Artisan: a Meta-Programming Approach For Codifying Optimisation Strategies

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

***Abstract*—This paper provides a novel compilation approach that addresses the complexity of mapping high-level descriptions to heterogeneous platforms, improving design productivity and maintainability. Our approach is based on a co-design methodology decoupling functional concerns from optimisation concerns, allowing two separate descriptions to be independently maintained by two types of programmers: application experts focus on algorithmic behaviour, while platform experts focus on the mapping process. Our approach supports two key requirements: (1) Customisable optimisations to rapidly capture a wide range of mapping strategies, and (2) Reusable strategies to allow optimisations to be described once and applied to multiple applications. To evaluate our approach, we develop Artisan, a meta-programming tool for codifying optimisation strategies using a high-level general-purpose programming language (Python 3), offering full design-flow orchestration of key components (source-code, third-party tools, and platforms). We evaluate Artisan using three case study applications and three reusable optimisation strategies, achieving at least 24 times speedup for each application on CPU and FPGA targets with little application developer effort.**

I. Introduction

In the last decade, there have been remarkable advances in the compute landscape, with hardware architectures becoming increasingly multi-core, heterogeneous, and distributed. This has led to exciting new opportunities to solve previously intractable problems in domains including Artificial Intelligence (AI), Big Data, and engineering. Reconfigurable computing is a key part of this revolution: reconfigurable architectures can be specialised at compile time to perform specific tasks, achieving high performance and low power consumption.

However, to make effective use of these advances, application developers need to have considerable expertise in programming each specific device type. Hence, there is an increasing demand for programming languages and models that provide higher-level abstractions from hardware details to increase developer productivity. In practice, design tools offer limited support for heterogeneous platforms: the onus of optimisation lies mostly on the developer, and often this considerable effort cannot be reapplied to other applications.

The limitations in design tool support are due to three main challenges. First, there is no common standard (such as the x86 instruction set) for heterogeneous computing devices which the compiler community can build upon. Each hardware vendor supplies its own set of tools, and they are largely incompatible with one another. Second, optimising high-level descriptions for heterogeneous platforms requires traversing a large multi-dimensional design space defined by many application, architectural, and runtime parameters. Third, developing design tools is expensive even when employing compiler frameworks [1].

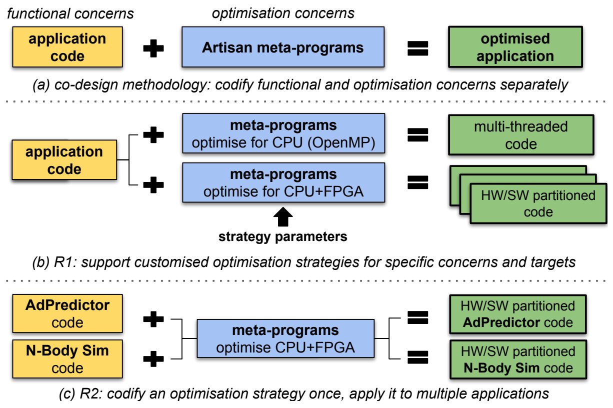

functional concerns
optimisation concerns
application code
Artisan meta-programs
optimised application
(a) co-design methodology: codify functional and optimisation concerns separately
application code
meta-programs optimise for CPU (OpenMP)
meta-programs optimise for CPU+FPGA
multi-threaded code
HW/SW partitioned code
strategy parameters
(b) R1: support customised optimisation strategies for specific concerns and targets
AdPredictor code
N-Body Sim code
meta-programs optimise CPU+FPGA
HW/SW partitioned AdPredictor code
HW/SW partitioned N-Body Sim code
(c) R2: codify an optimisation strategy once, apply it to multiple applications

Fig. 1. Our meta-programming approach.

We believe that to address the aforementioned challenges design tools need to support a co-design methodology [2] that allows *functional* and *optimisation* concerns to be described and maintained independently (Fig. 1(a)). In this context, functional concerns focus on capturing the algorithmic behaviour of the application, while optimisation concerns cover code mapping and optimisation, leading to the following requirements:

- **R1. Customisable optimisation strategies.** The rapid advances of technology and the sheer size of the design space require specialised optimisation techniques that can effectively exploit specific device capabilities (Fig. 1(b)).
- **R2. Reusable optimisation strategies.** To reduce development effort, optimisations that are codified once should be able to cover multiple applications (Fig. 1(c)).

This paper addresses the above two requirements and their underlying mechanisms to support design-flow orchestration and High-Level Synthesis (HLS). Our contributions are: (1) Artisan, a meta-programming optimisation approach that supports the aforementioned requirements; (2) a prototype of Artisan with a number of optimisations; and (3) an evaluation of our approach using three case-study applications.

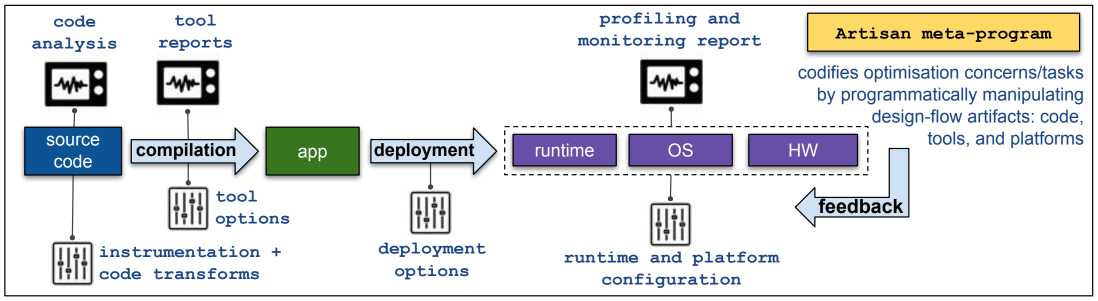

code analysis
tool reports
profiling and monitoring report
Artisan meta-program
codifies optimisation concerns/tasks by programmatically manipulating design-flow artifacts: code, tools, and platforms
source code
compilation
app
deployment
runtime
OS
HW
feedback
tool options
instrumentation + code transforms
deployment options
runtime and platform configuration

Fig. 2. Artisan meta-programs are written in Python 3, and have access to all design-flow artifacts as Python objects: source-code, tools and platforms. All stages of a design-flow can be optimised as part of a unified and coordinated strategy.

