Improving Loop Parallelization by a Combination of Static and Dynamic Analyses in HLS

FLORIAN DEWALD, MAIS chair, Dept. of Computer Science - TU Darmstadt, Germany

JOHANNA ROHDE, Computer Systems Group - TU Darmstadt, Germany

CHRISTIAN HOCHBERGER, Computer Systems Group - TU Darmstadt, Germany

HEIKO MANTEL, MAIS chair, Dept. of Computer Science - TU Darmstadt, Germany

High-level synthesis (HLS) can be used to create hardware accelerators for compute-intense software parts such as loop structures. Usually, this process requires significant amount of user interaction to steer kernel selection and optimizations. This can be tedious and time-consuming. In this article, we present an approach that fully autonomously finds independent loop iterations and reductions to create parallelized accelerators. We combine static analysis with information available only at runtime to maximize the parallelism exploited by the created accelerators. For loops where we see potential for parallelism, we create fully parallelized kernel implementations. If static information does not suffice to deduce independence, then we assume independence at compile time. We verify this assumption by statically created checks that are dynamically evaluated at runtime, before using the optimized kernel. Evaluating our approach, we can generate speedups for five out of seven benchmarks. With four loop iterations running in parallel, we achieve ideal speedups of up to 4 $\times$ and on average speedups of 2.27 $\times$, both in comparison to an unoptimized accelerator.

CCS Concepts: • **Hardware** $\rightarrow$ **High-level and register-transfer level synthesis; Hardware-software codesign; Datapath optimization;**

Additional Key Words and Phrases: FPGA, system-on-chip, high-level synthesis, scalar evolution analysis, loop parallelization

ACM Reference format:

Florian Dewald, Johanna Rohde, Christian Hochberger, and Heiko Mantel. 2022. Improving Loop Parallelization by a Combination of Static and Dynamic Analyses in HLS. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 31 (February 2022), 31 pages.

<https://doi.org/10.1145/3501801>

1 INTRODUCTION

High-level synthesis (HLS) is a method to create hardware implementations for programs written in traditional programming languages, like C. Often, HLS is not applied to whole applications but rather to compute-intense software parts only (i.e., kernels). The corresponding hardware

Florian Dewald and Johanna Rohde contributed equally to this research.

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

Authors' addresses: F. Dewald and H. Mantel, MAIS chair, Dept. of Computer Science - TU Darmstadt, Hochschulstraße 10, 64289, Darmstadt, Germany; emails: {dewald, mantel}@cs.tu-darmstadt.de; J. Rohde and C. Hochberger, Computer Systems Group - TU Darmstadt, Merckstraße 25, 64283, Darmstadt, Germany; emails: {rohde, hochberger}@rs.tu-darmstadt.de.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2022 Copyright held by the owner/author(s).

1936-7406/2022/02-ART31 \$15.00

<https://doi.org/10.1145/3501801>

Fig. 1. Three examples that visualize the different options for individual kernels.

implementation is called an accelerator. Such accelerators are often used in systems with **Field-Programmable Gate Arrays (FPGA)**, which can be reprogrammed for every new application.

Optimizing a program with HLS can be split into four tasks: (1) identifying compute-intensive software parts, i.e., kernels, that deserve acceleration, (2) analyzing for each kernel whether and how it could be optimized, (3) the selection of optimizations to be implemented, and (4) the actual high-level synthesis. Most HLS tools handle these four tasks requiring substantial user intervention (e.g., parameter selection, code annotations, or rewriting/restructuring the source code) [3, 8, 18, 26]. This process is usually tedious and time-consuming.

Our overall objective in this research project is to reduce this effort, both by enlarging the scope and by increasing the benefits of fully autonomous hardware acceleration of kernels. More specifically, we focus on loop parallelization where iterations of a loop can be executed independently and in parallel on multiple loop processing units. We propose an approach that identifies loop-level parallelism using methods of static code analysis. When static analysis does not suffice, it identifies necessary conditions for loop iterations to be independent from each other. Testing these conditions at runtime allows us to gain more parallelism in the kernel implementation and thereby increase the speedup that we can achieve. This works particularly well for nested loops.

That is, given a kernel, our approach determines whether the kernel can be parallelized always, under specific runtime conditions, or never. With this information, either a fully parallelized kernel implementation, a kernel with selectable parallelism, or a traditional scalar kernel is implemented.

Figure 1 shows a motivating example for these three cases. Analysis and parallelization can occur on any loop nesting level. For the sake of simplicity, the examples always show only two nesting levels. Figure 1(a) shows the case where static analysis can prove that parallel execution of inner loops is legal. The element written is only dependent on its own previous value. Thus, all lines could be processed in parallel. The picture shows a duplication of the inner loop, i.e., a parallelization factor of two. Yet, higher parallelization factors are possible. We perform this type of parallelization directly in our hardware implementation, whereas it requires additional user-annotations and infrastructure in software compilers. Figure 1(b) illustrates the case where static analysis can prove that parallel execution of inner loops is impossible. Elements written in the

current line depend on previous lines and thus, parallel execution would be erroneous. Finally, Figure 1(c) is an example where static analysis is inconclusive due to statically unknown values for pointers *a* and *b* and an unknown number of lines. Hence, at runtime a check must be executed to test whether a parallel execution of inner loops is legal. This runtime check is generated during HLS and executed at runtime. In our evaluation, this check enables parallel execution in more than half of the benchmarks. To the best of our knowledge other HLS tools do not support this optimization.

We implemented¹ our approach into the PIRANHA HLS plugin [6] and evaluate a set of benchmark applications previously used in the field of HLS. We also compare our tool against the leading industry HLS tool optimized for Xilinx FPGAs.² We show that our tool achieves equal or better speedups without any manual intervention by the user.

Section 2 discusses competing HLS tools and their approaches. Section 3 presents the technical background of our work. Section 4 explains the static analyses of our approach in detail. Section 5 augments the static analyses with dynamic analyses and explains them in detail. Section 6 describes the relevant implementation details. Section 7 presents a detailed evaluation. Finally, Section 8 concludes.

2 RELATED WORK

In this section, we provide an overview to areas related to our work. These areas are user-directed parallelization, autonomous parallelization, and the particular task of detecting aliasing when deciding autonomously what to parallelize. Each of these areas covers a different subset of the four HLS tasks named in the introduction and relies on user input to tackle the remaining tasks.

2.1 User-directed Parallelization

User-directed approaches delegate the responsibility of choosing parallelizable and compute-intensive software parts to the user. Given the four HLS tasks from the introduction, Task 1 to Task 3 are realized by user-provided annotations (i.e. pragmas).

On the commercial side, Xilinx with Vivado HLS and Intel's HLS compiler require user interaction to optimize loop structures [8, 26]. For the outermost loop, the user has to split the problem size manually and include more than one accelerator into the system. The structure of both, software and system, needs to support this split. If they do not, the user might need to perform excessive restructuring of both. In summary, Vivado HLS and Intel's HLS compiler require the user to manually parallelize and do not provide automatic splitting based on user-provided annotations.

A way to automatically implement tiling and coarsening is provided in Reference [23]. In this work, the authors provide a tool that implements multiple Vivado HLS kernels or deploys an NDRange kernel in OpenCL. Nevertheless, their approach uses a domain specific language as input.

On the academic side, one of the most popular tools is LegUp. In Reference [5], Choi, Brown, and Anderson propose to use OpenMP annotations and manually parallelized code with pthreads to identify parallelizable regions. Those regions are implemented by duplicated processing units. Note that the commercial version of LegUp uses the same concept [7].

Similar to that, Lattuada and Ferrandi use OpenMP annotated code as input for their vectorization approach implemented in Bambu [10]. The focus of their work is on the implementation of do-all loops with vectorized instructions that compute multiple iterations at the same time using a single-instruction, multiple-data approach.

¹You can download our implementation at <https://www.mais.informatik.tu-darmstadt.de/trets2021.html>.

²For licensing reasons, we cannot disclose the product name or version of this tool.

All of these user-directed approaches require the user to refactor the code and to actively select the optimizations to be applied, or even implement them themselves. Selecting which optimizations can be applied where in the code, without endangering correctness, is a challenge for developers and requires expertise [4, 17]. Additionally, estimating whether an optimization potentially yields a benefit requires a lot of knowledge in the field of HLS. Otherwise, it will result in a trial-and-error-style development that is tedious and time-consuming. We aim for an autonomous approach that does not require user interaction. This increases usability, lowers the barrier-to-entry, and decreases development time for the user.

2.2 Autonomous Parallelization

In an autonomous approach, the HLS tool itself implements an analysis that identifies parallel loops. Given the four HLS tasks from the introduction, Task 1 to Task 4 need to be covered by an autonomous approach.

G. Liu et al. propose ElasticFlow, an HLS tool that uses multiple loop processing units to process loops with a dynamically bounded inner loop in parallel [11]. The iterations of the outer loop are distributed dynamically. Loops with dynamically bounded inner loops are detected autonomously. The authors acknowledge that their approach is neither applicable nor can detect all loop-carried memory dependencies. As a result, they can implement parallelism autonomously but risk erroneously breaking memory dependencies in the outermost loop.

Autonomous approaches that combine static and dynamic checks to detect parallelism are also used outside of HLS. Approaches exist for thread-level parallelism [9, 15] and loop parallelism [20, 24]. The approach by Sampaio et al. uses dynamic checks to verify a speculatively created optimized version of a loop [24]. They highlight that the dynamic check they create runs once for a whole loop and does not need to re-run in each loop iteration. This is beneficial for performance. Based on the check's outcome, an if-then-else region starts either a sequential or parallel execution.

ElasticFlow transforms a restricted class of programs autonomously to use available parallelism, but their decision could be erroneous in case loop-carried memory dependencies are present. The approach by Sampaio et al. detects available parallelism and emphasizes that their generated dynamic check is efficient. Our approach is fully autonomous, i.e., it performs all four HLS tasks autonomously. We ensure that the execution of an accelerator will be correct by only running loops in parallel when we can disprove dependencies statically or dynamically. Furthermore, our optimization focuses on general purpose loops with arbitrary depth and is not specific to a particular loop structure, e.g., loops with dynamic bounds. Performance-wise, our runtime checks need to run only once before each loop, providing similar advantages to tools outside of HLS.

