Enhancing High-Level Synthesis using a Meta-Programming Approach

Jessica Vandebon*, Jose G. F. Coutinho*, Wayne Luk*, Eriko Nurvitadhi†
*Imperial College London, United Kingdom
Email: {jessica.vandebon17, gabriel.figueiredo, w.luk}@imperial.ac.uk
†Intel Corporation, San Jose, USA
Email: eriko.nurvitadhi@intel.com

Abstract—In today's increasingly heterogeneous compute landscape, there is high demand for design tools that offer seemingly contradictory features: portable programming abstractions that hide underlying architectural detail, and the capability to optimise and exploit architectural features. Our meta-programming approach, Artisan, decouples application functionality from optimisation concerns to address the complexity of mapping high-level application descriptions onto heterogeneous platforms from which they are abstracted. With Artisan, application experts focus on algorithmic behaviour, while platform and domain experts focus on optimisation and mapping. Artisan offers complete design-flow orchestration in a unified programming environment based on Python 3 to enable accessible codification of reusable optimisation strategies that can be automatically applied to high-level application descriptions. We have developed and evaluated an Artisan prototype and a set of customised meta-programs used to automatically optimise six case study applications for CPU+FPGA targets. In our experiments, Artisan-optimised designs achieve the same order of magnitude speedup as manually optimised designs compared to corresponding unoptimised software.

Index Terms—Heterogeneous Computing, Meta-Programming, FPGA, High-Level Synthesis

1 Introduction

In this paper, we address the challenge of bridging the ever-growing gap between design productivity and our current computing landscape, in which the adoption of heterogeneous and specialised hardware accelerators, such as GPUs and FPGAs, has allowed orders of magnitude faster performance over CPUs.

However, in order to leverage these resources to maximise performance and resource utilisation, developers are required to perform a number of tasks, including exposing program concurrency, identifying code regions worth accelerating, partitioning code, and optimising each partition to harness the capabilities of each processing element. These tasks require effort as well as platform expertise: developers must take into consideration several factors, including data movement, memory systems, parallel compute units, and specialised resources at different levels of granularity. Despite continuing advances in compiler technology, there is still significant manual effort required to optimise applications.

In this context, we present a design-flow approach that allows (A) behavioural and (B) optimisation concerns to be independently described, and then automatically merged to derive optimised designs. Optimisation concerns are described using *meta-programs*, which codify platform- and domain-specific expertise and describe how applications should be effectively mapped to specific platforms. For this purpose, we provide meta-programming mechanisms to allow source-code, tools, and platforms to be programmatically manipulated and analysed in order to codify optimisation strategies.

Our approach has the following key benefits:

1) new platforms can be targeted by adding, revising, or parameterising meta-programs;
2) optimisations described in one meta-program can be reused for multiple input applications;
3) decoupling functional application descriptions from optimisation descriptions allows each to be developed and maintained separately.

This paper extends our work in [1], introducing more complex analysis and optimisation strategies, such as polyhedral analysis, pattern-matching for non-trivial transformations such as line-buffering, and multi-kernel execution models with channels for direct kernel-to-kernel communication. In addition, we cover additional benchmarks from the Rosetta HLS (High-Level Synthesis) benchmark suite [2], namely: digit recognition, 3D-rendering, and optical flow.

This paper provides four key contributions:

C1. A meta-programming approach for codifying and automating optimisation strategies;
C2. An open-source meta-programming framework, named Artisan, which targets C/C++ applications;
C3. A meta-program repository for optimisation and mapping using Intel HLS tools on CPU+FPGA platforms;
C4. An evaluation of automated optimisation strategies in terms of performance and development effort.

The rest of this paper is organised as follows: Section 2 outlines the motivation for our work; Section 3 explores related work; Section 4 introduces our approach (**C1**); Section 5 details the Artisan meta-programming framework (**C2**); Section 6 overviews our current meta-program repository (**C3**); Section 7 provides an evaluation (**C4**); Section 8 provides a discussion about correctness and the limitations of our approach; and finally, Section 9 concludes our findings and discusses future work.

2 Motivation

Consider a C++ application developed by a physics expert. To optimise this application for a heterogeneous computing platform and leverage the capabilities of its processing elements, including CPUs, GPUs, and FPGAs, the following must be taken into account:

1. Partitioning and Mapping. First, the application must be partitioned into code regions, and the most profitable processing elements must be selected to execute each part. Factors such as partition granularity, computing capabilities, and data movement must be considered. Heuristics can be used to guide this process based on code inspection, for instance by looking at data-types, memory and arithmetic intensity. This process can also be informed by runtime data, such as *hotspot* regions [3] and the amount of data transferred between partitions. For complex applications and platforms, the partition and mapping process can be refined using design-space exploration (DSE).

2. Optimising Partitions. Once partitions and their mappings are established, developers must isolate code parts so that they can be compiled independently with target-specific compilers. Moreover, extra code must be written to connect partitions through remote procedure calls (RPC). While there have been considerable advances in optimising compiler technology for high-level code targeting GPUs and FPGAs, software code still requires human effort to fully leverage the specific hardware architecture capabilities. This includes employing vendor API calls to support efficient I/O between devices, performing loop transforms to improve parallelism and pipelining, mapping scalars and arrays across memory systems to improve data locality, and optimising arithmetic operations with built-in specialised compute units (e.g. extended SIMD instructions in CPUs or block multipliers in FPGAs).

3. Reusing optimisations. Manual optimisations are performed on and tailored to specific applications even though they are based on best practices [4] [5] and generic strategies. This manual optimisation effort cannot be reused, and must be re-applied to every new application.

4. Maintainability. Once code has been partitioned, mapped and optimised, it may be difficult to update functionality and to port the code to new platforms since behavioural and optimisation concerns become intertwined.

In this paper, we address the above issues by providing a design-flow approach that decouples functional and optimisation concerns, allowing optimisation steps to be codified and automated independently from application source-code. The two key challenges addressed in this paper are:

(i) **effective analysis** to extract static and dynamic information (i.e. dependence analysis, code pattern, hotspots) from high-level code to inform optimisation decisions;
(ii) **codifying strategies** to automate and reuse human effort in optimising a full application onto a heterogeneous platform, covering all stages of the design-flow.

Additional challenges and limitations, such as semantic-preserving transformations, are discussed in Section 8.

3 Related Work

In this section we explore existing approaches that decouple application behaviour from optimisation to address productivity, portability, and maintainability issues. In our context, application behaviour describes functionality, while optimisation describes how functional description is transformed to run efficiently on a specific heterogeneous platform or device. Table 1 overviews and compares the features of a variety of approaches:

- **Concerns Location.** Although optimisation and behavioural concerns are decoupled in all listed approaches, their description is not necessarily separated. Optimisation description can either be *embedded* in the application source, or *separated* from it. For example, Halide [6] is a DSL that allows (1) image processing computations and (2) *schedules* to be described within the same source. We contend that separated descriptions, such as LARA [12] and Delite [8], are better suited to allow the development of generic optimisations strategies that can be reused and applied to multiple applications.
- **Support Existing Software Descriptions.** The majority of the approaches surveyed provide a new programming model [6], [7], [8], [10], [11] using a domain-specific language (DSL) or a programming API to codify application behaviour. This enables uniform and robust optimisation strategies since the application domain is restricted, however developers are required to learn a new programming model and possibly rewrite existing code. In contrast, the ability to support existing general-purpose software languages allows a wider range of application domains and to target legacy code, as well as having a low-learning curve for application developers. In contrast, LARA [12] targets full application source-code, in a number of languages, including C++, MATLAB and Java.
- **Optimisation Reusability.** Optimisations described using these approaches can be limited in their reusability. Typically, if optimisation descriptions are embedded in application source-code, they are application specific, and strategies need to be redefined and customised to target other applications. Domain-specific compiler frameworks (e.g. developed with Delite [8]) can improve productivity by enabling optimisation reuse across applications within the same domain. However, these still do not allow cross-domain optimisations.
- **Optimisation Target.** The scope on which these optimisation approaches operate also differs. Most of the approaches in our survey are limited to self-contained code partitions, or *kernels*. For example, loo.py [10] provides a Python based DSL for describing a kernel's behaviour and provides a library of optimising transformations based on polyhedral analysis that can be applied. In contrast, LARA [12] operates beyond a single code region, supporting analysis and manipulation at the application source-level with programmatic access to existing vendor compiler tools and DSE. However, each LARA description (known as an aspect) operates on a single target model, and thus it relies on multiple meta-programming tools (known as weavers) to execute different fragments of an optimisation strategy.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3096429, IEEE Transactions on Computers

