 *2D convolution*

```
Require: $A \in \mathbb{R}^{N\times M}, ker \in \mathbb{R}^{P\times Q}$
Ensure: $B \in \mathbb{R}^{N\times M} : B = A * ker$
  procedure CONV($ker, A, B$)
    for $(i \leftarrow 0; i < N; i \leftarrow i+1)$ do
      for $(j \leftarrow 0; j < M; j \leftarrow j+1)$ do
        $tmp \leftarrow 0$
        LOOP_M: for $(m \leftarrow 0; m < P; m \leftarrow m+1)$ do
          LOOP_N: for $(n \leftarrow 0; n < Q; n \leftarrow n+1)$ do
            $ii \leftarrow i + m - Q/2$
            $jj \leftarrow j + n - P/2$
            if $(ii \geq 0 \ \& \ ii < N \ \& \ jj \geq 0 \ \& \ jj < M)$ then
              $tmp \leftarrow tmp + A[ii][jj] \cdot ker[m][n]$
            end if
          end for
        end for
        $B[i][j] \leftarrow tmp$
      end for
    end for
  end procedure
```

A cache associated with A requires P ways to achieve a high hit ratio, since all the lines belonging to a window can be stored in the cache, *effectively implementing a line buffer without source code changes*.

Cache lines sizes of $n \times Q$ enable prefetching n windows.

To keep in cache windows which are not aligned to the cache line size, the cache should have two sets.

- *ker* is accessed $N \times M$ times, by rows. Since its size is typically small, its cache can be configured to fit the whole *ker* in the L1 cache.
- B is sequentially accessed once per element. B has a low impact on performance, since it is accessed only once every $P \times Q$ accesses to A and *ker*. A single-line cache helps HLS to efficiently infer bursts.

All test cases use the same 8-bit integer data type and matrix sizes: $N = 1080$, $M = 1920$, $P = Q = 15$. In all implementations, the innermost loop (`LOOP_N`) was pipelined with $II = 1\,\text{CC}$.

In the tests including our cache, each matrix was associated with a cache configured according to the previous considerations. Table 2.5 summarizes the tested cache configurations, where n is 1, 2, 4, 8, and 16. Since our cache only supports power-of-2 words, ways, and sets, all the parameters were rounded to the next power of 2.

Table 2.5 *Conv2D*: tested cache configurations.

Implementation	Array	Words	L2 sets	L2 ways	L1 sets	L1 ways
Single-level (L2)	*A*	$n \cdot Q$	2	*P*	0	0
	ker	*Q*	1	1	*P*	1
	B	32	1	1	0	0
Multi-level (L1)	*A*	$n \cdot Q$	1	1	2	*P*
	ker	*Q*	1	1	*P*	1
	B	32	1	1	0	0

With the multi-level test cases, we further improved the performance exploiting the multi-port feature to enable partial loop unrolling while keeping the II of LOOP_K at one (by setting the number of ports of *A* and *ker* as the unrolling factor). We unrolled LOOP_M, instead of the innermost LOOP_N, for reasons similar to those explained in Section 2.5.2, and shown in Fig. 2.7. We tested unrolling factors of 3, 5, 8, and 15 (complete unroll).

The *Manual* reference design was implemented by Xilinx Inc. [58], according to the LCS pattern. It is not possible to implement a meaningful *RTLcache* test case, since the Xilinx System Cache can only provide up to 4 ways, but the *A* cache requires at least 15 ways to achieve a sufficiently high hit ratio.

All tested cache configurations had hit ratios higher than 99 %. Figure 2.14 shows the trade-offs between performance and area, in different test cases. Figure 2.15 shows the details of some relevant test cases, including (a) the reference designs, i.e., *Baseline* and *Manual*, (b) the least resource-demanding cache configuration, i.e., *L2:16*, (c) a cache configuration balanced between performance and resources, i.e., *L1:64 (5 ports)*, and (d) the fastest cache configuration, i.e., *L1:64, (15 ports).*

Our caches introduce multiple trade-offs in the PPA space, which perform better than the *Baseline* case, in exchange for higher resource usage. The *Manual* design is significantly faster than all the tested cache configurations, since it is able to process a whole window per CC (255 MAC operations per CC), while our cache configurations process at most one window column per CC (15 MAC operations).

Figure 2.16 again shows the results of using regression to predict the resource usage to achieve a given execution time with our cache. According to the regression prediction, to achieve performance on par with the *Manual* reference design, our

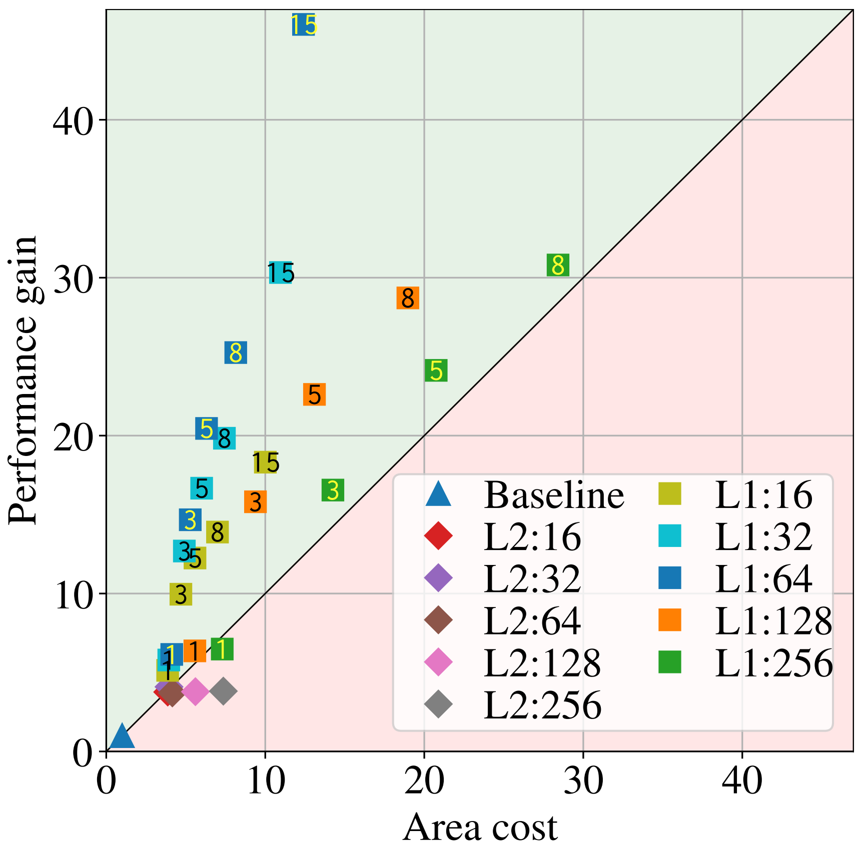
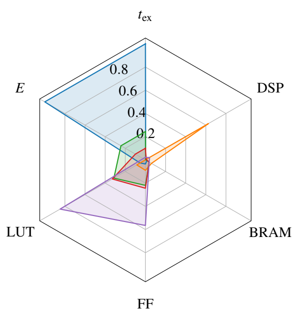


Fig. 2.14 *Conv2D*: performance gain with respect to area cost. *Single-level Cache* is labelled as *L2:WORDS*, and *Multi-level* as *L1:WORDS*. The *WORDS* suffix stands for the number of words per line of the *A* cache.



	Baseline	**Manual**	**L2:16**	**L1:64 (5 ports)**	**L1:64 (15 ports)**
t_{ex} (s)	32.69	0.03	8.69	1.60	0.71
P (W)	4.59	4.54	4.55	4.64	4.82
E (J)	150.0	0.1	39.5	7.4	3.4
LUT	3766	6082	21602	30880	59828
FF	4962	12670	30068	46458	79717
BRAM	3	13	8	8	8
DSP	1	225	1	5	15
Perf. gain	1.0	1089.7	3.8	20.4	46.0
Area cost	1.0	65.9	3.9	6.3	12.4

Fig. 2.15 *Conv2D*: PPA of some significant test cases.

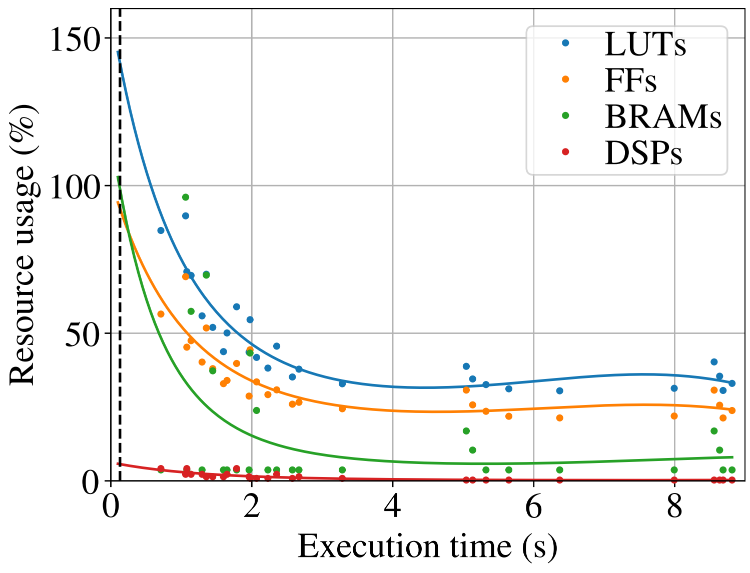


Fig. 2.16 *Conv2D*: regression of resource usage with respect to the execution time of the test cases with our caches.

Table 2.6 *Conv2D*: maximum achievable clock frequency of some test cases.

Test case	**Maximum clock frequency** (MHz)	**Relative maximum clock frequency** (%)
Baseline	330	100
Manual	330	100
L2:16	330	100
L1:16 (1 p)	330	100
L1:64 (15 p)	200	61

caches would require roughly 50 % more LUTs than those available on the target FPGA, while the other kinds of resources should suffice.

Note that the objective of our cache is not to compete with manually optimized designs, but rather to introduce new trade-offs between PPA and design effort. Our caches provided suboptimal results in terms of PPA, but required very low design effort, while being much more efficient than the designs automatically generated by the HLS tool, both in terms of execution time, reduced by up to 46 times, and in terms of energy consumption, reduced by up to 44 times, at the cost of an area overhead up to 12 times.

2D Convolution timing analysis

Table 2.6 reports the maximum clock frequency achieved by some test cases. Simi-

Algorithm 5 *Bitonic sorting*

```
Require: $a \in \mathbb{R}^N : N = 2^n$
Ensure: $a[i] \leq a[j], \forall i \geq j$
  procedure SORT($a$)
    for ($b \leftarrow 1; b < n; b \leftarrow b+1$) do
      for ($s \leftarrow b-1; s \geq 0; s \leftarrow s-1$) do
        LOOP_I: for ($i \leftarrow 0; i < N/2; i \leftarrow i+1$) do
          $dir \leftarrow (i/2^{b-1})\&1$
          $dir \leftarrow dir \wedge 1$
          $step \leftarrow 2^s$
          $pos \leftarrow 2i - (i\&(s-1))$
          $a_0 \leftarrow a[pos]$
          $a_1 \leftarrow a[pos+step]$
          if ($a_0 > a_1 \neq dir$) then
            $tmp \leftarrow a_0$
            $a_0 \leftarrow a_1$
            $a_1 \leftarrow tmp$
          end if
          $a[pos] \leftarrow a_0$
          $a[pos+step] \leftarrow a_1$
        end for
      end for
    end for
  end procedure
```

larly to the *MatMult* case, the *Baseline* design is very simple: it consists of a loop that computes a MAC operation per iteration using a DSP. Therefore, it can run at the higher-bound clock frequency of 330 MHz. Even the single-port test cases (*L2:16* and *L1:16 (1 p)*), despite being characterized by a large amount of cache ways (16), do not introduce any critical path limiting the frequency below the 100 %. Only with the multi-port test case (*L1:64 (15 p)*), which also involves an application loop unrolling by a factor of 15, we face a frequency degradation of 39 %.