2.3 Aliasing Detection in Autonomous Parallelization and Optimization

Detecting memory dependencies and aliasing is crucial for correctly and precisely performing Task 2 of the four HLS tasks, i.e., to detect what parts of a program can be parallelized. Memory dependencies and aliasing information are also important for other optimizations.

J. Liu et al. apply dynamic dependency testing to loop pipelining [12–14]. They focus on cases where multiple accesses to the same array might have loop-independent or loop-carried data dependencies caused by offsets that are unknown at compile time or different strides causing the memory accesses to overlap at a certain point. Using polyhedral analysis, they create a runtime check based on which either an optimized or conservative loop is executed. It should be mentioned that this approach only improves the loop pipelined schedule. It does not lead to a parallel execution of inner loops.

Dynamic techniques that rely on runtime information are also developed outside of HLS. Rauchwerger and Padua use dynamic analyses to gain information about irregular array accesses [21].

They replay loops with shadow arrays to record what indices are accessed and derive dependencies based on that. They extend this check to speculatively parallelize [22]. If a dependent access has been detected, then they restore original values and restart the loop sequentially.

In contrast to the approach by J. Liu et al., our approach is not restricted to the innermost loop. We apply runtime checks to the whole loop nest. Furthermore, we support dependencies between different arrays and pointer arithmetic. We even support these when relevant values, such as iteration counts, offsets, and strides, are only known at runtime, e.g., because their expressions depend on user input. Like the non-HLS approach, we test independence of memory accesses at runtime, but do not need to add logic to restore values in case of a failed check. Our approach computes access bounds for memory accesses statically and evaluates them at runtime before the parallel execution is started. This leads to runtime checks that are very efficient and that do not impose an additional time penalty in case a runtime check fails.

3 PRELIMINARIES

In this section, we introduce the concepts underlying our approach and implementation. First, we introduce the control and data flow graph together with its underlying concepts of flows and dependencies as our representation of programs. Second, we introduce the particular structure of loops that we work with. Finally, we introduce induction variables and their relevancy for memory accesses together with a compact representation to capture their evolution.

Since we use the GCC as our underlying compiler, these concepts have a close relation to GCC's **intermediate representation (IR)** GIMPLE and analyses performed by GCC. In particular, loops in the IR generated by the GCC during compilation are in the loop structure that we work with.

3.1 Flows and Data Dependencies

Data flow expresses how data is propagated between instructions. If an instruction writes a value that might be read by a consecutive instruction, then there is a data flow between these two instructions. We refer to such a flow of data as a *data dependency*.

Control flow is a second kind of flow. Control flow expresses the sequential execution order between instructions. If an instruction might be executed directly after another instruction, then there is a control flow between these two instructions. The related *control dependency* expresses that an instruction influences whether another instruction is executed or not.

In this article, we are concerned mostly with data dependencies. We represent a data dependency by a pair of instructions. A data dependency (i, j) represents that i writes a value that j might read.

A common distinction is whether data dependencies are loop independent or loop carried [2]. A data dependency (i, j) is *loop independent* if i and j are not in the body of the same loop or if j is executed after i in the same loop iteration. A data dependency (i, j) is *loop carried* if i and j are part of the same loop and the read by j might be in a consecutive loop iteration. This distinction is important for us, since our decision whether to parallelize a loop is based on it.

In Figure 2, the data dependencies $(1, 2)$ and $(2, 3)$ are loop independent, since Instruction 1 is not part of a loop and Instruction 3 is executed after Instruction 2 in the same loop iteration, respectively. The data dependency $(4, 2)$ is loop carried, since both instructions are in the same loop body and Instruction 2 reads the value written by Instruction 4 in a consecutive loop iteration.

3.2 Control and Data Flow Graphs as Program Representation

A **control and data flow graph (CDFG)** is a graphical representation of a program. Nodes in the graph represent instructions. Two different types of edges between nodes represent control flow and data flow between the instructions. In our approach and toolchain, we perform analyses and transformations on the CDFG representation of a program.

Fig. 2. Structure of a regular loop.

We represent the CDFG of a program in a compressed form by combining multiple instructions to a *basic block*. A basic block contains a list of instructions that is guaranteed to execute consecutively, i.e., except the first and last instruction, each instruction has a unique control flow predecessor and successor, respectively. Nodes in the compressed form of CDFGs are basic blocks instead of individual instructions. Edges between them represent control flow between the last instruction of a basic block and the first instruction of another basic block. A label on an edge represents the control condition that is required for the control flow. Possible data flow between instructions is represented by identically named variables in assignments and expressions.

Figure 2 represents a CDFG of a loop. Instruction 2 and Instruction 3 form a basic block. Instruction 3 is a conditional, hence, the two outgoing arrows of that basic block are annotated.

3.3 Loop Structure

Loops that we tackle have a particular basic block structure. Certain tasks in a loop, such as preparing data for use in a consecutive loop iteration, are fulfilled by dedicated basic blocks. If loops are nested in one another, then we refer to them as a *loop nest*. The structure is visualized in Figure 2.

The basic block that is the unique direct predecessor of a loop is the *preheader*. Usually, the preheader initializes variables that are used privately in the loop. Its direct and unique successor, the entry point of a loop, is called the *header*. Since the successor of a preheader is unique, a loop runs for at least one iteration as soon as its preheader is reached. Assignments to variables for consecutive loop iterations are performed only in the *latch* block. The latch block is directly executed before the next execution of the header block starts. A control flow edge between latch and loop header is called a *back edge*. Intuitively, a new loop iteration starts when control flow traverses a back edge. The *body* of a loop consists of all basic blocks that are executed as part of the loop. Finally, a *terminal* block is the first basic block executed after a loop's exit condition is met.

This structure is created by the GCC and preserved by further HLS tools that we use for pre-processing input programs. Hence, the input CDFGs to our approach are in this structure.

3.4 Induction Variables and Memory Accesses

The iteration count of loops and indices for memory accesses are often realized by variables that act as counters. These variables are incremented in each loop iteration and typically read by either memory accesses or used in conditional instructions to compare against the loop bound. Such a variable that is incremented or decremented by a constant value in each loop iteration is called an *induction variable* [1]. We call an instruction an *increment instruction* if it increments or decrements an induction variable by a constant.

Memory accesses in loops often make use of induction variables to specify what address to access. In this context, induction variables are used in two different settings. First, the memory access can use an induction variable as an index. A base variable refers to the address of the first

element in an array and the induction variable provides the element index that should be accessed. Second, the base variable itself can be an induction variable without an explicit additional index, but rather where the next element is accessed by increasing the base variable directly. Both settings can have an additional offset. While the first setting is comparable to accessing memory via an array and an index, the second setting is comparable to accessing memory directly via a pointer.

The two settings of how memory is addressed correspond to instructions generated by the GCC in its GIMPLE intermediate representation.

We refer to the first address that a memory access accesses in a loop as the memory accesses *base address*. We refer to the amount of bytes the accessed address increases in each iteration of a specific loop as the memory accesses *stride* for that loop. Note that a memory access can have multiple different strides when it is in a loop nest. Together, base address and stride characterize what memory addresses might be accessed in a loop by a specific memory address.

In Figure 2, variable `iv` is an induction variable and Instruction 2 its increment instruction.

3.5 Trees of Recurrence as Representation for Evolution of Induction Variables

The **Tree of Recurrence (TREC)** representation is a compact representation that captures the evolution of variables in loops. Concretely, it captures the evolution of induction variables as a function of the loop's iteration indices [19]. Since memory accesses make use of induction variables, TRECs are well suited to infer the base addresses and strides of memory accesses based on them.

The syntax of a TREC Φ is inductively defined as³

$$\Phi = \{\Phi_a, +, \Phi_b\}_k \text{ or } \Phi = c,$$

where Φ_a and Φ_b are TRECs, c is either an integer constant or a loop invariant expression computed outside the loop, and k is an identifier for a loop. Φ_a is called the *initial value* and Φ_b is called the *increment* of a TREC. Intuitively, Φ_a corresponds to the value of an induction variable in the first loop iteration and Φ_b to its constant increment in each consecutive loop iteration. The value $\Phi(i)$ of a TREC Φ for an induction variable represents the induction variable's value in the i th loop iteration. For a more detailed introduction, we refer to Reference [19].

When an induction variable is used in a memory access, the TREC of the induction variable is conceptually similar to the base address and stride of the memory access. While a TREC refers to concrete values of an induction variable, the base address and stride refer to memory addresses. However, their interpretation is similar. The initial value and the base address represent the first value and address of the first loop iteration, respectively. The increment and stride express how that value and address increases per loop iteration, respectively. Some care is needed when translating between the two, e.g., because of element sizes and the different ways induction variables are used in memory accesses. We explain the details of how we translate between them in Section 4.1.

In this work, we use the GCC's **scalar evolution (SCEV)** analysis, which takes as input a variable, a loop in which the variable is used, and a basic block. If necessary, it outputs a symbolic TREC that can be evaluated at runtime in the given basic block.

The example in Figure 3 and its CDFG in Figure 4 shows the flattened access to a 2D-array. The induction variable `j` is initialized with 0 and incremented by 2 with every iteration of Loop 2. Therefore, its evolution is expressed by the TREC $\{0, +, 2\}_2$. Its value is given by $\{0, +, 2\}_2(x) = 0 + \sum_{k=0}^{x-1} 2$.

TRECs can also represent variables within loop nests. E.g., in Figure 3, the index expression `i*M+j` for the array `inout` is represented by the TREC $\{\{0, +, M\}_1, +, 2\}_2$. Intuitively, the inner TREC represents the evolution caused by iterations of Loop 1 which iterates over each row of the

³To stay consistent with prior work on TRECs, we use curly instead of round brackets to denote the tuple of a TREC.

Fig. 5. Overview of approach and tool architecture.

Fig. 6. Accumulation loop.

formula that decides at runtime whether to run a parallelized or sequential version of a loop. Next, the information needed for parallelization, such as loop structure and increment instructions, are collected. Finally, the CDG transformation performs the parallelization using the information from all the previous steps and takes care of including potential runtime checks. Its output can be synthesized by standard HLS tools.

Our approach supports input programs where each loop has a unique terminal block, memory accesses are regular, and expressions for strides evaluate to positive values. We support parallelization of loops with independent iterations or additive and multiplicative reduction patterns.