TABLE 1
Comparison of Approaches That Decouple Optimisation and Behavioural Concerns

Approach	Concerns Location	Support Existing Software Descriptions	Optimisation Reusability	Optimisation Target	Analysis Scope
Halide [6]	embedded	no	no	kernel code	static
HeteroCL [7]	embedded	no	no	kernel code	static
Delite [8]	separated	no	yes (domain specific)	kernel code	static
TeML [9]	embedded	no	no	kernel code	static
Loo.py [10]	embedded	no	no	kernel code	static
Tiramisu [11]	embedded	no	no	kernel code	static
LARA [12] [13]	separated	yes	yes	app code and tools (fragmented)	static and dynamic
Artisan [1]	separated	yes	yes	app code and tools (unified)	static and dynamic

- **Analysis Scope.** Finally, most of the approaches in our list rely on static code analysis. We believe that this is not enough to support a wide range of optimisations, which require runtime information, for instance hotspot detection (used to partition code), monitoring data flow across partitions, and DSE. Acquiring runtime information requires the ability to instrument code to monitor and register the desired information.

Our approach, *Artisan*, is designed with the above factors in mind. In particular, optimisation strategies are codified in meta-programs separately from application descriptions, not entangled with application logic, in order to support reusable optimisations. Optimisations are self-contained and paramaterisable, such that they can be codified once and then, where applicable, be reused to multiple applications.

Artisan operates on an application's full source-code, allowing a wider scope for analysis and transformations, as described in Section 2. In addition, Artisan currently targets C/C++ code, and thus strategies can be applied to legacy code, not requiring algorithms to be re-written and adapted to new programming models. We have also designed Artisan to be more accessible by fully supporting a well-known general-purpose language, Python. In particular, we provide modules that can be imported into Python to support our meta-programming approach, and a repository of available meta-programs to optimise applications for specific platforms [14].

Our meta-programming approach includes different analysis tasks: querying source-code for code patterns, parsing tool reports, and acquiring monitoring platform data. In addition, it supports tasks such as code instrumention and transforms, invoking compile tools with specific options, and specifying platform parameters for deployment. All these tasks can be part of a coordinated optimisation strategy that is played out in a single and unified execution environment.

Note that Artisan and LARA are inspired by the aspect-oriented programming (AOP) [15] methodology of separation of concerns. However, AOP is restricted to functional concerns, allowing secondary functionality (such as logging) to be codified in separate modules from the main functional description, and then merged during execution. In contrast, we focus on decoupling functional and optimisation concerns, which requires more complex analysis and manipulation of the original source.

4 Artisan: Optimisation Approach

As mentioned in Section 3, we propose a meta-programming approach that decouples optimisation and behavioural descriptions. Using our framework, Artisan, platform- and domain-specific expertise are captured in parameterisable meta-programs. Artisan provides developers with programmatic access to artifacts including source-code, tools, and platforms, enabling coordinated optimisation strategies to be codified and reused across multiple applications.

The aim of meta-programs is to automate the human effort currently required to effectively map applications onto heterogeneous platforms (Section 2), and therefore they complement vendor compiler tools. For example, in the context of HLS, since meta-programs have access to the entire application source they are able to perform optimisation tasks that a single compiler tool cannot, including host-side and inter-kernel analysis and instrumentation, as well as runtime analysis, and automatic DSE.

Fig. 1 shows the three distinct stages involved in our approach. During **application development**, application experts write target-independent software descriptions and verify functionality on a CPU. During **meta-program development**, optimisation strategies are codified in Python. Note that both stages are performed autonomously. In the third stage, **mapping and deployment**, optimisation meta-programs and software descriptions are automatically merged, outputting an optimised description for a specific platform. In this stage, meta-program parameters are speci-

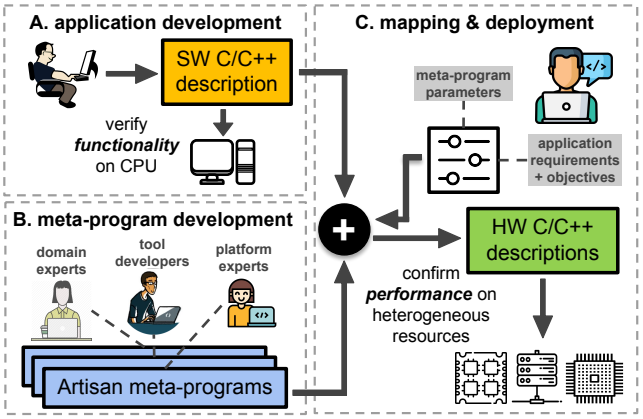

Fig. 1. The three distinct stages and roles of our meta-programming optimisation approach.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3096429, IEEE Transactions on Computers

fied according to application requirements (e.g. acceptable output error) and optimisation goals (e.g. minimising execution time).

In practice, although there are three distinct sets of responsibilities, the same person might take on more than one role. For example, an application developer might also map and deploy their application. Furthermore, interaction between those with different responsibilities may prove beneficial, but is not necessary.

Our approach addresses the two key mapping challenges described in Section 2. With programmatic access to source-code, tools, and platforms, meta-programs can be used to perform effective static and dynamic application analysis. Powerful query and instrumentation mechanisms enable meta-programs to effectively match patterns in input source code and apply source-to-source transformations programmatically (Section 5). Furthermore, Artisan enables scope for DSE, where optimisation decisions are automatically guided by feedback from tool reports and analysis of runtime behaviour and static code features.

The following section covers the process of codifying Artisan meta-programs.

5 Developing Artisan Meta-Programs

As mentioned, we have designed Artisan to be accessible to platform and domain experts. For this purpose, we have developed a set of Python modules to support our meta-programming approach mechanisms, allowing optimisation strategies to be codified and executed.

By definition, meta-programs are regular programs that analyse and manipulate source-code as data. However, as explained earlier, in order to effectively optimise applications for heterogeneous targets, we need to go beyond modifying source-code. That is, we need to analyse and instrument code, set compiler tool options, parse tool reports, specify deployment options, configure runtime and platform settings, and perform DSE based on feedback from tool reports and platform monitoring. A key challenge in making Artisan accessible is to identify all the mechanisms required to support any optimisation strategy. We describe these mechanisms in this section, and demonstrate their use in Section 6.

5.1 Design-Flow Artifacts

Our meta-programming approach supports a unified programming environment that exposes three types of *design-flow artifacts* as first-class Python objects: source-code, tools, and platforms, as illustrated in Fig. 2. With programmatic access to these artifacts, we are able to capture and automate coherent optimisation strategies for specific target platforms. For instance, programmatic access to the *source-code* artifact enables program description traversal to find code-patterns, as well as manipulation to support source-to-source transformations. We cover the source-code artifact in more detail in Section 5.2.