II. Overview

Optimising an application for a heterogeneous platform with hardware accelerators, such as GPUs and FPGAs, requires considerable human effort and expertise, even when employing sophisticated design tools, such as High-Level Synthesis (HLS). Application developers need to be aware of the target platform's architectural details, including available parallelism, memory hierarchies, and specialised features, to effectively distribute their workload and harness the platform's full potential. For this purpose, there are a number of *optimisation tasks* required, such as: analysing code features (static analysis), characterising runtime behaviour (dynamic analysis), annotating code with compiler directives, refactoring and applying code transformations, running a specific sequence of compilation optimisations, and experimenting with different application, tool, and platform parameters.

To improve the above process, we propose a co-design approach (Fig. 1(a)) which allows two distinct programming descriptions to be developed and maintained separately: (a) functional code that captures the application's algorithmic behaviour, written by application experts; and (b) *meta-programs* that capture *optimisation tasks*, programmed by platform experts. By allowing these concerns to be independently codified and automatically applied, application developers need not become conjurers of optimisation tricks, and application code does not need to tie to specific optimisations, thus improving both design productivity and maintainability.

One key question is whether optimisations can be described independently of application code. In practice, most *optimisation tasks* are not tied to the application, but instead can be generalised, as demonstrated by manuals and best practice documentation provided by hardware and tool vendors [3] [4] [5] [6]. Our co-design approach focuses on enhancing state-of-the-art HLS tools, although it is not limited to this domain. While common optimisation techniques are well documented, they are still performed manually, hindering developer productivity and design maintainability.

This paper aims to demonstrate that it is possible to codify established optimisation strategies independently of application code, that these strategies can be customised to support different platforms, and that such effort can be automated and reused to optimise multiple applications.

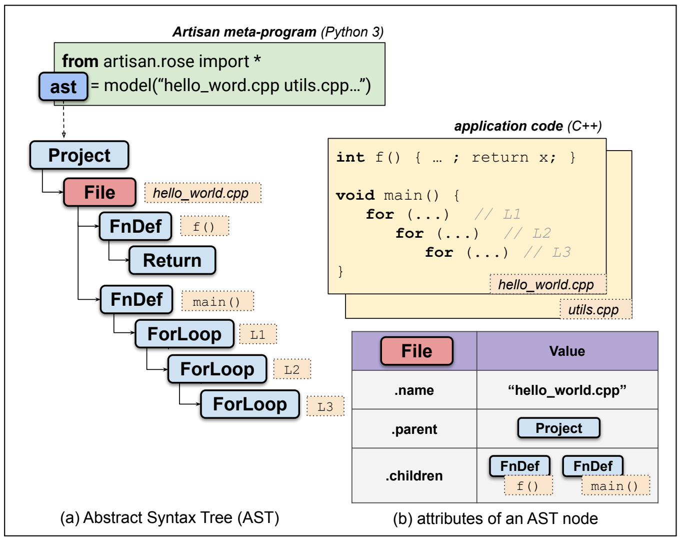
III. Codifying Strategies

This section covers the basics of crafting Artisan meta-programs. Meta-programs are written in Python 3, a well-known high-level general-purpose interpreted programming language. We choose Python to make the process of codifying optimisations as accessible as possible. A novel element of our approach with respect to others [2] is that we expose all *design-flow artifacts* as first-class Python objects (Fig. 2). Artifacts include: source-code, tools, runtimes, operating systems, and hardware platforms. With programmatic artifact access, we are able to devise effective, customised, coordinated strategies that meet specific optimisation concerns, such as performance, on a particular target platform.

To automate these strategies, meta-programs codify the *optimisation tasks* mentioned previously, such as (a) code analysis, (b) code instrumentation, (c) setting compiler tool options, (d) tool report analysis, (e) specifying deployment options, (f) configuring runtime/OS/platform options, and (g) performing design-space exploration (DSE) based on feedback from reports and runtime monitoring. These tasks are critical to optimising applications for heterogeneous compute platforms; but, despite advances in vendor tools, they are often performed manually. This is an expensive and error-prone process, and requires significant expertise.

To validate our approach, we focus on high-level synthesis (HLS) by developing meta-programs that translate agnostic C++ code into HLS-enhanced C++ code. Agnostic C++ captures functionality without any optimisations, while HLS-enhanced C++ may have: `#pragma` annotations to direct the HLS tool to perform low-level optimisations, loop transformations, and low-level library calls to specify memory interfaces, built-in operators, and customised numerical representations. By providing a programmatic model of the platform, tools and source-code, meta-programs can automate the translation between agnostic and HLS-enhanced descriptions (Fig. 2).

Artifacts such as *tools* and *platforms* are exposed as Python objects, with methods that execute actions such as running the tools with specific compiler options, as well as providing tool reports and monitoring information. The *source-code* artifact, on the other hand, requires more complex mechanisms to model, analyse, and manipulate. We cover the source-code artifact in more detail in the remaining subsections.



```
from artisan.rose import *
ast = model("hello_word.cpp utils.cpp…")
```

```
int f() { … ; return x; }

void main() {
   for (...)  // L1
      for (...)  // L2
         for (...)  // L3
}
```

File	Value
.name	"hello_world.cpp"
.parent	Project
.children	FnDef f(), FnDef main()

Fig. 3. AST nodes and attributes.