Throughout this section, we will use two examples. The first example is the previous example shown in Figures 3 and 4 of a loop nest that increments every second element of a flattened 2D-array by 10. The second example is the accumulation loop shown in Figure 6.

4.1 Memory Analysis

The memory analysis receives as input a CDG and a specific memory access. It calculates the base address and strides of the instruction. It outputs both values as either constants or expressions.

Recall that a memory access can specify the address to access in one of two ways (cf. Section 3.3): either via a base variable or via a base variable and an index variable, both with an optional offset. Intuitively, these variants correspond to accesses via a pointer and a pointer with an array index, respectively. Our analysis supports both variants.

For a memory access using a base variable only, we run GCC’s SCEV analysis on the base variable. It provides the TREC $\{\Phi_a, +, \Phi_b\}_k$ for the innermost loop k that contains the memory access.

Φ_a is another TREC that is either nested (Φ_{k-1} , in the case of nested loops) or a constant. The initial value of TREC Φ_1 (i.e., the TREC of the outermost loop) represents the base address of the memory access. The stride for loop j with $j \leq k$ is the increment of its TREC Φ_j , where we require that the increment of Φ_j is a constant.

For a memory access using a base variable and an index variable, we run GCC's SCEV analysis on both variables. We require one of them to be a constant c and the other to be a TREC. We apply an analysis to the TREC as in the previous case, but perform two additional computations before we return the result. First, we scale the resulting strides by the element size of the array. Second, the constant c is added to the resulting base address.

We apply two techniques to improve precision in case the initial SCEV analysis fails. First, we optimize what basic block we pass to the SCEV. As a default, we always pass the preheader of the outermost loop to the analysis, since a successful analysis with it as the basic block provides most information. If the SCEV analysis fails with that basic block, then we improve the overall precision by iteratively retrying with the preheader of the next directly outer loop. Second, we can work with expressions throughout our approach. More concretely, the memory analysis can output strides, iteration counts, and base addresses as expressions that can be evaluated at runtime. These expressions can refer to values only known at runtime, e.g., function parameters.

Throughout this step, we do not speculate or create approximate values. The base address and strides for memory accesses enable the static and dynamic analyses of their independence.

Given the CDFG in Figure 4, we run the memory analysis on the read instruction in Line 9 and the write instruction in Line 11. Both instructions conform to the first type of memory accesses that use a base variable only. Therefore, the memory analysis passes the variable `_4` to the SCEV analysis. The intended target basic block to evaluate the TREC is the preheader of Loop 1 (Basic Block 3). The result of the SCEV analysis is $\{\{inout_1, +, M * 2\}_1, +, 4\}_2$. The memory analysis outputs the function parameter `inout_1` as base address and strides 4 for Loop 2 and the expression $M*2$ for Loop 1. Note that the factor two in both strides is caused by the size of the data type of array `inout` that is two bytes for a short.

In case of the example shown in Figure 6, we run the memory analysis on the read instruction in Line 5. The instruction conforms to the second type of memory access because the induction variable `i_1` is used as index to access array `in`. Therefore, the memory analysis runs the SCEV analysis separately on variables `in` and `i_1`. The evaluation point is the preheader of Loop 1 (Basic Block 3). The resulting TREC for the array `in` is the variable itself as a constant. The resulting TREC for variable `i_1` is $\{0, +, 1\}_1$. This satisfies our requirement that one SCEV analysis result is constant and one is a TREC. The analysis computes an initial stride of 1 and initial base address of 0. Afterwards, the memory analysis multiplies the initial stride by the element size, which is 4 for `int` arrays, and adds the constant `in` to the initial base address. Therefore, the memory analysis outputs the function parameter `in` as base address and a stride of 4 for Loop 1.

4.2 Static Independence Check

The static independence check receives as input a CDFG and a specific loop. It analyzes whether the iterations of that loop are independent or not. It provides as output one of three answers: yes, no, and maybe (i.e., a runtime check is needed).

The analysis is composed of two steps. First, we analyze whether accesses to scalar variables are independent. Second, we analyze whether accesses to memory are independent. We directly output “yes” if the analyses in both steps pass. We output “maybe” if only the analysis in the first step passes. Otherwise, we output “no.”

4.2.1 Independence of Scalar Variables. We start by collecting data dependencies of the CDFG and classifying them in loop-carried and loop-independent. A data dependency (i, j) is classified as loop-carried, if i can reach j via a back edge. Otherwise, it is loop-independent.

Loop-carried data dependencies prevent parallelization in general. However, there are two exceptions.

- (1) Computations by increment instructions on induction variables produce a loop-carried data dependency. However, the computation performed by the increment instruction can be statically split into different iterations and the loop can be parallelized.
- (2) A loop-carried data dependency that is completely contained inside an inner loop is private to that inner loop. Hence, an outer loop can still be parallelized.

Our analysis takes both exceptions into account. It works as follows: We detect increment instructions by their structure. They have one variable operand with a loop-carried data dependency and a constant or expression that is constant at runtime as their other operand. Finally, we take the set of all dependencies that enter the latch block and are hence relevant in a consecutive loop iteration. Notice that this excludes loop-carried dependencies that are contained in inner loops only (cf. Exception 2), because they traverse via the latch blocks of these inner loops. If all the dependencies in this set result from increment instructions (cf. Exception 1), then the scalar independence check passes.

Consider the loop nest shown in Figure 3 with its CDFG in Figure 4. We want to parallelize the outer loop and, hence, care for dependencies entering the latch block of Loop 1 (Basic Block 9). The set of data dependencies entering the latch block is the singleton set containing the data dependency (15, 17). Notice that it does not contain dependencies of the inner loop, respecting Exception 2. The assignment $i_{14} = i_{22} + 1$ in Line 15, which created the single dependency, has the variable i_{22} with a loop-carried data dependency (17, 15) as its first operand and a constant as its second operand. Thus, we detect it as an increment instruction, respecting Exception 1. In summary, all dependencies entering the latch block result from increment instructions and the scalar independence check passes for the outer loop.

4.2.2 Independence of Memory Accesses. At compile time, we detect two cases in which memory accesses are independent.

- (1) A loop performs read accesses only.
- (2) There is at most one write and one read instruction, and the locations accessed by these instructions are identical and unique across all iterations.

The first condition is fulfilled if all memory accesses in a loop's body are reading accesses. For the second condition, we check typical row-wise and column-wise reading and writing access patterns. If we detect such a pattern and it is identical for both accesses, then the second condition is fulfilled.

The first condition is straightforward to check. To check the second condition, we use information provided by the memory analysis. It provides base addresses, strides, and loop iteration counts for each loop and each memory access. We detect the two access patterns described above via this information. If the strides are such that one iteration of an outer (respectively, inner) loop skips all accessed addresses of an inner (respectively, outer) loop, then the loop has a row-wise (respectively, column-wise) access pattern. We calculate the number of addresses accessed as the product of stride and the loop iteration count decremented by one. The access pattern manifests if the number of accessed addresses is greater than the stride of the outer (respectively, inner) loop. Finally, the access patterns are identical when both accesses have identical base addresses

and strides. If the access patterns are detected in the code and are identical, then the check for the second condition passes.

Consider the loop shown in Figure 3. It contains a reading and a writing access to `inout`, hence, we need to check for the second condition. Both stride and iteration count as given by the memory analysis are symbolic, since they depend on the function's input parameters `N` and `M`. While this allows us to evaluate the equality of the stride's expressions (since their expressions are identical), we can not compare their size to determine whether the stride of the outer loop is sufficiently large to skip the accessed addresses of the inner loop. Hence, for this example, we resort to a runtime check. In contrast, the loop shown in Figure 6 contains reading accesses to the `in` array only. Hence, we detect that it fulfills the first condition and the check performed by the memory analysis passes.

4.2.3 Supporting Reduction Patterns. A *reduction* uses an associative (in our case also a commutative) operation to combine a collection of values to a single value [16]. We call such an operation a *reduction instruction*. Since it summarizes a value across loop iterations, it causes a loop-carried data dependency and has a non-constant value as its second operand. At the same time, it does not prevent parallelization, since partial summaries can be created independently. The partial summaries can later be summarized to the final value. The created loop-carried data dependency is not covered by the exceptions listed in Section 4.2.1. We introduce a special exception to support reductions.

We first identify reduction instructions. Candidates are all addition or multiplication instructions that have a cyclic dependency with an assignment in the latch block. Out of the candidates, reduction instructions are those that have a non-constant second operand. Finally, we consider them when checking the set of dependencies entering the latch block.

Consider the loop in Figure 6. The two dependencies (7, 9) and (6, 10) enter the latch block. Since the instruction in Line 7 is an increment instruction, it is irrelevant due to Exception 1 from before. We thus inspect (6, 10). The assignment `result_2 = _1 + result_1` in Line 6 involves an addition operation and has two operands. The first operand, `_1` is non-constant. The second operand is cyclic with the assignment to `result_1` in Line 10. Hence, we detect the instruction in Line 6 as reduction instruction and can parallelize the loop.

4.3 Information and Selection of Parallelizable Loops

The collection of information and selection of parallelizable loops takes as input a CDFG and a list of parallelizable loops. It analyzes the structure of these loops and provides one loop and its structure out of each loop nest as output.

More precisely, we identify the loop's preheader, terminal block, and all basic blocks in the loop's body. We collect more information for vital instructions such as increment and reduction instructions. For both, we include their operands and initial values. For increment instructions, we additionally signal whether their outputs are used in a loop exit condition.

Finally, we select one loop out of each loop nest. We sort loops according to their nesting structure. Out of each loop nest, we select the outermost loop for parallelization. Note that if a CDFG contains multiple sequential loop nests, then overall multiple loops could be parallelized.

The information is used in the upcoming transformation process. Hence, we ensure that all necessary information is available for a loop that we select for parallelization.

Consider the loop nest in Figure 3. We detect the nesting structure of the two loops and output parallelization information for the outer loop first. We identify the structure of basic blocks in the loop as they are shown in Figure 4. Vital instructions for this example are its increment instruction

Fig. 7. Structure of a parallelized loop with a parallelization factor $P = 2$.

$i_{14} = i_{22} + 1$ with its parameters and the information that it is used in the loop's exit condition and the exit condition itself. The loop shown in Figure 6 outputs similar information together with the reduction instruction $result_2 = _1 + result_1$ and its parameters.