The *Tool* object wraps a specific set of tools, such as g++ and HLS tools, allowing programmatic control over the compilation process. Each tool object has a `run()` method with generic parameters, such as folder location and project name, as well as tool-specific parameters, such as triggering

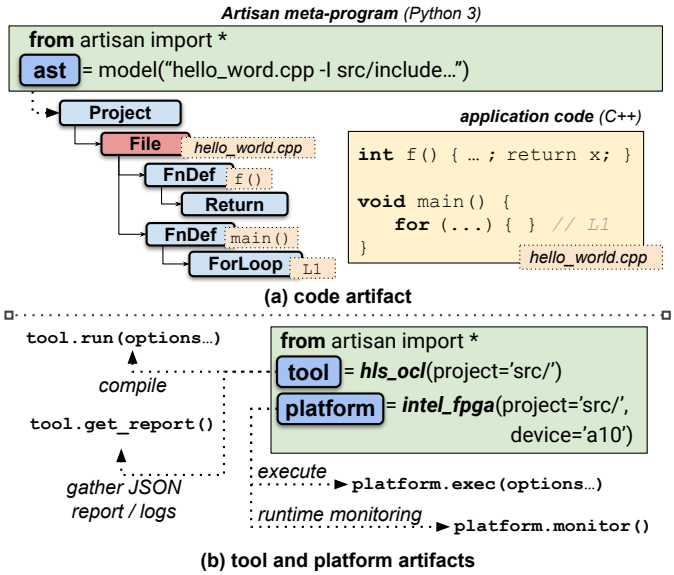

Fig. 2. Artisan design-flow artifacts.

compiler optimisations. The `get_report()` method returns key results produced by the tools. For example, results from synthesis reports, such as II (initiation interval) and resource utilisation metrics. Using the `Platform` object we can run compiled applications and extract key details about execution, such as runtime device and power utilisation, on target platforms using the `exec()` and `monitor()` methods.

5.2 Source-Code Analysis and Manipulation

In Artisan, source-code is represented by an abstract-syntax tree (AST), which can be analysed and manipulated using a Python object. The AST closely reflects the structure of the code as written by the developer without losing information. Thus, when new code is generated from a manipulated AST, the updated source-code remains familiar to the developer, instead of deriving this code from a lower-level IR.

In this section, we present the main mechanisms implemented in Artisan which allow meta-program developers to traverse and manipulate application source-code: query and instrument. For more implementation details, including how an AST is modelled from input code and how code is versioned, refer to our previous work [1].

5.2.1 *Code Traversal, Queries, and Pattern Matching*

Effective methods for AST traversal are critical in order to identify code regions that match syntactic or computational patterns with well-known optimising transformations, or to extract relevant information to perform analyses.

To facilitate pattern matching, every AST node has a `query()` method, which returns a table including all node sequences in the invoking node's sub-tree that match a specified search pattern and conditions. The `query()` method has two parameters: `match` defines the search pattern, and `where` specifies a condition to filter matches. The match expression specifies a sequence of entities separated by the edge => operator ("*is ancestor of*"). We demonstrate the query mechanism with three examples in Fig. 3, explained below.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3096429, IEEE Transactions on Computers

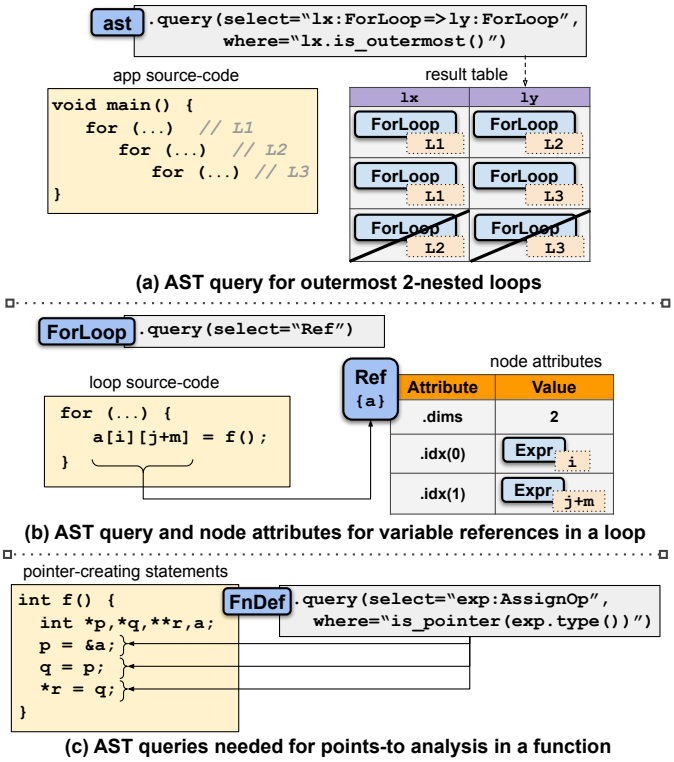

Fig. 3. Artisan query mechanism.

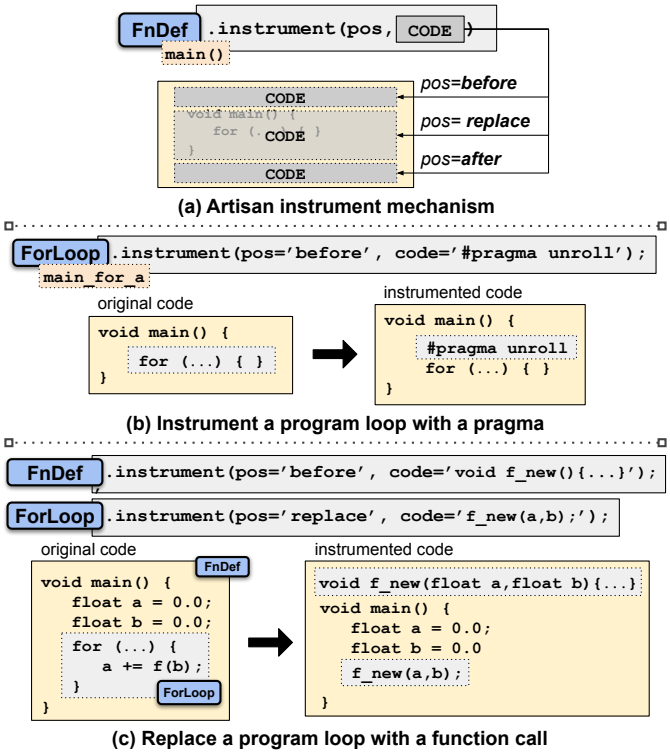

Fig. 4. Artisan instrument mechanism.

Fig. 3(a) shows a simple pattern match example, finding all two-nested for-loops in an AST where the outer of the two is an outermost loop. The results of a query are structured in a table, where each row represents a match and each column corresponds to the *entity* specified in the match expression. Entities include common source code constructs (e.g. function definitions, loops, expressions). In our example, columns correspond to `ForLoop` entities and are identified by labels `lx` and `ly`. Our program has three two-nested loops, but our query filters out the third match, as specified by the `where` parameter, since loop $L2$ is not outermost.

In Fig. 3(b), we find all variable references within a loop body. The results can be used in a meta-program performing memory access pattern analysis. Each entity (node instance type) is associated with a specific set of attributes providing information about the corresponding AST node. In our example, we find references to variable `a`, and a sample of attributes are shown in the depicted attribute table, including the dimensions and the index expressions in each corresponding dimension.

In Fig. 3(c), we match all statements needed to construct a points-to graph for a function. The code on the left shows three different pointer-creating statements that we can capture with a single query mechanism. We match all assignment operation expressions and filter based on whether matched expressions evaluate to pointer types. Meta-programs can use the results of this query to perform points-to analysis and determine if pointers may alias.

We include a more complex example of pattern matching in Section 6, combining various queries with extra meta-program logic to determine if a program might benefit from line buffering.

5.2.2 Code Instrumentation

In order to inject monitoring primitives that collect runtime information, or to perform source-to-source transformations, a mechanism to manipulate source code is necessary. With Artisan, every AST node can be instrumented with the `instrument(pos, code)` method. This method has two parameters: `pos`, which defines the position where the instrumented code will be placed, and the `code` itself which can be an arbitrary string. This allows developers to manipulate source code in a way that is familiar, injecting high-level code as strings rather than operating on lower-level IR.

Fig. 4(a) demonstrates the instrument method with three position arguments: *before* the target construct, *replace* the target construct, or *after* the target construct. The method also supports *begin* and *end* positions, allowing code insertion at the beginning and end of basic block scopes.

Fig. 4(b) shows an example of code instrumentation using the *before* position. In particular, we insert a pragma statement before the loop construct to instruct the compiler to fully unroll it. In general, pragma injection allows HLS code to be annotated to guide the synthesis process.

