Memory Aware Design Optimisation for High-Level Synthesis

André Bannwart Perina[1*], Jürgen Becker[2] and Vanderlei Bonato[1]

[1*]Institute of Mathematics and Computer Science, University of São Paulo, São Carlos, Brazil.

[2]Institute for Information Processing Technologies, Karlsruhe Institute of Technology, Karlsruhe, Germany.

*Corresponding author(s). E-mail(s): abperina@alumni.usp.br;
Contributing authors: becker@kit.edu; vbonato@usp.br;

Abstract

The FPGA environment is traditionally exotic to high-level software developers, mainly due to the large difference in the development methodologies. This can be mitigated through High-Level Synthesis (HLS) tools. By incorporating complex models and code analyses, these tools allow the use of software languages as input for FPGA designs. This paper presents a Design Space Exploration (DSE) approach that uses an estimator named Lina. This approach iterates over hundreds/thousands of combinations of HLS compiler directives in search for the best one, while avoiding HLS runs. Lina approximates the performance and resource usage that the HLS compiler would reach for each combination of directives. It uses simpler models, and as such it runs generally faster than HLS compilation. This version of Lina has an off-chip memory model, allowing the estimation of off-chip memory accesses considering several aspects such as burst detection, data packing, and effect of other compiler directives. For a simple convolution kernel with off-chip accesses, explorations using Lina correctly inferred all but one of the compiler directives when searching the solution with best performance. The best points given by the approach were always in the top-10, reaching at least **720**× speed-up without considering start/end data transfers. Explorations of 4 kernels from Parboil benchmark are also presented. Although there is still significant non-optimised data transfer overheads, speed-ups were reached in all kernels.

Keywords: Reconfigurable Computing, High-Level Synthesis, Design Space Optimisation, Model-based Estimation

1 Introduction

Since its debut nearly four decades ago, Field-Programmable Gate Arrays have been used for a multitude of applications, such as glue logic, custom hardware acceleration and hardware prototyping. Considering the current challenges on the semiconductor level (e.g. dark silicon), architectures that are tailored for specific compute patterns are increasingly more desirable, rather than having a fully general-purpose one. These architectures are able to provide better performance and/or energy efficiency to certain applications than the general-purpose counterpart, making better use of the resources provided [1].

For example, Graphics Processing Units (GPUs) excel at matrix calculations due to their inherent Single-Instruction Multiple-Data (SIMD) design. Specialised languages such as CUDA and OpenCL are normally used in the GPU development flow. For FPGAs, Register-Transfer Level

(RTL) languages are used to define the computation circuits. Using RTL on FPGAs has a steeper learning and effort curve as compared to GPU design with CUDA/OpenCL, since several concepts exotic to software development are often faced. In addition, FPGA compilation (synthesis) can take from several minutes to hours. All these aspects hinder FPGA programmability as compared to usual high-level languages. In general, FPGA development requires more man-hours and specialised workforce [1].

High-Level Synthesis (HLS) tools offer an alternative development flow for FPGAs using high-level software codes as input, typically C/C++. These tools contribute on reducing the programmability gap to software development, which in turn allows a broader usage of FPGAs with less specialised workforce and less man-hours. However, multiple studies [2, 3] have shown that optimisation effort on the code is essential to achieve reasonable results when using HLS. It mostly comprises of compiler pragmas/directives and code manipulation, often quite related to the FPGA world. This goes against the primary goal that HLS seeks to achieve: the abstraction of the FPGA world for the high-level developer.

Most HLS frameworks provide off-the-shelf optimisation features (e.g. loop unroll, pipeline, array partition) that can provide significant speed-up and increased energy efficiency when properly applied. Considering that the synthesis (compilation) of a single hardware design takes a non-negligible time to complete, evaluating all possible combinations of options may be unfeasible due to large combinatorial spaces. A Design Space Exploration (DSE) methodology is suitable in this scenario.

This paper presents a fast estimation-based DSE approach that optimises the execution time of a high-level C/C++ kernel. The estimator at the core of this approach is named Lina, first being presented here [4, 5] and in more details here [6]. Lina allows the estimation of high-level kernels while also considering common HLS optimisations such as compiler pragmas. Lina runs orders of magnitude faster than FPGA synthesis, allowing a fast and parallel traversal across large design spaces composed of combination of pragmas. This allows a more automated and faster approach towards finding the best optimisation directives for a given high-level code.

Our approach is able to estimate resource usage of the design points, which in turn allows an exploration that optimises not only performance but also resource footprint. In this paper however, focus is given on performance optimisation. The resources estimations are only used as tiebreaker criteria. For a more comprehensive analysis on Lina's resource-aware explorations, refer to the previous paper [5].

Lina's timing model allows the exploration of different clock frequencies, and the resource model supports both floating-point and integer datapaths. Lina is based on Lin-analyzer [7]: it inherits the trace-based scheduling while adding features such as support to non-perfect loop nests, different operating frequencies and a lightweight model for resource estimation.

The version of Lina here presented includes a memory model that is able to estimate the latency of off-chip transactions, including common memory patterns or features such as coalescing, data packing and memory banking. This is essential for real-case use, since input/output data of compute kernels is often located on off-chip memories.

In addition, Lina has been updated to include a new caching logic that reduces the amount of I/O accesses during parallel exploration. That allows multiple parallel exploration threads to be executed with significantly less I/O bottlenecking.

We validate our memory optimisation model by two test cases. First, a simple convolution kernel is passed through our framework, on which loop unroll, pipeline and array partitioning are explored. The input arrays are marked as off-chip, and thus Lina estimates their accesses considering the off-chip memory model. Speed-ups of at least 720× were reached, as compared to the baseline code used for HLS.

Our second experiment considers 4 kernels from the Parboil benchmark [8], also with off-chip access. The selected optimisations resulted in speed-ups up to 3.2× when compared to the baseline applications on the same platform and using the same HLS compiler. If only the computation loop is considered while disregarding the initial and final data transfers between host and device memory spaces, speed-ups are up to 24.2×.

This paper is structured as follows: in Section 2 we present a summarised background on pertinent subjects, Section 3 presents relevant related work, Section 4 describes the proposed off-chip memory model, Section 5 the model's caching mechanism, Section 6 presents the experimental setup, followed by the validation results in Section 7. Section 8 discusses the results. Finally, Section 9 presents our final considerations and concludes the paper.

2 Background

Given a high-level sequential code, the HLS compiler analyses it and extracts a dependency graph of the operations that compose the code. This graph is used to identify which operations may execute in parallel while respecting the dependencies between them. At the end, the HLS compiler generates an RTL source composed of a finite state machine that orchestrates multiple instances of Functional Units (FUs). These FUs perform simple operations (add, multiply), and are used to execute the operations identified from the high-level code.

Figure 1 presents an example of m-to-n reduction, on which an array of m elements is reduced (i.e. elements are summed) into an array of n elements. If the code is supplied without any directives (i.e. without the `pragma` directives), the resultant design has a latency of 7021 clock cycles. By using the pragmas as presented on the figure, the latency drops to 70 clock cycles, a speed-up of 100×. This shows the importance of using compiler directives to assist the compiler on the right direction.

```
#define M 1000
#define N 100
#define R (M / N)

void example(float A[M], float C[N]) {
#pragma HLS array_partition variable=A block factor=(2*R)
#pragma HLS array_partition variable=C complete
    for(int j = 0; j < R; j++) {
#pragma HLS pipeline
        for(int i = 0; i < N; i++) {
            C[i] += A[j + (R * i)];
        }
    }
}
```

Fig. 1 Code example for m-to-n reduction: m elements from an array are accumulated into n output elements. The code is annotated with compiler pragmas that significantly improves performance of the HLS output.

The amount of possible combinations of pragmas can easily explode in size, and evaluating each with the HLS compiler itself may be unfeasible. In some cases, the HLS compilation per se (without full FPGA synthesis) can take hours [7]. Moreover, even with hardware expertise it might not be straightforward to decide the best combination of pragmas. When pipeline is enabled, for example, it is not trivial to infer proper unroll and partition pragmas that adequately provide enough read/write ports to avoid memory contention. A tool that automatically performs such exploration and suggests the best combination of pragmas is of great interest.

2.1 Design Space Exploration

The approach adopted in this paper for inferring the best combination of compiler directives is through design space exploration. Figure 2 presents the approach. A coordinating script iterates through a selection of possible directives, dispatching parallel executions of the Lina estimator for each combination. At last, the best identified combinations are provided to the user. Multiple solutions are possible, since this exploration can be multi-objective (e.g. to minimise design latency and FPGA resources).

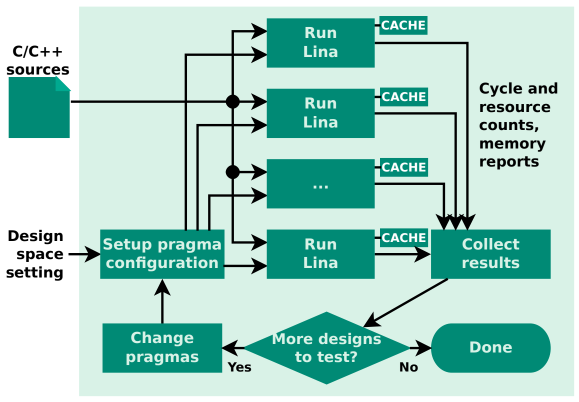

Fig. 2 Typical flow of our approach. Combination of pragmas are selected by a dispatcher script, which configures and spawns parallel instances of Lina for each. At the end, the approach selects the best solutions according to the objectives to be minimised.

Each Lina execution is independent, and thus multiple instances can run in parallel. In addition, our approach implements a caching mechanism that speeds up Lina across different iterations.

2.2 Timing and Resource Awareness

From the flow in Figure 2, the design points are estimated for cycle and resource counts, and are used to decide the best outcomes. Our approach is considered timing-aware because it allows multiple input clock frequencies to be tested, supporting values from a continuous frequency range. Our approach is resource-aware because it contains several models that allow the estimation of FPGA resource counts such as LUTs, FFs, DSPs and BRAMs. The resource estimations can be used in conjunction with estimated latencies to provide a set of solutions to the user. These may be performance-optimised, resource-optimised, or a balance in between.

The timing awareness is achieved mainly by adjusting the estimation according to a selected input frequency. In fact, Lina supports the input frequency in a similar fashion as any other compiler directive.

In general, HLS compilers provide a set of configurations for each supported FU. These configurations vary in: number of clock cycles needed to perform a full operation (i.e. latency, or l); and in amount of time needed within a clock period to perform the intermediate steps (critical path delay, or t_{cp}). Lina replicates this behaviour by including a hardware profile library that contains all these configurations. Frequencies supported range from 16.66MHz to 500MHz. For each FU used within an analysed code, Lina selects the best configuration according to the provided clock frequency. The hardware profile library was constructed beforehand and once for each supported FPGA platform through a binary-traversal-guided microbenchmarking.