4.4 CDFG Transformation

The final stage takes as inputs the CDFG of a loop that was selected for parallelization and a parallelization factor P (more information on the parallelization factor P will be given in Section 6). It transforms the CDFG so multiple iterations are processed in parallel and outputs the transformed CDFG.

The loop is accelerated by distributing its iterations statically and in a round robin fashion to P identical and independent **loop processing units (LPU)**. We transform the CDFG so after processing one iteration the next $P-1$ iterations are skipped.

We transform the preheader, the loop body, and the terminal separately. Changes to the regular loop structure in Figure 2 are visualized in Figure 7.

4.4.1 Transformation of the Preheader. In the preheader the initial values for the induction variables are calculated. With P LPUs, only the first unit can use the original initial value. Hence, we add additional increment instructions.

As explained in Section 3.3, preheaders have exactly one successor block. Therefore, there is always at least one iteration to be processed once the preheader is entered. If the number of iterations is less than P , then not all LPUs can be used. Therefore, we add additional instructions checking the loop condition of the first $P-1$ iterations. The results are later assigned to the enable signal of the corresponding LPUs.

4.4.2 Transformation of the Loop. We transform the loop body so after calculating one iteration, $P-1$ iterations are skipped.

In the beginning, we temporarily remove all increment instructions and the loop exit condition. The remaining instructions are responsible for calculating a single iteration. Afterwards, we insert a new induction variable for every increment instruction and replace all uses of the old one. The induction variable is incremented P times. If the increment operand is a constant, then this operation is folded into a single instruction. Otherwise, we insert P separate instructions.

Fig. 8. Transformed accumulation loop shown in Figure 6(b) with a parallelization factor $P = 2$.

The loop exit condition has to be evaluated for the processed iteration as well as for the skipped ones, because most loop exit conditions optimized by the GCC are equality checks. In consequence, if an induction variable is part of the loop exit condition, then we add compare instructions for all $P-1$ intermediate induction variables. The result is the conjunction of all compare instructions.

4.4.3 Transformation of the Terminal. We transform the terminal block only if a reduction pattern was detected. In this case, we add additional instructions that retrieve the partially reduced values of the LPUs. Each value is copied into a separate temporary variable and reduced to the final result in the terminal block.

Only accumulation and multiplication patterns are supported. For both operations, the reduction can be done in a minimum-height balanced tree to leverage instruction level parallelism and to keep the overhead in terms of latency low.

4.4.4 Example. The result of the CDFG transformation on the accumulation loop shown in Figure 6(b) is shown in Figure 8. A parallelization factor of $P = 2$ was used. Statements inserted by the transformation are painted in red.

It can be seen that two separate induction variables i_1_dup0 and i_1_dup1 are created in the preheader. They are passed to the induction variable i_1 inside the LPUs. Furthermore, the enable signal for the second LPU is calculated. The special case $N = 0$ where no iteration is executed is already checked in Basic Block 2. Therefore, the first LPU is always enabled. Inside Basic Block 4, the induction variable is incremented twice and the condition $N > i$ is evaluated for both versions of the induction variable. Inside the latch block only i_3 is copied back into i_1 , effectively skipping one iteration. As the example contains a reduction pattern, the results of LPUs are copied into two separate result variables that are accumulated inside Basic Block 6.

5 RUNTIME CHECKS TO COMPLEMENT STATIC ANALYSIS

In the previous section, we presented our static analysis to identify parallelizable software parts. The example in Section 4.2.2 showed, however, that the static analysis has insufficient information to judge that the code in Figure 3 can be parallelized. This is a typical situation and limitation of static analyses in general. If a static analysis cannot reliably deduce a result, then it errs on the safe side.

In this section, we introduce a dynamic analysis that complements the capabilities of our static analysis. In more detail, if our static analysis is inconclusive, then we implement both a parallelized and sequential version at compile time and use a runtime check to make the final verdict which of these two versions to run. This increases precision, because the runtime check can work with concrete values not known statically but only at runtime.

5.1 Where and What to Check

The information what loops might benefit from an additional runtime check is generated by the static analysis from Section 4.2. Our runtime check can provide more precise results when we verify whether memory accesses in a loop are independent or not. Hence, we create a runtime check for loops where we could show the independence of scalar variable accesses statically, but not the independence of memory accesses. Typical situations in which this occurs are cases when the memory analysis can only provide certain values as expressions, e.g., the iteration count of loops, or strides and base addresses of memory accesses.

We place runtime checks directly before the preheader of a loop. Hence, our runtime checks are able to deduce the independence of memory accesses in all loop iterations jointly.

A runtime check verifies whether the conjunction of two properties holds inside a loop.

- (1) Memory addresses that one instruction writes to are not accessed again by another instruction.
- (2) A memory address is written to at most once.

The first property ensures that there are no loop-carried data dependencies between a writing memory access and other memory accesses. The second property ensures that the order in which writes to an address are performed is irrelevant.

Our approach for verifying the first property is to calculate what memory addresses are accessed by each memory access. We represent the accessed addresses by intervals. The check then verifies that these intervals are pairwise disjoint.

Our approach for verifying the second property is similar to a condition we use in the static check. We use the strides available from the memory analysis to compute how the accessed addresses behave in nested loops. In a nested loop, the stride for the outer (respectively, inner) loop must be large enough to skip over the accessed addresses by the inner (respectively, outer) loop. This corresponds to row- (respectively, column-) wise access of an array.

5.2 Creating Runtime Checks

We create a separate check for each of the two properties from above. The resulting runtime check is the conjunction of these two separate checks.

5.2.1 Verifying Disjoint Writing Accesses. To check Property 1, we first construct intervals that contain the accessed memory addresses for each memory access. Then, we compare these intervals pairwise to see whether they are disjoint.

We represent the intervals by their two bounds. The information provided by the memory analysis helps to determine these bounds. The first bound of the interval is given by the base address

```

void f3(short* in, short*out, unsigned N, unsigned M){
    for(int i = 0; i < N; i++)          // Loop 1
        for(int j = 0; j < M; j += 2) // Loop 2
            out[i*M+j] = in[i*M+j] + 10;
}

```

Fig. 9. Example of a loop nest with separate pointers for input and output values.

of the memory access. The base address is the memory address that is accessed in the first loop iteration. Afterwards, since strides represent a regular access pattern, the accessed addresses in consecutive loop iterations are either all greater or all smaller than the base address. This makes the base address a true bound of the interval. The second bound of the interval is the accessed memory address with the furthest distance from the base address, i.e., the address accessed in the last loop iteration. Let s be the stride size and it the loop iteration count. We calculate that distance as $s \times (it - 1)$. Note that one is subtracted from the loop iteration count, since the induction variable is incremented but not accessed in the last loop iteration. In loop nests, the products for each individual loop in the loop nest are summed up to obtain the complete distance. Finally, the furthest accessed memory address is the sum of base address and the calculated distance. We show an example in Section 5.3.

Finally, we create a condition for each pair of memory accesses where at least one is a write. Each condition verifies whether the bounds of the two intervals do not overlap. The resulting check is the conjunction of all of these conditions.

5.2.2 Verifying Writing Only Once. Consider a typical implementation of row-wise access to a two-dimensional array via a pointer. An inner loop accesses each element in a row by incrementing the pointer by the size of a single element. An outer loop skips a complete row by incrementing the pointer (pointing to the first array element in a row) by the size of a complete row. Our check for Property 2 verifies that all writes in a nested loop follow this pattern.

Recall from the previous check that we have the total accessed distance of a memory access for a single loop available. We compare the distance of an inner loop to the stride of its directly outer loop. The stride in the outer loop follows the pattern described above if it is at least as big as the accessed distance of the inner loop. The resulting formula is the conjunction of all these comparisons for each pair of adjacent nested loops in a loop nest.

5.3 Example

Consider the example in Figure 9. The code is a variation of the previous example in Figure 3 in the sense that it uses separate pointers for input and output values. The initial values for both of these pointers are only known at runtime and not at compile time. For certain input values, these reads and writes might overlap. Hence, our static check for memory independence fails and we create a runtime check. Notice that the static check for independence of scalar dependencies passes with the same arguments as presented in Section 4.2.1.

The memory analysis provides expressions $s_{inner_{in}}$, $s_{outer_{in}}$, $s_{inner_{out}}$, and $s_{outer_{out}}$, that evaluate to the strides for the inner and outer loops of the accesses to `in` and `out`, respectively. Both accesses have a constant stride of 4 for the inner loop (increment of 2 times the size of a `short`). GCC provides expressions it_{inner} and it_{outer} that evaluate to the iteration counts of the inner and outer loop, respectively. We calculate the distance by

Fig. 10. Visualized calculation of distance for `in` pointer.

$$d_{in} = s_{outer_{in}} \times (it_{outer} - 1) + s_{inner_{in}} \times (it_{inner} - 1) \text{ and} \\ d_{out.} = s_{outer_{out.}} \times (it_{outer} - 1) + s_{inner_{out.}} \times (it_{inner} - 1).$$

The calculation of d_{in} is visualized in Figure 10. The size of d_{in} (1) is composed of two components. The inner loop (2) performs multiple accesses, each with a stride of $s_{inner_{in}}$ from the previous access. In total, a complete execution of the inner loop performs it_{inner} accesses. Thus, the accessed distance of the inner loop is calculated by $s_{inner_{in}} \times (it_{inner} - 1)$. The starting point of the accessed distance is the furthest distance reached by the outer loop (3). This distance is calculated similarly by multiplying the stride of the outer loop $s_{outer_{in}}$ with its decremented iteration count $(it_{outer} - 1)$. Note that the two distances (2) and (3) do not overlap and, hence, the total distance d_{in} is their sum.

Next, we create the access intervals and verify that they are disjoint. The condition verifying that the interval of reads to `in` is completely before the interval of writes to `out` is

$$\begin{aligned} & in \leq out \quad \wedge \quad in + d_{in} \leq out \\ & \wedge \quad in \leq out + d_{out.} \quad \wedge \quad in + d_{in} \leq out + d_{out.} \end{aligned}$$

A similar alternative condition is created for the other direction where all accesses to `in` are after all accesses to `out`. Combined, these two conditions verify Property 1.

We create the check for a row-wise access pattern. It verifies whether the stride of the outer loop is larger than the amount of accessed addresses in a single run of the inner loop. The condition for the example in Figure 9 is

$$s_{inner_{out}} \times it_{inner} \leq s_{outer_{out.}}. \quad (1)$$

A similar alternative condition is created for a column-wise access pattern. Combined, these two conditions verify Property 2.