Fig. 4(c) demonstrates how to replace a loop with a call to a new function, as well as to insert the function's definition into the program. This is useful in order to extract hotspot regions into isolated functions for analysis, optimisation, and eventual offloading to hardware. In our example, the original `ForLoop` entity is *replaced* with a function call string, and the function's definition is inserted *before* `main()`. The function definition string and corresponding call string would be constructed using extra logic and query mechanisms to determine suitable arguments and corresponding call parameters.

Section 6 includes examples of meta-programs employing multiple coordinated `query()` and `instrument()` calls.

5.3 Meta-Program Classification

We classify meta-programs into three types:

- **Analysis**. Analysis meta-programs acquire static and runtime application information to guide the optimisation process. For instance, identifying hotspots, points-to analysis, and polyhedral dependence analysis;
- **Transforms**. Transform meta-programs perform actions on design-flow artifacts, changing their state. Examples include inserting `pragma` annotations to specific constructs or refactoring code;
- **Control**. Control meta-programs [16] orchestrate the optimisation to achieve a specific objective. For instance, to maximise or minimise an utility function such as execution time or resource utilisation, by performing DSE.

Control meta-programs implement optimisation strategies by relying on Analysis meta-programs to provide relevant information about the application, and employ Transform meta-programs to apply *optimisation tasks*. High-level control strategies can also invoke lower-level Control meta-programs to implement a divide-and-conquer strategy. Note that meta-program developers are responsible for ensuring that transformations are sufficiently correct, as discussed in Section 8.

6 High-Level Synthesis Meta-Programs

In this paper, we focus on the Intel OpenCL SDK for FPGAs to validate our approach. Artisan supports a generic interface for tools, languages, and hardware targets so meta-programs can in principle be customised and extended to support other platforms. In the following sub-sections, we outline our design-flow and provide an overview of our current meta-program repository.

6.1 Control

Fig. 5 outlines the Artisan design flow for optimising applications. The input is an unoptimised C++ application, and the output is an optimised CPU+FPGA application. We implement a high-level Control meta-program which coordinates the optimisation process, employing analysis and transform meta-programs to perform the following:

(1) Code normalisation. *Normalising transformations* are applied to put the input application into a canonical form, such that the code can be effectively parsed and instrumented. For example, normalising loops. This allows us to be robust against different coding styles.

(2) Hotspot detection/extraction. The application is instrumented with timers to identify program loops or functions which comprise greater than a specified threshold of program execution time. These regions are extracted into isolated functions for hardware offloading.

(3) FPGA project generation. HW/SW partitioning is performed, generating OpenCL FPGA project files: `*.cl` contains program hotspots extracted into OpenCL kernels, and `*.cpp` contains remaining application and hardware management logic.

(4) Optimisation: design space exploration. Transform meta-programs are applied, employing feedback from analysis meta-programs to guide optimisation. For example, P&R reports are used when unrolling program loops to determine the unroll factor that maximises resource utilisation without over-mapping the device.

6.2 Analysis

Table 2 overviews the analysis meta-programs from our current meta-program repository. Static analysis meta-programs (AS-X) inspect source code to generate reports without performing instrumentation or requiring code execution. For example, checking for code constructs that are not synthesisable with Intel OpenCL, such as function pointers and recursion, or performing static dependence analysis. Dynamic analysis meta-programs (AD-X), on the other hand, execute instrumented code in order to generate reports. Note that instrumented versions are discarded once reports are generated. For example, timing functions at runtime or parsing design reports to gather resource utilisation values.

Before we introduce meta-program code, consider the C++ k-means classification application code included in Fig. 6. The main function contains two loops: one initialises classes, and one calls a function (`radius()`) to perform computations. Within `radius()`, there are two nested loops, inside which the squared distance between each data point and class centre is calculated. We use this example to show the effects of the meta-programs which follow. Below, we include two analysis meta-program examples, one dynamic, and one static.

6.2.1 Dynamic Example: Loop Timing

Loop timing (AD-1) is an example of dynamic analysis. It instruments input code with timers on every loop, such that when the instrumented code is executed the time spent in each loop is reported. This is useful to detect hotspot code regions, for which loops are often candidates.

The code for AD-1 (`time_loops.py`) is included in Fig. 7. For simplicity, we consider only for-loops, but the code can trivially be changed to check while-loops as well. First, an AST (provided as a parameter) is queried for for-loops (line 2). Each loop is instrumented with an `artisan::Timer` object (line 4). The AST is queried for the main function (line 5), which is instrumented with a call to `artisan::report` (line 6). Finally, the global scope is instrumented to include artisan utility files (line 7).

The bottom of Fig. 7 shows instrumented k-means classification code (`k_means_timing.cpp`) after applying AD-1. This code begins with a new `#include` statement, each loop has an `artisan::Timer` object, and there is an `artisan::report` call at the end of `main`. When executed, all timer values are written to `loop_times.json`.

6.2.2 Static Example: Parallel Loop

The parallel loop meta-program (AS-3) uses polyhedral dependence analysis to statically determine whether a program loop contains loop-carried dependencies. If a loop is free from dependencies, it can be parallelised.

The key insight of polyhedral analysis is to model a program's *dynamic execution instances*, which correspond to operations executed at run-time. For instance, a single static code operation inside a program loop may correspond to many dynamic execution instances depending on the structure and number of iterations of the loop. Mathematical objects, such as Presburger Sets and Maps, are used to represent these instances and reason about their relations. We extract relevant program information with Artisan, and use islpy [17] [18], in order to model loops and run dependence analysis.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3096429, IEEE Transactions on Computers

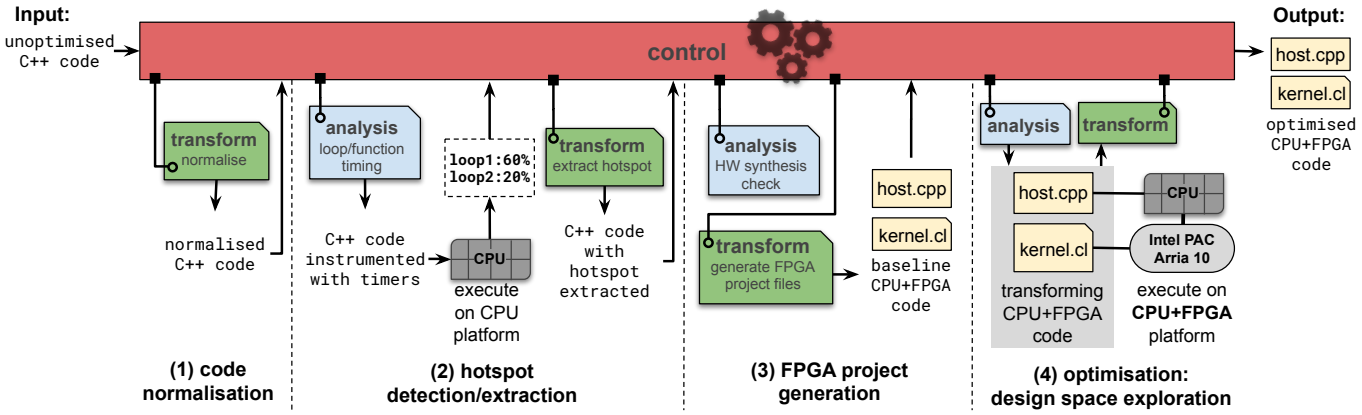

Fig. 5. Artisan Design Flow (Intel OpenCL SDK for FPGAs).