2.5.4 Bitonic Sorting

Bitonic sorting (*BitSort*) is a sorting algorithm, whose implementation is shown in Algorithm 5. From the memory access point of view, at each inner loop (`LOOP_I`) iteration: (1) $a[pos]$ is read, (2) $a[pos+step]$ is read, (3) $a[pos]$ is written, and (4) $a[pos+step]$ is written. Therefore, the cache associated with the a array should be set-associative with at least two sets, so that the interleaved accesses to pos and $pos+step$ do not overwrite the related cache lines.

In the designs under test, the inner loop was pipelined, but due to the data dependencies on the a array the pipeline performance is limited. The pipeline of the *Baseline* test case (accessing a directly from DRAM) requires a very high $II = 142$ CCs because it must guarantee the dependency on the slow AXI interface.

Table 2.7 *BitSort*: tested cache configurations.

Implementation	**Words**	**L2 sets**	**L2 ways**	**L1 sets**	**L1 ways**
Single-level (L2)	16	1	2	0	0
	32	1	2	0	0
	64	1	2	0	0
Multi-level (L1)	16	1	2	1	1
	32	1	2	1	1
	64	1	2	1	1

Our cache allows shortening the dependency distance and building a more performant pipeline, with an $II = 6$ CCs. All the tests use the same data type (32-bit integers) and sizes ($N = 2^{20}$). Table 2.7 shows the tested cache configurations.

We were unable to implement a *Manual* design for an optimized reference, since the irregular access pattern, makes the on-chip data buffering challenging, especially considering that the array is accessed both in read and in write mode, introducing data dependencies. We believe that caching is the most convenient solution for optimizing this algorithm.

The *RTLcache* test case inserts the Xilinx System cache between the HLS kernel and the AXI interface. We set the total cache size to 32 kB (the minimum possible), with 2 ways, 64 words per line, and, by consequence, 128 sets.

The selected cache configurations achieve high L2 hit ratios, above 96 %. The L1 hit ratios are instead very low, from 8 % to 24 %, since our L1 caches use the write-through consistency policy.

Figure 2.17 plots the performance gain with respect to the area overhead of each test case with our cache. All the test cases provide significantly more performance gains than area cost. The L1 caches introduce a very limited performance advantage, because of their low hit ratio.

Figure 2.18 reports the full information on (a) the reference designs (*Baseline* and *RTLcache*), (b) the least resource-demanding cache configuration, i.e., *L2:16*, (c) the best cache configuration in terms of performance gain and area cost ratio, i.e., *L2:32*, and (d) the fastest cache configuration, i.e., *L1:64*.

The *RTLcache* is worse than the *Baseline* in all dimensions in the PPA space. This is because the cache module is inserted after HLS, therefore HLS optimizes the circuit as if all memory accesses were off-chip. In particular, the loop pipeline

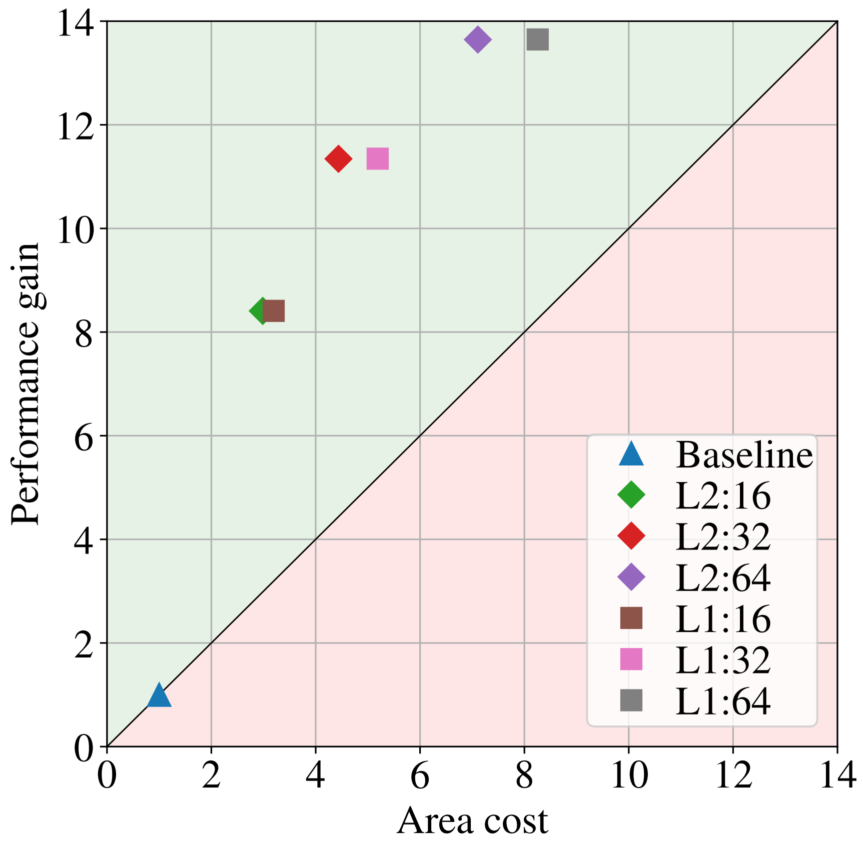
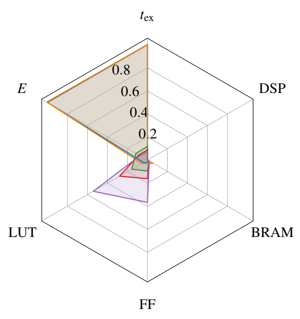


Fig. 2.17 *BitSort*: performance gain with respect to area cost. *Single-level Cache* is labelled as *L2:WORDS*, and *Multi-level* as *L1:WORDS*. The *WORDS* suffix stands for the number of words per line of the *a* cache.



	Baseline	**RTLcache**	**L2:16**	**L2:32**	**L1:64**
t_{ex} (s)	156.6	156.9	18.6	13.8	11.5
P (W)	4.52	4.54	4.39	4.45	4.64
E (J)	707.8	712.3	81.7	61.5	53.3
LUT	2896	4077	11089	19667	38206
FF	3270	4270	13454	22751	51758
BRAM	1	12	4	4	4
DSP	0	0	0	0	0
Perf. gain	1.0	1.0	8.4	11.3	13.6
Area cost	1.0	3.7	3.0	4.4	8.3

Fig. 2.18 *BitSort*: PPA of some significant test cases.

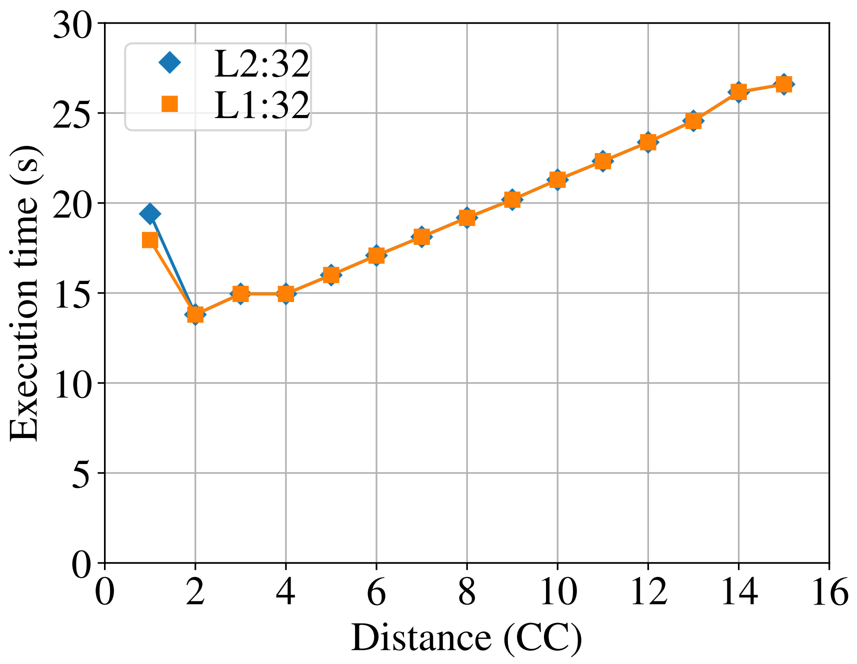


Fig. 2.19 *BitSort*: execution time with respect to L2 cache request-response distance.

is still characterized by a very high II. This is another example showing that it is counterproductive to insert an RTL cache module a posteriori, after HLS. It is only introducing overhead, not only in terms of area, but also in terms of t_{ex} and power.

On the other hand, our cache improves the performance and the energy consumption by one order of magnitude compared to the *Baseline*. The *RTLcache*, despite having 128 sets instead of 1, consumes significantly less LUTs and FFs than our cache. It could be useful to combine the advantages of the source-level implementation with the resource efficiency of the RTL description to achieve the best performance at the lowest area cost. This could be achieved by exploiting the *Vitis HLS* capabilities to embed RTL code within HLS source code.

Bitonic Sorting request-response distance

To evaluate the performance of the default L2 request-response distance for RW cache configurations (2 CCs), we characterized the t_{ex} with respect to distance in a couple of test cases. As Fig. 2.19 shows, we chose the optimal value that balances the L2 cache pipeline exploitation (higher distance values better exploit it) and the algorithm loop II (the distance corresponds to the RAW dependency distance,

Table 2.8 *BitSort*: maximum achievable clock frequency of some test cases. The relative maximum clock frequency is normalized over the maximum clock frequency of the AXI adapter.

Test case	**Maximum clock frequency** (MHz)	**Relative maximum clock frequency** (%)
Baseline	330	100
L2:16	330	100
L1:32	300	91
L1:64	230	70

Table 2.9 *BitSort*: Performance achieved for some test cases.

Test case	**Maximum clock frequency** (MHz)	t_{ex} (s)
Baseline	330	63.4
L2:16	330	10.19
L1:32	300	8.45
L1:64	230	7.62

and, by consequence, to the II). The data points of the multi-level configuration approximately overlap the single-level ones, because the L1 hit ratio is low. In a test case with high L1 hit ratio, the optimal distance value would probably be in 1 CC, since it would not need to exploit the L2 cache, and it could minimize the loop II.

Bitonic Sorting timing analysis

The maximum achieved clock frequencies for some test cases are shown in Table 2.8. Unlike the previous experiments, we encounter a slight maximum frequency degradation even with single-port cache configurations. This is due to the additional logic required for supporting both read and write operation within a single cache, differently from the read-only and write-only caches of *MatMult* and *Conv2D*. The performance i.e. execution time, achieved for these clock frequencies is shown in Table 2.9.

Chapter 3

GNN-based Prediction Model for HLS QoR

Part of the work described in this chapter has been previously published in [26].

Current commercial HLS tools do not provide reliable estimates of the final QoR [9]. As a result, designers are unable to make cost/performance trade-offs and guarantee that the design will meet the requirements because the estimation results in terms of timing and resource usage often significantly differ from the actual QoR after implementation.

In this chapter, we propose a a GNN-based framework to predict the quality of results of an HLS design based on its HLS intermediate-representation (IR). We formulate the QoR prediction problem as a multi-objective regression task to estimate post-implementation resource usage and timing without invoking the back-end of the HLS tool.

Fig. 3.1 shows our overall framework flow, and also shows the relationship between the general HLS-based hardware design workflow and our proposed predictive framework for estimating the QoR of an HLS-based design. Our proposed predictive framework is shown on the right side of the flowchart, showing both training and inference flows. In both phases, the IR-based graph is used as input data. During the training process, in addition to the input data, the corresponding ground truth (i.e., the actual number of resources and clock period) is also required, which is extracted from the post-implementation report. On the other hand, the inference process uses the trained model to predict the QoR of a new HLS design based on its IR.