Recall the example from Section 4.2.2. Since its static test did not pass, it needs a runtime check. The check for Property 1 statically reduces to “true” when simplified in the implementation. It still needs a check for Property 2 that is similar to the example shown in Equation (1).

5.4 Implementing Runtime Checks in the CDFG Transformation

The CDFG transformation receives the formula of the runtime check along with the information on the loop selected for parallelization.

At runtime there are two possible scenarios. In the first, the runtime check passes and the loop can be processed in parallel. In the second, the runtime check fails and the loop has to be processed sequentially. We implement this behavior into the CDFG by providing a fallback solution for the

Fig. 11. Structure of a parallelized loop with a parallelization factor $P = 2$ and runtime checks.

sequential execution. For that, only the first LPU is started in a specific mode that processes the loop sequentially. This means that we apply transformations to the preheader and the loop body.

Figure 11 shows the changes to the regular structure in Figure 2 for the case that a runtime check was detected. Instructions highlighted in blue separate parallel and sequential operation of a LPU.

5.4.1 Preheader. In the beginning, the formula of the runtime check is split into a set of instructions. Afterwards, we insert them into the preheader and assign the result to a special variable. For the first LPU, the result serves as indicator whether the LPU operates parallelly or sequentially. For the other $P-1$ LPUs, the result is joined with the results of the loop conditions and disables the LPU in case the check failed.

5.4.2 Loop Body. Additionally, we include a switch between parallel and sequential mode into the loop body.

If we added a runtime check into the preheader, then we use its result as an indicator to determine whether the value that was incremented once or multiple times is copied back into the induction variable. For this purpose, we modify the copy instruction in the latch block. Furthermore, either the result of the joint loop exit conditions or the original loop exit condition result is used in the control instruction. Note that the overhead for the fallback mode is limited to inserting multiplexers, because the incremented induction variable and the result for the loop exit condition are already needed for the parallel mode.

6 IMPLEMENTATION

We implemented our approach in two separate tools. We used PIRANHA for HLS- and GCC-related tasks and implemented the static independence check, the creation of runtime checks, and the collection of loops described in Section 4 in roughly 2,450 lines of Python code as an external tool.

The **Plugin for Intermediate Representation Analysis and Hardware Acceleration (PIRANHA)** [6] hooks as a plugin into the GCC and handles code analysis, kernel selection, and accelerator generation autonomously. However, our approach is not limited to PIRANHA. Its prerequisites on an HLS tool are that it produces a loop structure as shown in Section 3.3 and provides a SCEV analysis as shown in Section 3.5.

Fig. 12. Experimental setup using a MicroBlaze soft-core with automatically adapted firmware, a cache system for direct memory accesses, and accelerators created by PIRANHA.

PIRANHA implements the memory analysis (cf. Section 4.1) based on the GCC's SCEV analysis. Its results and the CDFG of the kernel are exported into a file and provided to the external tool.

The input file is parsed into a dependence graph and the information from the memory analysis is embedded. We use NetworkX [25] for performing analyses on the graph, such as identifying back edges and computing paths.

Our tool's output contains a list of parallelizable loops including needed information for parallelization (cf. Section 4.3), such as their structure and potential runtime checks formatted in a tree-like structure. The output is read back by PIRANHA.

PIRANHA transforms the selected loops and the CDFGs are scheduled under resource constraints. The available resources are divided evenly between the LPUs. If the parallelization factor P is not a divisor of the number of resources, then some resources will not be used. Nevertheless, this might lead to performance degradations. For the same reason, P must be low enough such that the created LPUs do not exceed the available resources.

Note that loop parallelization is an optimization applied early in the HLS process. It does not interfere with other HLS optimizations such as speculation, chaining, and loop pipelining.

The LPUs are implemented as independent hardware modules that can be instantiated multiple times within the main accelerator. The values of live-in and live-out variables are passed directly through dedicated ports. Instead of processing the loop directly, the main accelerator starts the LPUs by assigning the input and enable signals and waits for the end of the execution by polling the done signal.

7 EVALUATION

In the following, we discuss our experimental results. In the beginning, we describe the evaluated platform and the benchmarks used in the evaluation. Afterwards, we evaluate the benchmarks with regard to speedup and area. Finally, we compare our results to a commercial HLS tool optimized for Xilinx FPGAs and to the vectorization approach presented in Reference [10].

7.1 Evaluation Platform

In this evaluation, we used a system-on-chip that consists of a MicroBlaze soft-core [27], a cache system, and the hardware accelerators. It is shown in Figure 12. The system contains one or more accelerators, depending on the number of kernels found in the firmware. The firmware is adapted automatically by PIRANHA to call the accelerators. The cache system consists of one 16 KB cache connected to the MicroBlaze and four 8 KB caches connected to the accelerators. Therefore, the

Table 1. Characteristics of the Evaluated Benchmarks

Benchmarks	Complexity		Time per Data Transfer		
	Function Calls	Unbound Innermost Loops	Clock Cycles MicroBlaze	Input/Output Data [Bytes]	Clock Cycles/Byte of Data
Add	1	X	147K	40,000	3.69
Blackscholes	63	✓	77,734K	256	303,648.98
Boxfilter	1	X	1,820K	24,024	75.74
Dotproduct	1	X	126K	48,000	2.62
Hash	0	X	3,727K	7,216	516.55
Histogram	1	X	893K	144,020	6.20
Mandelbrot	0	X	441,052K	65,536	6,729.92

Color coded classification: Green means beneficial, yellow neutral, and red undesirable.

accelerators have direct memory access. Before the execution of the benchmark, the input data is either stored in the core cache or the main memory.

All systems were synthesized for a Nexys Video Artix-7 (XC7A200T) FPGA board. The DDR3 memory was used as main memory. Clock cycles are measured for the whole application excluding only result checking and printing. Note that this includes parts of the algorithm that are not accelerated and therefore executed by the MicroBlaze.

7.2 Characteristics of the Evaluated Benchmarks

For better comparability of our work, we chose a set of benchmarks previously used in the field of HLS. They are distributed with the LegUp HLS tool [3] and identical to the benchmarks used to evaluate the vectorization approach proposed by Lattuada and Ferrandi in Reference [10].

The speedup of a hardware accelerator is mainly limited by two factors: The benchmark's complexity and its sensitivity towards the memory bandwidth.

Table 1 shows two metrics that we use to characterize a benchmark's complexity: The number of function calls inlined by PIRANHA and whether the algorithm contains unbound innermost loops.

A small number of function calls is considered positive (green), because it most likely results in simpler control flow. While most benchmarks have none or at most one function call inside the kernel, *Blackscholes* includes 63 partially nested function calls.

The countability of a loop is an important indicator for a benchmark's complexity, because loops that are not countable can not be optimized as efficiently. Especially unbound, innermost loops can prevent optimizations such as loop unrolling and the presented loop parallelization optimization. It can be seen that *Blackscholes* is the only benchmark that contains innermost loops of which the loop count can neither be expressed as constant nor as expression. Again, this indicates a rather complex control flow structure.

Besides the complexity, it is important to characterize the benchmarks' sensitivity towards the memory bandwidth. As explained in Section 7.1, the cache system provides four accelerator caches. Therefore, the accelerator can process up to four memory accesses each clock cycle. Nevertheless, this can only be achieved for a perfect cache hit rate of 100%. The actual cache hit rate depends on many different factors, such as cache size, access pattern, and data reuse. Therefore, the actual bandwidth is hard to predict.

Ahead of the execution of the benchmark, the input data is either stored in the core cache or the main memory. In any case, the input data has to be transferred into the accelerator caches through the coherence bus or the interconnect to the main memory. Likewise, the output data has to be

Table 2. Experimental Results Evaluated for Setup Shown in Figure 12 on a Xilinx Artix-7 XC7A200T FPGA

Benchmark	P	Cycles (Speedup)	LUT	Area (Ratio)		FMax [MHz] (Ratio)
				FF	DSP	
Add	1	58K (1.00)	613 (1.00)	733 (1.00)	0 (-)	98 (1.00)
	2	39K (1.49)	1,561 (2.55)	1,561 (2.13)	0 (-)	95 (0.97)
	4	33K (1.78)	2,804 (4.57)	3,144 (4.29)	0 (-)	93 (0.95)
Blackscholes	1	3,252K (1.00)	12,136 (1.00)	9,964 (1.00)	188 (1.00)	91 (1.00)
	2	1,628K (2.00)	32,043 (2.64)	22,468 (2.25)	342 (1.82)	77 (0.84)
	4	816K (3.98)	73,269 (6.04)	46,347 (4.65)	444 (2.36)	71 (0.77)
Boxfilter	1	392K (1.00)	2,489 (1.00)	2,496 (1.00)	0 (-)	93 (1.00)
	2	204K (1.92)	6,151 (2.47)	5,335 (2.14)	0 (-)	92 (0.98)
	4	142K (2.76)	11,091 (4.46)	9,646 (3.86)	0 (-)	91 (0.98)
Dotproduct	1	61K (1.00)	727 (1.00)	990 (1.00)	3 (1.00)	96 (1.00)
	2	44K (1.38)	1,791 (2.46)	1,928 (1.95)	6 (2.00)	92 (0.96)
	4	44K (1.38)	2,792 (3.84)	3,304 (3.34)	12 (4.00)	91 (0.95)
Hash	1	184K (1.00)	18,218 (1.00)	15,630 (1.00)	18 (1.00)	79 (1.00)
	2	184K (1.00)	18,217 (1.00)	15,630 (1.00)	18 (1.00)	77 (0.97)
	4	184K (1.00)	18,235 (1.00)	15,630 (1.00)	18 (1.00)	76 (0.96)
Histogram	1	268K (1.00)	3,500 (1.00)	4,684 (1.00)	0 (-)	92 (1.00)
	2	268K (1.00)	5,103 (1.46)	6,482 (1.38)	0 (-)	91 (0.99)
	4	268K (1.00)	6,225 (1.78)	8,315 (1.78)	0 (-)	91 (0.99)
Mandelbrot	1	3,343K (1.00)	1,577 (1.00)	1,309 (1.00)	60 (1.00)	98 (1.00)
	2	1,672K (2.00)	3,264 (2.07)	2,599 (1.99)	120 (2.00)	95 (0.96)
	4	838K (3.99)	6,416 (4.07)	4,861 (3.71)	240 (4.00)	91 (0.93)

transferred back. Both coherence bus and memory interconnect can only process one request at a time, creating a bottleneck. Therefore, a good and simple estimation for a benchmark's sensitivity towards the memory bandwidth is to evaluate the amount of input and output data in relation to the time spent by the processor alone to process the benchmark.