TABLE 2
Analysis Meta-Programs (LOC denotes lines of code)

Label	Type	Meta-Program	LOC	Description
AS-1	Static	Call-graph	12	Determines all functions called by a specified top-level function and returns a call-graph string.
AS-2	Static	Variable R/W	8	Determines whether a specified variable is accessed and overwritten (RW), only accessed (R), or only written to (W).
AS-3	Static	Parallel loop	13	Uses polyhedral analysis to determine if iterations of a specified loop are independent.
AS-4	Static	Loop dependencies	8	Identifies loop carried dependencies on variables using polyhedral analysis.
AS-5	Static	Function Data-flow	10	Analyses function inputs/outputs to determine how they interact with one another.
AS-6	Static	HW Synthesis Check	38	Checks if a specified function contains constructs that are not synthesisable using Intel OpenCL (e.g. recursion, function pointers)
AD-1	Dynamic	Loop Timing	8	Instruments program loops with timers.
AD-2	Dynamic	Function Timing	6	Instruments a specified function with a timer.
AD-3	Dynamic	Kernel Timing	22	Instruments a call to a specified FPGA kernel from the host with a timer.
AD-4	Dynamic	Resource Utilisation	15	Parses P&R reports to determine estimated FPGA resource utilisation.
AD-5	Dynamic	Hotspot detection	53	Identifies loops or functions where time spent is above a given threshold of overall execution time.

TABLE 3
Transform Meta-Programs (LOC denotes lines of code)

Label	Type	Meta-Program	Class	LOC	Description
TN-1	Normalisation	Scopify	n/a	9	Ensure all application scopes are block entities (i.e.enclosed in curly brackets {}) to enable effective parsing and instrumentation.
TN-2	Normalisation	For-loop Normalisation	n/a	10	Transforms for loops such that loop variables are normalised to start at 0 and increase by a constant each iteration.
TG-1	Generation	Loop-to-Function Extraction	n/a	20	Extracts a specified program loop into a new function.
TG-2	Generation	Generate OpenCL FPGA Project	n/a	>100	Generates OpenCL host (<main>.cpp) and kernel (<kernel>.cl) files.
TO-1	Optimisation	Unroll Loop	Scalability	4	Inserts `#pragma unroll F` above a specified loop to unroll it by factor `F`.
TO-2	Optimisation	NDRange Params	Replication	11	Sets the work group size and number of SIMD work-items for an NDRange kernel using `#define` statements.
TO-3	Optimisation	Local Mem. Buffer	Pipelining	16	Makes a local copy of a specified array, populating it with data from global memory.
TO-4	Optimisation	Function Inline	Pipelining	60	Inlines a specified function.
TO-5	Optimisation	Element Packing	Memory Enhancing	30	Modifies a kernel interface by packing multiple elements into a larger data type.
TO-6	Optimisation	Remove Dependency	Pipelining	20	Introduces register to replace accesses to a specified variable within a loop.
TO-7	Optimisation	Line Buffering	Memory Enhancing	16	Introduces local line buffers (i.e. N lines of a 2D array) using shift registers to update values.
TO-8	Optimisation	Window Buffering	Memory Enhancing	16	Introduces local window buffers (i.e. NxN elements of a 2D array) using shift registers to update values.
TO-9	Optimisation	Introduce Channels	Scalability	25	Introduces Intel OpenCL channels for direct kernel-to-kernel communication (replaces global memory accesses).
TO-10	Optimisation	OpenMP Loop	Scalability	25	Injects `#pragma omp parallel for num_threads(N)` above a specified program loop.

```
1  void radius(int n, float **data,
               class_t *classes, int **out){
2    for (int i = 0; i < CLASSES; i++) {
3      float dist2 = 0.0;
4      for (int j = 0; j < DIM; j++) {
5        float dist = classes[i].cntr[j] - data[n][j];
6        dist2 += dist*dist;
7      }
8      out[n][i] = dist2 < classes[i].rdi2 ? i : -1;
9    }
10 }
11 int main() {
12   float data[NUM_POINTS][DIM];
13   class_t classes[CLASSES];
14   float result[NUM_POINTS][CLASSES];
15   read_data(data);
16   for (int i = 0; i < CLASSES; i++)
17     init_class(&classes[i]);
18   for (int n = 0; n < NUM_POINTS; n++)
19     radius(n,data,classes,result);
20 }
```

Fig. 6. Example C++ input application (k_means.cpp)

```
1  def time_for_loops(ast):
2    loops = ast.query("ForLoop")
3    for loop in loops:
4      loop.body().instrument(pos='begin',
         code='artisan::Timer __timer__("%s")'%loop.tag());
5    main_func = ast.query("FnDef{main}")
6    main_func.body().instrument(pos='end',
         code='artisan::report("loop_times.json");')
7    ast.global_scope().instrument(pos='begin',
         code="#include <artisan/timer.hpp>")
```

meta-program: time_loops.py

```
1  #include <artisan/timer.hpp>
2  void radius(...){ ... }
3  int main() {
4    float ...;
5    read_data(data);
6    for (int i = 0; i < CLASSES; i++) {
7      artisan::Timer __timer__("main_for_a");
8      init_class(&classes[i]);
9    }
10   for (int n = 0; n < NUM_POINTS; n++) {
11     artisan::Timer __timer__("main_for_b");
12     radius(n,data,classes,result);
13   }
14   artisan::report("loop_times.json");
15 }
```

app: k_means_timing.cpp

Fig. 7. AD-1: Loop Timing

This meta-program performs 7 steps to determine if a loop is parallel, as indicated by comments in the code excerpt:

- **I.** query for all statements in the loop (line 2). In our example, four statements labelled from A to D are identified;
- **II.** derive statement instances according to their loop index and iteration space (line 3, e.g. $A(n,i)$: $0{\leq}n{<}N \wedge 0{\leq}i{<}C$);
- **III.** construct a schedule capturing the execution order of statement instances, by the position of each code statement;
- **IV.** query for all variable references (line 5, e.g. $data[n][j]$);
- **V.** identify all reference instances and define their constraints (line 6, e.g. $data(a,b)$: $0{\leq}a{<}N \wedge 0{\leq}b{<}D$);
- **VI.** define the mapping between statement and reference instances, separating them into read and write sets (line 7, e.g. $\{B(n,i,j){\rightarrow}data(a,b)\colon a{=}n \wedge b{=}j\} \in reads$);
- **VII.** compute the dependence flow on the polyhedral model using islpy (line 8).

For k-means, the outermost loop is determined not to have loop-carried dependencies, so the meta-program returns `True`.

```
1  def is_parallel(loop):
   # I. find all stmts in loop
2  stmts = loop.query("s:Stmt")
   # II. model loop stmts: names, iteration domains
3  model = model_stmts(stmts)
   # III. build statement schedule
4  schedule = build_stmt_schedule(model)
   # IV. find all references in loop (array and scalar)
5  refs = loop.query("r:Ref")
   # V. determine array constraints
6  arr_constrnts = get_array_constraints(refs,loop,ast)
   # VI. define access relations for each stmt and ref
7  reads,writes = get_access_relations(model,refs,arr_constrnts)
   # VII. run dependence analysis (true -> parallel)
8  return analyse_loop_deps(reads,writes,schedule)
```

meta-program: is_parallel.py

```
1  for (int n = 0; n < N; n++) {
2    for (int i = 0; i < C; i++) {
3  A   float dist2 = 0.0;
4      for (int j = 0; j < D; j++) {
5  B     float dist = classes[i].cntr[j] - data[n][j];
6  C     dist2 += dist*dist;
7      }
8  D   out[n][i] = dist2 < classes[i].rdi2 ? i : -1;
9    }
10 }
```

app: k_means.cpp

Fig. 8. AS-3 (partial): Parallel Loop (Polyhedral Dependence Analysis)

6.3 Transforms

Table 3 overviews the transform meta-programs from our current meta-program repository. Transforms are categorised as normalisation (TN-X), generation (TG-X), and optimisation (TO-X). Normalisation meta-programs transform input applications into a canonical form to facilitate their instrumentation by other meta-programs. Generation meta-programs generate code in a different format from the input application, for example, to comply with specific compilation tools (i.e. creating OpenCL host and kernel files). Finally, optimisation transforms apply techniques to make code more efficient.

The work in [19] identifies three transformation classes: (1) pipelining; (2) scaling; (3) memory enhancing. In their paper, they outline a variety of optimising transformations within each class and describe how to apply them manually. With Artisan these transformations can be codified and applied to applications automatically. The optimising transforms we present in this paper are all categorised according to these transformation classes (Table 3).

In the following subsections, we provide examples of transform meta-programs.