A. Source-Code Model

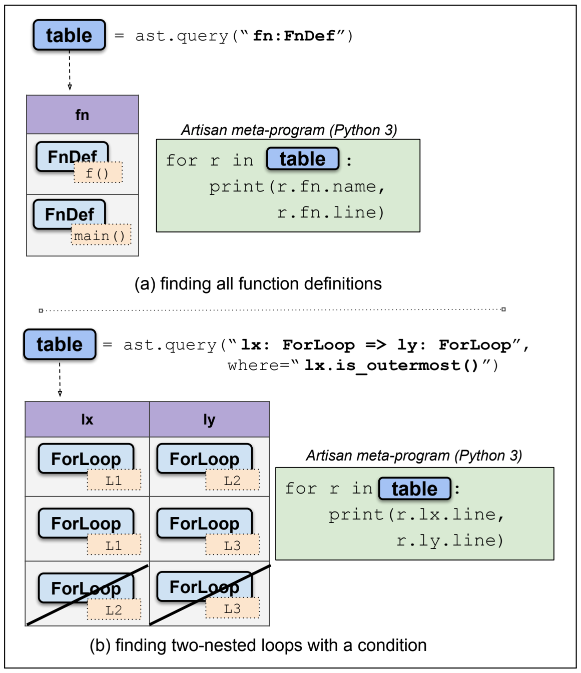
In order to analyse and manipulate application source-code, we expose an abstract-syntax tree (AST) data-structure that closely reflects the code's structure in Python. Our AST representation is designed to capture the source-code as written by the developer without losing information, as opposed to exposing a lower-level intermediate representation (IR) of the program. This allows optimisations to be expressed at the application-level, rather than at the compiler-level, and helps combine manual and automated efforts if required.

Fig. 3 provides an example of a program, written in two source-code files: $hello\_world.cpp$ and $utils.cpp$. In this figure, we use the $model()$ Python function, which is part of the Artisan core library, to generate an AST representation of the application. The arguments to $model()$ are similar to any C/C++ compiler: a list of paths to source files with optional macro definitions (-D) and include header locations (-I). Currently, our prototype works with C (ANSI C and C99), C++ (C++89 and C++11), OpenCL and OpenMP programs, while our approach can be extended to support other programming languages where code can be represented by an AST.

Each AST node represents a source-code construct exposed as a standard Python object organised in a tree structure. Each node has a specific set of properties and methods that allow developers to access and operate on the AST. Meta-programs can find code patterns by traversing the tree (i.e. querying as explained in Section III-B), access node attributes to extract code features, and manipulate source-code by invoking specific methods (e.g. instrumentation as explained in Section III-C).

B. Queries

Artisan supports a powerful mechanism for identifying code patterns. Every AST node has a `query()` method, which performs a depth-first search (DFS) on the tree starting with the node that invokes this method. The results of a query correspond to all of the AST node sequences that match the search pattern and verify the condition supplied, as we


Artisan meta-program (Python 3)
(a) finding all function definitions
Artisan meta-program (Python 3)
(b) finding two-nested loops with a condition

```
table = ast.query("fn:FnDef")
```

fn
FnDef f()
FnDef main()

```
for r in table:
    print(r.fn.name,
          r.fn.line)
```

```
table = ast.query("lx: ForLoop => ly: ForLoop",
                  where="lx.is_outermost()")
```

lx	ly
ForLoop L1	ForLoop L2
ForLoop L1	ForLoop L3
~~ForLoop L2~~	~~ForLoop L3~~

```
for r in table:
    print(r.lx.line,
          r.ly.line)
```

Fig. 4. Source query and results (based on the code in Fig. 3).

explain next. Note that currently all code patterns considered in our research use DFS, however we can support other search algorithms if required.

The `query()` method has two parameters: `match` defines the search pattern, and `where` specifies a condition to filter matches. The `match` expression has the following format: $\boxed{\text{a: } Entity_a \text{ => b: } Entity_b \text{ => ...}}$, where $Entity \in \{FnDef, File, ForLoop, ...\}$, one for each AST node type. The match expression specifies a sequence of entities separated by the edge => operator, and each entity in the expression is associated with a unique label (e.g. a and b). The edge operator represents the relationship *"is ancestor of"*. Thus a match corresponds to a sequence of AST nodes [$node_x$, ..., $node_z$] resulting from query $\boxed{\texttt{x:}Entity_x \texttt{ ... => z:}Entity_z}$, such that $node_x$ and $node_z$ are instances of $Entity_x$ and $Entity_z$, respectively, and $node_x$ is an ancestor of $node_z$. Note that Artisan queries are polymorphic, and can capture AST nodes using the *parent* entity. For instance, the query $\boxed{\texttt{s:}Stmt}$ returns all statements, including loops and conditionals, since $ForLoop$ and $CondStmt$ inherit from the $Stmt$ entity.

The results of a query are structured in a table (Fig. 4). Each row represents a match, while each column corresponds to the *entity* specified in the match parameter expression and identified by the corresponding label. The table can be iterated to access each result. Each iteration returns a row object where each node can be accessed through its column (label) name. Fig. 4(a) shows an example where we find all function definitions in the AST. In Fig. 4(b), we query two-nested loops in the program presented in Fig. 3, where the first loop is an outermost loop. Note that while the program has three two-nested loops, we filter out the third match, as specified by the query `where` parameter, since loop $L2$ is not outermost.

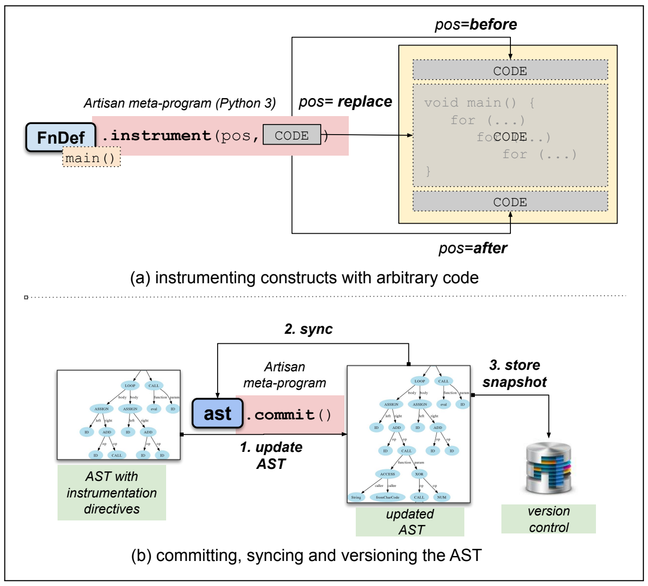


Fig. 5. Source-code instrumentation and version control.

C. *Instrumentation*

We provide mechanisms for instrumenting code for two reasons: first, to inject monitoring primitives such that when the augmented code is executed we acquire runtime information; and second, to transform code in order to optimise it.