It is important to note that not always the best performance is located at the highest frequency, which motivates Lina's timing awareness. The higher the frequency, the tighter the timing budget is. If an intermediate step of a multi-cycle FU takes longer than a clock cycle in such scenario, this step must be split into further clock cycles. Thus, increasing the frequency may increase the FU's latency. Furthermore, high frequencies tend to put more pressure on the FPGA's routing mechanism, since long paths become more critical under tight timing budgets.

Reducing resource footprint of an FPGA design is also advantageous: it eases routing due to less resources competing for routes, and it opens more space for other parts of a project to share the same FPGA.

Using more resources does not always imply in direct gain of performance. In some cases, applying unroll or pipeline to an HLS design incurs in equivalent performance but with the latter being more resource-efficient. Consider the code in Figure 1 but using only a single pipeline OR full unroll directive at the inner loop. Table 1 presents the outcomes considering each optimisation. While performance is roughly equivalent, the pipeline version is more resource-efficient.

Table 1 HLS outcomes for code in Figure 1 with pragma only at innermost loop (resources estimated by Vivado HLS).

Pragma	Full Unroll	Pipeline
Design latency	1001	1007
LUTs	3661	468
FFs	3984	494
DSPs	4	2

Lina implements multiple resource models. Thus, it allows identifying and discarding resource-hungry points that do not contribute in better performance, as exemplified in previous paragraph. Lina's resource awareness include:

- Estimation of resources used by FUs, sensitive to compiler directives and input clock frequency;
- Estimation of resources used by on-chip arrays, sensitive to array partitioning directives;
- A complete resource estimation considering both calculations above, and also the resources used by other parts of the design, like auxiliary logic (with equations adapted from [9]).

The timing and resource awareness can be found in detail in [5] and [6].

2.3 Memory Awareness

Modern FPGAs include interfaces to communicate with off-chip memories such as DDR SDRAM, HBM, etc. These are essential for communication with the outer world, for example to integrate on a heterogeneous system.

Differently from on-chip memories, off-chip latencies are affected by peculiarities of each memory technology, and may be severely affected by access patterns. Therefore, optimisation approaches considering on-chip accesses may not reflect in similar gains when considering off-chip transactions [10].

A simplified and agnostic model to consider off-chip memory read/write is composed of three steps: `setup`, `action` and `commit`. An off-chip memory transaction starts with the `setup` phase, that prepares the terrain for the memory access (e.g. row fetch from DDR memory). Then, successive `action` phases can be triggered, each performing a single read/write on incremental memory positions. To finish the transaction, a `commit` step may be required. Usually, the `setup` or `commit` phases are longer than the `action` phase. The long steps for example can be related to buffer flushes on the off-chip memory. Depending on how the memory architecture is constructed, not all steps may be required for read or write. Considering this model, the total latency to perform a single off-chip transaction is given by c_{total}^{mem}:

$$c_{total}^{mem} = c_{setup}^{mem} + (n \times c_{action}^{mem}) + c_{commit}^{mem} \tag{1}$$

where c_{setup}^{mem}, c_{action}^{mem}, and c_{commit}^{mem} are the number of clock cycles required to perform the `setup`, `action` and `commit` phases, respectively, and n is the number of contiguous `action` phases to be performed in a coalesced manner.

Lina considers the model above as a base for its off-chip estimation logic. Thus, it is able to reflect its estimation considering not only the combination of directives, but also considering off-chip access. Lina looks for coalescing opportunities, attempts to perform data packing, and reports missed optimisation attempts.

2.4 Trace-based Estimation

In order to identify the parallelism potential of a high-level code, the HLS compiler uses dependency information generated statically, such as the Program Dependency Graph (PDG) [11]. Depending on the code's complexity, performing HLS compilation can take a non-negligible time. For exploration purposes, the design metrics of HLS compilations are of more interest than the complete generated design themselves. And these design metrics can be reasonably approximated by simpler models, instead of running the whole HLS compilation process.

Approximation models that use dependency graphs can be divided in static or dynamic approaches. Lina is a dynamic-based approach. That is, a profiled execution from the input high-level code is used to: identify dependencies; construct dependency graphs; and perform estimation. These dependency graphs will be henceforth called Dynamic Data Dependency Graph, or DDDG [12].

The advantage of using dynamically-generated structures like DDDGs is that global code motion optimisation is inherently enabled. All control and false data dependencies are implicitly resolved by the execution that generates the software trace. This provides an optimistic notion of the parallelism capabilities of the code, which can then be constrained to reflect realistic parallel architectures [12].

While Lina has additional features as compared to its base implementation Lin-Analyzer, both share the same estimation concept: the trace generated from profiled execution is used to construct DDDGs. Then, a model that approximates the HLS behaviour performs scheduling of the DDDG. At last, an estimation of the design latency can be derived. On Lina's case, the scheduling is also used to estimate FPGA resources.

Figure 3 presents an example of a DDDG, its scheduling and the latency derived from it. The DDDG represents a single iteration of a floating-point vector add, and the numbers located near each node represent the latency required to execute each node. Nodes with latency zero are not synthesised and are optimised away during schedule.

Lina's estimation model is separate from the HLS compiler scheduling models. Inaccuracies in the estimation will not cause the HLS compiler to generate functionally incorrect designs.

3 Related Work

There are several contributions to the HLS estimation field, and most can be divided in static [13–15] and dynamic [11, 16] approaches. The

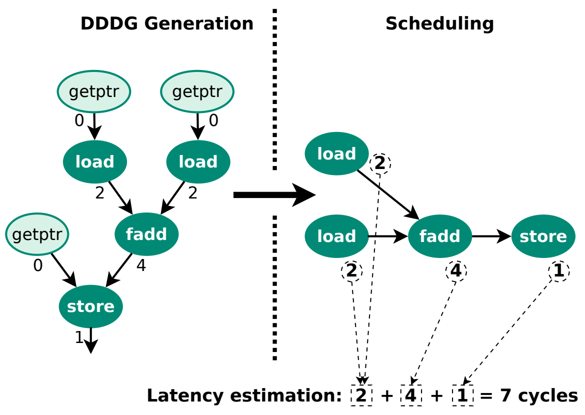

Fig. 3 Example of DDDG for a vector add (single iteration), a schedule solution for it, and the latency derived from the resultant schedule. Adapted from [6].

difference between them is the source of information used to perform estimation. Static approaches rely on information acquired solely through the high-level code, whereas dynamic approaches use traced executions to extract information useful for estimation.

Similar interest exist on the DSE field, with several contributions over the last years [17–20]. Relevant aspects to consider are: whether synthesis is required to evaluate each design point of the exploration or not; whether design frequency is supported as an exploration knob; and if the resource estimation is used not only to guide the exploration, but also as exploration objective. In addition, there are multiple HLS researches acknowledging that off-chip memory access must also be considered in the models for real world use cases [10, 21–23].

FlexCL [21] is an analytical performance estimation for HLS using OpenCL with average absolute error below 10%. It is composed of a computation and a memory model, that together are used to approximate latency. Its memory model is able to identify eight off-chip memory patterns. FlexCL focuses on a different language and execution model than the one focused on this paper, and it assumes a fixed operating frequency. The hardware profile library from FlexCL assumes an average latency for each operation considering different frequencies. This adds up inaccuracy, and its estimation is still nonetheless invariant to frequency.

Rajagopala et al. [10] presents Volcan, a methodology that performs source-to-source transformation on high-level code for HLS. Volcan analyses the code in search for potential coalesced and packed off-chip memory access, towards reducing the design latency. This approach also performs DSE by varying compiler directives. Results indicate performance gains on 24 out of 27 test cases, with speed-ups ranging from 1.5× to 11×. Although a design space pruning technique is performed, each design point must be synthesised and tested on board for acquiring runtime values.

Dávila-Guzmán et al. [22] presents an analytical model to estimate performance of memory-bound application on Intel FPGA OpenCL platforms. For 9 memory-bound applications, the estimation error remained below 9%. This approach uses the RTL code generated from HLS to assist on its estimations. Since in some cases HLS compilation may also take considerable time [7], it may not be viable for large design spaces.

Sherlock [23] is a multi-objective design exploration approach. It implements active learning, on which multiple regression models are continuously tested, and Sherlock uses the most promising results on the way to guide its exploration. Each design point is evaluated using design synthesis, and thus exploration reaches dozens of hours. The authors affirm that Sherlock is well-suited for applications on which the design space's characteristics are not known.

AutoScaleDSE is a design space exploration engine proposed by Jun et al. [24] that explores multiple interacting loop nests, and then it finds the best solution that satisfies the whole code. AutoScaleDSE performs an initial exploration for each loop nest, then it uses a random forest classifier and a genetic algorithm to merge the results and create a global optimised solution. During the process, one or more HLS compilations may be needed in order to fine tune the result, and the whole process might take few hours. In exploration of large-scale applications, a maximum speed-up of 12× is reported. AutoScaleDSE tackles a much broader problem by solving the DSE for more than one loop, while our approach is more focused on solving a single loop. In its core, AutoScaleDSE uses the ScaleHLS [25] framework to both apply code transformations on the code, and to perform the initial DSE of each loop nest. ScaleHLS' design exploration uses a latency/area estimator named "QoR estimator", which shares many aspects from our approach. For example, it is also synthesis-less and uses similar calculations as Lin-analyzer and Lina (such as ALAP scheduling), although it is an analytical model-based approach. In addition, it lacks a comprehensive off-chip memory model as presented in this paper, focusing the estimations on the on-chip accesses. In fact, our approach could be complementary to ScaleHLS by adding a finer-grain approach to the off-chip memory scheduling. The QoR estimation supports different clock frequencies, although only 100MHz is currently implemented, and it is not part of the DSE itself as in our approach.

In summary, most contributions presented above make use of design synthesis or HLS compilation to guide their explorations. Although design space pruning is used, the exploration may still reach multiple hours for just a single case. If compared to other synthesis-less approaches - such as ScaleHLS' "QoR estimator" - our solution provides a more comprehensive model for off-chip memory analysis and scheduling. Lina leverages dynamically-acquired information to analyse and find potential off-chip optimisations, not requiring any lengthy compilation step to guide its exploration.

4 Off-Chip Memory Model

This section details the off-chip memory model added to Lina. This model is used in three steps:

- **First stage DDDG transform:** load/store nodes in a DDDG that access off-chip region are substituted by one or more nodes related to off-chip transactions (e.g. `setup`, `commit`);
- **Second stage DDDG transform:** a memory trace is used to identify optimisations such as coalescing, data packing. Then the nodes added at the first stage are modified according to identified optimisations;
- **Scheduling:** if Lina reaches an off-chip read-/write candidate node during scheduling, it consults the memory model to know if this current candidate can be scheduled or not. The memory model considers multiple aspects to take such decision, for example: the number of ports available, ongoing coalesced transactions and memory policy.