6.3.1 *Normalisation Example: Scopify*

Scopify (TN-1) is an example of a normalising transformation. It ensures that all scoped statement bodies are surrounded by

```
1  def scopify(ast):
2    scopes = ast.query("Scope")
3    for scope in scopes:
4      if hasattr(scope, 'body'):
5        body = scope.body()
6        if body and body.entity != 'Block':
7          body.instrument(pos="before", code="{")
8          body.instrument(pos="after", code="}")
```

meta-program: scopify.py

```
18   for (int i = 0; i < NUM_POINTS) {
19     read_data_point(&data[i]);
20   }
```

app: k_means_normalised.cpp

Fig. 9. TN-1: Scopify

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3096429, IEEE Transactions on Computers

curly brackets. This is important to ensure that subsequent meta-programs can operate effectively. For instance, if a loop or a conditional body is not surrounded by curly brackets, they will not be identified as code blocks (entity `Block`), and therefore we cannot add statements to them.

Fig. 9 shows the code for scopify (`scopify.py`), which takes as parameter an AST. All AST scoped nodes (e.g. loops, conditionals) are queried for (line 2), and if they contain a body which is not a `Block`, they are instrumented with surrounding curly brackets (lines 7-8).

6.3.2 Optimisation Example: Line buffering

Line buffering (TO-7) is a common memory optimisation for image and video processing applications to exploit data reuse in stencil loop computations [2] by buffering a number of input rows at all times. Fig. 10 shows a general loop example that may benefit from line buffering, as well as a depiction of how the buffer operates. A variable suitable for buffering is accessed within a two-nested loop, with reads performed across a fixed number of rows (n) and columns (m) in each iteration. The buffer caches n rows, introducing and removing an element at each iteration.

To implement line buffering, we first check whether this pattern is matched. We find all two-nested loops in an input program using Artisan's query mechanism, and then check each nest for buffer candidates using the code in Fig. 11 (`match_linebuf_pattern.py`). The bottom of Fig. 11 shows how the pattern is matched in two different input applications, optical flow and sobel filter. For brevity, we assume that the loop nest iterates through rows then columns, but we can extend the code to check for the opposite case as well, and transpose the loops if required.

First, we find all read-only 2D array variables (lines 3-4). For optical flow and sobel filter, `grad` and `frame` are identified, respectively. Next, for each variable, we check whether the first dimension accessed is always a linear translation of the outer loop index (i.e. the rows loop, lines 6-7), and we verify that the number of rows accessed is constant and greater than 1 for each iteration (lines 8-10). In optical flow, the first dimension of `grad` accessed is of the form "`r-i`", which is a linear translation of the loop index "`r`". The row span of `grad` is constant at 7 (`[r-0][c]...[r-6][c]`). Finally, we check whether the second dimension accessed is always a linear translation of the inner loop index (i.e. the columns loop, lines 11-12), and verify that the number of columns accessed is constant. In sobel filter, the second dimension of `frame` accessed is of the form "`x-1+j`", which

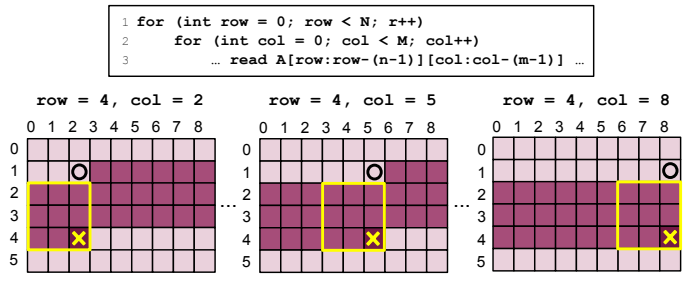

Fig. 10. The code pattern to match for line buffering and buffer operation for an example where N=6, M=9, n=m=3. The darker elements are buffered. In each depicted iteration, the element with an O is removed from the buffer, the element with an X is added to the buffer, and all elements in the yellow square are read, following a sliding window pattern.

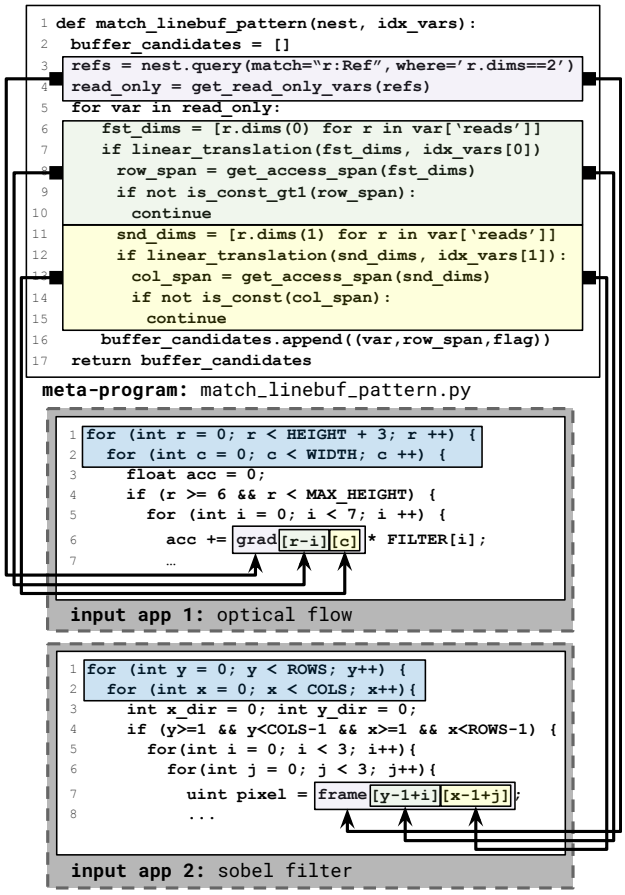

Fig. 11. TO-7: Pattern Matching for Line Buffering

is a linear translation of the loop index "`x`", and the column span is constant at 3 (`[...][x-1]...[...][x+1]`). Note that for optical flow, one column is accessed in each iteration, corresponding to a vertical 1D stencil, in contrast to the 2D stencil in sobel filter.

Once we have a candidate variable for buffering, we instrument the loop code to introduce a buffer. Fig. 12 includes `line_buffer.py`, a meta-program that performs the transformation, and shows how it operates on both applications. First, we ensure the loop nest iterates through rows and then columns. If this is not already the case, we transpose the loops (lines 2-3). Next, we transform the loop into a normalised form, where the access pattern corresponds to our example in Fig. 10 (line 4). That is, for the variable to buffer, the bottom right element accessed corresponds to the current nest iteration. If the loop is not already in this format, horizontal and/or vertical translations are performed. Once the loop nest and accesses are in the expected format, we declare a buffer (line 7-8) and insert it above the loop. We insert buffer update code at the top of the inner loop body (lines 9-10), and we change variable references to read from the buffer (lines 11-12).

Fig. 12 shows how the transformation applies to our optical flow and sobel filter examples. Optical flow is already normalised, whereas the sobel filter example requires a translation, the result of which is shown (normalised app 2). The buffer declarations, shift and update code, and modified accesses for each application are highlighted in the optimised code excerpts. Note that we enable the Intel OpenCL compiler to infer a shift buffer in our buffer update

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3096429, IEEE Transactions on Computers