Every AST node representing a program construct can be instrumented (Fig. 5(a)) with the `instrument(pos, code)` method. This method has two parameters: `pos`, which defines the position of the code to be placed, and the `code` to be injected. The `code` can be any string, for instance, a preprocessor directive (`#include`), a pragma statement, a C/C++ extract, or just a comment. The meta-program developer is responsible for ensuring that inserted code leads to a program that can be parsed by a C/C++ compiler. The `pos` argument can have one of three values: *before* the target construct, *replace* the target construct, or *after* the target construct. The *before* and *after* positions are usually used in the context of monitoring, since they do not remove the original code. The *replace* position, on the other hand, replaces the whole program construct.

When instrumenting code, there are no actual changes to the AST. Instead, annotations are stored on relevant AST nodes. To update the AST, meta-programs must invoke the `commit()` method of the `ast` object (Fig. 5(b)). This method triggers three steps: first, the instrumented source-code base is generated and re-parsed to derive the new AST which is a direct reflection of the instrumentation annotations; second, the `ast` variable now references the updated AST object, so all further query and instrumentation operations are performed on this updated structure; finally, the new code base is automatically stored in a version control database.

Code versioning allows us to trace intermediate transformation results and experiment with optimisation strategies in parallel. Any committed version can be accessed for further analysis, transformation, and execution.

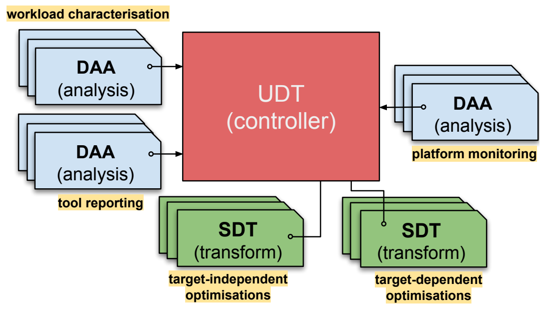


Fig. 6. Artisan meta-program categories.

IV. Meta-Program Repository

We build a repository of scalable and reusable Artisan meta-programs (Requirement R2) to support HLS. We classify them into three categories according to their roles (Fig. 6):

- **Design and Application Analysis (DAA)**. DAAs acquire static and runtime application information to guide the optimisation process. For instance, identifying hotspots and non-synthesizable code regions, performing dynamic range analysis, and parsing data from tool reports.
- **Syntax-Directed Transformations (SDT)**. An *SDT* performs actions on design-flow artifacts. Examples include adding `pragma` annotations to specific constructs, transforming code, refactoring code, applying specific tool options, and configuring the platform runtime.
- **Utility-Directed Transformations (UDT)**. *UDTs* [7] provide control logic for optimisation and are objective-based. The goal of a UDT is to maximise or minimise a utility function such as execution time or resource utilisation, either by applying a set of well-known steps, or by performing DSE. Optimisation techniques, such as hill-climbing, simulated annealing or integer linear programming, may be used.

Meta-program developers (i.e. platform experts) are responsible for ensuring that transformations are sound, which can be checked by hand or using tools (e.g. [8]). In addition to verification, pre-conditions must be employed to ensure correct optimised code. For instance, it must be verified that there are no loop-carried dependencies when parallelising a loop with OpenMP. SDTs must also define transformation scope. As an example, if an SDT only operates on normalised `for`-loops, this must be checked. UDTs, on the other hand, are guided by optimisation objectives; they rely on DAAs to provide analysis reports, and on SDTs to execute *optimisation tasks*.

Our current repository contains meta-programs that automatically optimise C++ applications for two hardware targets: (1) multi-threaded CPUs using OpenMP, and (2) CPU+FPGA platforms using Intel HLS/OpenCL tools. For OpenCL, we derive two types of kernels: a *single work-item* kernel which leverages pipeline parallelism, and an *NDRange* kernel which operates concurrently in a SIMD execution fashion [4] to exploit data parallelism. Fig. 7 shows our current optimisation workflow, while Table I summarises the meta-programs in our repository.