Table 1 shows the number of clock cycles per byte of data. A low number (<10 , red) indicates that the bottleneck of the accelerator will be most likely the memory bandwidth. Benchmarks with an intermediate number (≥ 10 and $<1,000$, yellow) are most likely influenced by both: the bandwidth as well as the benchmark's complexity. A high number ($\geq 1,000$, green) indicates that the accelerator is most likely only limited by the structure of the benchmark and the amount of available parallelism exploited through different optimizations. In this case, the overall time spent on read and write operations is negligible.

7.3 Benchmark Evaluation

The results for different parallelization factors P are shown in Table 2. The four available caches for the accelerator define the limits of P . The speedup and the ratios are given relative to an accelerator without loop parallelization ($P = 1$).

Out of these seven benchmarks, static analysis suffices only for *Mandelbrot* to identify parallelizable loops at compile time. For *Add*, *Blackscholes*, *Boxfilter*, *Dotproduct*, and *Histogram*, checks were included to verify independence at runtime. *Hash* is the only benchmark that does not contain any parallelizable loops.

The *Hash* benchmarks calculates hash values for a set of data and collects the number of collisions in an array. To access this array, the hash itself is used as index. Therefore, the read and write accesses are random and might cause aliasing. This behavior is detected correctly by our analysis and the loops are not parallelized. As a result, clock cycles, area, and maximum clock frequency are unchanged for all values of P .

The *Histogram* algorithm contains a loop that is parallelized. Nevertheless, this loop does not contribute significantly to the execution time. The main loop of the algorithm in turn uses a temporary array to collect the data. This array is written at fixed indexes and can not be shared between LPUs. Our tool legitimately detects this behavior and does not parallelize the loop.

The speedups of *Blackscholes* and *Mandelbrot* scale perfectly with P ($2\times$ for $P = 2$, $\sim 4\times$ for $P = 4$). As indicated in Table 1, both benchmarks are compute-intensive, as the vast majority of operations are arithmetic instructions. It is important to note that the complex structure of *Blackscholes* does not prevent parallelization, since the memory accesses themselves follow a regular pattern that is analyzable by the SCEV analysis. The complex structure caused by the 63 function calls and the small, unbound, innermost loops only affects calculations based on scalar variables. Therefore, the static independence check can analyze the CDFG without problems.

Finally, the speedups of *Add*, *Boxfilter*, and *Dotproduct* improve with increasing P even though they do not scale linearly. *Add* has a speedup of $1.49\times$ for $P = 2$ and $1.78\times$ for $P = 4$. *Boxfilter* scales almost perfectly for $P = 2$ ($1.92\times$) but not for $P = 4$ ($2.76\times$). For *Dotproduct* the speedup of $1.38\times$ is identical for $P = 2$ and $P = 4$. This is caused by the memory dominated nature of these benchmarks, which makes them more sensitive to memory bandwidth. In consequence, LPUs wait for the completion of memory accesses most of the time.

Table 3 shows the overhead produced by the initialization, the runtime check, the LPU control, and the result reduction. It can be seen that the additional clock cycles spent are less or equal than five for *Add*, *Blackscholes*, *Boxfilter*, *Dotproduct*, and *Histogram*. The reason for this is that the parallelized region of these benchmarks is only entered once. Therefore, the overhead is only computed once. Merely *Mandelbrot* takes 512 additional clock cycles. The reason for this is that not the innermost but the middle loop is parallelized. As a result, the overhead has to be computed for every iteration of the outermost loop. For a single iteration, four additional clock cycles are needed. Note that *Mandelbrot* does not implement a runtime check. Nevertheless, the additional clock cycles are spent on initialization, LPU control, and the result reduction of the identified reduction pattern. In any case, the overhead is negligible in comparison to the total runtime of the benchmarks.

The evaluation of the area in Table 2 shows that the numbers of **look up tables (LUT)**, **flip flops (FF)**, and DSP blocks scale linearly with P . This is expected, because effectively, the loop body is implemented P times. A special case is the *Blackscholes* benchmark. Here, the LUT consumption increases by a factor of 6.04 for $P = 4$, while the DSP utilization only increases by a factor of 2.36. This is caused by PIRANHA's resource-sharing algorithm. Specifically, the DSP consuming 64-bit multiplication operators are shared more often for $P = 4$ than for $P = 1$. Hence, additional LUTs are needed to implement input multiplexers while the number of DSPs does not grow linearly with P .

Some additional area overhead is needed for the interconnect of the LPUs, the runtime checks, and the instructions inserted to compute multiple iteration conditions. The resource consumption is shown in Table 3. It can be seen that *Mandelbrot* requires the fewest additional LUTs and FFs, because it is the only benchmark for which no additional runtime checks were needed. The size of the runtime check depends on the number of memory accesses that have to be checked for aliasing. Therefore, *Boxfilter* (9 reads/1 write) and *Histogram* (7 reads due to unrolling/1 write) have the largest overhead. For *Add*, *Blackscholes*, and *Dotproduct* (1 or 2 read/1 write) the overhead is between 166 and 383 LUTs and between 106 and 262 FFs. No DSP blocks were needed.

Table 3. Overhead Produced by Initialization, Runtime Check, LPU Control, and Results Reduction Evaluated on a Xilinx Artix-7 XC7A200T FPGA

Benchmark	P	Total Cycles	Cycles	Overhead		
				LUT	FF	DSP
Add	1	58K	0	0	0	0
	2	39K	5	269	119	0
	4	33K	5	257	106	0
Blackscholes	1	3,252K	0	0	0	0
	2	1,628K	3	166	197	0
	4	816K	3	226	261	0
Boxfilter	1	392K	0	0	0	0
	2	204K	4	1,336	579	0
	4	142K	4	1,349	632	0
Dotproduct	1	61K	0	0	0	0
	2	44K	4	383	204	0
	4	44K	5	318	262	0
Hash	1	184K	-	-	-	-
	2	184K	-	-	-	-
	4	184K	-	-	-	-
Histogram	1	268K	0	0	0	0
	2	268K	4	815	405	0
	4	268K	4	844	526	0
Mandelbrot	1	3,343K	0	0	0	0
	2	1,672K	512	53	164	0
	4	838K	512	65	228	0

In terms of clock frequency it can be seen in Table 2 that most benchmarks run at around 93 MHz with little deviation, depending on P. In all cases the critical path is part of the cache system. There are a couple of outliers. *Mandelbrot* can run on up to 98 MHz for P = 1, while *Blackscholes* only runs at 71 MHz for P = 4. The drop in the clock frequency is caused by congestions on the FPGA due to the large design. Nevertheless, the lower clock frequency is compensated by the speedup in cycles.

All benchmarks are self-checking, and the evaluation showed that no erroneous parallelization decision was made. Furthermore, it can be summarized that compute-intensive benchmarks run P times as fast as the unoptimized version. For P = 4, an arithmetic average speedup of 2.27× was achieved. At the same time, the consumption of LUTs, FFs, and DSP blocks was increased by the parallelization factor with some overhead. Since the speedup was essentially gained without any user interaction, we consider this a good tradeoff.

7.4 Comparison to a Commercial HLS Tool

Next, we will compare the PIRANHA generated accelerator to an industrial-strength HLS tool that is optimized for Xilinx FPGA. In contrast to PIRANHA, the commercial tool lets the user choose which parts of the code to accelerate, which optimizations to apply, and how to interface the accelerator.

Unfortunately, we are not able to compare with the Intel HLS compiler or LegUp. Both would require specific Intel FPGAs to fully operate. Since we do not have access to such hardware, we cannot use these two HLS tools.

Fig. 13. Experimental setup used to evaluate a commercial HLS tool (CT) using a MicroBlaze soft-core with user-adapted firmware, dual-ported BRAMs for input and output data, and accelerators created by the commercial HLS tool.

Table 4. Optimizations Used to Create the Performance-optimized Version with the Commercial Tool

Benchmark	Inlining	Unrolling	Pipelining	Array Partitioning
Add	✓	✓	✓	
Blackscholes				
Boxfilter	✓	✓	✓	✓
Dotproduct	✓	✓	✓	
Hash	✓		✓	✓
Histogram	✓	✓	✓	✓
Mandelbrot		✓	✓	✓

The experimental setup can be seen in Figure 13 and consists of a MicroBlaze, a single cache connected to the main memory, and one or more accelerator. The commercial tool automatically creates BRAMS for all input and output arrays of the kernel. The data has to be transferred to and from the accelerator ahead of execution. Therefore, the firmware has to be adapted. In contrast to PIRANHA where all adaptations are inserted automatically into the firmware, this has to be done manually for the accelerators created with the commercial HLS tool.

Two accelerator versions were created with the commercial HLS tool. For one, as little user intervention as possible was used to simulate similar conditions to PIRANHA where optimizations are chosen automatically. For the other, a performance-optimized version was created by telling the HLS tool explicitly to use inlining, unrolling, loop pipelining, and array partitioning. For each benchmark the optimizations were configured so the resulting execution time was minimized. A detailed overview on which optimizations were used for which benchmark is shown in Table 4.

The results are shown in Table 5. For a better comparison, the latencies using PIRANHA were color-coded comparing PIRANHA without parallelization with the minimum user intervened version of the commercial tool and PIRANHA with a parallelization of $P = 4$ with the performance-optimized version of the commercial tool. Green means PIRANHA performed better than the commercial tool. Red means the commercial tool performed better than PIRANHA.