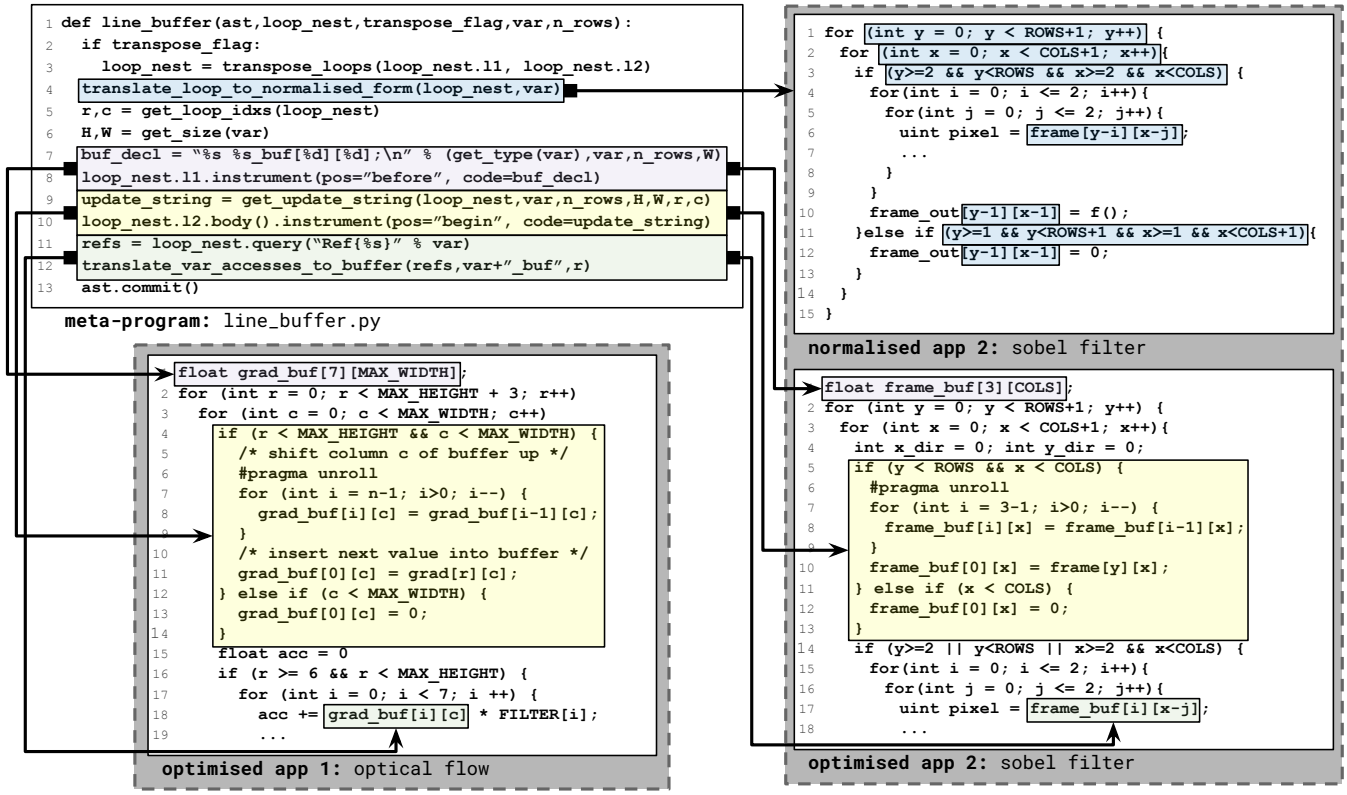

Fig. 12. TO-7 (partial): Line Buffering

code by unrolling the column shift loop.

7 Evaluation

7.1 Experimental Setup

We evaluate our Artisan meta-programs with FPGA and CPU targets. Baseline CPU as well as FPGA experiments are executed on the Intel Academic Compute Cloud. CPU input applications are programmed in C++, compiled with G++ 4.8.5 and target an Intel Xeon Platinum 8280 CPU @ 2.70GHz. FPGA applications use the Intel OpenCL SDK for FPGAs included in Quartus Prime Pro 17.1.1 and target an Intel Arria10 PAC board. Multi-threaded CPU experiments are compiled with G++ 5.4.0 which supports the OpenMP 4.0 standard, and target 8x Intel Core i7-9700K CPUs @ 3.60GHz. Meta-programs are programmed in Python3 using Artisan developed on top of the ROSE compiler framework [20].

We apply meta-programs to six benchmark applications: AdPredictor, K-Means Classification, N-Body Simulation, Digit Recognition, 3D-Rendering and Optical Flow. The first three benchmarks are extended from our work in [1], and the latter three are from the Rosetta HLS benchmark suite [2]. These benchmarks cover domains including machine learning, physics and image processing.

7.2 Optimisation Strategies

Table 4 includes the execution model selected for each benchmark as well as a list of the optimisations applied using Artisan. AdPredictor and K-Means both employ a single work-item execution model. Both applications are compute-bound and well-suited to pipelining optimisations. For each, we unroll loops (TO-1), checking resource utilisation (AD-4) to determine the maximum unroll factor that does not overmap the device. For K-Means, we also introduce local memory buffers (TO-3) for kernel arguments reused across iterations that fit in on-chip memory.

N-Body Simulation and Digit Recognition use ND-Range execution models, with multiple work-items performing computations in parallel. These applications are compute-bound and suited to scalability optimisations. For each we use a hill-climbing DSE based on kernel timing (AD-3), to identify the ND-Range parameters (work group size and number of SIMD items) that minimise execution time (TO-2). For N-Body Simulation, we remove a loop dependency (AS-4,TO-6). For Digit Recognition, we inline all functions (TO-4) and partially unroll fixed-bound loops (TO-1).

3D-Rendering and Optical Flow both follow a multi-kernel with channels execution model, where kernels communicate directly via on-chip channels. Both applications consist of multiple kernels and are memory-bound, so introducing channels (TO-9) for inter-kernel communication is critical to reduce overhead of global memory accesses. For 3D-Rendering, 8-bit inputs are packed into unsigned integers (TO-5) for efficient host to kernel transfer. For Optical Flow, we introduce line and window buffers (TO-7, TO-8).

7.3 Performance Results

To evaluate performance, we compare the execution time results achieved automatically using Artisan to manually hand-tuned FPGA designs. Table 4 includes speedups achieved compared to the baseline, unoptimised input software. We also include speedups achieved by designs automatically optimised using Artisan with OpenMP to target multiple

TABLE 4
Summary of Performance Results and Optimisation Strategies for Each Benchmark

Benchmark	Speedup			Execution Model	Applied Artisan Optimisations	Extra/Different Manual Optimisations
	OpenMP CPU	Artisan A10	Manual A10			
AdPredictor (LOC = 156)	10.3×	98.5×	98.5× (LOC +37%)	Single Work-Item	Unroll fixed loops fully	
K-Means Classification (LOC = 93)	9.1×	37.2×	41.8× (LOC +96%)	Single Work-Item	Local memory buffers	Unroll fixed loops (F=16)
					Unroll fixed loops (F=32)	
N-Body Simulation (LOC = 119)	11.7×	23.5×	25.5× (LOC +42%)	ND-Range	Set NDRange params	Struct padding
					Remove dependency	Single precision FP
Digit Recognition (LOC = 171)	11.2×	408.6×	415.8× (LOC +10%)	ND-Range	Set NDRange params	Simplified inlining
					Inline functions	
					Unroll fixed loop (F=16)	
3D-Rendering (LOC = 286)	n/a	3.2×	3.5× (LOC +53%)	Multi-Kernel w/ Channels	Kernel channels	Struct padding
					Data packing	
					Remove dependency	
Optical Flow (LOC = 477)	n/a	16.7×	18.5× (LOC +37%)	Multi-Kernel w/ Channels	Kernel channels	Struct padding
					Line buffering	Fewer channels
					Window buffering	

CPU threads, following the strategy presented in [1]. For 3D-rendering and optical flow loops are not parallelisable, so there is no applicable OpenMP optimisation.

Overall, the Artisan-optimised designs achieve the same order of magnitude speedup as our manually hand-tuned designs. Speedups achieved range from 3.2 times to 408.6 times compared to unoptimised CPU designs.

The final column in Table 4 includes optimisations that were applied to the manual design and not to the automated design. These explain the slight discrepancy in performance between the two. In some cases, these differences are due to limitations of the current Artisan framework. For instance, it is not yet possible (but will soon be supported) to query for complex struct definitions and therefore padding struct types for efficient memory alignment cannot be automated (N-Body Simulation, 3D-Rendering, Optical Flow).

In other cases, more complex optimisation techniques must be implemented. For instance, word-length optimisation to automatically demote types from double- to single-precision floating point (N-Body Simulation), and supporting DSE incorporating multiple metrics, such as performance and resource utilisation. In the case of K-Means, automatically unrolling until overmapping the device determines an unroll factor of 32, whereas manually observing kernel execution times with different unroll factors confirms that a factor of 16 achieved better execution time results despite utilising less of the device.