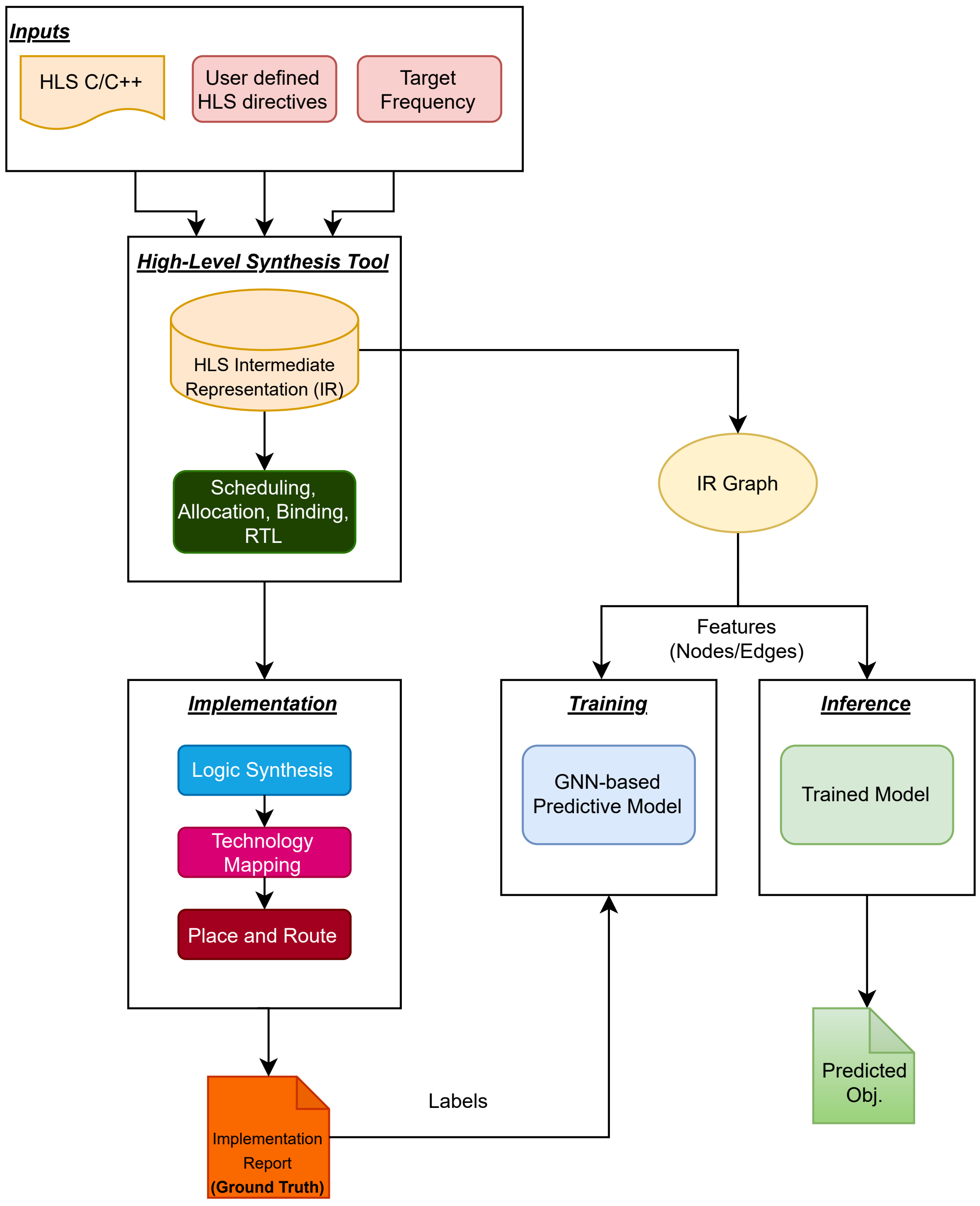

Inputs
HLS C/C++
User defined HLS directives
Target Frequency
High-Level Synthesis Tool
HLS Intermediate Representation (IR)
Scheduling, Allocation, Binding, RTL
IR Graph
Features (Nodes/Edges)
Implementation
Logic Synthesis
Technology Mapping
Place and Route
Training
GNN-based Predictive Model
Inference
Trained Model
Predicted Obj.
Labels
Implementation Report (Ground Truth)

Fig. 3.1 Overall framework flow and the relationship between the general HLS-based hardware design work flow (left-hand side) and our proposed framework (right-hand side) for estimating quality of results (QoR) of an HLS-based design.

3.1 Background

3.1.1 Low Level Virtual Machine

Low level virtual machine (LLVM)is a collection of compiler and tool-chain technologies that facilitate the creation of front-end and back-end components for programming languages across different Instruction Set Architecture (ISA) [2]. The primary component of the system is a language-independent Intermediate-representation (IR), which functions as a portable high-level assembly language and can be enhanced to generate a novel IR. The novel IR can subsequently undergo translation and integration into platform-specific assembly language code that is contingent upon the characteristics of the machine.

LLVM is capable of providing the intermediate layers of a compiler system by accepting an IR code from a compiler and producing an optimized IR. The instructions are presented in the Static single assignment form (SSA), which encompasses a language-independent instruction set system. This helps in the examination of the inter-relationships among different variables as each variable (also referred to as a register) is allocated only once and subsequently rendered immutable.

LLVM Intermediate Representation

The LLVM infrastructure relies on a language-independent intermediate-representation which is as an essential component. The LLVM IR is a low-level code representation and is specifically intended for utilisation within the LLVM compiler framework. The framework designed to be independent of any particular programming language, allowing it to effectively represent code written in different programming languages. The main benefit of the LLVM acIR is its ability to be extracted as a graph, which can be represented in a number of structures, including a Data Flow Graph (DFG) or Control Data Flow Graph (CDFG). A Control Data Flow Graph (CDFG) represents both data and control dependencies and a Data Flow Graph (DFG) only shows the data dependencies that exist between different instructions in the program. The utilisation of graphs in performance optimization and analysis of the program proved to be highly beneficial as they offer invaluable insights into program behaviour.

LLVM IR is commonly encoded in *.bc* or *.ll* formats. The *.bc* is a binary format that shows greater efficiency in terms of storage and processing capabilities when handling substantial volumes of code. Instaed, The *.ll* is a human-readable format where the program is represented in assembly like instructions which are easier to be understandable from the programmer point of view. To converting LLVM bit code (.bc files) into LLVM human-readable bitcode (.ll files), LLVM Disassembler is used which is one of the integral component of the LLVM compiler infrastructure.

3.1.2 Design as Graph

Most major compiler tools use graphs for optimization and transformation, and the first step in their compilation process is always to transform the input program into a *IR* graph.

Modern HLS tools are designed and based on state-of-the-art compilers such as LLVM [32] and GCC [18]. The input is a *high-level code* that represents the functionality of the design and can be written in programming languages such as C/C++. The input to the front end of the HLS tool undergoes several IR transformations, and the final output *IR* is geared towards hardware circuit generation, for example by using bit-width-accurate operations such as 7-bit additions. The *HLS IR* consists of *basic blocks*, where each block contains assembler-like instructions with no incoming or outgoing branches except at the beginning and end of the block. The HLS IR can be represented as different types of graphs such as *control flow*, *data flow*, and *call-flow* for specific information extraction.

In the control flow graph, the nodes correspond to the LLVM instructions of the program, and the edges represent how the instructions are sequenced, including conditional branches. The data flow graph contains the same nodes, while the edges represent values flowing between instructions, from results to input operands, using the SSA form. Finally, the call graph represents the calling relationship between sub-functions, starting from the top function being synthesized. *We considered a combination of these three graphs for our experiments.*

3.1.3 Embedding Layer

Embedding is a technique used for encoding categorical data in Machine Learning. Category representation in embedding is in the form of dense vectors of continuous values, as opposed to one-hot encoding, which uses high-dimensional, sparse vectors to represent categories. The idea behind embedding is to convert categorical data into a continuous space and learn a mapping lookup table where similarity is captured by representing similar categories by similar vectors in that space. In PyTorch, one can use embedding layer [46] to implement it. Embedding layers are not limited to representing just text or categorical variables; they can also be used to represent numerical values. Embedding is superior to one-hot encoding in several ways. First, it leads to low-dimensional dense vectors, which are more computationally efficient to handle. Second, embedding captures the underlying structure and relationships between categories, which can improve the ability of the model to generalize to previously unseen categories.

3.2 Graph Neural Network

Graphs are a type of data structure that helps represent complex information explicitly by establishing relationships between objects. Recently, there has been a significant surge of interest in using deep neural networks, also graph neural networks [55], to analyze graph-structured data. Graph neural network are a type of Neural Networks (NNs) that are designed to handle and process data that is structured in the form of a graph. Associating feature vectors (also known as node embeddings) with graph nodes gives GNNs the ability to capture both structural and contextual information.

Graphs are a form of data structure that facilitates us to represent complex information by establishing connections between objects. An edge-weighted graph can be represented as $\mathscr{G}(V,E,A)$, where V and E are the set of vertices and edges, respectively, and $A \in \mathbb{R}^{n\times n}$ is an adjacency matrix where A_{ij} is the weight of the directed edge from vertex i to vertex j, or 0 if no such edge exists. An undirected graph can be represented by a symmetric adjacency matrix, and an unweighted graph can be represented by a matrix where all entries are either 0 or 1. Additionally, a graph can be associated with a matrix of *node features* $X \in \mathbb{R}^{n\times d}$, where the feature vector (with d elements) of the i-th node is the i-th row.

Architecture

A GNN is a model that learns a trainable nonlinear mapping function F such that $H = F(A, X)$, where $H \in \mathbb{R}^{n \times k}$ is the output feature matrix. A GNN model iteratively computes a *sequence* of feature matrices from the input node feature matrix through a series of cascading layers. All layers have exactly the same graph structure, but their feature vectors can have different sizes. An aggregate (*AGG*) and an update (*UPDATE*) functions are applied to each node in each layer. The *AGG* function receives the feature vector information from the neighboring nodes of the ith node in the tth layer and sends the aggregated information to the *UPDATE* function to update the ith node features in the $(t+1)$th layer based on the aggregated value and possibly the ith node feature value in the tth layer. Note that this may change the size of the feature vector, while the graph remains the same across all layers. In addition, some classes of GNNs, such as dynamic graph attention network (GAT), also use edge features in the aggregation process. A *READOUT* function, such as mean or max pooling, is applied after the last layer to summarize the features from all nodes and produce a *single graph-level feature vector*, which is typically used as input to the multi-layer perceptron (MLP) that produces the final GNN output(s).

For example, a convolutional neural network where all layers have exactly the same "image" size (along the x and y directions) can be seen as a special case of GNN, where (1) each node in the graph (except the "boundary" nodes, which have smaller neighborhoods) has a set of neighbors (i.e., nodes connected to it by edges) of the same size and shape as the convolution filter, (2) the number of channels in each layer is the number of elements in the node feature vector, (3) the *AGG* function computes a convolution (and pooling) operation over the neighbors feature vectors, (4) the *UPDATE* function is a ReLU, and (5) the *READOUT* function concatenates the features of the last layer.

The message passing based architecture [17] of a generic GNN model can be summarized as follows

$$m_i^{t+1} = \mathrm{AGG}\left(h_u^t \mid u \in \mathcal{N}(i) \cup \{i\}\right) \tag{3.1}$$

$$h_i^{t+1} = \mathrm{UPDATE}\left(m_i^{t+1}\right) \tag{3.2}$$

$$h_{\mathscr{G}} = \mathrm{READOUT}\left(h_i^T, i \in V\right) \tag{3.3}$$

where the feature vector for node i at layer (also called iteration) t is denoted by h_i^t, $N(i)$ represents the neighborhood of node i, i.e. the set of nodes with an edge to i, where m_i^{t+1} represents the aggregated message from neighbors, V is the node set of the graph $\mathscr{G}$, T is the number of layers, i.e. message-passing iterations, of the GNN, and $h_{\mathscr{G}}$ is the final graph-level feature vector sent to the output MLP.