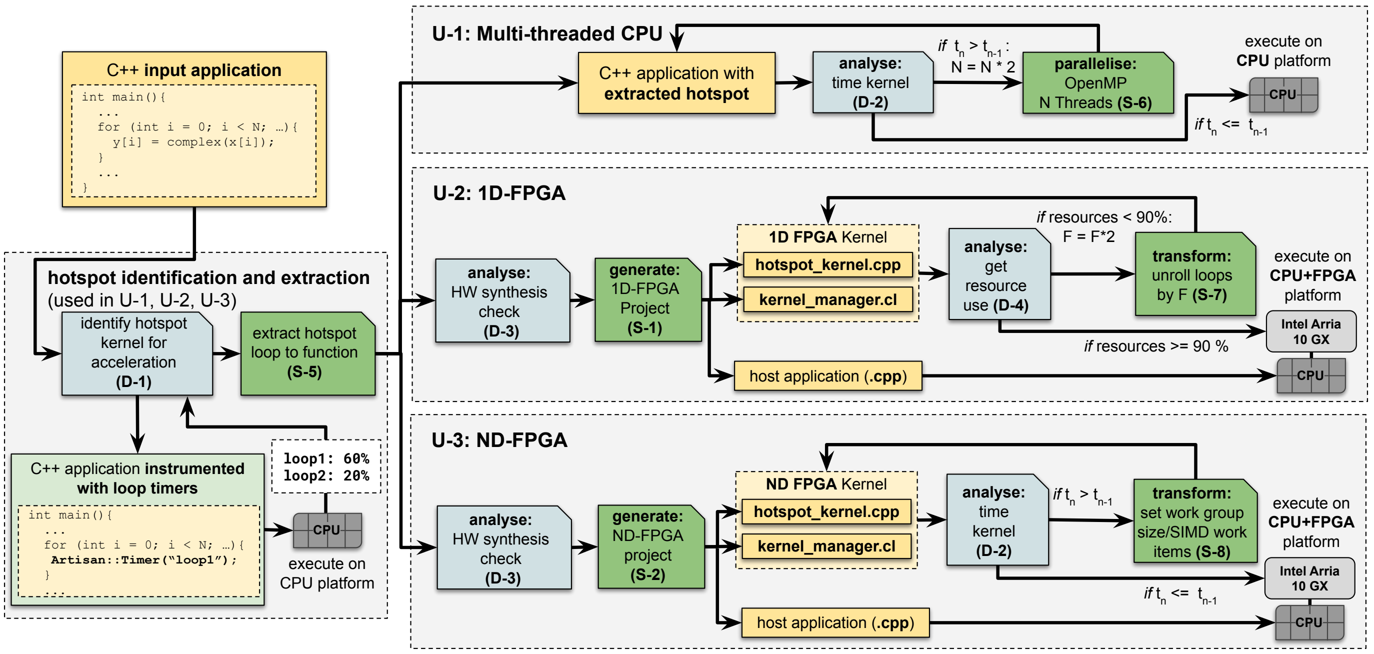

C++ input application
int main(){ ... for (int i = 0; i < N; …){ y[i] = complex(x[i]); } ... }
hotspot identification and extraction (used in U-1, U-2, U-3)
identify hotspot kernel for acceleration (D-1)
extract hotspot loop to function (S-5)
C++ application instrumented with loop timers
int main(){ ... for (int i = 0; i < N; …){ Artisan::Timer("loop1"); } ... }
loop1: 60%
loop2: 20%
CPU
execute on CPU platform
U-1: Multi-threaded CPU
C++ application with extracted hotspot
analyse: time kernel (D-2)
if t_n > t_n-1 : N = N * 2
parallelise: OpenMP N Threads (S-6)
if t_n <= t_n-1
execute on CPU platform
U-2: 1D-FPGA
analyse: HW synthesis check (D-3)
generate: 1D-FPGA Project (S-1)
1D FPGA Kernel
hotspot_kernel.cpp
kernel_manager.cl
analyse: get resource use (D-4)
if resources < 90%: F = F*2
transform: unroll loops by F (S-7)
if resources >= 90 %
host application (.cpp)
execute on CPU+FPGA platform
Intel Arria 10 GX
U-3: ND-FPGA
analyse: HW synthesis check (D-3)
generate: ND-FPGA project (S-2)
ND FPGA Kernel
hotspot_kernel.cpp
kernel_manager.cl
analyse: time kernel (D-2)
if t_n > t_n-1
transform: set work group size/SIMD work items (S-8)
if t_n <= t_n-1
host application (.cpp)
execute on CPU+FPGA platform
Intel Arria 10 GX

Fig. 7. Our current optimisation workflow to support Intel HLS. Meta-program labels and descriptions can be found in Table I. We have applied this workflow to three different applications (AdPredictor, K-Means Classification, and N-Body Simulation), and our results are presented in Table II.

TABLE I
SUMMARY OF THE ARTISAN META-PROGRAM REPOSITORY. LOC: LINES OF CODE.

Label	Meta-program	Type	LOC	Description
U-1	**Multi-CPU** `app` → `app`$_{omp}$	UDT	36	Optimises an application (`app`) by parallelising hotspot loops using OpenMP.
U-2	**1D-FPGA** `app` → `app`$_{1d}$	UDT	87	Optimises an application (`app`) by deriving a single work-item FPGA kernel that maximises resource utilisation.
U-3	**ND-FPGA** `app` → `app`$_{Nd}$	UDT	86	Optimises an application (`app`) by deriving an ND-Range FPGA kernel that maximises data-parallelism and minimises execution time (`app`$_{Nd}$).
D-1	**Hotspot Detection** `(app, T)` → `hotspots`	DAA	32	Identifies loops (`hotspots`) in `app` where time spent is above a given threshold (`T%`) of overall execution time.
D-2	**Kernel Timing** `(app, fn)` → `time`	DAA	22	Returns the time spent (`time`) in a specified SW or HW function/kernel (`fn`) during program execution.
D-3	**HW Synthesis Check** `(app, fn)` → `bool`	DAA	49	Returns whether a specified function (`func`) contains constructs that are not synthesizable using Intel HLS (e.g. recursion or function pointers).
D-4	**Resource Utilisation** `design` → `(ALUT, FF, RAM, DSP, MLAB)`	DAA	15	Parses P&R reports for an FPGA `design` to determine estimated resource utilisation (`ALUT,FF,RAM,DSP,MLAB`).
S-1	**1D-FPGA Project** `(app, kernel)` → `(host, manager, hls_kernel)`	SDT	298	Generates code for a single work-item OpenCL FPGA project. This includes a single work-item OpenCL kernel manager, a C++ host application (S-3), and a C++ HLS kernel file (S-4).
S-2	**ND-FPGA Project** `(app, kernel)` → `(host, manager, hls_kernel)`	SDT	300	Generates code for an NDRange OpenCL FPGA project. This includes an NDRange OpenCL kernel manager, a C++ host application (S-3) and a C++ HLS kernel (S-4). For a kernel function (`kernel`) to be NDRange-compatible, it must have an outer loop with `idx=0-N`. This outer loop is removed, and the index variable (`idx`) is added as a function argument. This argument is set to the work-item ID in the kernel manager.
S-3	**C++ Host application** `(app, kernel)` → `host`	SDT	103	Instruments the input program (`app`) to generate an OpenCL host program replacing the call to the specified kernel function (`kernel`) by enqueuing an OpenCL kernel. The OpenCL platform is initialised and all OpenCL kernel arguments and buffers are created and set.
S-4	**C++ HLS Kernel** `(app, kernel)` → `hls_kernel`	SDT	71	A C++ kernel targeted for hardware synthesis using Intel HLS is generated by copying a specified hotspot kernel function (`kernel`) into a separate `.cpp` file including required HLS headers. This HLS kernel can be compiled and linked with the OpenCL manager and host.
S-5	**Loop To Function** `(app, loop)` → `app`$_{loop}$	SDT	28	Extracts a specified program `loop` into a new function, and replaces the original loop with a function call.
S-6	**OpenMP Loop** `(app, loop, N)` → `app`$_{omp}$	SDT	19	Injects `#pragma omp parallel for num_threads(N)` above the `loop`.
S-7	**Loop Unroll** `(app, loop, F)` → `app`$_{unrolled}$	SDT	12	Inserts `#pragma unroll F` above a specified `loop` to unroll it by factor `F`.
S-8	**Set Work-Group Size and SIMD Work-Items** `(manager, WG, SIMD)` → `manager`$_{wg-simd}$	SDT	14	Sets the work-group size (`WG`) and number of SIMD work-items (`SIMD`) for an NDRange kernel in the kernel manager code (`manager`).