First, we compare those benchmarks that have a low clock cycles per data ratio according to Table 1 and are therefore considered bandwidth-sensitive. Those are *Add*, *Dotproduct*, and *Histogram*. For all three benchmarks it can be seen that the accelerators generated by the commercial tool can not generate any speedup. Optimizing the accelerator only has a minor effect on the

Table 5. Comparison of PIRANHA to a Commercial HLS Tool (CT) Evaluated on a Xilinx Artix-7 XC7A200T FPGA

Benchmark	Version	Cycles	Fmax [MHz]	Latency [ms]	Area		
					LUT	FF	DSP
Add	No Accelerator	147K	97	1.52	-	-	-
	PIRANHA P = 1	58K	98	0.60	613	733	0
	PIRANHA P = 4	33K	93	0.36	2,804	3,144	0
	CT - Min. User	238K	98	2.42	135	227	0
	CT - Perf. Opt.	228K	99	2.31	138	216	0
Blackscholes	No Accelerator	77,734K	97	799.90	-	-	-
	PIRANHA P = 1	3,252K	91	35.60	12,136	9,964	188
	PIRANHA P = 4	816K	71	11.54	73,269	46,347	444
	CT - Min. User	1,232K	82	15.04	4,391	3,684	89
	CT - Perf. Opt.	1,232K	82	15.04	4,391	3,684	89
Boxfilter	No Accelerator	1,820K	97	18.72	-	-	-
	PIRANHA P = 1	392K	93	4.21	2,489	2,496	0
	PIRANHA P = 4	142K	91	1.56	11,091	9,646	0
	CT - Min. User	405K	98	4.12	898	967	0
	CT - Perf. Opt.	128K	97	1.32	14,494	11,358	0
Dotproduct	No Accelerator	126K	97	1.30	-	-	-
	PIRANHA P = 1	61K	96	0.64	727	990	3
	PIRANHA P = 4	44K	91	0.49	2,792	3,304	12
	CT - Min. User	276K	98	2.81	278	311	3
	CT - Perf. Opt.	258K	101	2.56	286	303	3
Hash	No Accelerator	3,727K	97	38.36	-	-	-
	PIRANHA P = 1	184K	79	2.34	18,218	15,630	18
	PIRANHA P = 4	184K	76	2.44	18,235	15,630	18
	CT - Min. User	332K	99	3.36	1,521	1,759	3
	CT - Perf. Opt.	46K	92	0.49	3,783	2,957	3
Histogram	No Accelerator	893K	97	9.19	-	-	-
	PIRANHA P = 1	268K	92	2.91	3,500	4,684	0
	PIRANHA P = 4	268K	91	2.93	6,225	8,315	0
	CT - Min. User	893K	86	10.34	655	523	0
	CT - Perf. Opt.	857K	92	9.32	736	579	0
Mandelbrot	No Accelerator	441,052K	97	4,538.51	-	-	-
	PIRANHA P = 1	3,343K	98	34.02	1,577	1,309	60
	PIRANHA P = 4	838K	91	9.19	6,416	4,861	240
	CT - Min. User	2,007K	99	20.36	409	338	12
	CT - Perf. Opt.	352K	92	3.83	11,788	6,759	582

Color-coded comparison: PIRANHA P = 1 vs. CT - Min. User and PIRANHA P = 4 vs CT - Perf. Opt. Green means PIRANHA performed better than the CT. Red means the CT performed better than PIRANHA.

latency. The reason for this is that most of the execution time is spent transferring the data into and out of the BRAMs. In contrast, PIRANHA decreases the latency by factors of 2.5 (*Add*), 2.0 (*Dotproduct*), and 3.2 (*Histogram*) even without parallelization. Even though the parallelization of *Add* and *Dotproduct* does not decrease the latency by a factor of P, there is still a noticeable speedup.

Comparing the resource consumption of those three benchmarks, one can see that the accelerators generated by PIRANHA are larger than those created by the commercial tool. Nevertheless, the resources are well spent, considering the improvements in the latency.

Next, we compare those benchmarks that have an intermediate clock cycles per data ratio according to Table 1. Those are *Boxfilter* and *Hash*. Comparing the latencies of PIRANHA's unparallelized version with the commercial tool's version with minimum user intervention, PIRANHA performs almost equally or outperforms the commercial tool (*Boxfilter*: 4.21 ms vs. 4.12 ms, *Hash*: 2.34 ms vs. 3.36 ms). Nevertheless, the commercial tool can generate significant improvements if inlining, unrolling, loop merging, pipelining, and array partitioning are applied. The reason for this is more efficient data reuse inside the accelerator. As discussed in Section 7.3, PIRANHA can only parallelize the *Boxfilter* algorithm. As result, the parallelized accelerator runs only slightly slower than the user-optimized accelerator created by the commercial tool (1.56 ms vs. 1.32 ms). Parallelizing the *Hash* is not possible. Therefore, PIRANHA is outperformed by the commercial tool.

Comparing the area of the *Boxfilter*, one can see that the smallest and the largest accelerators are created by the commercial tool. Even though PIRANHA increases the area by $4.46\times$ (LUT) and $3.86\times$ (FF) once parallelization is activated, the performance optimizations by the commercial tool increase the area consumption even more.

For the *Hash* algorithm, the commercial tool creates significantly smaller accelerators. The reason for this is that a lot of area inside PIRANHA's accelerator is occupied by the implementation of the modulo operations. It can be assumed that the commercial tool uses a smaller implementation.

Last, we evaluate *Mandelbrot* and *Blackscholes*, which have a high clock cycles per data ratio. Unfortunately, for the *Blackscholes* algorithm, performance optimization could not be applied by the commercial tool or it led to a kernel exceeding the resources of the FPGA. There are two reasons for this. First, the total of 63 function calls could not be inlined without increasing the resource consumption excessively. Second, the algorithm contains a lot of small, unbound, innermost loops that prevent unrolling, loop merging, and pipelining.

Comparing the latencies of the *Blackscholes* algorithm, one can see that PIRANHA outperforms the commercial HLS tool for $P = 4$ (11.54 ms vs. 15.04 ms). However, the accelerator created with the commercial HLS tool requires less resources on the FPGA. The reason for this are more efficient resource sharing and reutilization, because inlining is not applied as excessively.

For *Mandelbrot*, the commercial tool is better than PIRANHA. Especially the performance optimizations reduce the execution time significantly. The reason is that accessing the data through BRAMs is not a bottleneck, and the structure of the benchmark is regular. Therefore, the commercial tool can create a highly efficient pipeline. Similar to the *Boxfilter* benchmark, the commercial tool creates the smallest and the largest accelerator for *Mandelbrot*. The resource consumption is increased by factors of 28 (LUT), 20 (FF), and 48 (DSP) once performance optimizations are enabled.

Finally, we compare the latencies of the parallelized accelerators created by PIRANHA with the minimum user intervened accelerators created with the commercial tool. One can see that PIRANHA performs better for each of the seven benchmarks. Note that in this scenario it is still less effort to integrate hardware accelerators using PIRANHA than the commercial tool, because PIRANHA applies optimizations automatically and takes care of the integration. In contrast, even the accelerators created with the commercial tool with minimum user intervention need to be integrated manually by the user.

7.5 Comparison Using Different Problem Sizes

To better understand the characteristics of the benchmarks, we swept the problem size and determined its impact on the performance of PIRANHA with the implemented parallelization approach as well as the commercial tool.

Fig. 14. Sweep of the problem size comparing the accelerators generated by PIRANHA with and without parallelization to the performance optimized accelerators created with the commercial tool. All benchmarks were evaluated on a Xilinx Artix-7 XC7A200T FPGA.

We used the same experimental setup as in the previous evaluations in Sections 7.3 and 7.4. We swept the problem sizes of the seven benchmarks logarithmically from 1/100 of the original size to 5× the original size. The only exception was *Blackscholes*, because the original problem size was too small to be scaled downwards sufficiently. Therefore, we swept the problem size between 1 and 500.

The results can be seen in Figure 14. Each graph shows the execution time of the benchmark optimized by PIRANHA with and without parallelization as well as the user-optimized version generated by the commercial tool. Furthermore, the software execution is shown. For better comparability, both axes are scaled logarithmically and the original problem size is marked.

For benchmarks that, according to Table 5, have an intermediate or high clock cycles per byte of data ratio (*Blackscholes*, *Boxfilter*, *Hash*, and *Mandelbrot*), the clock cycle latency scales linearly

Fig. 15. Experimental setup used in Reference [10] to evaluate the vectorization approach including a dual-ported BRAM and an accelerator created by Bambu. The accelerator is evaluated as standalone.

with the problem size for all four versions. As a result, the problem size does not influence the speedup achieved by PIRANHA or the commercial tool.

The outliers in the evaluation of the *Blackscholes* algorithm for the problem sizes of one and two can be explained by the fact that no accelerator was built by PIRANHA. The reason is that PIRANHA selects kernels on loop level but the GCC eliminated the main loop in both cases due to the limited number of iterations.

For *Add*, *Dotproduct*, and *Histogram* (all benchmarks with a low clock cycles per byte of data ratio) PIRANHA performs better for smaller problem sizes. In contrast, the performance of the commercial HLS tool is not influenced by the problem size. The reason is the cache system. Benchmarks with a low clock cycles per data ratio are more sensitive towards cache delays. Once the problem size exceeds the cache size, the number of coherence misses increases and data needs to be fetched from main memory. This increases the latency to load the data and therefore reduces the speedup.

In conclusion, the question of whether PIRANHA or the commercial tool performs better does not depend on the problem size but rather on the number of calculations per data. Furthermore, PIRANHA achieves better speedups when the data does not exceed the cache limit.

7.6 Comparison to Vectorization Approach

As explained in Section 2, in Reference [10], Lattuada and Ferrandi present an optimization to parallelize do-all loops using vectorized instructions. Their objective is to save resources by enabling resource sharing between iterations. In contrast to our approach, they do not implement any analysis that determines whether a loop can be parallelized. Instead, they rely on OpenMP annotations in the code.

Lattuada and Ferrandi implemented their approach into the Bambu HLS tool [18]. The experimental setup can be seen in Figure 15. The accelerators are evaluated as standalone. No processor core is integrated. Instead, the accelerators are connected directly to dual-ported BRAM where the data is stored ahead of execution.