3.2.1 Graph Neural Network Models Variants

In general, different GNN models differ from each other based on different aggregate and update functions [63]. Any permutation-invariant operation can serve as an *AGG* function, and any differentiable function can be used as an *UPDATE* function. Fig. 3.2 shows a general information flow of GNN model. *In our experimentation, we have used 4 different GNNs, namely: graph convolutional network [30], dynamic graph attention network [6], graph isomorphism network [61], and deep adaptive graph neural network [34]. In this work, we refer to these models as GCN, GAT, GIN and DAGNN respectively.*

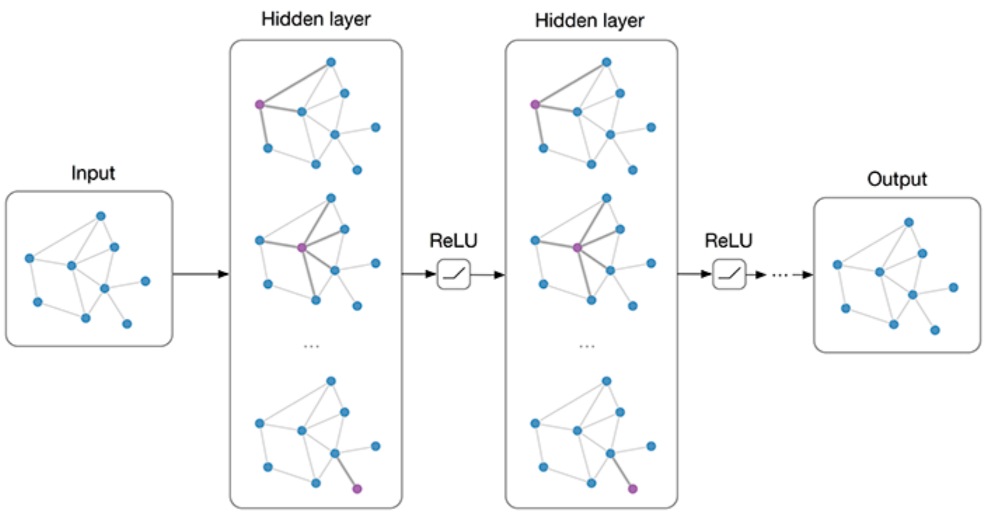


Fig. 3.2 A general graph neural network model [29]

3.2.2 Graph Convolutional Network

Graph convolutional network (GCN)[30] is a specialized adaptation of the Convolutional Neural Network (CNN), tailored for processing graph data.. Essentially, GCN extends the idea of convolution, which is commonly applied to images for different tasks like image classification and object detection. To achieve this, GCN uses a localized graph convolution operation where each node aggregates information from its neighbors by taking a weighted sum of their feature vectors. It involves message passing between neighboring nodes in a graph.

In GCN, an aggregation function is a simple mean or sum aggregation. A GCN architecture can be composed of multiple layers, each of which conducts a graph convolution operation followed by a non-linear activation function. It is a relatively simple model and computationally efficient compared to more complex GNN models. On the other hand, it has limited expressiveness which can limit its ability to capture complex relationships in the underlying data.

3.2.3 Dynamic Graph Attention Network

Dynamic graph attention network (GAT) [6] aims to overcome the limitations of GCN, which assigns equal weights to each neighboring node's. GAT employs an attention mechanism to learn the importance of node neighbors, which enables it to assign different weights to nodes in the neighborhood according to their contributions, making it more expressive than GCN.

GAT introduces a dynamic attention mechanism that only considers the transformed embedding of the target node to compute attention coefficients. It also involves message passing like GCN. Due to the attention mechanism, it is computationally more expensive.

To improve the expressive power of the model, GAT typically employs multiple attention heads in parallel, allowing the model to capture different aspects of the graph structure simultaneously.

3.2.4 Graph Isomorphism Network

Graph isomorphism network (GIN)[61] is a powerful GNN variant with significant expressive capacity. It is designed to capture graph isomorphism making it suitable for tasks where distinguishing between similar graphs with different structures is important. Graph isomorphism is a problem of determining whether two graphs are structurally identical.

The universal approximation theorem, which allows for arbitrary precision approximations of any permutation-invariant function when given sufficient depth, is the foundation for the expressive capability of GIN. In order to provide GIN the capacity to distinguish between various graphs, it uses MLPs as its aggregation functions.

3.2.5 Deep Adaptive Graph Neural Network

Deep adaptive graph neural network (DAGNN)[34] addresses the over-smoothing problems brought on by the deeper GNN model by introducing an adaptive adjustment mechanism.

Although neighborhood aggregation is performed via graph convolutions, one layer of these neighborhood aggregation methods only takes into account immediate neighbors, which results in limited receptive fields. To obtain larger receptive fields, the depth of the graph convolution layer needs to be increased. However, performance suffers as one goes deeper. The cause of this performance degradation is the over-smoothing problem, in which individual nodes' embedding after aggregation and update tends to be identical, resulting in the degradation of performance.

To address this issue, DAGNN implements an adaptive adjustment mechanism, which adaptively adjusts the balance between local and global neighborhood information from each node. This mechanism is realized by introducing an adaptive weighting matrix that modulates the contribution of different graph layers to the final node embeddings. By learning this weighting matrix, DAGNN is able to adaptively incorporate information from large receptive fields, mitigating the over-smoothing problem and thus preserving the discriminative power of node embeddings. In addition, DAGNN decouples the representation transformation and propagation in the current graph convolution operation to boost the performance of deeper GNNs that can be used to achieve a larger receptive field.

3.3 Transductive and Inductive Learning

GNNs can be divided into two groups based on the learning method: *transductive* and *inductive*. Transductive-based GNNs need to see the whole graph structure to learn each node feature vector during training. If there is a change in the structure of the graph, a model has to be retrained. Therefore, they are not able to generalize to unseen graphs. On the other hand, the inductive-based GNN learns a trainable function that aggregates the feature vectors from a node neighborhood to generate feature vectors for the nodes in the graph. Because this trainable function is shared across the graph, like the filter of a CNN layer, the learned model can be applied to unseen graphs without re-training, making it generalizable. *In this work, we have performed the training via inductive learning for the GNN models.*

3.4 Related work

ML techniques have been successfully applied to address various challenges during the chip design flow [22]. These techniques have also been used to resolve the difference between QoR estimates in HLS and post-implementation results. Some of this work is shown in Table 3.1.

Dai et al. [13] and Makrani et al. [42] propose *non-graph-based* ML models to estimate a design post-implementation resource usage and timing by extracting global features from HLS synthesis reports, thus requiring the most time-consuming HLS steps, namely scheduling and binding. Dai et al. [13] use a linear model (Lasso), an artificial neural network (ANN), and XGBoost to recalibrate the results generated from HLS reports. Makrani et al. [42] use Linear Regression, ANN, Support Vector Machine (SVM), Random Forest (RF), and an ensemble of the four models. However, their methods require the HLS reports as input, and their ability to correctly estimate *unseen* designs is questionable.

On the other hand, our solution input is the HLS LLVM IR after the front-end execution, so we generate the QoR estimates earlier, before the back-end execution. Therefore, the HLS back-end itself could benefit from our estimates (e.g., HLS scheduling could use our critical time path predictions when balancing pipeline stages).

Another problem with previous work is the limited generalization capabilities since the inputs (features) to the model can only be extracted after scheduling and binding. This means that for each new and unseen design, one must run time-consuming phases of HLS, which can take hours for larger designs, to collect the features needed to estimate QoR.

Wu et al. [53] and Ustun et al. [50] use graph-based ML models to perform HLS prediction tasks. Wu et al. [53] proposes an end-to-end reinforcement learning-based framework for design space exploration. A GNN-based performance predictor (GPP) is integrated into the framework to predict post-implementation resource utilization and timing based on the data flow graph (DFG) representation. They use a separate GNN-based model for each objective. Ustun et al. [50] builds a customized GNN-based model to automatically learn operation mapping patterns to improve operation delay prediction for HLS-based designs. Their approach improves the estimation accuracy by 72 % with respect to *Vitis HLS*. Both works only consider data flow graphs (DFGs). These works show the effectiveness of using graph neural networks even though they do not include pragmas in their input representation and focus on predicting only DSP clustering and clock period.

De et al. [14] compare both graph-based and non-graph-based machine learning models to improve delay prediction accuracy for ASIC HLS and propose a hybrid model that considers both local (structural) and global (domain knowledge) features. The global features are extracted from HLS reports, so one has to run HLS during the inference phase, which can take a long time for large designs.

Wu et al. [54] propose a graph-based ML approach to estimate resource usage and timing based on the results of different HLS stages. The input graphs are constructed from the IR operator information (`*.adb` file) and the features are extracted from both `*.adb` files and other HLS intermediate results. They formulate the prediction problem as a single-objective task; that is, a separate GNN-based model for each type of resource and for timing. Furthermore, the format of the `*.adb` files is not documented and can be changed at any time. Also, [54] does not consider the HLS synthesis directives, whereas we do.

Table 3.1 summarizes the relevant state-of-the-art ML-based approaches for HLS prediction tasks and compares them with our contributions.

Table 3.1 *Comparison of ML-based Approaches for high-level synthesis (HLS) Prediction Tasks*

Work	**ML model**		**Target**		**Task**	**Feature Source**	**Tool**
	Graph	**Non-Graph**	**FPGA**	**ASIC**			
[13]		✓	✓		Resource Usage and Timing	HLS reports	Vivado HLS
[42]		✓	✓		Resource Usage and Timing	HLS reports	Vivado HLS
[53]	✓		✓		DSE	DFG	Vivado HLS
[50]	✓		✓		Operation Delay	Operation Type and Bitwidths from HLS IR code	Vivado HLS
[54]	✓		✓		Resource Usage and Timing	IR operator information (*.adb) and HLS report	Vitis HLS
[14]	✓	✓		✓	Timing	HLS reports	Stratus HLS
This Work	✓		✓		Resource Usage and Timing	HLS LLVM IR	Vitis HLS

3.5 DataSet Generation

The fundamental step in any ML problem is to obtain the data on which the ML model can be trained, validated, and tested. This section describes how the dataset is built by using different designs from various application domains and how a graph is constructed and generated for an HLS design.

3.5.1 Data Collection

The dataset should include a variety of designs from different applications so that the trained ML model is robust enough to generalize. For this purpose, we choose 30 designs from the well-known HLS benchmark suites, namely MachSuite [47], Polyhedral [36], and Rosetta [64]. The application domains of these designs cover a wide range of areas such as linear algebra, image and signal processing, computer graphics, data mining, stencils, sorting, and ML. The detail of the designs with their respective application domain is shown in Table 3.2.

Table 3.2 HLS-based Designs used in DATASET

Design	**Application Domain**
2mm	Linear algebra computations
3mm	Linear algebra computations
3d-rendering	Computer graphics
atax	Linear algebra computations
bicg	Numerical Linear Algebra
correlation	Digital signal processing

Continued on next page

Table 3.2 – *Continued from previous page*

Design	**Application Domain**
covariance	Digital signal processing
doitgen	Linear algebra computation
fdtd_2d	Finite-difference time-domain simulation
fft_strided	Digital signal processing, Image processing, Recursive formulation of the Fast Fourier Transform
gemm	General matrix multiplication
gemm_blocked	General matrix multiplication with better locality
gemm_ncubed	General matrix multiplication for dense matrix multiplication
gemver	Linear algebra computations
gesummv	Linear algebra computations
jacobi_1d	Linear system solver
jacobi_2d	Linear system solver
lu	Linear algebra computations
md	Molecular dynamics simulations
merge	Sorting
mvt	Linear algebra computations
optical flow	Image processing
seidel_2d	Stencils
spam_filter	Spam filtering using Naive Bayes classifier
spmv_crs	Sparse matrix-vector multiplication, using variable-length neighbor lists
spmv_ellpack	Sparse matrix-vector multiplication, using fixed-size neighbor lists
stencil2d	A two-dimensional stencil computation, using a 9-point square stencil
symm	Linear algebra computations
trisolv	Linear system solver
trmm	Linear algebra computations

To create multiple hardware implementations for each design, we used different *HLS pragmas*, i.e., synthesis directives (see Table 3.3) and various clock periods (2.5 ns, 5 ns, 7.5 ns, and 10 ns). This allows our predictive model to learn designs

Table 3.3 *Synthesis* Pragma configurations

Pragma	**Configuration**
Loop Pipelining	Enabled/Disabled
Loop Unrolling	Unrolling Factor
Loop Flattening	Yes/No
Array Partitioning	Block/Reshape/Cyclic/Complete
Function Inline	Yes/No

Table 3.4 Overall *Summary* of designs in our DATASET

	# of **LUTs**	# of **DSPs**	# of **FFs**	**CP** (ns)
Minimum	8	0	24	1.5
Maximum	53 239	360	31 004	8.562
Average	2456	28	2619	3.72

with different area-delay tradeoffs. Thus, a dataset is built with a total of 2465 data points. Each design point in our dataset is synthesized with *Vitis HLS* 2021.2 [56] and implemented it with *Vivado* 2021.2 [57] (using *Vivado* defaults for synthesis and implementation) targeting a *Zynq UltraScale+* FPGA device.