Next, we provide detailed explanations of some meta-programs in Table I:

A. Multi-CPU Strategy (U-1)

The goal of this strategy is to parallelise loops using OpenMP. First, a hotspot loop is identified by automatically timing all loops in the application (D-1). Next, the strategy verifies if the candidate loop can be parallellised. If the loop is parallel, it is extracted to a function (S-5) in order to isolate it from other parts of the code. An OpenMP `parallel for` pragma is inserted on top of the hotspot loop with a specified number of threads (S-6). To determine the minimum number of threads that maximises performance for a given dataset and CPU target, hill-climbing is used, starting with 2 threads and doubling this number until the observed execution time no longer improves. To determine execution time during this profiling stage, the hotspot function call is automatically instrumented with a timer (D-2).

B. 1D-FPGA Strategy (U-2)

This strategy generates a single work-item kernel from the hotspot loop, and maximises FPGA resource utilisation. A single work-item kernel is derived by unrolling the candidate loop by a specific factor in order to pipeline computation stages. Controlling the loop unroll factor allow us to adjust pipeline parallelism by mapping more or less loop iterations onto the physical FPGA space [3].

The strategy is as follows. First, a hotspot loop is identified and extracted to a function (D-1, S-5), then it is checked for constructs not supported by Intel HLS, such as function pointers or recursion (D-3). If the function is synthesizable, HW/SW partitioning is performed, generating a single work-item FPGA project based on the hotspot function (S-1). This generated FPGA project contains 3 files: (i) the C++ host application, (ii) the OpenCL kernel manager, and (iii) the HLS kernel file. The C++ host calls the hotspot C++ function residing in the HLS kernel via the OpenCL kernel manager. The kernel manager acts as the intermediary between the host and the kernel, handling the communication between them.

After the FPGA project is generated, the HLS kernel is instrumented to unroll the candidate loop by a specific factor (S-7). This factor begins at 2, and is doubled iteratively in a DSE process until the synthesis tools report that the design is *over-mapped*. More specifically, for each unroll factor, intermediate RTL compilation is used to generate Intel Design reports, which are parsed to get resource utilisation (ALUTs, FFs, RAMs, DSPs, MLABs) estimates (D-4). If any resource is reported at over 90% utilisation, the unroll factor is rolled back and the DSE stops. This 90% threshold value is parameterisable. The final HLS C++ kernel is compiled into OpenCL, and linked with the kernel manager during synthesis using the Intel OpenCL compiler, producing the final single work-item FPGA design.

C. ND-FPGA Strategy (U-3)

This strategy generates an NDRange FPGA kernel from the hotspot loop to maximise data parallelism. In contrast to U-2, which pipelines the entire loop, the hardware kernel in this strategy implements one loop iteration, corresponding to a single work-item. The number of work-items running concurrently corresponds to the number of loop iterations. To achieve higher throughput, we further group multiple work-items to execute operations in a SIMD manner [9].

As with the 1D-FPGA strategy, a hotspot function is automatically identified for hardware acceleration, and an FPGA project is generated consisting of a C++ host, OpenCL kernel manager, and a HLS kernel (S-2). The number of data-parallel work-items for the NDRange kernel is specified in the C++ host. Kernel vectorization is adjusted by two parameters: the work-group size (`WG`) and number of SIMD work-items (`SIMD`), both of which are specified in the kernel manager: `WG` specifies the number of work-items provisioned for each work-group, while `SIMD` instructs the compiler to vectorize the datapath where possible within a work-group [9].

The strategy uses hill-climbing, increasing the values of `WG` and `SIMD` in the kernel manager (S-8) for various problem sizes, and observing the synthesized kernel execution time at each step, until execution time stops improving. Table IV presents an example of the values found for both parameters.

D. Hotspot Identification (D-1) and Extraction (S-5)

The above three strategies use the same process to identify hotspots, that is, regions of code where most time is spent. For this purpose, input applications are automatically instrumented with timers on each outermost loop (D-1). The instrumented code is compiled and run, such that all loops comprising greater than 50% of the program's execution time are identified. This 50% threshold is parameterisable. Note that, while tools like *gprof* provide similar information, they work at the function-level which is too coarse-grained, or at the line-level which is too fine-grained. We focus on loops, which are often hotspot candidates for hardware acceleration. Once a hotspot is identified, it is extracted into a function and replaced in the original source with a call to that function (S-5).

V. Evaluation

A. Experimental Setup

Artisan is developed using the ROSE compiler framework [1], designed to support source-to-source C++ transformations. In particular, we implement the query and instrumentation mechanisms on top of a customised version of ROSE, and wrap this functionality inside the Python 3 environment.

We evaluate our approach with CPU and FPGA targets using three case study applications: AdPredictor [10], K-Means Classification [11], and N-Body Simulation [12]. These applications include constructs that are often found in complex AI and engineering applications, such as computationally intensive loops with predictable memory access patterns and large input data sizes. Because of these similarities, we expect the optimisation strategies described in this paper to be applicable to other programs.

For CPU baseline and OpenMP experiments, we use C++11 and compile with -O2 using g++ 7.4 targeting a platform with

TABLE II
SUMMARY OF PERFORMANCE RESULTS AND DEVELOPMENT EFFORT REQUIRED FOR EACH STRATEGY AND APPLICATION.

	Optimisation Strategy		
	Multi-CPU (U-1)	**1D-FPGA (U-2)**	**ND-FPGA (U-3)**
AdPredictor	Speedup: **7.5**× LOC: +2	Speedup: 37.8× LOC: +163	Speedup: 4.0× LOC: +167
K-Means Classification	Speedup: 7.0× LOC: +2	Speedup: **132.5**× LOC: +149	Speedup: 0.4× LOC: +153
N-Body Simulation	Speedup: 7.4× LOC: +2	Speedup: 1.9× LOC: +144	Speedup: **24.3**× LOC: +148

eight i7-9700K CPU @ 3.60GHz cores. For FPGA experiments, we use the Intel HLS compiler [13] and Intel OpenCL SDK for FPGAs [14] with Quartus Prime Pro 19.3 [15] targeting an Intel Arria 10 GX FPGA Development Kit [16].