Their measurement procedure and setup deviates from ours. Data transfer times to move input and output data to and from the accelerator were not taken into account when measuring the execution time of the accelerators. Furthermore, the accelerator can issue two independent memory accesses in each clock cycle without any delay due to the dual-ported nature of the BRAM. In comparison, in our setup the data is stored either in the core cache or in the main memory before the execution of the benchmark. When accessing the data parallelly through the accelerator caches, the interconnect to the main memory and the coherence bus effectively sequentializes the requests creating a bottleneck. As a result, an exact comparison is impossible, because benchmarks

Table 6. Comparison of PIRANHA Evaluated on a Xilinx Artix-7 XC7A200T FPGA to the Vectorization Approach Presented in Reference [10] for a Xilinx Virtex-7 xc7vx690t FPGA

Benchmark	P	Parallelization				Vectorization, [10]			
		Speedup	Ratio			Speedup	Ratio		
			LUT	FF	DSP		LUT	FF Pairs	DSP
Add	1	1.00	1.00	1.00	-	1.00	1.00	-	-
	2	1.49	2.55	2.13	-	2.00	0.88	-	-
	4	1.78	4.57	4.29	-	4.02	0.88	-	-
Blackscholes	1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	2	2.00	2.64	2.25	1.82	1.84	1.58	2.16	2.16
	4	3.98	6.04	4.65	2.36	3.35	2.70	4.83	4.83
Boxfilter	1	1.00	1.00	1.00	-	1.00	1.00	-	-
	2	1.92	2.47	2.14	-	1.91	1.90	-	-
	4	2.76	4.46	3.86	-	3.03	3.67	-	-
Dotproduct	1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	2	1.38	2.46	1.95	2.00	1.33	1.27	2.00	2.00
	4	1.38	3.84	3.34	4.00	1.60	1.60	4.00	4.00
Hash	1	1.00	1.00	1.00	1.00	1.00	1.00	-	-
	2	1.00	1.00	1.00	1.00	2.00	1.73	-	-
	4	1.00	1.00	1.00	1.00	2.91	2.29	-	-
Histogram	1	1.00	1.00	1.00	-	1.00	1.00	-	-
	2	1.00	1.46	1.38	-	1.80	1.44	-	-
	4	1.00	1.78	1.78	-	2.93	2.32	-	-
Mandelbrot	1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	2	2.00	2.07	1.99	2.00	2.00	1.62	2.00	2.00
	4	3.99	4.07	3.71	4.00	3.92	2.88	4.00	4.00

limited by the memory bandwidth will show lower speedups in our setup, since the accelerator can not access the data uninterrupted and in parallel. Nevertheless, we believe that an evaluation focusing on the quality of the approach is possible.

The speedups, as they were presented in Reference [10], are shown in Table 6. In difference to our results, Lattuada and Ferrandi are able to achieve speedups for *Hash* and *Histogram*. For *Histogram*, they are supporting thread local arrays. By using OpenMP as input, the compiler activates OpenMP specific optimizations that run before the HLS tool. As a result, this array is duplicated and each thread can access their own copy. In our setup this array exists only once and therefore cannot be shared. The *Hash* benchmark can only be parallelized if the implementation supports atomic memory accesses for the collision result array. The results suggest that they support it.

As shown in Table 1, *Mandelbrot* and *Blackscholes* are computation-dominated. Therefore, it is possible to compare the speedups directly. For the *Blackscholes* algorithm, Reference [10] achieves a speedup of $3.35\times$ for $P=4$. The reason why they are not able to exploit the full parallelization potential is that their approach uses predication where operations of all control paths get executed. For *Mandelbrot*, both approaches achieve similar results.

Add, *Boxfilter*, and *Dotproduct* are benchmarks that highly rely on memory bandwidth. In fact, even the authors of Reference [10] identify this as the main reasons why the speedup grows less than the factor P . In consequence, significant higher bandwidth achieved through the dual ported BRAM is the main cause for their approach resulting in better speedups. Additionally, the data

transfer time to copy the data into and out of the BRAMs was not considered in the evaluation. As shown during the previous evaluation in Section 7.4, data transfer times can contribute a significant amount of time to the overall execution time of a benchmark.

The relative area consumption presented in Reference [10] is also shown in Table 6. In comparison to our evaluation, LUT and FFs are not evaluated separately but as LUT-FF-Pairs. This is less precise. Therefore, only a simplistic comparison can be given. The results can be summarized as follows: Accelerators optimized using the vectorization approach grow less than ours because the vectorization approach provides the flexibility to selectively save area for instructions where performance is not critical. They can choose to either compute P values parallelly or sequentially. The sequential version is slower but saves space.

8 CONCLUSION

In this article, we presented an end-to-end approach to improve autonomous loop parallelization in HLS. Our approach analyzes what and how loop nests can be parallelized correctly. Further, it realizes this potential by distributing the iterations of a loop to multiple loop processing units, all without any annotations or interventions needed by the user.

Our analyses support several features to make them more applicable to a wide range of applications. We can accelerate do-all loops and loops implementing a reduction pattern. If a decision whether a loop can be parallelized or not is inconclusive at compile time, we postpone the decision via a runtime check. This check is created statically and can refer to values and expressions that are only known at runtime, e.g., concrete values of pointers or numbers of loop iterations.

We evaluated our approach using established benchmarks and compared it to academic and commercial tools for HLS. With four loop iterations running in parallel, we achieve ideal speedups of up to 4× and on average speedups of 2.27×, both in comparison to an unoptimized accelerator. For all codes in our evaluation, we were able to achieve higher performance autonomously than a commercial tool without expert-guided optimizations. For compute-intensive code, we were able to achieve speedups autonomously that exceed or are comparable to speedups achieved by a vectorizing academic tool with user-provided annotations.

ACKNOWLEDGMENTS

We thank the reviewers for their helpful comments.

REFERENCES

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd Edition). Addison-Wesley Longman Publishing Co., Inc.
- [2] Randy Allen and Ken Kennedy. 1987. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.* (Oct. 1987), 491–542. DOI : <https://doi.org/10.1145/29873.29875>.
- [3] Jason Anderson. 2020. LegUp High-Level Synthesis. Retrieved from <http://legup.eecg.utoronto.ca/>.
- [4] Rohit Atre, Zia Ul-Huda, Felix Wolf, and Ali Jannesari. 2019. Dissecting sequential programs for parallelization—An approach based on computational units. *Concurr. Comput.: Pract. Exper.* 31, 5 (2019), e4770. DOI : <https://doi.org/10.1002/cpe.4770>
- [5] J. Choi, S. Brown, and J. Anderson. 2013. From software threads to parallel hardware in high-level synthesis for FPGAs. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. 270–277.
- [6] G. Hempel, C. Hochberger, and M. Raitza. 2012. Towards GCC-based automatic soft-core customization. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL)*.
- [7] Microchip Technology Inc. 2021. LegUp 9.1 Documentation. Retrieved from <https://download-soc.microsemi.com/FPGA/HLS-EAP/docs/legup-9.1-docs/index.html>.
- [8] Intel Corporation. 2020. Intel® High Level Synthesis Compiler Pro Edition—Reference Manual. Retrieved from <https://www.intel.com>.

- [9] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. 2007. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. Association for Computing Machinery, 205–214. DOI : <https://doi.org/10.1145/1229428.1229474>
- [10] Marco Lattuada and Fabrizio Ferrandi. 2017. Exploiting vectorization in high level synthesis of nested irregular loops. *J. Syst. Archit.* 75 (2017), 1–14. DOI : <https://doi.org/10.1016/j.sysarc.2017.03.001>
- [11] G. Liu, M. Tan, S. Dai, R. Zhao, and Z. Zhang. 2017. Architecture and synthesis for area-efficient pipelining of irregular loop nests. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 36, 11 (2017), 1817–1830.
- [12] J. Liu, S. Bayliss, and G. A. Constantinides. 2015. Offline synthesis of online dependence testing: Parametric loop pipelining for HLS. In *Proceedings of the IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 159–162.
- [13] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides. 2017. Run fast when you can: Loop pipelining with uncertain and non-uniform memory dependencies. In *Proceedings of the 51st Asilomar Conference on Signals, Systems, and Computers*. 126–130.
- [14] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides. 2018. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 37, 9 (2018), 1802–1815.
- [15] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. 2006. POSH: A TLS compiler that exploits program structure. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. Association for Computing Machinery, 158–167. DOI : <https://doi.org/10.1145/1122971.1122997>
- [16] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA.
- [17] Mohammad Norouzi, Felix Wolf, and Ali Jannesari. 2019. Automatic construct selection and variable classification in OpenMP. In *Proceedings of the ACM International Conference on Supercomputing (ICS'19)*. Association for Computing Machinery, 330–341. DOI : <https://doi.org/10.1145/3330345.3330375>
- [18] C. Pilato and F. Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Proceedings of the 23rd International Conference on Field programmable Logic and Applications*. 1–4. DOI : <https://doi.org/10.1109/FPL.2013.6645550>
- [19] S. Pop, P. Clauss, A. Cohen, V. Loechner, and G.-A. Silber. 2004. *Fast Recognition of Scalar Evolutions on Three-address SSA Code*. Technical Report. Technical Report A/354/CRI, Centre de Recherche en Informatique (CRI), École des mines de Paris, 2004. Retrieved from <http://www.cri.enscm.fr/classement/doc/A-354.ps>.
- [20] Easwaran Raman, Neil Va hharajani, Ram Rangan, and David I. August. 2008. Spice: Speculative parallel iteration chunk execution. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*. Association for Computing Machinery, 175–184. DOI : <https://doi.org/10.1145/1356058.1356082>
- [21] Lawrence Rauchwerger and David Padua. 1994. The privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization. In *Proceedings of the 8th International Conference on Supercomputing (ICS'94)*. Association for Computing Machinery, 33–43. DOI : <https://doi.org/10.1145/181181.181254>.
- [22] L. Rauchwerger and D. A. Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.* 10, 2 (1999), 160–180.
- [23] Oliver Reiche, Mehmet Akif Özkan, Frank Hannig, Jürgen Teich, and Moritz Schmid. 2018. Loop parallelization techniques for FPGA accelerator synthesis. *J. Sig. Process. Syst.* 90 (2018), 3–27. DOI : <https://doi.org/10.1007/s11265-017-1229-7>
- [24] D. Sampaio, A. Ketterlin, L. Pouchet, and F. Rastello. 2016. Hybrid data dependence analysis for loop transformations. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 439–440.
- [25] Daniel A. Schult. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*. 11–15.
- [26] Xilinx. 2020. Vivado Design Suite UserGuide - High-Level Synthesis (UG902). Retrieved from <https://www.xilinx.com>.
- [27] Xilinx Inc. 2018. UG984 - MicroBlaze Processor Reference Guide. Retrieved from <https://www.xilinx.com>.

Received May 2021; revised September 2021; accepted November 2021