The *ground truth* (actual resource usage and critical path timing) of each of these design points are extracted from the implementation and timing reports generated after the place and route phase for maximum prediction accuracy. Table 3.4 shows the range of values of the target objectives in our dataset.

3.5.2 Graph Generation

The input to the proposed GNN-based predictive model is an HLS IR graph representing the functionality of the design, extracted after the front-end compilation step. As mentioned in Section 3.1.2, we used a combination of control flow, data flow, and call flow graphs for our experiments. A program semantic information representation tool, ProGraML [12] is used to extract the graphs from a given IR and combine them into a single graph. It merges information from control and data flow graphs and also preserve the function hierarchy by incorporating the call flow.

The out-of-the-box configuration of ProGraML converts the LLVM statements into nodes of the generated graph with some special features like the opcode. By using the LLVM language reference manual [1], we have extended ProGraML capabilities to retrieve more critical information from the LLVM statements, namely operand bit width, and opcode category, and append it to the feature vector of each corresponding node in the graph, as discussed below.

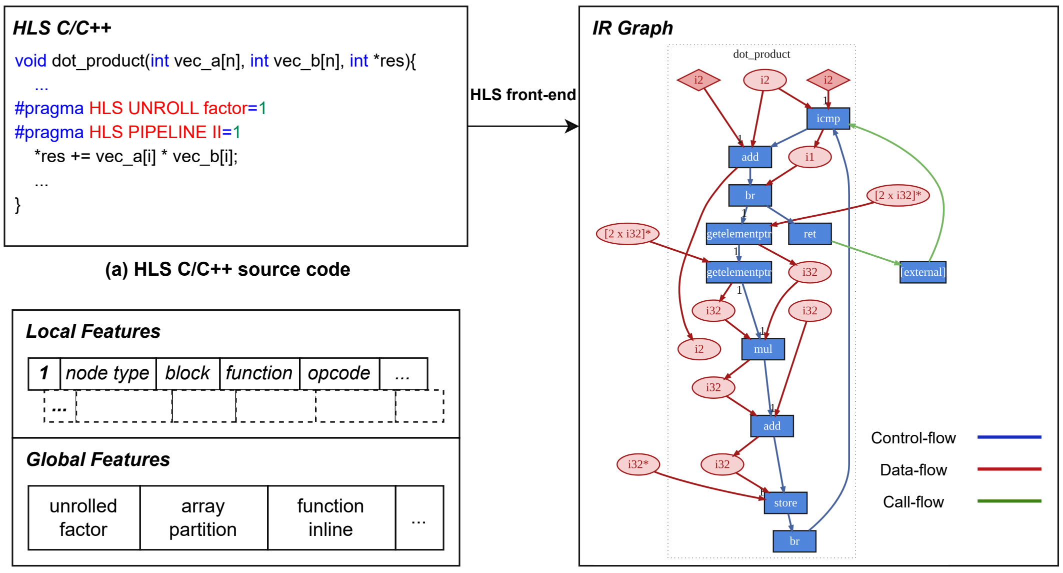
Bit-width describes the number of bits in the instruction operands, which is highly relevant to the hardware resource prediction. For example, an instruction that operates on 7-bit operands require fewer hardware resources than an instruction that operates on 32-bit operands.

The Opcode category identifies the functionality of an LLVM instruction. In general, the resource requirements and behavior of different LLVM instruction categories vary, affecting the overall performance and resource utilization. For example, the The arithmetic category contains statements that perform arithmetic operations, such as adding or multiplying. In principle, this information could be learned by a GNN from the opcode, but this would require more layers and more training data. We decided to provide it directly because it is design-independent and easy to derive automatically.

Algorithm 6 shows the steps to construct the graph for an HLS design.

Algorithm 6 Graph generation for HLS design

```
Require: HLS design
  LLVM IR Bitcode ← Vitis HLS Front-end (HLS design)
  LLVM IR ← LLVM Disassembler (LLVM IR Bitcode)
  Graph Representation ← Modified ProGraML (LLVM IR)
```



```
void dot_product(int vec_a[n], int vec_b[n], int *res){
    ...
#pragma HLS UNROLL factor=1
#pragma HLS PIPELINE II=1
    *res += vec_a[i] * vec_b[i];
    ...
}
```

(a) HLS C/C++ source code

(c) Features

(b) IR Graph

Fig. 3.3 An High-level synthesis (HLS) design example with its graph representation (a) shows the HLS code for an implementation of dot product with two sample synthesis directives provided as an input to the HLS tool (b) shows its graph representation based on an intermediate-representation (IR) which is extracted after the HLS front-end compilation (c) shows the local and global features used by the model. Local features are extracted directly from the IR graph. Global features are user-defined synthesis directives.

Fig. 3.3 shows a toy example of how the input graph to our model is generated. For illustration purposes, only the most relevant nodes in the graph are shown. For example, we have omitted zero extension and alloca nodes in the figure, while our model considers them as well. Fig. 3.3 (a) shows the HLS code for implementing the dot product of two input vectors, including two sample HLS pragmas inside the loop. Fig. 3.3 (b) shows its graph representation, extracted after the HLS frontend compilation. The graph has two types of nodes. The LLVM statements are represented by the blue nodes, which are connected according to the control flow. The variable values and constant values that represent the operands and results of the statements in the data flow are represented by the nodes in red. Three different colors are used to symbolize different types of edges: blue, red, and green for control, data, and call, respectively. Fig. 3.3 (c) shows the local and global features that can be extracted directly from the IR graph and user-defined optimization directives (see Section 3.6 for details).

3.6 Features

Our proposed approach uses two different sets of features that are useful for predicting post-implementation QoR. These feature sets are extracted from two different sources, the HLS IR code and user-defined HLS synthesis directives. We call these sets local and global features, respectively. Local features contain structural and contextual information. Structural information describes the connectivity of nodes in the graph and is encoded as an adjacency matrix. Contextual information refers to node and edge properties, and this information is explicitly encoded as a feature vector for each node and edge.

For each node, its *type*, *category*, *opcode*, *block ID*, *function ID*, and *bitwidth* are taken into account. For example, *type* indicates whether the node is a statement, a variable, or a constant. For an edge, we only considered its *type*, which basically tells whether the edge belongs to control, data, or call flow. Details of local features are listed in Table 3.5. Edge features are only considered by one of the models we used, namely GAT.

Table 3.3 displays the list of global features. We feed the global feature vector into our prediction model by concatenating it with the final graph-level feature vector generated by *READOUT*. These features can be useful for integrating domain-

Table 3.5 *Local Features*: Nodes and Edges

	Feature	Description	Example Values
Node	Type	Node Type	Instruction, Variable, Constant
	Block	LLVM Block ID	0, 1, 2 etc.
	Function	Function ID	0, 1, 2 etc.
	Opcode Category	Opcode type based on LLVM	Unary, Binary, Terminator, etc.
	Opcode	Opcode of the node	add, icmp, shl etc.
	Bitwidth	Bitwidth of the operand	8, 16, 32, etc.
Edge	Type	Flow Type	Control, Data, Call

specific knowledge into the model and improving its predictive capabilities. They have a direct impact on the timing and resource requirements of a design. For this reason, we explicitly provide this information to our model. These features are exactly the same as the main user-defined HLS synthesis directives, so they are well-known to the designer.

These global features could also be automatically extracted from the HLS IR code via *ssdm intrinsic functions* (function calls that encode both user-written and tool-generated synthesis directives) and encoded as local features for the nodes. However, as we show in Section 3.8.2, the estimation accuracy is significantly improved by using global features as well. We leave to future work the exploration of different sets of local features and/or GNN architectures to overcome the need for global features and handle more complex kernels, including, for example, multiple pipelined loops.

3.7 Model

We formulate the QoR prediction problem as a multi-objective regression task to estimate the post-implementation *timing* and *resource usage* for LUT, FF, and DSP of a given design based on its HLS IR without scheduling and binding. We do not address the BRAM estimation problem because the HLS estimates are already quite reliable in this case, although we could. We use multi-task learning, where a single GNN model is trained and the generated graph feature vector is fed to a set of MLPs to estimate the different objectives.

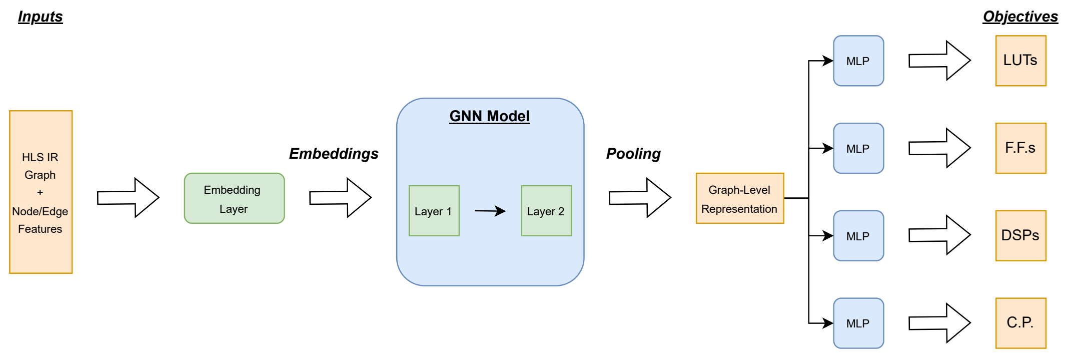


Fig. 3.4 General structure of the framework to evaluate different graph neural network (GNN) models. Features are passed to the trainable embedding layer to create their dense vector representations. This vector representation with the corresponding HLS IR based graph is fed as an input to GNN model. A pooling operation is applied across all the nodes to create a single graph-level feature vector which is fed to four separate MLPs for each prediction objective (lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.)).

The general structure of our model architecture is shown in Fig. 3.4. It takes the graph representation of the design as input and creates the initial feature vector by converting the features extracted from LLVM (Table 3.5) via a trainable embedding layer [46]. This information is then passed to the GNN model, which iteratively updates the feature vectors layer by layer. These updated feature vectors are used to generate a graph-level representation vector via *mean pooling*, i.e. by computing an average vector of all final feature vectors of all nodes, which is then concatenated with the global feature vector and passed to a set of MLPs to predict multiple targets. We evaluated different GNN models (Section 4.2), using the same flow for a fair comparison (so the only difference is the type of GNN layers).

Fig. 3.5 shows the *training flow* of the proposed framework. Before training, we preprocess the data and apply normalization so that each objective (timing, LUTs, DSPs, and FFs) can contribute equally to the training loss. For example, in the case of resource utilization, we normalize the resource utilization by dividing it by the total number of resources available on the FPGAs, converting it to a percentage utilization. Once the dataset and associated features are available, we perform the

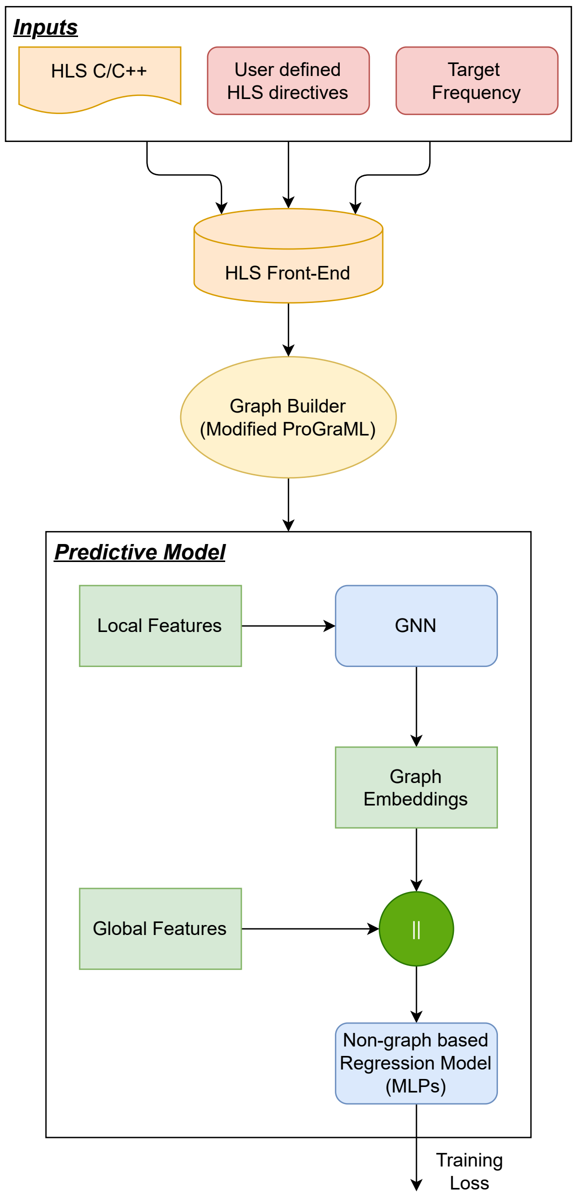


Fig. 3.5 Training phase of the proposed framework.