Figure 4 presents an example of DDDG transformation being applied to some nodes. First, the user indicates to Lina which arrays should be considered off-chip. Then, all load/store nodes related to these arrays are transformed similar to the example presented.

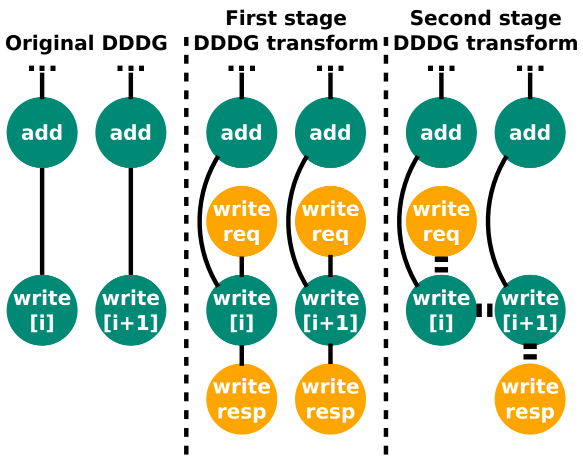

Fig. 4 Example of DDDG transformations for inserting off-chip transactions. First, loads and stores are substituted by off-chip equivalents. Then the memory model identifies a coalescing potential and merges both identified off-chip transactions (coalesced transaction indicated by thicker dashed-edges).

The traced execution of the input kernel is also used to generate a memory trace, containing all accessed data addresses. This is used in conjunction with the memory model to decide the scheduling of the off-chip accesses.

The next subsections present the memory model features and behaviour, such as potential memory optimisations, how they transform the DDDGs, scheduling policies, and interaction with pragmas. When Lina finishes, it presents a memory report to the user indicating the optimisations that were detected, and the ones that were blocked along potential reason. Appendix A.1 presents a report snippet.

4.1 Memory Optimisations and Second Stage DDDG Transform

After first stage, Lina performs multiple analyses in order to identify potential memory optimisations and then transforms the DDDGs during second stage accordingly. Three optimisations are evaluated by Lina: burst, data packing, and memory banking. Figure 5 presents examples of these optimisations, which are detailed in the following sub-sections.

4.1.1 Burst

Two types of bursts are analysed: intra- and inter-iteration. In the first case, Lina searches for contiguous off-chip transactions within a loop iteration. If any is found, the generated memory trace is used to verify whether the coalescing pattern applies to other loop iterations. If positive, Lina performs intra-iteration burst by grouping the off-chip transction into a single transaction. An example of this DDDG transform is presented in Figure 5.

Intra-iteration burst reduces latency by grouping coalesced transactions, but the time-consuming steps (e.g. `setup` and/or `commit`) are still executed at every iteration. On the other hand, inter-iteration bursts reduce latency even further by running the time-consuming steps of an off-chip transaction just once before and/or after the loop body. The second column in Figure 5 presents a code snippet on which off-chip transactions are coalesced across successive loop iterations. In this case, a single `setup` can be executed prior to loop execution ("pre-loop" region on figure), then successive `read` or `write` calls are performed. Finally, the `commit` step is performed after loop is executed ("post-loop" region on figure). Lina uses the memory trace to identify if such pattern is possible within the code being analysed.

4.1.2 Data Packing

Modern off-chip memories usually provide wide interfaces, for example GDDR5 that is 256-bit wide. Common scalar data types have in general no more than 64 bits. In order to make full use of wide data buses, the developer must adapt the code to use vector types (e.g. `float4` type on OpenCL). This can also be performed as an optimisation in some HLS compilers. Intel FPGA's OpenCL, for example, provides the `num_simd_work_items` attribute that attempts to pack scalar computations within a kernel automatically.

If data packing is possible for a certain array in a certain region of a code, it must also be applicable to every other place where the same array is used. Consider for example the code snippet at the "Data packing" column of Figure 5. Its computation pattern allows the array to be always accessed as a vector element with four packed elements. Data packing is possible in this case. Now consider that this snippet is wrapped by an outer loop that accesses array `A` as a single element per iteration (e.g. `A[k] = ...;`). If the HLS compiler or platform being utilised does not allow masking packed writes (that is, writing only certain elements of a vector element back to memory), then packing is not possible. The innermost loop would allow a vector type with 4 elements, but the outer loop requires access as single elements.

Lina evaluates all generated DDDGs for a single loop nest in search for potential packed off-chip accesses. For each evaluated array within a DDDG, Lina stores the potential packing values. Then, it selects the largest common value among all DDDGs. This ensures that if data packing occurs, it is consistent throughout all evaluated code regions. The selected value is used to transform the DDDGs as exemplified in Figure 5. In this example, the array is being accessed as four contiguous (and bus aligned) transactions at every iteration. After data packing, the first three `action` steps are silenced (i.e. they have their latency reduced to zero). Only the last `action` step is left unchanged. During scheduling, all four `action` steps will be scheduled at once, which is latency-wise equivalent to a data-packed off-chip transaction.

The report snippet in Appendix A.1 presents an example of messages reported during data packing analysis.

4.1.3 Memory Banking

On-chip arrays are implemented on FPGA with separate read/write interfaces, allowing parallel access to them. Off-chip arrays, however, are dependent on how the memory access is configured and physically implemented. If just a single memory chip is connected, memory access for any off-chip array will compete for the same interface. Some architectures are able to implement memory

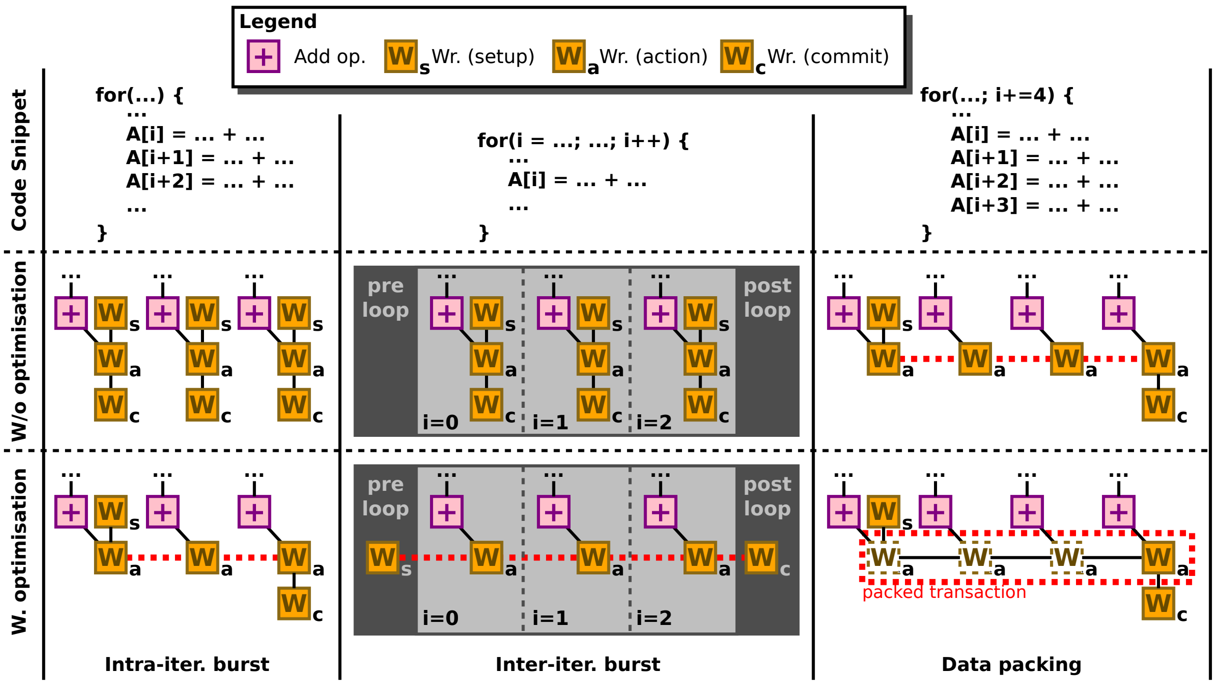

Fig. 5 Examples of code snippets, their DDDG equivalents without and with the optimisations indicated underneath. The thicker dashed lines indicate the optimisation paths created. Nodes with dashed borders are silenced (i.e. latency is zero).

banking. That is, off-chip arrays are placed on separate memory banks, which in turn allows parallel access similar to on-chip arrays.

Lina replicates memory banking by analysing each array as a separate memory space with separate interfaces if enabled. When disabled, all off-chip arrays are treated to be in the same memory space, sharing the same interface.

4.2 Interaction Between Multiple Transactions and Pragmas

Off-chip transactions occupy multiple cycles, and they may or may not overlap depending on several conditions. Every time an off-chip node enters the scheduler, the off-chip memory model is consulted on whether it is possible to schedule that node in current cycle or not. The decision making is based on a memory policy, that may vary between HLS compilers and platforms.

Since we use Vivado HLS as base for Lina, we include two memory policies that are based on off-chip memory behaviour of Xilinx Zynq UltraScale+ system (our test platform). The first policy (`permissive`) allows multiple off-chip transactions to occur in many scenarios, whereas the other policy (`conservative`) is more restrictive on allowing overlaps. Details for each policy are available in Appendix A.2.

We could not draw a perfect conclusion on what triggers each policy on Vivado HLS, but we noticed however that when a read is performed after a write to a same off-chip array, the `conservative` policy is automatically applied by Vivado HLS. If banking is disabled, a single off-chip read after any off-chip write suffices. We name this pattern as `read-after-write` for further reference. Lina is able to identify such pattern, and emits a warning for the user that performance degradation is expected due to conservative scheduling policy. These warnings are presented on the generated memory report, along with potential solutions.

Not only multiple off-chip transactions affect themselves, but optimisation pragmas do as well. The following sub-sections present how loop unroll and loop pipeline affect off-chip scheduling.

4.2.1 Loop Unroll

Unrolling a loop virtually replicates its body. Thus, off-chip transactions within the loop are also replicated. How this may impact in performance is dependent on the HLS compiler.

To test the off-chip transactions in our Xilinx Zynq UltraScale+ platform, we used Xilinx SDSoC toolchain, that uses Vivado HLS as its core. Enabling loop unroll generally incurred in performance degradation. Off-chip optimisations that were being previously detected were disabled due to unroll. This happens for example if both read and writes are included within the loop body. When it is unrolled, the virtual code replication causes a `read-after-write` pattern, that switches the memory policy to `conservative`, in turn disabling coalescing.

As an example, consider a very simple loop with one input array and one output array being accessed in a monotonic manner (e.g. `B[i] = A[i]; i += 1`). Without any unroll directive, inter-iteration burst is detected. With unroll enabled, the `read-after-write` pattern occurs and severely degrades performance. As an additional test, we attempted to manually unroll the code (i.e. copy and pasting the loop body) while positioning all read operations before the write operations. In this case, we were able to circumvent the `read-after-write` pattern, but it nonetheless did not detect inter-iteration burst as in the code with no unroll. From all three cases, the code without unroll provided the best performance. There seems to be some missing exchange of information between the SDSoC and Vivado HLS optimisers.