In principle, strategies could be developed to minimise these discrepancies and to improve automated designs. For example, the unroll until overmap DSE can be guided by P&R reports, which take longer to produce but are more precise, instead of using estimated resource utilisation results which are faster to generate but may lead to less efficient designs. It is up to meta-program developers to consider trade-offs and to codify the most suitable strategies.

7.4 Application Developer Effort

Table 4 includes the LOC count for each unoptimised benchmark application (column 1) as well as the added LOC percentage for each manually optimised Arria10 design (column 4). The added LOC ranges from 10% for Digit Recognition, to 96% for K-Means. This is an indication of the manual effort required to hand-tune each design which is effectively removed when using Artisan to automate the process. This added LOC metric does not, however, capture the expertise and experience required to effectively manually optimise a design. This requires understanding of each application, the target platform, and experience with the design tools, which often equates to months or years of developer experience. These percentages are therefore lower bounds on the estimation of developer effort saved when using Artisan. We discuss the effort to develop meta-programs in the next section.

8 Discussion

One challenge when codifying any optimisation is to ensure *correctness*: that the resulting code preserves the behaviour of the original code. Ensuring correctness by verification is of critical importance, since it would also clarify the precise conditions under which an optimisation is valid. Verification is, however, beyond the scope of this paper. Moreover, it is possible that different tools will be needed to verify different transformations. We hope in future to interface Artisan to verification tools such as model checkers and theorem provers, to facilitate the verification of key Artisan transformations.

Another aspect about developing Artisan meta-programs is that it requires platform and/or domain expertise. Tables 2 and 3 include LOC (lines of code) counts for each meta-program. These meta-programs were developed by a single developer in three weeks. We contend that operating at source-level (human-level) is easier than at IR level (compiler-level), since it does not require developers to learn a new model representation. Note that this does not preclude the use of more complex compiler analysis, such as polyhedral analysis, to support more aggressive transformations.

Finally, while Artisan is designed to automate human optimisation effort, it can only do so as long as there are systematic steps that allow a developer to codify them. Furthermore, the speedup achieved is still dependent on the algorithm employed by the application developer, and its coding style. For instance, obscured code that heavily relies on function and data pointers, instead of static calls and arrays, may be less effectively optimised if meta-programs rely on static information. Both cases can be mitigated by supporting code analysis that guides developers to write code that is more maintainable and more amenable for optimisation.

9 Conclusion

In this paper, we address the complexity of mapping high-level application descriptions onto heterogeneous platforms using our meta-programming approach: Artisan. Artisan decouples functional and optimisation descriptions: application experts focus on algorithmic behaviour, and platform experts focus on optimisation and mapping using meta-programs. Artisan offers design-flow orchestration using a single unified programming environment based on Python 3, enabling platform and domain experts to codify reusable, parameterisable optimisation strategies and apply them to applications automatically. We have developed and evaluated an Artisan prototype and used it to optimise six case study applications for CPU+FPGA targets, achieving the same order of magnitude of speedup (up to 409 times) as manually optimised designs, when compared to the corresponding unoptimised software versions.

Future work includes extending our meta-program repository to support additional hardware platforms and programming models, such as Intel OneAPI [21]). It would also be of interest to interface our approach to sophisticated optimisation and verification tools.

Acknowledgments

The support of Intel and the U.K. EPSRC (EP/L016796/1, EP/N031768/1, EP/P010040/1, EP/S030069/1, EP/L00058X/1) is gratefully acknowledged.

References

[1] J. Vandebon, J. G. F. Coutinho, W. Luk, E. Nurvitadhi, and T. Todman, "Artisan: a Meta-Programming Approach For Codifying Optimisation Strategies," in *Int'l Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2020.

[2] Y. Zhou et al., "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[3] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, 2008.

[4] "Intel High Level Synthesis Compiler Pro Edition: Best Practices Guide," accessed Mar-2020. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/nml1505158467345.html

[5] "Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide," accessed Mar-2020. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html

[6] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, 2012.

[7] Y-H Lai et al., "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing," in *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[8] A. K. Sujeeth et al., "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. on Embedded Computing Systems*, vol. 13, p. 134, 2014.

[9] A. Susungi, N. A. Rink, A. Cohen, J. Castrillon, and C. Tadonki, "Meta-Programming for Cross-Domain Tensor Optimizations," in *Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*, 2018.

[10] A. Klöckner, "Loo.py: Transformation-based code generation for gpus and cpus," in *Int'l Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY)*, 2014.

[11] R. Baghdadi et al, "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code," in *Int'l Symp. on Code Generation and Optimization (CGO)*, 2019.

[12] J. M. Cardoso et al., "LARA: an aspect-oriented programming language for embedded systems," in *Int'l Conf. on Aspect-oriented Software Development (ICAOSD)*, 2012.

[13] P. Pinto, T. Carvalho, J. Bispo, M. A. Ramalho, and J. M. Cardoso, "Aspect composition for multiple target languages using LARA," *Computer Languages, Systems & Structures*, vol. 53, 2018.

[14] Intel Corporation, "Intel FPGA SDK for OpenCL," https://www.intel.co.uk/content/www/uk/en/software/programmable/sdk-for-opencl/overview.html, 2020, [Online; accessed Mar-2020].

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *European Conference on Object-Oriented Programming*. Springer, 2001, pp. 327–354.

[16] Q. Liu, T. Todman, W. Luk, and G. A. Constantinides, "Optimizing Hardware Design by Composing Utility-Directed Transformations," *IEEE Trans. on Computers*, vol. 61, no. 12, 2011.

[17] "islpy 2020.2.2 Documentation," accessed Feb-2021. [Online]. Available: https://documen.tician.de/islpy/index.html

[18] "Integer Set Library Documentation," accessed Mar-2021. [Online]. Available: http://isl.gforge.inria.fr/

[19] J. de Fine Licht, S. Meierhans, and T. Hoefler, "Transformations of High-Level Synthesis Codes for High-Performance Computing," *IEEE Trans. on Parallel and Distributed Systems*, vol. 32, 2021.

[20] "ROSE compiler infrastructure," accessed Mar-2020. [Online]. Available: http://rosecompiler.org/

[21] "Intel oneAPI Toolkits," accessed Mar-2020. [Online]. Available: https://software.intel.com/en-us/oneapi

Jessica Vandebon is a PhD student within the Custom Computing Group at Imperial College London under the supervision of Prof. Wayne Luk. Her research interests include high-level abstractions for heterogeneous and reconfigurable compute platforms. She received her B.S. in Computer Engineering from Columbia University in 2017, and her M.Sc. in Advanced Computing from Imperial College London in 2018.

Dr. Gabriel Figueiredo received his M.Sc. and PhD in Computing Science from Imperial College London respectively. Since 2005, Dr. Figueiredo has been working as a research associate at the Custom Computing Group in Imperial College London, and has been involved in several UK and EU research projects. In addition, he has published over 30 research papers in peer-reviewed journals and international conferences and has contributed as an author to two book chapters, and as an editor to one book.

Prof. Wayne Luk received his M.A., M.Sc. and D.Phil. in engineering and computing science from Oxford University. Currently Professor of Computer Engineering at Imperial College, he founded and leads the Computer Systems Section and the Custom Computing Group, and is also Visiting Professor at Stanford University. He is on the Program Committee of many international conferences such as FCCM, FPGA and DATE. He has been an author or editor for 6 books, 4 special journal issues, a patent and over 120 research papers in peer-reviewed journals and conference proceedings.

Dr. Eriko Nurvitadhi is a senior research scientist and manager of the FPGA Research Lab at Intel Labs. He works on hardware accelerator architectures (e.g., FPGAs, ASICs) for AI and data analytics, with over 30 academic publications and 20 patent filed/issued in this area. His research has contributed to Intel's FPGA and ASIC solutions for AI. He co-founded Intel academic programs on FPGAs (HARP, ISRA). He received his PhD in Electrical and Computer Engineering from Carnegie Mellon University.