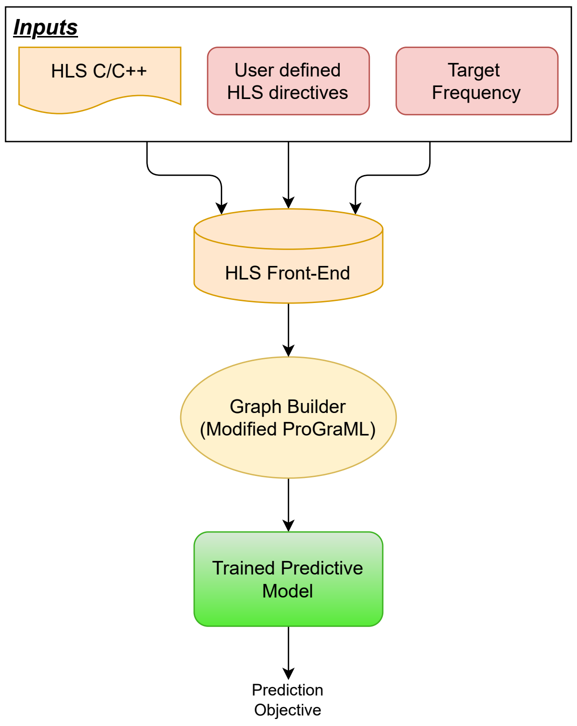


Fig. 3.6 Inference phase of the proposed framework.

training to obtain a predictive model. The ground truth labels are extracted from the post-implementation reports.

We train the GNNs model via supervised learning to learn the behavior of the underlying HLS heuristics and optimization techniques, such as scheduling, sharing, register allocation, etc., in order to quickly and accurately predict the desired objectives. During the training phase, an HLS design and its configurations are first fed into the HLS tool front-end, where the HLS IR is generated. The graph generator (see Section 3.5.2 for details) then converts the HLS IR into the graph used by the GNN model as discussed above.

To select the best GNN model, we first randomly set aside a 20 % of the data set, also called the hold-out or test set. This hold-out is not used during training and validation, but only at the end to evaluate the final performance and report the results. Then, we perform training by 5-fold cross-validation on the remaining 80 % dataset, where the hyper-parameters of the considered models are optimized and tuned.

The *inference flow* of the proposed framework is shown in Fig. 3.6. The main purpose of the inference phase is to achieve fast and accurate QoR prediction of the

Table 3.6 Inference time for the proposed model vs HLS time per design point

	HLS F.E. (s)	HLS B.E. (s)	Graph Generation (s)	Inference (s)	Total (s)
Model [1]	11.65	—	0.24	0.0054	11.90
HLS [2]	11.65	15.07	—	—	26.72

[1] Model Time = Front-End (F.E.) + Graph Generation + Inference
[2] HLS = Front-End (F.E.) + Back-End (B.E.)

design compared to the HLS baseline without going through the implementation process, which is time-consuming. In the inference phase, we apply the same pre-processing workflow to unseen designs (test set data points) to generate a graph and extract feature vectors. These are then fed into the already-trained model to perform target prediction.

It is worth noting that our predictive model is able to complete the inference to estimate resources and timing in milliseconds given a graph, as opposed to the implementation phase, which typically takes minutes to hours. Table 3.6 shows the inference time of the predictive model and the time HLS tool takes for a single data point. Our model provides better estimates of resource usage and timing in less than half of a time with respect to the HLS tool. If we exclude the common time needed to execute the Front-End, our model is more than 60x faster than the Back-End.

3.8 Experimental Results

3.8.1 Setup

Our framework is deployed using PyTorch and all GNN models mentioned in Section 4.2 are implemented and trained using PyTorch Geometric [16] library on a Linux machine with an Nvidia GeForce RTX 3060 graphical processing unit. The designs in the dataset (Section 3.5) are synthesized and implemented using *Vitis HLS* 2021.2 [56] and *Vivado* 2021.2 [57], respectively. The ground truth labels of the four objectives (LUT, FF, DSP, C.P.) are extracted from the post-implementation reports. The dataset is randomly divided into 70 % for training, 10 % for validation, and 20 % for testing.

For our experiments, each model has the structure shown in Fig. 3.4, with the following characteristics:

1. A trainable embedding layer, that converts the features (see Table 3.5) from the HLS IR into a feature vector of size 300 that is fed as an input to the GNN layers.
2. Two GNN layers, i.e., the *AGG* and *UPDATE* functions are run twice, with input feature vector size 300, internal feature vector size of 128 and output feature vector size of 64.
3. A mean pooling layer, taking the mean of each one of the 64 output features across all nodes, and generating a single graph-level vector of 64 features that are fed to the MLPs.
4. Four separate MLPs, one for each prediction objective (LUT, FF, DSP, C.P.), each with three layers with 32, 16 and 1 output features respectively.

We trained the GNN-based models in an *inductive setting* [21] with the Adam optimizer [28] for 100 epochs, using a learning rate of 0.001 with a weight decay of 0.0005, and an exponential linear unit [8] as the activation function. All of these hyper-parameters, including the number of layers and feature vector sizes, are tuned using the validation set during the training phase. Since the prediction problem is formulated as a regression task, we use *root mean square error (RMSE)* as a metric for evaluating the models. We perform 5-fold cross-validation to check the effectiveness and robustness of the models and to select the best-performing model. We also compare our models with the commercial HLS tool (*Vitis HLS*) used as a *baseline model* and with a graph learning-based performance prediction model [54]. Algorithm 7 shows the process of extracting the baseline and ground truth values for a given HLS design.

Algorithm 7 Baseline and ground truth (G.T.) extractor

```
Require: An HLS design
  RTL design, HLS report ← Vitis HLS (HLS design)
  Post-implementation report ← Vivado (RTL design)
  Baseline ← HLS baseline Extractor(HLS report)
  G.T. ← G.T. Extractor(Post-implementation report)
```

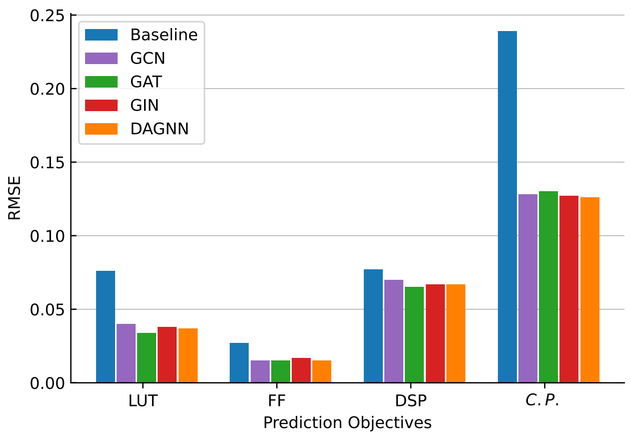


Fig. 3.7 Performance comparison of different GNN-based predictive models for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with HLS *Baseline* on the *test set* (the lower the better). Only Local Features are considered.

3.8.2 Model Evaluation and Model Selection

A GNN model transforms the local feature vectors into a single graph-level feature vector, which is then passed to the non-graphic regression model (in this case using a 3-layer MLPs). We first test the performance of the GNN models with *only local features* (i.e., no manual information from the designer).

Fig. 3.7 shows the performance evaluation of GNN models with respect to *baseline* regarding LUT, FF, DSP and C.P. in terms of RMSE, while Fig. 3.8 shows the prediction improvements of the models over the baseline. For LUT utilization prediction, GAT provides the best improvement, with more than 55 % over baseline. In the case of FF, GCN, GAT, and DAGNN give prediction improvements of more than 40 %. GAT is the best of all models at reducing prediction error relative to the HLS baseline model for DSP utilization. For C.P. timing, all models improve the prediction by more than 45 %. On average, GAT is the best model for improving

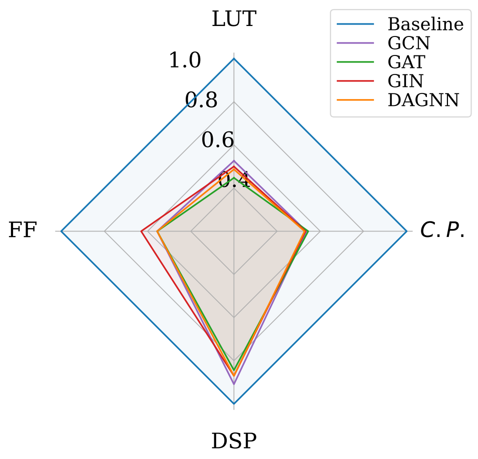


Fig. 3.8 Quality of results (QoR) prediction improvements of different GNN-based predictive models for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with respect to HLS *Baseline* on the *test set*. Only Local features are used.

resource utilization prediction, and DAGNN is the best for timing. Note that in practice, different models may be used for different objectives.

Fig. 3.9 shows the performance evaluation of the graph neural network (GNN) model with respect to the HLS baseline for the target objectives by *also using global features*. Global features are concatenated with the graph features generated by the GNN models and fed into the non-graph regression model, MLP. Fig. 3.10 shows the quality of results (QoR) improvements of the GNN-based predictive models over the baseline. All models provide performance prediction improvements of more than 50 % for LUT utilization. For FF utilization, GAT and DAGNN are the best models at reducing the prediction error over the baseline, improving the QoR prediction by up to 52 %. In the case of DSP utilization, GAT provides the best prediction improvement among all GNN models with respect to the HLS baseline. The GAT model with both local and global features gives a relative improvement of almost 19 % over the GAT model with local features only.

For C.P. timing, GCN, GAT, and DAGNN based models improve the prediction by more than 70 % with respect to the baseline, with DAGNN providing the best improvement at 72 %.

These GNN models provide a relative performance improvement of up to 47 % over the same models using only local features.

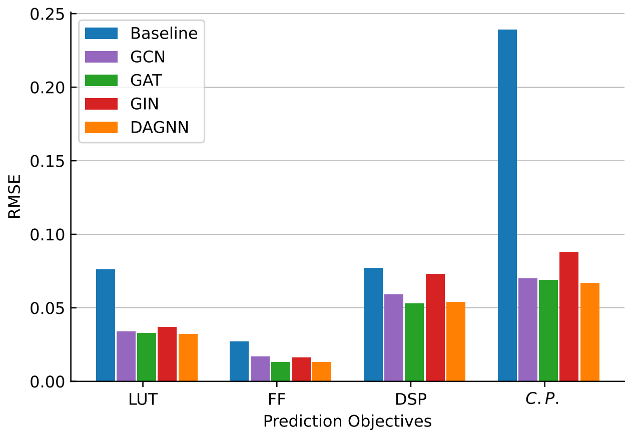


Fig. 3.9 Performance comparison of different GNN-based predictive models for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with HLS *Baseline* on the *test set* (the lower the better). In addition to Local features, Global features are also considered to evaluate their impact on performance.