Lina is able to detect such patterns and issue warnings if a loop unroll may cause a negative impact. In this case, loop unroll is only indicated when it facilitates other optimisations, such as coalescing with data packing.

4.2.2 Loop Pipeline

In order to calculate the latency of a loop when pipeline is enabled, Lina uses the calculations from [7, 26, 27], on which the initiation interval II is approximated by a "minimum initiation interval" MII. The value MII is calculated based on two factors that often limit the reachable minimum value of II: recurrence and resource constraints.

The $ResMII_{mem}$ is the resource constraint MII related to memory constraint. For on-chip arrays, Lina/Lin-analyzer calculate as follows:

$$ResMII_{mem} = \max_{m} \left\{ \left\lceil \frac{Nr_m}{Pr_m} \right\rceil , \left\lceil \frac{Nw_m}{Pw_m} \right\rceil \right\} \quad (2)$$

where: Nr_m and Pr_m are the number of reads within a DDDG and number of read ports for array m; Nw_m and Pw_m are the respective counterparts for write transactions.

This value builds upon the following rationale: in general, load and store nodes are either start or endpoints of a DDDG. Load nodes without dependency can be all moved and compacted to the beginning of the DDDG schedule, while store nodes that are terminal can all be moved to the end of the schedule. A limited number of values can be read/written at each clock cycle, as defined by the Pr_m and Pw_m values. Thus, if all load/store nodes are compacted to the edges of a schedule, they can be processed in $\left\lceil \frac{Nr_m}{Pr_m} \right\rceil$ cycles for read, and $\left\lceil \frac{Nw_m}{Pw_m} \right\rceil$ cycles for write. The largest between both values — and the largest among all arrays — is considered the MII related to memory resource constraint.

This, however, does not apply for load and store nodes that are dependent on other load/store nodes. This can happen for example if the HLS compiler is not able to statically infer an array read's index after a store has happened for the same array. In this case, the compiler takes the conservative approach of considering a dependency between the last store and the load. This indeed reduces the movement of both operations within the schedule. That is, they cannot be moved to the starting or ending of a schedule window in order to reduce II if pipelined. Lin-analyzer takes this into account for on-chip arrays by using an adjustment factor when calculating $ResMII_{mem}$. In our case, such calculation was not enough when considering off-chip arrays. Figure 6 presents two examples of pipeline allocations, one without any dependent load/stores, and one with a load/store dependency path.

Lina implements an improved calculation for $ResMII_{mem}$ that considers both on and off-chip transactions. We tested several different access

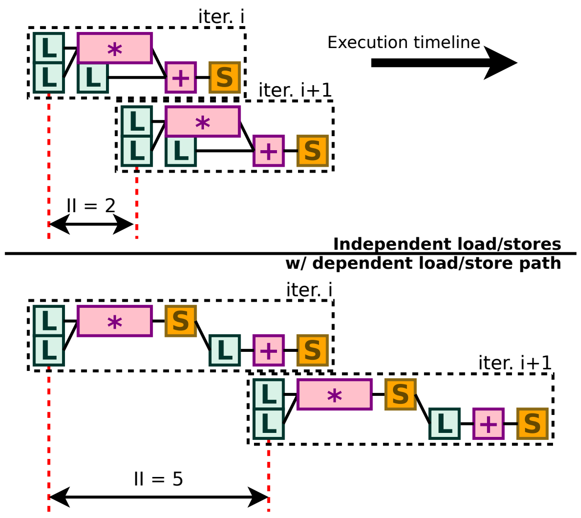

Fig. 6 Two examples of pipeline allocations. Each dashed box represents a single iteration of the pipelined loop. On the first example (above), no dependency exists between loads and stores, and the II is constrained by the memory resource (i.e. II of 1 would overlap too many loads to the same array, exceeding port budget). In the second example (below), there is a dependency path between loads and stores, and thus a larger II is allocated in order to respect the dependency. Adapted from [6].

patterns on Vivado HLS (via SDSoC), that led to a more generalistic equation as follows:

$$ResMII_{mem} = \max\{ResMII^{port}_{mem}, ResMII^{rec}_{mem}\} \tag{3a}$$

$$ResMII^{port}_{mem} = \max_m \left\{ \left\lceil \frac{Nr_m}{Pr_m} \right\rceil, \left\lceil \frac{Nw_m}{Pw_m} \right\rceil \right\} \tag{3b}$$

$$ResMII^{rec}_{mem} = \max_m \{\Delta' r_m, \Delta' w_m\} \tag{3c}$$

$$\Delta' r_m = (\Delta r_m + Cr_m) * \lambda_m \tag{3d}$$

$$\Delta' w_m = \Delta w_m + Cw_m \tag{3e}$$

$$\lambda_m = \begin{cases} 0 & \text{if } m \text{ is on-chip} \\ 1 & \text{if } m \text{ is off-chip} \end{cases} \tag{3f}$$

where $ResMII^{port}_{mem}$ is the MII value constrained by memory port, $ResMII^{rec}_{mem}$ is the MII counterpart constrained by dependent memory accesses, Δr_m is the largest schedule distance between dependent read transactions for memory interface m, Cr_m is the number of connected read dependency paths for memory interface m, Nr_m and Pr_m are the number of reads and number of read ports for memory interface m. All variables have their respective counterparts for write transactions, respectively: Δw_m, Cw_m, Nw_m, and Pw_m. This whole new calculation wraps around the previous value of $ResMII_{mem}$, that now became $ResMII^{port}_{mem}$.

The concept of “memory interface” is dependent on whether it is on or off-chip. For on-chip, each array has its own set of block RAMs and dedicated memory ports, thus each array has its own memory interface m. For off-chip arrays, if banking if disabled, all arrays share the same memory interface. With banking, off-chip arrays have their own interfaces similar to on-chip arrays.

Lin-analyzer only applies its adjustment factor to write transactions. For read transactions, the memory recurrence constraint as shown in (3) is not considered. Lina follows a similar approach: for on-chip transactions, the $ResMII^{rec}_{mem}$ value discards the read transactions through the $\lambda_m = 0$ multiplier.

These equations are calculated by Lina as follows: first, it counts all read/write transactions within the DDDG and calculates Nr_m and Nw_m. Data packing must be taken in consideration (e.g. a packed transaction of 4 loads should be seen as a single increment on Nr_m, not 4). Then during scheduling, Lina keeps track of all load/store nodes that are being scheduled in order to identify dependent load/store nodes. At the end of scheduling, Lina has a list of all paths with dependent loads and stores, which is used to calculate Δr_m, Δw_m, Cr_m and Cw_m. Finally, $ResMII_{mem}$ can be calculated. An example of this calculation, along with more explanation on Cr_m and Cw_m can be found in Appendix A.3.

5 Exploration Caching Mechanism

One notable overhead of our estimator is that dynamic traces are often very large, and points of interest for DDDG generation are usually scattered through the compressed trace file. That incurs in large overhead for reading and parsing the trace entries until these points of interest are found. This was partially handled on previous work [5], where a trace cursor cache was created to store known positions on the compressed file once they are reached. When needed, Lina simply reads the cache and if there is a hit, it seeks directly to the point of interest using the cached cursor, without having to re-parse large parts of the trace for every design point.

When we started to increase the number of parallel calls of Lina during kernel explorations (i.e. multiple design points evaluated at same time), we noticed that the trace file seek operations also started to become a bottleneck. Due to the decompression mechanism, the further away a seek operation points to, the longer it takes to reach that point. Since there is data locality across design points - that motivated the existing trace cursor cache - keeping on memory these regions that are frequently accessed can significantly reduce the bottleneck.

This new caching mechanism is composed of a daemon application named `linad`. This daemon maintains a shared memory area where these frequently-accessed regions are stored uncompressed. Instead of each Lina instance opening the trace file by itself, it communicates with an API that provides a file-like interface. The API uses Inter-Process Communication (IPC) to interact with the daemon, which is then responsible by seeking and decompressing further parts of the trace file when there is a miss, or to provide a shared memory pointer when there is a hit. Due to this mechanism, less seek and decompression operations are performed for each exploration. Figure 7 presents the daemon and how Lina jobs interface with it.

6 Experimental Setup

This section presents the experimental setup used to validate the off-chip model as part of design space exploration. We present the hardware platform, toolchain, and test kernels used as DSE inputs.

As opposed to the previous paper [5], we do not perform a pareto analysis between resource footprint and performance. Our focus here is on performance, and the resources estimations are only used for tiebreaking when needed.

6.1 Hardware/Software Platform

Our test platform is a ZCU104 development kit containing a Xilinx Zynq UltraScale+ device. The Zynq UltraScale+ architecture is composed of two

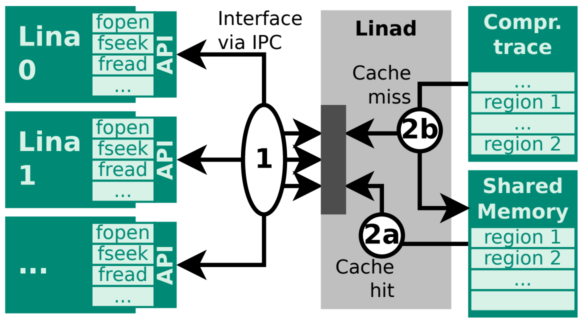

Fig. 7 Caching mechanism structure showing multiple Lina calls, the daemon, the compressed trace file, and a shared memory region. The multiple instances use a file-like interface to communicate with an API, which in turn uses IPC to communicate with the daemon (indicated as **1**). If the requested trace region is present on the shared memory (a hit), a pointer to this region is returned to the API (indicated as **2a**). If not present (a miss), the daemon decompresses the requested region, stores it on the shared memory and returns the pointer (indicated as **2b**).

major components: an ARM-based architecture (Processing System, PS) and an FPGA region (Programmable Logic, PL), both with strong coupling. This architecture is quite suitable for HLS solutions, as it allows software to be executed on the PS while compute-intensive tasks are offloaded to custom hardware on PL.

The SDSoC toolchain offered by Xilinx complements this architecture by supplying a user-friendly approach on such offload. Via C/C++, the developer is able to mark functions as to be offloaded to PL. These are synthesised via HLS, and automatically coupled to the remaining of the code that executes on PS.

OpenCL is also supported. In this case, the PS executes the host code, whilst the OpenCL kernels are offloaded to the PL using HLS. The integration happens in a similar fashion as to C/C++, but OpenCL-conformant. Although Lina is tailored for C/C++, in this paper we use OpenCL as the input language, but solely for its facilitations in regard to access to off-chip memory. An array can be stored and accessed from off-chip memory simply by adding the `__global` keyword to the array's declaration. Furthermore, OpenCL also facilitates data packing by providing packed data structures, such as `float4` or `int4`. Although OpenCL supports other execution models for kernels — such as the SIMD pattern using NDRange — we only use the single-task approach. In this case, an OpenCL kernel is executed likewise a C/C++ function[1].