This section shows how Artisan meets the two key requirements in Section I: *customisable* strategies that optimise an agnostic application code to multiple platforms (R1), and *reusable* strategies that can serve multiple applications (R2). For customisable optimisations, we apply and compare the performance of our three strategies to a single application (Section V-B). To demonstrate reusable strategies, we apply the same three strategies to the remaining two applications, determining the most suitable one for each (Section V-C).

A summary of our results for each application and strategy is included in Table II. The speedup values reported are for the largest application workload, i.e. the maximum speedup achieved by each optimised implementation relative to the input, single-threaded CPU implementation. For the FPGA strategies, we observe execution times of synthesised hardware designs. The added lines of code (LOC: +N) quantifies the additional code required to get from the input application to the optimised C++ or OpenCL implementation. This is a representation of the developer effort offloaded to Artisan.

B. Customisable Optimisation Strategies

This section evaluates the process of optimising a single application (AdPredictor) for multiple targets without hardware expertise. AdPredictor is a Bayesian machine-learning algorithm used by Microsoft's Bing search engine for advertisement ("ad") recommendations. Our code implements the training module which processes *ad impressions*. Impressions are tuples consisting of attributes related to a particular search session, such as the user ID, IP address, query terms, and the ad's relative page ranking. For each impression, the system observes whether the ad was clicked or not. Several ad impressions are logged for training, which defines the problem size as shown in Fig. 8. The goal of the training process is to update prior belief with a set of new observations to improve ranking of ads that will be most likely clicked.

Our workflow (Fig. 7), automatically analyses the platform-agnostic application code and identifies a hotspot loop to accelerate. In this case, the extracted loop, **a**, contains two inner loops nested at the same level, **_b** and **_c**. The number of iterations of **a** depends on the problem size supplied at execution time (N), while both inner loops have fixed bounds with 12 iterations each. The process of applying each strategy

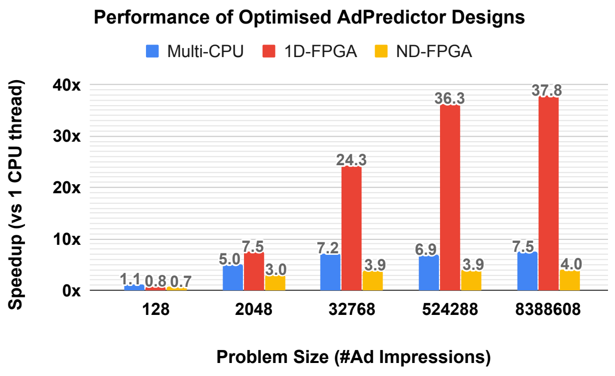


Fig. 8. Performance of different strategies for AdPredictor.

to AdPredictor is described below, and execution time results are included in Fig. 8.

- **Multi-CPU (U-1).** U-1 determines that the optimal number of threads for AdPredictor on our CPU target is 8 (as expected, since we have 8 cores). As shown in Fig. 8, the Multi-CPU speedup ranges from 1.1 times for 128 ad impressions to 7.5 times for 8,388,608 ad impressions.
- **1D-FPGA (U-2).** Table III outlines the process of applying U-2 to AdPredictor. Version 1 is the initial FPGA kernel with the unmodified hotspot, already more than 20 times faster than the input implementation for large problem sizes. DSE determines an unroll factor of 16 for fixed loops (**_b** and **_c**). The number of cycles decreases by a factor of 24. The final pipelined design is $\sim$1.5 times faster than the reference FPGA version, and up to 37.8 times faster than the input CPU version (Fig. 8).
- **ND-FPGA (U-3).** Table IV shows the hill-climbing process to determine the best work-group size (`WG`) and number of SIMD work-items (`SIMD`) for the ND-FPGA strategy. For AdPredictor, the values are automatically determined as `SIMD=4` and `WG=8`, achieving a 4 times speedup compared to the input CPU version.

It can be seen that 1D-FPGA is most effective for AdPredictor, achieving up to 38 times speedup compared to the input CPU implementation. The smallest workload (128), however, is best suited to Multi-CPU, since data transfer overhead outweighs accelerator performance. We can also see that out of the three strategies, the ND-FPGA strategy performs the worst, suggesting that the algorithm is more suited to pipeline parallelism than data parallellism.

The key takeaway from these experiments is that all three strategies can be applied to the same application with little or no developer effort or expertise, and thus the most suitable optimised version can be automatically selected.

C. Reusable Optimisation Strategies

This section demonstrates that optimisations can be described once and then applied to multiple applications. This is important to significantly reduce optimisation effort. We apply each of the three strategies to the two other platform-agnostic case study applications: K-Means and N-Body, following the same process described in Section V-B (see Table II).

TABLE III
ADPREDICTOR 1D-FPGA (U-2) STRATEGY (N = # AD IMPRESSIONS).

Version	f (MHz)	II			# Cycles	Resource Utilisation				Execution Time (ms)	
		a	_b	_c		ALUTs	FFs	RAMs	DSPs	N=2^{15}	N=2^{21}
0. Software Reference	-	-	-	-	-	-	-	-	-	46.20	2955.96
1. 1D FPGA Reference	310.00	≥ 1	1	1	$\geq 24N$	193326 (23%)	24346 (14%)	409 (15%)	211 (14%)	2.28	124.37
2. 1D FPGA Unrolled	315.97	1	n/a	n/a	N	102030 (12%)	189038 (11%)	499 (18%)	239 (16%)	1.78	78.89

TABLE IV
ADPREDICTOR ND-FPGA (U-3) STRATEGY.