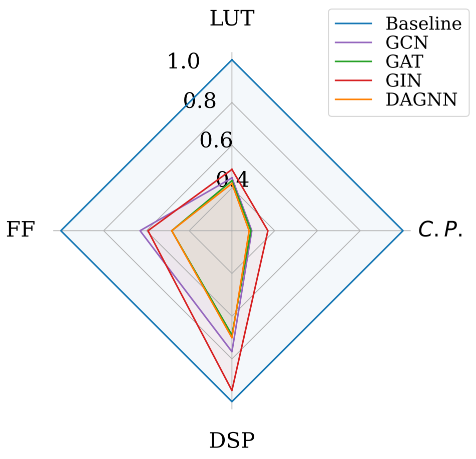


Fig. 3.10 Quality of results (QoR) prediction improvements of different GNN-based predictive models for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with respect to HLS *Baseline* on the *test set*. In addition to Local features, Global features are also used to evaluate their impact on performance.

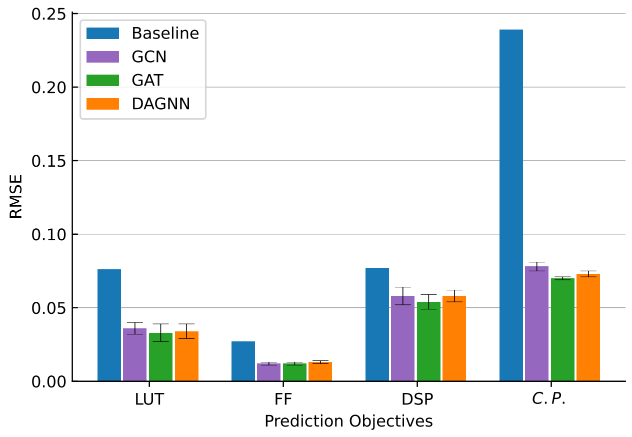


Fig. 3.11 Performance comparison of best performing GNN-based predictive models for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with HLS *Baseline* (the lower the better). A *5-fold cross-validation* with the *holdout set* is performed. Both Local and Global features are considered.

It is worth noting that all of these GNN-based prediction models perform better than the HLS baseline model for all target objectives. graph isomorphism network (GIN) provides the least benefit in terms of prediction error reduction among all the models tested, especially in the case of DSP usage prediction (see Fig. 3.10). Based on this empirical evidence, we drop GIN for the model selection phase.

For *model selection*, we use 5-fold cross-validation with a holdout test set (see Section 3.7 for details). Fig. 3.11 compares the performance of the GCN, GAT and DAGNN models with respect to the HLS *baseline*. Fig. 3.12 shows the QoR improvements of the selected model over the baseline for the target objectives. For the LUT and FF utilization predictions, all selected graph-based models give improvements of more than 50 %. The GAT based model gives the best result of 57 % for LUT, while the GCN based model gives the best result of 55 % for FF. In the case of DSP prediction, GAT is the clear winner, reducing the prediction error by more than 30 %. For C.P. timing, all graph-based models provide a prediction improvement of more than 67 %, with the GAT-based prediction model being the best (71 %). GAT outperforms other graph-based models in three out of four target objectives and is not far behind in predicting FF usage (where the GCN-based model performs best). *Based on these results, we choose the GAT-based model*, but we could use different models for different objectives, as mentioned above.

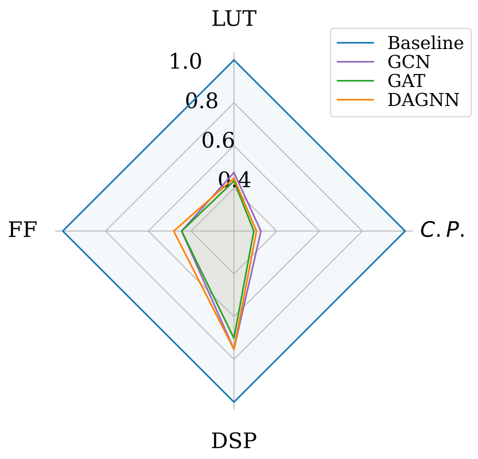


Fig. 3.12 Quality of results (QoR) prediction improvements of best performing GNN-based predictive models for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with respect to HLS *Baseline*. A *5-fold cross-validation* with the *holdout set* is performed. Both Local and Global features are used.

To evaluate the selected model over the entire dataset, we perform a generic 5-fold cross-validation. Fig. 3.13 shows the quality of results prediction improvements over the entire data. It is observed that the GAT based prediction model provides significantly better prediction with respect to the HLS baseline model, giving up to 74 % performance prediction improvements.

3.8.3 Generalization and Comparison

To check the performance of our chosen model on unseen kernels and for a quantitative comparison with the state-of-the-art, we choose four kernels (`gemm_ncubed`, `optical_flow`, `jacobi2d`, and `stencil2d`), including all their design variants created using different HLS synthesis directives and clock frequencies, and use them as a test set for evaluation (in 5-fold cross-validation, the test set was chosen at random, so training saw many variants of each design). `gemm_ncubed` is a dense matrix multiplication algorithm, `optical_flow` computes the motion of each pixel in a sequence of image frames, `jacobi2d` is an iterative method that computes and updates each grid point by averaging its neighbors, and `stencil2d` performs stencil computation using a 9-point square stencil. Note that all of these kernels have different code structures.

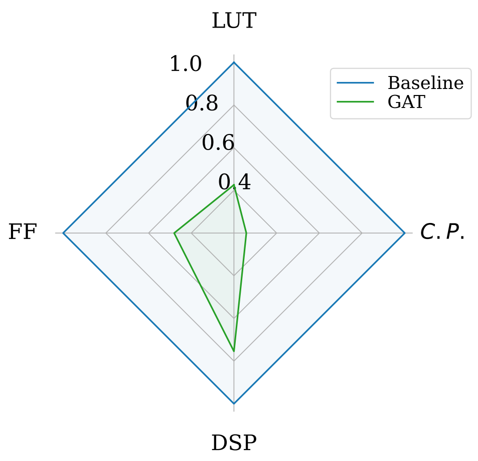


Fig. 3.13 Quality of results (QoR) prediction improvements of the selected GNN-based predictive model for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with respect to HLS *Baseline*. A *5-fold cross-validation* over the *whole dataset* is performed.

We compare our model quantitatively with the graph-based machine learning approach [54], which provides their tool in open source (as us). For comparison with other ML-based approaches, Table 3.1 provides a qualitative analysis.

We take the best-performing regression model in [54] for resource usage and timing prediction, train it to the best of our ability, and call it *PNA-HLS*. A separate PNA-HLS model is trained for each objective. Fig. 3.14 shows that our GAT-based prediction model reduces the prediction error for resource utilization and timing prediction by 68 % and 34 %, respectively, compared to HLS. Our proposed model also outperforms the state-of-the-art PNA-HLS model by 29 % and 22 % for resource utilization and timing prediction, respectively.

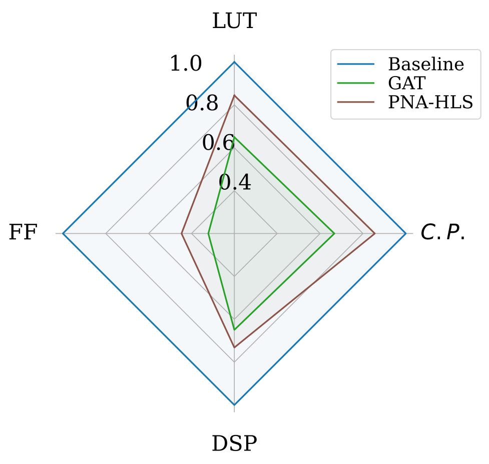


Fig. 3.14 Quality of results (QoR) prediction improvements of our best performing GNN-based model for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with respect to HLS *Baseline* and the state-of-the-art on *Unseen Kernels*.

Chapter 4

Conclusions and Future Work

4.1 Array-Specific Dataflow caches

The experimental results, summarized in Fig. 4.1 show that our approach of semi-automatically generating an LCS-like architecture through dataflow caches is an effective solution for significantly improving performance and energy consumption, without requiring high design effort. Designers simply need to perform a DSE of the cache configurations instead of extensively changing the algorithm for buffering data on-chip. Additionally, *for algorithms with irregular or data-dependent memory access patterns, caching would be the only way to actually improve memory access performance*.

To achieve performance comparable with the manually optimized designs of *MatMult* and *Conv2D*, our cache would require more resources than the ones provided by the small FPGA used in the tests. For *BitSort*, caching was the only feasible performance optimization we found, due to the irregular, but with good data locality, memory access pattern. Adding an RTL cache module post-HLS fails to provide any advantage, since the HLS-generated circuit is optimized for high-latency memory accesses, and cannot achieve any acceleration from an external cache.

It is worth noting that we collected the results from an embedded device, which provides a DDR4 memory.

Modern datacenter-level devices are equipped with HBMs. HBMs, compared with DDR4 memories, are characterized by the availability of many more ports,

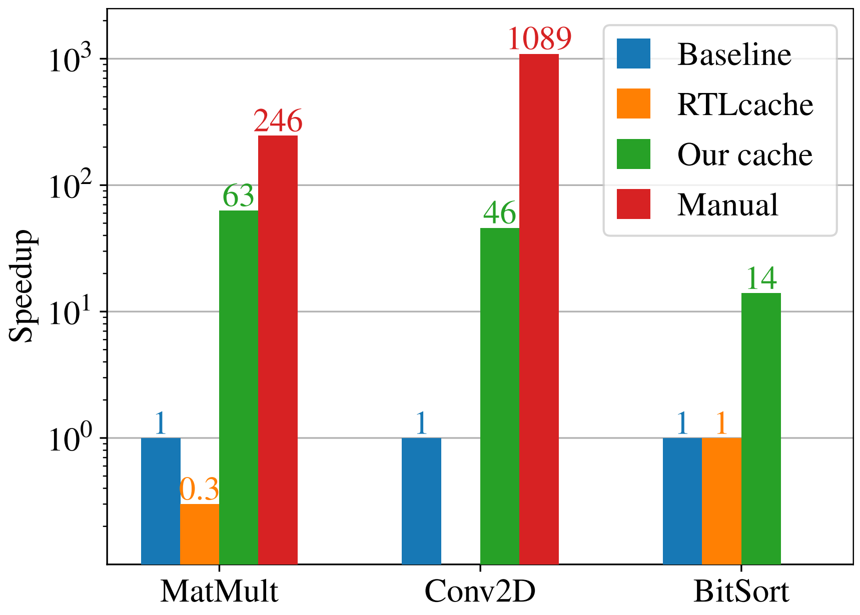


Fig. 4.1 Speedup of the tested benchmarks.

thus dramatically increasing bandwidth, while paying a price in terms of access latency (roughly 2 times larger, as benchmarked by Wang et al. [51]). Thanks to these characteristics, a cache potentially provides even greater advantages than experienced with our setup, since caches are precisely designed for mitigating the performance penalties of high-latency memories. Moreover, irregular memory access patterns require word-sized accesses, since the HLS tool is unable to optimize the accesses through bursting and interface widening, underutilizing the HBM ports bitwidth. On the other hand, caches always access the DRAM in lines, thus enabling the interface optimizations, resulting in better exploitation of the large interface bitwidth of HBM. We leave the evaluation of our caches on HBM-equipped HW, to quantitatively support these considerations, as future work.

We plan to automate the DSE for optimal cache parameter selection, by extending one of the state-of-the-art cache parameter optimization methods [49] to support the configuration space of our cache architecture for some additional dimensions with respect to standard caches, such as the request-response distance, the number of ports, and the address bit mapping.

To further improve performance, we are considering to implement a prefetching mechanism to anticipate the memory requests by loading data in advance, before they are needed by the computation, thus fully emulating the LCS pattern.

4.2 GNN based QoR Prediction

Although HLS provides great flexibility for optimizing designs for area and performance, HLS-estimated QoR often differ from actual post-implementation results. In Chapter 3, we propose an HLS tool-agnostic GNN-based framework for estimating quality of results of HLS designs, as long as the tool provides a publicly readable LLVM-based IR. Of course, the GNN must be trained differently for each new tool, but once trained, it can be reused for different results.