The kernels that we use in the experimental setup are originally in C/C++. These are used for exploration with Lina. Then, simple modifications are made in order to transform them to OpenCL, since the single-task execution model is used. The modifications are mostly comprised of adding OpenCL-specific keywords and creating the OpenCL host initialisation code for the kernels and related buffers. The kernel body is left practically intact.

A power sensing testbench was created in order to measure the consumed energy from the kernels. Appendix B presents the testbench.

6.2 Exploring a CNN Layer with Off-chip Access

Convolutional Neural Networks (CNN) are particular applications that have a code structure suitable to parallelism, and also have a large memory footprint for inputs and outputs. There are several studies focused on using smaller on-chip buffers and loop reordering/tiling [17, 28, 29]. Their primary focus is to optimise the access to off-chip memory while maximising the usage of on-chip resources.

Figure 8 presents the basic loop nest of a CNN kernel. It has a suitable code pattern for memory optimisation, which we perform using Lina on a single CNN layer. The design spaces (detailed in Appendix C) are composed of 680 valid points.

```
LOF: for(auto m = 0; m < M; m++)
  LIF: for(auto c = 0; c < C; c++)
    LSY: for(auto y = 0; y < E; y++)
      LSX: for(auto x = 0; x < E; x++)
        LFY: for(auto k = 0; k < E; k++)
          LFX: for(auto l = 0; l < E; l++) {
            auto p = I[c][y * S + k][x * S + l];
            auto w = W[m][c][k][l];
            O[m][y][x] += p * w;
          }
```

Fig. 8 Basic loop nest of a CNN kernel.

We explored two variations of a CNN loop nest, differing on how memory accesses on border cases are handled. The first version — `padmemory` — has padded buffers in such way that all memory accesses are always within array bounds. The second version — `padlogic` — uses unpadded buffers and conditionals to check whether a memory access would be off-bounds or not.

The input arrays `I` and `W` are left off-chip, while the output array `O` is buffered on-chip during execution. The results from `O` must be transferred back to the off-chip memory after the CNN layer completes. We leave memory banking always enabled.

Since this code has a rather simple loop body, we created an unroller tool to explicitly unroll the loop nest. First, we splitted the loop body in such way that all reads were placed before writes and added special annotations to guide the unroller tool. Then, this tool is able to explicitly unroll the code while keeping all reads before the writes. The intent of this is to evaluate if there is any performance gain on avoiding the `read-after-write` pattern as previously explained. An example can be found in Appendix D. In the remaining of this paper, we refer to this approach using the unroller tool as **explicit unroll**, and to the normal approach using unroll pragmas as **Vivado unroll**.

6.3 Parboil Benchmark

The Parboil benchmark [8] is a suite of applications that are presented with different variants targetting different architectures. We selected 4 applications and used their C/C++ baseline variants as input for DSE and HLS [2].

Table 2 presents the four kernels tested. These kernels were selected considering current Lina limitations. For example, current Lina implementation does not support loop nests with more than one loop per level, arbitrary precision data types or variable loop bounds. See [6] for a detailed discussion.

6.4 Parallel Execution of Lina

As presented in Section 2.1, the dispatcher script simply configures and calls Lina for each design

[1]OpenCL is only used as a facility for the optimisations estimated by our model. Other HLS languages could be used if the same features are provided, like data packing.

[2]Parboil was used in this experiment, since it was suitable for comparison between different architectures. Such comparison is out of scope of this paper, but it was performed and is detailed in [6].

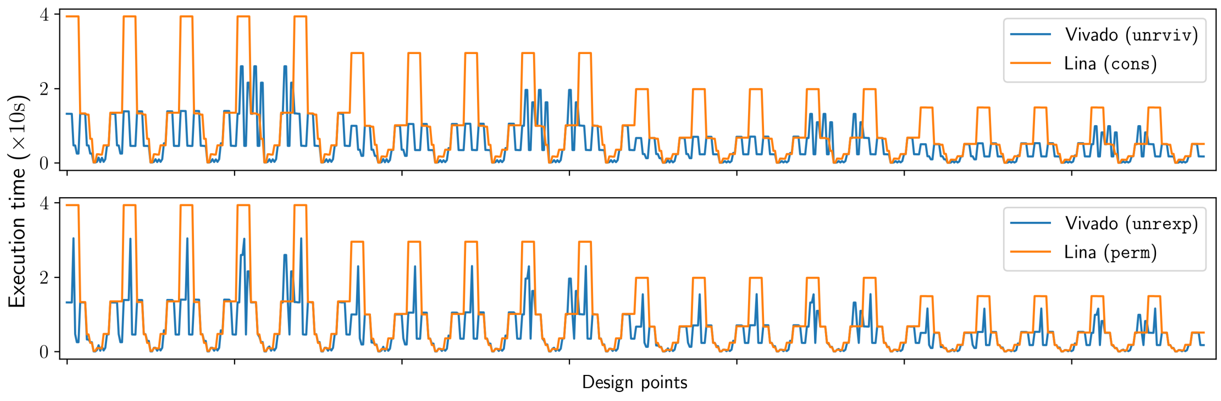

Fig. 9 Design space exploration results for `padmemory`. Horizontal axis presents the design points tested, and vertical axis presents the kernel execution time that was found/estimated for the associated design point. The subplot at top couples Vivado results using normal HLS attributes for unroll with Lina exploration in `conservative` mode. The bottom subplot presents Vivado results using our unrolling tool compared to Lina exploration in `permissive` mode.

Table 2 Parboil kernels tested.

Kernel name	Description
`histo`	Histogramming operation
`lbm`	Lattice-Boltzman Method simulation
`mri-q`	MRI non-cartesian Q matrix
`sgemm`	Single precision general matrix multiply

point. Thus, it can be easily parallelised by dispatching multiple concurrent calls. We perform such exploration and evaluate the total exploration time with different amount of threads. The four Parboil kernels were explored on a Linux computer running Ubuntu, with a six-core Intel i7-9750H CPU and 32GB RAM.

The new caching mechanism as presented in Section 5 allowed our exploration to run with more threads and further reduce exploration time. We adapted the dispatcher script to allow multi-node execution and tested on the Sorgan Cluster, located at the Computer Systems Laboratory (LSC) in UNICAMP, Brazil. This cluster is composed of four nodes with Intel Xeon Gold 6252 CPU (48-core) and 93 GB RAM.

7 Results

This section presents the results related to the experimental setups. We performed many experiments by toggling the memory policies, data packing directives, and unroll approach. On each experiment, the usual HLS pragmas are explored. These experiment toggles are represented by aliases as shown on Table 3.

Table 3 Experiment toggles.

Knob 1: Vectorisation	
`novec`	DSE performed without data packing analysis
`vec`	DSE performed with data packing analysis
Knob 2: Off-chip memory policy	
`cons`	DSE performed using `conservative` policy
`perm`	DSE performed using `permissive` policy
Knob 3: Unroll approach	
`unrviv`	HLS code unrolled using SDSoC directives
`unrexp`	HLS code explicitly unrolled using our tool

7.1 CNN Layer

We ran Vivado HLS for all points in the design space that were traversed by our approach, and collected the actual scheduling results from the compilations. These values are used as a golden model for comparison against our DSE. Figure 9 presents the comparison results for the `padmemory` kernel (estimations vs. golden model). Similar figure for the `padlogic` kernel can be found in Appendix C. We leave data packing / vectorisation enabled on Lina, since we detected that SDSoC was automatically performing vectorisation when possible. Since our unrolling tool allows reorganising the code in order to mitigate `read-before-write`, Vivado is more likely to work similar to the `permissive` mode of Lina. Thus, the bottom subplot presents the exploration using Lina in `permissive` mode compared to Vivado synthesis using our unroller tool (`unrexp`). The converse rationale is also valid. That is, using HLS unroll directives may present similar behaviour as of Lina's `conservative` mode, presented in the top subplot. Although there are visible deviations between the estimated execution times and ground truth, the lower regions of the plots were characterised by Lina with good accuracy. As our goal is to reduce execution time, our greater interest is at the lower regions.

Due to the large interval on the y-axis, it is not visually clear which subplot presents better kernel execution time values. In fact, the most optimal points vary on the millisecond scale. Table 4 presents the kernel execution times for the golden model's best point (`vivbest`) versus our exploration's best pick (`linbest`). For `vivbest`, using `unrexp` brought significant better results in `padmemory` kernel, but slightly worsened performance in `padlogic`. For `linbest`, we reached the optimal for `padmemory` with `unrviv`, but reached 71% of the optimal when using `unrexp`. Results for `padlogic` are similar, however Lina DSE reached 89% of the optimal kernel execution time with `unrexp`. Table 4 also presents the speed-ups achieved by these points as compared to HLS baseline. These values do not consider start/end data transfers, only the compute part (and consequently off-chip accesses during compute). The trends are the same as the execution time values.

Table 4 Achieved optimals by the explorations using HLS for each design point (`vivbest`), and by exploring using our DSE approach (`linbest`).

	`padmemory`		`padlogic`	
	`cons` `unrviv`	`perm` `unrexp`	`cons` `unrviv`	`perm` `unrexp`
`vivbest`	41.29ms	9.83ms	65.87ms	72.75ms
speed-up	241.36×	1013.59×	879.88×	796.65×
`linbest`	41.29ms	13.76ms	65.87ms	81.59ms
speed-up	241.36×	724.02×	879.88×	710.27×

Note that the maximum speed-ups for these experiments are only known because all explored design points were synthesised. In a real application scenario, the maximum is expected to be unknown, as in the Parboil experiments.

Table 5 ranks the design points in terms of speed-up. Our explorations correctly inferred the best frequency, loop unroll and pipeline configurations. It only slightly deviated when selecting proper array partition configurations. For all explorations, the best design points given by Lina DSE were always in the top-10 best points of the explored design spaces.

It is also possible to note from Table 5 that the baseline versions were not in the last rank positions (around 680). This further indicates that some pragmas may result in worse performance than baseline.

Table 5 Design points ranked: baseline, best pick from HLS exploration and best pick from Lina exploration. Pragma factors are presented in parentheses when applicable [6].

	Rank	Freq. MHz	Unroll level	Pipeline level	Array part. `0`
			`padmemory`		
`base`	527	100	off	off	off
`vivbest`	1	200	3 (2)	3	cyclic (4)
`linbest`	4	200	3 (2)	3	cyclic (8)
			`padlogic`		
`base`	546	100	off	off	off
`vivbest`	1	200	3 (2)	3	off
`linbest`	3	200	3 (2)	3	cyclic (8)

7.2 Parboil Benchmark

Figure 10 presents the execution time speed-up for the kernels presented in Table 2. For each kernel, more than one exploration was performed by toggling data vectorisation analysis and memory policy. And for each exploration, two speed-up values are presented: full application speed-up considering host code execution and data transfers (indicated as **full** on figure), and the computation loop speed-up, disregarding data transfers before and after kernel execution (indicated as **compute** on figure).