N	Execution Time (ms)				
(# ad impressions)	SIMD=2 WG=2	SIMD=4 WG=4	SIMD=8 WG=8	**SIMD=4 WG=8**	SIMD=4 WG=16
2^{12}	2.35	1.68	2.09	**1.67**	1.67
2^{15}	15.61	11.89	14.62	**11.66**	11.70
2^{18}	124.91	93.04	111.26	**92.86**	92.99
2^{21}	988.90	735.22	876.91	**733.84**	734.35

For each application, the Multi-CPU results achieve similar performance, 7.0-7.5 times speedup. The ND-FPGA strategy is the most suitable for N-Body, providing up to 24 times speedup, while it is the least suitable for both K-Means Classification and AdPredictor. For K-Means, ND-FPGA performs worse than the input CPU implementation, with over 2 times slowdown, while 1D-FPGA performs exceptionally well, with up to 133 times speedup. This suggests that the K-Means algorithm is well-suited to pipeline parallelism, while N-Body is better suited to data parallelism.

D. Discussion

The goal of Artisan is to capture and codify hardware and domain expertise in order to make it accessible to non-experts. Table II includes the added lines of code required to manually produce identical optimised outputs starting from our experiment inputs. For our FPGA strategies, over 100 extra lines are required, and for the Multi-CPU strategy, two are required. While these numbers do not reflect the effort required to master OpenCL and OpenMP, they represent the effort that Artisan captures and offloads from the application developer.

Furthermore, as demonstrated in Section V-C, developer productivity is improved since optimisation efforts (Artisan meta-programs) need only be described once and can then be reused for multiple input applications. In this paper, all meta-programs were developed by a single developer in two weeks. While our applications are relatively small, the same meta-programs can operate on larger, more complex programs, in which it is not obvious (a) where hotspots are located, and (b) how to apply optimisation transformations.

Note that the effectiveness of our approach depends on how the application code is written: unstructured and obscured code may be difficult to optimise [17]. Meta-programs can be used not only to automate optimisations, but also to guide developers to revise their programs to make them more amenable to optimisation. For instance, they can instruct developers to use arrays instead of pointer arithmetic, or use iterative instead of recursive algorithms.

VI. RELATED WORK

Approaches, such as Delite [18], focus on developing Domain-Specific Languages (DSLs) to restrict application language semantics. This allows developers to express their algorithms in a language that naturally models their problems. However, it forces developers to choose one domain, as the inter-operability between domains (necessary for larger problems) is limited, and supporting new transformations requires expertise with the compiler architecture. HeteroCL [19], on the other hand, provides a Python-based DSL which decouples algorithm specification from hardware customisation. While HeteroCL provides a high level of abstraction to define and customise applications for heterogeneous targets, it still requires application developers to learn a new programming model and to understand optimisation techniques, and cannot be used to optimise existing software code.

Artisan's co-design methodology is similar to LARA [2], [20], however it has different goals. LARA is an aspect-oriented programming DSL that was extended to support compile-time optimisations [21] for multiple target languages, such as C and MATLAB. In contrast, Artisan meta-programs are based on a well-known general-purpose language (Python 3), focusing on design-flow orchestration, with artifacts such as source-code, tools and platforms handled as first-class objects.

VII. CONCLUSION

In this paper, we present a novel compilation approach that addresses the complexity of mapping high-level software source-code to heterogeneous hardware architectures. Our meta-programming approach, Artisan, decouples functional and optimisation concerns, maintaining two independent descriptions: application experts focus on algorithmic behaviour, and platform experts focus on the mapping process. Artisan offers design-flow orchestration based on two key requirements: customisable optimisations and reusable strategies. We have developed and evaluated an Artisan prototype using three case study applications and three reusable optimisation strategies, achieving at least a 24 times speedup for each application on CPU or FPGA targets with little application developer effort.

Future work includes: extending our meta-program repository to support more hardware platforms and tools, and to explore new programming models such as Intel OneAPI [22].

ACKNOWLEDGEMENT

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

REFERENCES

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

[2] J. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "LARA: an aspect-oriented programming language for embedded systems," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM, 2012, pp. 179–190.

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

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

[5] Mentor, "Catapult® High-Level Synthesis," https://www.mentor.com/hls-lp/catapult-high-level-synthesis, 2020, [Online; accessed Mar-2020].

[6] University of Toronto, "LegUp 4.0 Documentation," http://legup.eecg.utoronto.ca/docs/4.0/index.html, 2020, [Online; accessed Mar-2020].

[7] Q. Liu, T. Todman, W. Luk, and G. A. Constantinides, "Optimizing hardware design by composing utility-directed transformations," *IEEE Transactions on Computers*, vol. 61, no. 12, pp. 1800–1812, 2011.

[8] K. W. Susanto, T. Todman, J. G. F. Coutinho, and W. Luk, "Design validation by symbolic simulation and equivalence checking: A case study in memory optimization for image manipulation," in *SOFSEM*, 2009.

[9] Intel Corporation, "Intel FPGA SDK for OpenCL Pro Edition: Programming Guide," https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html, 2020, [Online; accessed Mar-2020].

[10] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich, "Web-scale Bayesian Click-through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine," in *ICML*. Omnipress, 2010, pp. 13–20.

[11] Maxeler, "Classification of N-dim data by Euclidean measure," https://github.com/maxeler/Classification, 2015, [Online; accessed Mar-2020].

[12] ——, "N-Body Particle Simulation," https://github.com/maxeler/NBody, 2015, [Online; accessed Mar-2020].

[13] Intel Corporation, "High-Level Synthesis Compiler - Intel HLS Compiler," https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html, 2020, [Online; accessed Mar-2020].

[14] ——, "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] ——, "Intel FPGA Development Tools: Intel Quartus Prime Software Suite," https://www.intel.co.uk/content/www/uk/en/software/programmable/quartus-prime/overview.html, 2020, [Online; accessed Mar-2020].

[16] ——, "Arria® 10 GX FPGA Development Kit," https://www.intel.com/content/www/us/en/programmable/products, 2020, [Online; accessed Mar-2020].

[17] M. J. Wolfe, C. Shanklin, and L. Ortega, *High Performance Compilers for Parallel Computing*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[18] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, p. 134, 2014.

[19] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, New York, NY, USA, 2019, p. 242–251.

[20] 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, pp. 1 – 26, 2018.

[21] 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.

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