First, a method is developed to extract a graph-based representation of a design directly from the HLS front-end output, encoding both program semantics and HLS synthesis directive information. Then, a multi-objective GNN-based learning model is proposed to predict resource usage and timing of HLS designs within milliseconds without invoking the HLS back-end to perform scheduling and binding. To address the issue of the limited ability of a pure GNN-based model to be aware of global information such as design guidelines, this information is explicitly passed to the learning model in addition to the local features automatically extracted from the IR. The experimental results show that our proposed prediction model outperforms both a commercial HLS tool and a state-of-the-art GNN-based tool [54] for realistic benchmark applications from different domains. It also shows that our model is capable of extending the learned knowledge and generalizing it to unseen design cases.

In the future, we plan to extend our framework to larger designs and more application domains. We also plan to add the support for additional performance parameters like throughput and latency to support faster design space exploration.

References

[1] (2022a). *LLVM Language Reference Manual*. LLVM Project.

[2] (2022b). The LLVM Compiler Infrastructure Project. https://llvm.org/.

[3] Adler, M., Fleming, K. E., Parashar, A., Pellauer, M., and Emer, J. (2011). Leap scratchpads: Automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 25–28, New York, NY, USA. Association for Computing Machinery.

[4] Avnet Inc. (2018). *Ultra96 Hardware User's Guide*.

[5] Brignone, G., Usman Jamal, M., Lazarescu, M. T., and Lavagno, L. (2022). Array-specific dataflow caches for high-level synthesis of memory-intensive algorithms on fpgas. *IEEE Access*, 10:118858–118877.

[6] Brody, S., Alon, U., and Yahav, E. (2022). How Attentive are Graph Attention Networks? In *Proc. 10th Int. Conf. Learn. Represent. (ICLR)*.

[7] Choi, J., Nam, K., Canis, A., Anderson, J., Brown, S., and Czajkowski, T. (2012). Impact of cache architecture and interface on performance and area of fpga-based processor/parallel-accelerator systems. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 17–24.

[8] Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2016). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). In *Proc. 4th Int. Conf. Learn. Represent. (ICLR)*.

[9] Cong, J., Lau, J., Liu, G., Neuendorffer, S., Pan, P., Vissers, K., and Zhang, Z. (2022). FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Trans. Reconfigurable Technol. Syst.*, 15(4):1–42.

[10] Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., and Zhang, Z. (2011). High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 30(4):473–491.

[11] Cong, J., Zhang, P., and Zou, Y. (2012). Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 1233–1238, New York, NY, USA. Association for Computing Machinery.

[12] Cummins, C., Fisches, Z., Ben-Nun, T., Hoefler, T., O'Boyle, M., and Leather, H. (2021). ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proc. 38th Int. Conf. Mach. Learn. (ICML)*, pages 2244–2253.

[13] Dai, S., Zhou, Y., Zhang, H., Ustun, E., Young, E. F., and Zhang, Z. (2018). Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning. In *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, pages 129–132.

[14] De, S., Shafique, M., and Corporaal, H. (2023). Delay Prediction for ASIC HLS: Comparing Graph-based and Non-Graph-based Learning Models. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 42(4):1133–1146.

[15] de Fine Licht, J., Besta, M., Meierhans, S., and Hoefler, T. (2021). Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1014–1029.

[16] Fey, M. and Lenssen, J. E. (2019). Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop Represent. Learn. Graphs Manifolds*.

[17] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural Message Passing for Quantum Chemistry. In *Proc. 34th Int. Conf. Mach. Learn. (PMLR)*, page 1263–1272.

[18] GNU (2022). GCC, the GNU Compiler Collection.

[19] Guo, Z., Liu, M., Gu, J., Zhang, S., Pan, D. Z., and Lin, Y. (2022). A Timing Engine Inspired Graph Neural Network Model for Pre-Routing Slack Prediction. In *Proc. 59th ACM/IEEE Des. Automat. Conf. (DAC)*, pages 1207–1212.

[20] Haaswijk, W., Collins, E., Seguin, B., Soeken, M., Kaplan, F., Süsstrunk, S., and De Micheli, G. (2018). Deep Learning for Logic Optimization Algorithms. In *Proc. IEEE Int. Symp. Circuits Sys. (ISCAS)*, pages 1–4.

[21] Hamilton, W. L., Ying, R., and Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. In *Proc. 31st Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, page 1025–1035.

[22] Huang, G., Hu, J., He, Y., Liu, J., Ma, M., Shen, Z., Wu, J., Xu, Y., Zhang, H., Zhong, K., Ning, X., Ma, Y., Yang, H., Yu, B., Yang, H., and Wang, Y. (2021). Machine Learning for Electronic Design Automation: A Survey. *ACM Trans. Des. Automat. Electron. Syst.*, 26(5):1–46.

[23] Intel® (2021a). *Arria® 10 EMIF Latency*.

[24] Intel® (2021b). *Avalon® Memory-Mapped Host Interfaces and Load-Store Units*.

[25] Intel® (2021c). *Intel® High Level Synthesis Compiler Pro Edition Reference Manual*.

[26] Jamal, M. U., Li, Z., Lazarescu, M. T., and Lavagno, L. (2023). A graph neural network model for fast and accurate quality of result estimation for high-level synthesis. *IEEE Access*, 11:85785–85798.

[27] Jo, G., Kim, H., Lee, J., and Lee, J. (2020). Soff: An opencl high-level synthesis framework for fpgas. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 295–308.

[28] Kingma, D. P. and Ba, J. (2015). Adam: A Method for Stochastic Optimization. In *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*.

[29] Kipf, T. (2023). Graph convolutional networks.

[30] Kipf, T. N. and Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks. In *Proc. 5th Int. Conf. Learn. Represent. (ICLR)*.

[31] Kirby, R., Godil, S., Roy, R., and Catanzaro, B. (2019). CongestionNet: Routing Congestion Prediction Using Deep Graph Neural Networks. In *Proc. IFIP/IEEE 27th Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, pages 217–222.

[32] Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Symp. Code Generation Optim. (CGO)*, pages 75–86.

[33] Lin, Z., Yuan, Z., Zhao, J., Zhang, W., Wang, H., and Tian, Y. (2022). PowerGear: Early-Stage Power Estimation in FPGA HLS via Heterogeneous Edge-Centric GNNs. In *Proc. Conf. Exhib. Des. Automat. Test Eur. (DATE)*, pages 1341–1346.

[34] Liu, M., Gao, H., and Ji, S. (2020). Towards Deeper Graph Neural Networks. In *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discovery & Data Mining*, pages 338–348.

[35] Lopera, D. S., Servadei, L., Kiprit, G. N., Hazra, S., Wille, R., and Ecker, W. (2021). A Survey of Graph Neural Networks for Electronic Design Automation. In *ACM/IEEE 3rd Workshop Mach. Learn. CAD (MLCAD)*, pages 1–6.

[36] Louis-Noël Pouchet, Uday Bondugula, T. Y. (2022). PolyBench/C.

[37] Lu, Y.-C., Pentapati, S., and Lim, S. K. (2020). VLSI Placement Optimization using Graph Neural Networks. In *Proc. 34th Adv. Neural Inf. Process. Sys. (NeurIPS) Workshop ML Sys.*, pages 6–12.

[38] Lu, Y.-C., Pentapati, S., and Lim, S. K. (2021). The Law of Attraction: Affinity-Aware Placement Optimization Using Graph Neural Networks. In *Proc. Int. Symp. Physical Des. (ISPD)*, pages 7–14.

[39] Ma, L., Lavagno, L., Lazarescu, M., and Arif, A. (2017). Acceleration by inline cache for memory-intensive algorithms on fpga via high-level synthesis. *IEEE Access*, PP:1–1.

[40] Ma, Y., He, Z., Li, W., Zhang, L., and Yu, B. (2020). Understanding Graphs in EDA: From Shallow to Deep Learning. In *Proc. Int. Symp. Physical Des. (ISPD)*, pages 119–126.

[41] Ma, Y., Ren, H., Khailany, B., Sikka, H., Luo, L., Natarajan, K., and Yu, B. (2019). High performance Graph Convolutional Networks with Applications in Testability Analysis. In *Proc. 56th ACM/IEEE Des. Automat. Conf. (DAC)*, pages 1–6.

[42] Makrani, H. M., Farahmand, F., Sayadi, H., Bondi, S., Dinakarrao, S. M. P., Homayoun, H., and Rafatirad, S. (2019). Pyramid: Machine learning Framework to Estimate the Optimal Timing and Resource Usage of a High-Level Synthesis Design. In *Proc. 29th Int. Conf. Field Program. Logic Appl. (FPL)*, pages 397–403.

[43] Marjanovic, J. (2021). Exploring the ps-pl axi interfaces on zynq ultrascale+ mpsoc.

[44] Matthews, E., Doyle, N. C., and Shannon, L. (2015). Design space exploration of l1 data caches for fpga-based multiprocessor systems. FPGA ’15, page 156–159, New York, NY, USA. Association for Computing Machinery.

[45] Pouchet, L.-N., Zhang, P., Sadayappan, P., and Cong, J. (2013). Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA ’13, page 29–38, New York, NY, USA. Association for Computing Machinery.

[46] PyTorch (2022). Embedding - PyTorch 2.0 documentation.

[47] Reagen, B., Adolf, R., Shao, Y. S., Wei, G.-Y., and Brooks, D. (2014). Machsuite: Benchmarks for accelerator design and customized architectures. In *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, pages 110–119.

[48] Teich, J. (2012). Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430.

[49] Upadhyay, B. R. and Sudarshan, T. S. B. (2016). Design space exploration of cache memory — a survey. In *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, pages 2294–2297.

[50] Ustun, E., Deng, C., Pal, D., Li, Z., and Zhang, Z. (2020). Accurate Operation Delay Prediction for FPGA HLS Using Graph Neural Networks. In *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD)*, pages 1–9.

[51] Wang, Z., Huang, H., Zhang, J., and Alonso, G. (2020). Shuhai: Benchmarking high bandwidth memory on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–119.

[52] Winterstein, F., Fleming, K., Yang, H.-J., Wickerson, J., and Constantinides, G. (2015). Custom-sized caches in application-specific memory hierarchies. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 144–151.

[53] Wu, N., Xie, Y., and Hao, C. (2021a). Ironman: GNN-Assisted Design Space Exploration in High-Level Synthesis via Reinforcement Learning. In *Proc. Great Lakes Symp. VLSI (GVLSI)*, pages 39–44.

[54] Wu, N., Yang, H., Xie, Y., Li, P., and Hao, C. (2022). High-Level Synthesis Performance Prediction Using GNNs: Benchmarking, Modeling, and Advancing. In *Proc. 59th ACM/IEEE Des. Automat. Conf. (DAC)*, pages 49–54.

[55] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Yu, P. S. (2021b). A Comprehensive Survey on Graph Neural Networks. *IEEE Trans. Neural Netw. Learn. Syst.*, 32(1):4–24.

[56] Xilinx (2021a). *Vitis High-Level Synthesis User Guide*.

[57] Xilinx (2021b). *Vivado Design Suite User Guide*.

[58] Xilinx Inc. (2021a). Design and analysis of hardware kernel module for 2-d video convolution filter.

[59] Xilinx Inc. (2021b). *PYNQ: Python productivity for Xilinx platforms*.

[60] Xilinx Inc. (2021c). *System Cache LogiCORE IP Product Guide (PG118))*.

[61] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2019). How Powerful are Graph Neural Networks? In *Proc. 7th Int. Conf. Learn. Represent. (ICLR)*.

[62] Zhang, Y., Ren, H., and Khailany, B. (2020). GRANNITE: Graph Neural Network Inference for Transferable Power Estimation. In *Proc. 57th ACM/IEEE Des. Automat. Conf. (DAC)*, pages 1–6.

[63] Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., and Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81.

[64] Zhou, Y., Gupta, U., Dai, S., Zhao, R., Srivastava, N., Jin, H., Featherston, J., Lai, Y.-H., Liu, G., Velasquez, G. A., Wang, W., and Zhang, Z. (2018). Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs. In *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, page 269–278.