Due to some explored design space sizes (e.g. `mri_q` with 12800 explored points), it is unfeasible to fully synthesise all points. It is also unfeasible

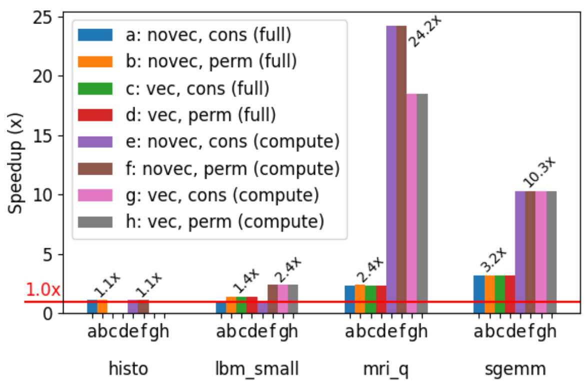

Fig. 10 Performance speed-ups for each kernel. Four experiments were performed using Lina, indicated by a, b, c, d (**full** application speed-up) and e, f, g, h (**compute** loop speed-up). The highest speed-up of each kernel and for each speed-up type is indicated. The horizontal line - labelled 1.0x - indicates the baseline kernel with no optimisation pragmas at 100 MHz. Adapted from [6].

to acquire the full application metrics (execution time, consumed energy), since this step required a semi-automatic procedure for result collection. Therefore we did not compare the Parboil results to ground truth. We believe that Lina validation has already been performed by previous papers and by the convolution results.

Considering the **full** application speed-ups, one exploration actually presented a slowdown (i.e. `novec, cons` exploration of `lbm_small`). Nonetheless, all kernels presented speed-ups when considering the best exploration of each: 1.1× for `histo`, 1.4× for `lbm_small`, 2.4× for `mri_q` and 3.2× for `sgemm`. In fact, most explorations presented nearly similar results for each kernel, apart from the one slowdown in `lbm_small` and two Lina explorations of `histo` that did not synthesise (letters `c` and `d` on Figure 10). These syntheses failed on late timing analysis.

Considering the **compute** loop speed-ups (letters `e`, `f`, `g` and `h` on Figure 10), kernels `mri_q` and `sgemm` present a great discrepancy as compared to **full** application: max. 24.2× and 10.3×, respectively. Kernel `lbm_small` has a greater speed-up but in lesser extent (2.4× compared to full application's 1.4×), and `histo` is unchanged. The `histo` explorations that did not synthesise were not tested (letters `g` and `h` on figure).

The kernels with great discrepancy between **full** application and **compute** loop speed-ups indicate that data transfers take non-negligible times. These could be mitigated by optimisation strategies, such as interleaving data transfers with computation. Currently no such optimisation is performed, and it is out of scope of this paper.

Figure 11 presents the energy efficiency gains for each kernel. These are using the same optimisation parameters as the results presented in the previous figure.

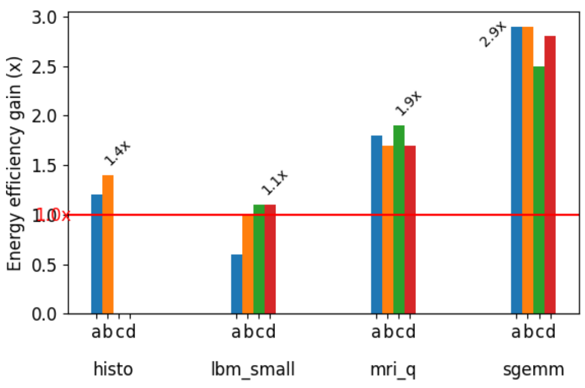

Fig. 11 Energy efficiency gains for each kernel. The four experiments for each kernel are presented similar to Figure 10. The highest efficiency gain of each kernel is indicated. Adapted from [6].

The energy efficiency gain is a speed-up analogy, but for energy consumption: how many times the optimised kernel is more efficient than baseline. We have only considered the **full** applications in this case, since it would be tricky to isolate the exact energy consumption of the compute loops using our experimental setup. Energy efficiency gain values are close to the attained speed-ups, differing by at most 0.7 points: 1.4× for `histo`, 1.1× for `lbm_small`, 1.9× for `mri_q` and 2.9× for `sgemm`. Similar to the previous figure, the `histo` explorations that did not synthesise were not tested (letters `c` and `d`).

7.3 Lina Parallel Execution Time Analysis

Figure 12 presents the exploration results for each kernel when using our approach, with varying number of parallel threads in the i7-9750H computer. Apart from `histo`, all other kernels took a couple of minutes to generate the dynamic trace file. This step is sequential and does not benefit from the parallel setup. In this computer, 16 threads appeared to be the optimal configuration, with most kernels having only a marginal gain (or even a slight slowdown) when using 32 threads.

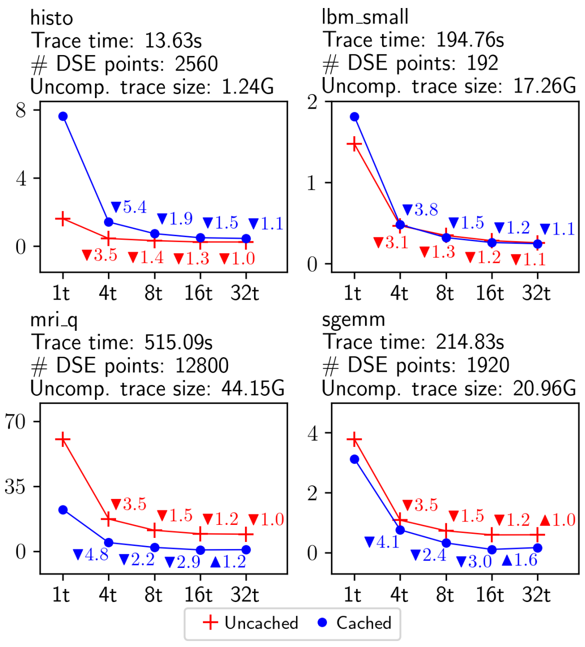

Fig. 12 Exploration statistics for the Parboil kernels on the i7-9750H computer. The x-axis represents the number of threads, and y-axis represents total Lina DSE execution times (1000s of seconds). Each point (apart from 1 thread) are accompanied with a numerical value that depicts its speed-up or slowdown when compared to the previous point (for example, ▼5.0 means a 5× speed-up). Values with ▲ represent slowdowns.

For `mri_q` and `sgemm`, the caching solution as presented in Section 5 gave better results than running the exploration without it. Notably the `mri_q` had an exploration time reduction of 9406 to 731 seconds on 16 threads by using the cache. For the `histo` kernel, the caching mechanism presented worse results. We believe that this is due to the small trace size for `histo`, which leads to low I/O intensity in overall. In this case, the caching mechanism overhead becomes more visible. For `lbm_small`, the cached solution is worse on lower threads count, but slightly better on higher counts.

In order to avoid unnecessary overhead, a simple toggle mechanism could be used to enable or disable the dynamic trace cache. For example, the exploration could initially start with cache disabled. Then if it detects that the dynamic trace cursor is constantly reaching very high values, the cache could be enabled from this point onwards. Very high dynamic trace cursors will incur in decompression and I/O overhead that can be avoided by the caching mechanism. An alternate simpler approach is to consider a trace size threshold to decide whether to use caching or not. For example, `histo` has a very small trace size that discourages the use of the cache. Using a threshold of $5GB$ for example would make all but `histo` explorations to be performed with cache. This is consistent with the fastest results presented.

The speed-up when increasing the number of threads was higher on the cached executions as compared to the uncached ones. This is specially noticeable on `mri_q` and `sgemm`. A speed-up of $\sim 3\times$ was reached when increasing from 8 to 16 threads.

The total exploration time (including trace and with 16 threads) for each kernel is: 267 seconds for `histo`, 479 seconds for `lbm_small`, 1246 seconds for `mri_q`, and 321 seconds for `sgemm`. This considers four experiments for each kernel, by switching the toggles as presented in Table 3.

Figure 13 presents the explorations performed on the computer cluster. We used the $5GB$ threshold as explained above for toggling the cache. In this setup, 32 threads seemed to be the optimal approach. Partitioning the design space into multiple nodes brought improvements as expected. For kernel `mri_q`, the exploration time (excl. trace generation) reduced from 575 to 233 seconds when increasing from 1 to 4 nodes.

8 Discussion

The CNN layer experiments present the importance of using directives in order to boost HLS quality. With simple unroll and pipeline pragmas, speed-ups with two orders of magnitude were achieved by improving the off-chip memory access on the loop nest.

After understanding the compiler's limitation — e.g. the `read-after-write` pattern that degrades performance — our results have also shown that explicit code manipulation in order to alleviate such limitations can also lead to performance improvements, as in `padmemory`. Conversely, they also lead to performance degradation in the case of `padlogic`. This non-linearity

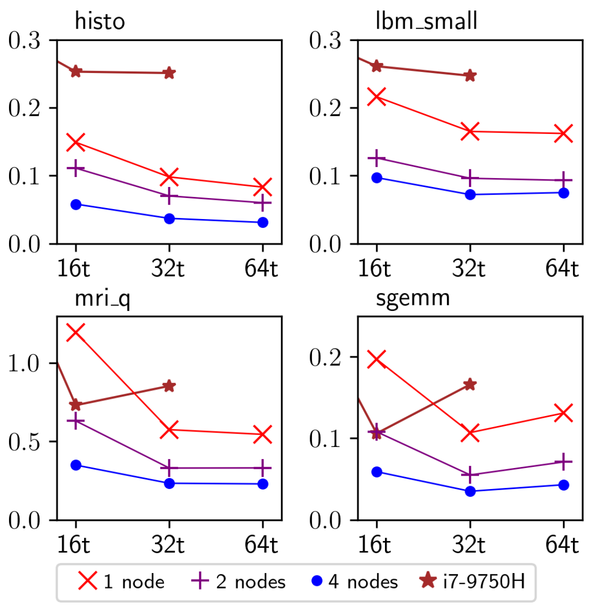

Fig. 13 Exploration statistics for the Parboil kernels on the computer cluster. The x-axis represents the number of threads, and the y-axis is the total exploration time in thousands of seconds. The results with 16 and 32 threads from Figure 12 are also presented here for reference. For `histo` these points are from the uncached version. For other kernels, these points are from the cached version.

between cause and effect further suggests the use of automated exploration tools.

Lina DSE correctly inferred the best operating frequency, loop unroll factors and pipeline directives in all CNN explorations. The results were always in the top-10 best design points of each explored design space.

The kernels explored from Parboil presented more concrete use cases of Lina DSE, since in this case the whole applications were considered, including data transfers from a host machine to the FPGA device. Considering the best exploration performed for each kernel, Lina DSE was able to provide performance speed-up (avg. 2.03×) and energy efficiency gains (avg. 1.83×). When analysing the speed-up of the computation loop only (disregarding data transfers), the improvements were even greater (avg. 9.5×). This is not unexpected, as Lina does not currently consider shadowing memory transfers with computations. Nonetheless, we believe that the results indicate that Lina DSE effectively improves the performance of computation loops that may access off-chip elements, sequentially or not.

As an extra experiment, we adapted our unroller tool used in the CNN kernel to accept more generic codes, as the ones from Parboil. The input kernel must be adapted to adopt a pattern similar to the code presented in Appendix D. These versions did not bring any result that were both synthesisable and better than compared than the ones already presented. According to HLS compilation reports, explicitly unrolling the code and enabling vectorisation would improve the speed-up of `mri_q` from 24.2× to 85.1×. But, this solution failed to pass full synthesis. The bottom line is that although rearranging all reads before write can improve the quality of HLS scheduling policies, it also increases the code size, that in turn pushes the HLS compiler on other edges. This can eventually lead to performance degradation — as seen in `padlogic` — or to synthesis failure, as in `mri_q`.

The experimental results further corroborates the viability of performing synthesis-less design space exploration. Related work - such as ScaleHLS - also contributes towards this goal. However, Lina provides a more robust off-chip memory model. For example, their *ResMII* calculation does not consider off-chip memory accesses. Using a dynamic trace for memory analysis has the benefit of having all addresses already calculated through code execution.

9 Conclusion

This paper presented a design space exploration approach that uses Lina, a tool capable of estimating performance of HLS-generated designs including several aspects. The most important herein presented being an off-chip memory model, that estimates accesses to off-chip memory. Potential optimisations are considered, such as burst and data packing.

For a simple convolution kernel with off-chip memory accesses, Lina correctly inferred the best frequency, the best loop unroll configuration and the best loop pipeline configuration, only with a slight deviation when selecting the array partition configuration. The best points given by Lina were always one of the top-10 best points in the design space, reaching significant speed-ups.

We also presented the exploration of 4 kernels from Parboil benchmark. Lina DSE was able to provide performance speed-ups on all kernels, considering the whole application execution. However, there is still significant data transfer overhead before and after the computation loops, that our approach does not currently optimise.

Although our approach is quite fine-tuned in several aspects to Vivado HLS and SDSoC, our methodology has further shown the benefits of using dynamic traces for decision making when performing DSE. For example, the memory accesses are already resolved by the trace, not requiring any static complex inferring that could incur in larger estimation times.

Our model assists on reducing the hardware burden when programming FPGAs via HLS. It does so by providing guidance on optimisation that would take much longer to be manually found. It also helps to reduce the gap between (high-level) hardware and software workflows, broadening FPGA accessibility.

Acknowledgements. The authors would like to thank São Paulo Research Foundation (FAPESP) for the financial support. A great appreciation also goes to Prof. Guido Araujo and the rest of the Computer Systems Laboratory (LSC) in UNICAMP for the support given via the Sorgan cluster. At last, we thank the reviewers for all the orientations given.

Data Availability. Lina and the design space exploration framework used in this paper are available at https://github.com/comododragon/cirith-fpga. For other requests, please contact the corresponding author.

Author Contributions. Model development, validation, interpretation, and writing of paper: Perina, André B.; Concept design, interpretation, and content review: Becker, Jürgen, and Bonato, Vanderlei.

Funding. This work was supported by São Paulo Research Foundation (FAPESP) under grants no. 2016/18937-7 and 2018/22289-6.

Declarations

Competing Interests. The authors declare no conflicts of interest.

References

[1] Takach, A.: High-Level Synthesis: Status, Trends, and Future Directions. IEEE Design & Test **33**(3), 116–124 (2016)

[2] Zohouri, H.R., Maruyama, N., Smith, A., Matsuda, M., Matsuoka, S.: Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 35 (2016). IEEE Press

[3] Weller, D., Oboril, F., Lukarski, D., Becker, J., Tahoori, M.: Energy Efficient Scientific Computing on FPGAs using OpenCL. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 247–256 (2017). ACM

[4] Perina, A.B., Becker, J., Bonato, V.: Lina: Timing-Constrained High-Level Synthesis Performance Estimator for Fast DSE. In: 2019 International Conference on Field-Programmable Technology (ICFPT), pp. 343–346 (2019). IEEE

[5] Perina, A.B., Silitonga, A., Becker, J., Bonato, V.: Fast Resource and Timing Aware Design Optimisation for High-Level Synthesis. IEEE Transactions on Computers **70**(12), 2070–2082 (2021)

[6] Perina, A.B.: Lina: a fast design optimisation tool for software-based FPGA programming. PhD thesis, Universidade de São Paulo (2022)

[7] Zhong, G., Prakash, A., Liang, Y., Mitra, T., Niar, S.: Lin-analyzer: a High-level Performance Analysis Tool for FPGA-based Accelerators. In: 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6 (2016). IEEE

[8] Stratton, J.A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G.D., Hwu, W.-m.W.: Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Center for Reliable and High-Performance Computing

127, 29 (2012)

[9] Makni, M., Niar, S., Baklouti, M., Abid, M.: HAPE: A high-level area-power estimation framework for FPGA-based accelerators. Microprocessors and Microsystems **63**, 11–27 (2018)

[10] Rajagopala, A.D., Sass, R., Schmidt, A.: Impact of Off-Chip Memories on HLS-Generated Circuits. In: FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers, pp. 1–10 (2019). VDE

[11] Shao, Y.S., Reagen, B., Wei, G.-Y., Brooks, D.: Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), pp. 97–108 (2014). IEEE

[12] Austin, T.M., Sohi, G.S.: Dynamic Dependency Analysis of Ordinary Programs. In: Proceedings of the 19th Annual International Symposium on Computer Architecture, pp. 342–351 (1992)

[13] Enzler, R., Jeger, T., Cottet, D., Tröster, G.: High-Level Area and Performance Estimation of Hardware Building Blocks on FPGAs. In: International Workshop on Field Programmable Logic and Applications, pp. 525–534 (2000). Springer

[14] Kulkarni, D., Najjar, W.A., Rinker, R., Kurdahi, F.J.: Compile-Time Area Estimation for LUT-Based FPGAs. ACM Transactions on Design Automation of Electronic Systems (TODAES) **11**(1), 104–122 (2006)

[15] Bilavarn, S., Gogniat, G., Philippe, J.-L., Bossuet, L.: Design Space Pruning Through Early Estimations of Area/Delay Tradeoffs for FPGA Implementations. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **25**(10), 1950–1968 (2006)

[16] Bjureus, P., Millberg, M., Jantsch, A.: FPGA Resource and Timing Estimation from Matlab Execution Traces. In: Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627), pp. 31–36 (2002). IEEE

[17] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J.: Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-programmable Gate Arrays, pp. 161–170 (2015)

[18] Zhong, G., Prakash, A., Wang, S., Liang, Y., Mitra, T., Niar, S.: Design Space Exploration of FPGA-based Accelerators with Multi-level Parallelism. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pp. 1141–1146 (2017). IEEE

[19] Choi, Y.-k., Cong, J.: HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds. In: 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–8 (2018). IEEE

[20] Zhao, J., Feng, L., Sinha, S., Zhang, W., Liang, Y., He, B.: Performance Modeling and Directives Optimization for High Level Synthesis on FPGA. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2019)

[21] Wang, S., Liang, Y., Zhang, W.: FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs. In: Proceedings of the 54th Annual Design Automation Conference 2017, pp. 1–6 (2017)

[22] Dávila-Guzmán, M.A., Tejero, R.G., Villarroya-Gaudó, M., Gracia, D.S.: An Analytical Model of Memory-Bound Applications Compiled with High Level Synthesis. In: 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 218–218 (2020). IEEE

[23] Gautier, Q., Althoff, A., Crutchfield, C.L., Kastner, R.: Sherlock: A Multi-Objective

Design Space Exploration Framework. ACM Transactions on Design Automation of Electronic Systems (TODAES) **27**(4), 1–20 (2022)

[24] Jun, H., Ye, H., Jeong, H., Chen, D.: AutoScaleDSE: A Scalable Design Space Exploration Engine For High-Level Synthesis. ACM Transactions on Reconfigurable Technology and Systems **16**(3), 1–30 (2023)

[25] Ye, H., Hao, C., Cheng, J., Jeong, H., Huang, J., Neuendorffer, S., Chen, D.: ScaleHLS: A New Scalable High-Level Synthesis Framework On Multi-Level Intermediate Representation. In: 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 741–755 (2022). IEEE

[26] Li, P., Zhang, P., Pouchet, L.-N., Cong, J.: Resource-Aware Throughput Optimization for High-Level Synthesis. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 200–209 (2015)

[27] Rau, B.R.: Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops. In: Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 63–74 (1994)

[28] Stoutchinin, A., Conti, F., Benini, L.: Optimally Scheduling CNN Convolutions for Efficient Memory Access. arXiv preprint arXiv:1902.01492 (2019)

[29] Peemen, M., Setio, A.A., Mesman, B., Corporaal, H.: Memory-Centric Accelerator Design for Convolutional Neural Networks. In: 2013 IEEE 31st International Conference on Computer Design (ICCD), pp. 13–19 (2013). IEEE

Appendix A Off-chip Memory Model Material

Additional material regarding the off-chip memory model can be found here.

A.1 Reporting

Figure 14 presents a snippet of the report generated by Lina's memory model. Notice that two data packing (vectorisation) attempts are made. The first one fails, since Lina detects that not all DDDGs show compatibility with 4 packed elements. Then, data packing is successful when packed as twos.

```
[WARN] Burst possibility between loop iterations failed
       for array A
       at loop level 1
       at region before the loop nest
       Reason: detected reads for the same array
               at loop level 2
               within the loop nest
...
[WARN] Vectorisation attempt failed
       for array B
       with 4 elements vectorised
       Reason: cannot align write with pack size
               due to 2 unused element(s) after
               at loop level 1
               at region before the loop nest
...
[INFO] Vectorisation attempt successful
       for array B
       with 2 elements vectorised
```

Fig. 14 Snippet of a memory report generated by Lina.

A.2 Implemented Policies

Table 6 presents how several aspects of the off-chip memory model behave depending on selected memory policy. The concept of "memory space" herein used is similar as to the "memory interface" concept previously presented, that is: if banking is enabled, each off-chip array has its own memory space. Otherwise, all share the same memory space.

A.3 Additional Insights Considering Loop Pipeline

The extended $ResMII_{mem}$ calculation in (3) considers chained load/store operations, and relaxes II so that it avoids potential memory conflicts. Figure 15 presents an example of the delta calculations. Consider that all loads are in the same interface and it is single-ported. According to (3), the longest delta is selected, in this case Δ_3. However, if Δ_3 was set as $ResMII^{rec}_{mem}$, there would still be overlap between these chained loads, which

Table 6 The effect of scheduling policies on the memory model [6].

Aspect	Permissive Policy	Conservative Policy
Inter-iteration bursts (read)	Allowed when in same or deeper loop level there are: no other reads for same array; and no other writes for same array	Allowed when in same or deeper loop level there are: no other reads for same memory space; and no other writes for same array
Inter-iteration bursts (write)	Allowed when in same or deeper loop level there are: no other reads for same array; and no other writes for same array	Allowed when in same or deeper loop level there are: no other writes for same memory space; and no other reads for same array
Setup (read)	Allowed when active reads and writes in same memory space are for regions with no overlap	Allowed when: active reads are not burst; and writes in same memory space are for regions with no overlap
Setup (write)	Allowed when active reads and writes in same memory space are for regions with no overlap	Allowed when there is no other active write in same memory space
Commit (write)	Always allowed	Allowed when active reads in same memory space are for regions with no overlap
Promoted nodes *(nodes promoted from inner loop levels due to inter-iteration burst optimisation)*	Can execute concurrently if all active transactions in same memory space are non-overlapping	**For read:** can execute concurrently if all active transactions in same memory space are non-overlapping **For write:** cannot execute concurrently with any other transaction in same memory space

can lead to port contention. In order to avoid that, $ResMII^{rec}_{mem}$ should comprise at least all these chained loads. Since HLS will attempt to optimise and pack memory operations together at start or end of schedulings, successive chained memory operations will often be placed one after another in a "stairs" pattern (indicated on figure). The minimum value that comprises all these chained loads is equal to the largest delta, plus the distance created by the "stairs" pattern. Each isolated set of chained operation adds one step to the staircase (i.e. one clock cycle), and this is why we add the number of connected sets Cr_m to the largest delta when calculating $ResMII^{rec}_{mem}$.

Back to Figure 15, there are three connected sets of loads, and the largest delta is 12. Thus, the MII must be at least $12+3 = 15$ in order to avoid overlap between any of these sets. The rationale is similar for write operations and Cw_m.

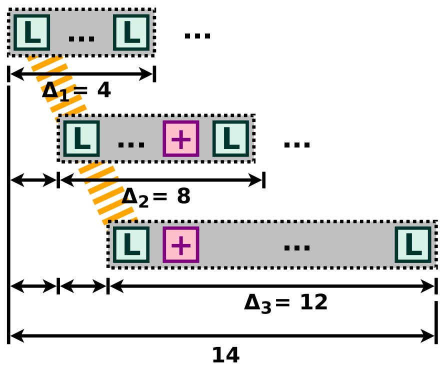

Fig. 15 Example of three chained sets of load operations, all for a same single-ported interface. The largest delta is $\Delta_3 = 12$. There are three chained sets, thus $Cr_m = 3$. Then, $ResMII^{rec}_{mem} = 12 + 3 = 15$. This value is relaxed enough to conservatively avoid any contention between these chained sets. The "stairs" pattern explained on text is indicated by the dashed diagonal stripe in background.

Appendix B ZCU104 Testbench

Figure 16 presents the FPGA platform, including the additional modules used for power sensing.

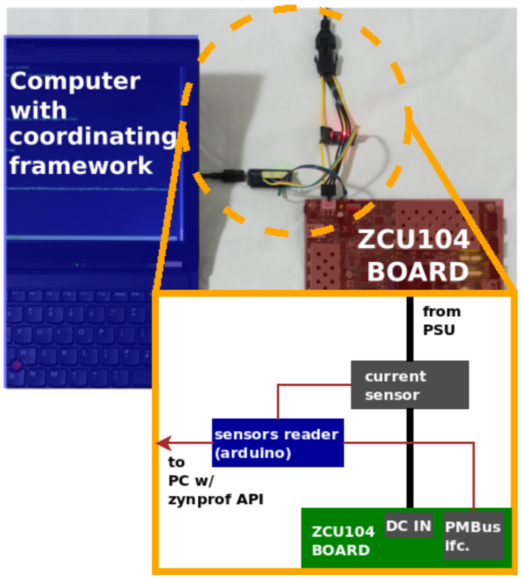

Fig. 16 FPGA experimental setup. The power sensing system is highlighted. An Arduino module reads the PSU current sensor and communicates with the regulators using the PMBus interface. The raw information is collected and sent to the host machine, which calculates consumed energy. Adapted from [6].

Appendix C CNN Experiment Materials

For the CNN experiment presented in Section 6.2, we used ZFNet's 6th layer configuration as presented in [28] and in Table 7. These parameters relate to the constants used in Figure 8.

Table 7 ZFNet CNN layer configuration used [6].

Name	Description	Value
`C`	# input feature maps	256
`M`	# output feature maps	256
`H`	Input feature map size (`H` × `H`)	6
`R`	Convolution kernel size (`R` × `R`)	3
`S`	Convolution kernel stride	1
`E`	Output feature map size (`E` × `E`)	6

Table 8 presents the knobs that compose the design spaces for both CNN kernels tested. The design spaces have 680 valid points.

Table 8 Optimisation knobs for the CNN kernels [6].

Loop knobs			
Name	Loop depth	Unroll factors	Pipeline
LOF	1	Off	Off
LIF	2	Off	Off
LSY	3	Off, 2	Off, on
LSX	4	Off, 2	Off, on
LFY	5	Off, 3	Off, on
LFX	6	Off, 3	Off, on
Array knobs			
Name	I/O	Partitioning	
I	Input	No partitioning (off-chip access)	
W	Input	No partitioning (off-chip access)	
O	Output	**Off**; **Block:** 4, 8; **Cyclic:** 4, 8	
Frequencies			
Values (MHz)		75, 100, 150, 200	

Figure 17 presents the DSE results for the `padlogic` kernel.

Appendix D Unroller Tool Examples

There are two automated unroller tools presented in this paper. The first one - mentioned in Section 6.2 - was tailored for CNN layer. Figure 18 presents an example of code generated by this tool. All read statements (lines 10 to 17) were placed before all write statements (lines 19 to 22).

The second tool - mentioned in Section 8 - is an expansion of the more domain-specific used for the CNN experiments. It is a simple tool and thus it relies on code manipulation and annotations. Figure 19 presents the `sgemm` code adapted for the tool. Notice the presence of various keywords, such as `<LITER>`, `<LINCR>`, etc. The `<LFRONT>` defines the loop boundary between reads and writes. All off-chip reads should be placed before this front, and all off-chip writes after that. The code must be further adapted so that the tool is able to replace the `<LCTR>` and `<LITER>` keywords with unrolled values.

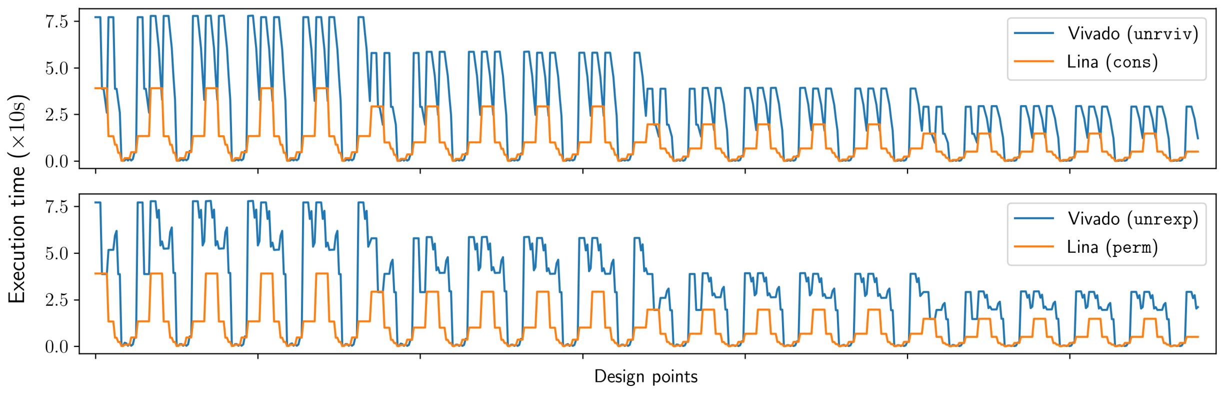

Fig. 17 Design space exploration results for `padlogic`. Horizontal axis presents the design points tested, and vertical axis presents the kernel execution time that was found/estimated for the associated design point. The subplot at top couples Vivado results using normal HLS attributes for unroll with Lina exploration in `conservative` mode. The bottom subplot presents Vivado results using our unrolling tool compared to Lina exploration in `permissive` mode.

```
LOF: for(auto m = 0; m < M; m++)
  LIF: for(auto c = 0; c < C; c++)
    LSY: for(auto y = 0; y < E; y++)
      LSX: for(auto x = 0; x < E; x++)
        LFY: for(auto k = 0; k < E; k++)
          LFX: for(auto l = 0; l < E; l += 4) {
            auto __ySk = y * S + k;
            auto __xSl = x * S + l;

            auto __p_0 = I[c][__ySk][__xSl];
            auto __w_0 = W[m][c][k][l];
            auto __p_1 = I[c][__ySk][__xSl + 1];
            auto __w_1 = W[m][c][k][l + 1];
            auto __p_2 = I[c][__ySk][__xSl + 2];
            auto __w_2 = W[m][c][k][l + 2];
            auto __p_3 = I[c][__ySk][__xSl + 3];
            auto __w_3 = W[m][c][k][l + 3];

            O[m][y][x] += __p_0 * __w_0;
            O[m][y][x] += __p_1 * __w_1;
            O[m][y][x] += __p_2 * __w_2;
            O[m][y][x] += __p_3 * __w_3;
          }
```

Fig. 18 Example of convolutional kernel, unrolled (factor of 4) using the unroller tool. All off-chip reads are placed before off-chip writes.

```
<LOOP_0_1> for(int mm = 0; mm < M; mm++) {
  <LOOP_0_2> for(int nn = 0; nn < N; nn++) {
    float c = 0.0f;

    <LOOP_0_3> for(int i = 0; i < K; i += <LINCR>) {
      <LPREAMB>
      float a_<LCTR>;
      float b_<LCTR>;

      <LBEGIN>

      a_<LCTR> = <ARR_A>[mm + (i + <LITER>) * M];
      b_<LCTR> = <ARR_B>[nn + (i + <LITER>) * N];

      <LFRONT>

      c += a_<LCTR> * b_<LCTR>;

      <LEND>
    }

    <ARR_C>[mm + nn * M] =
        <ARR_C>[mm + nn * M] * BETA + ALPHA * c;
  }
}
```

Fig. 19 Example of `sgemm`'s loop nest adapted for the unroller tool.