Autotuning High-Level Synthesis for FPGAs Using OpenTuner and LegUp

Pedro Bruel
and Alfredo Goldman
University of São Paulo, Brazil

Sai Rahul Chalamalasetti
and Dejan Milojicic
Hewlett Packard Labs, USA

***Abstract*—Changes in Moore's law and Dennard's scaling made hardware accelerators critical for performance improvement, but configuring them for performance, area, and energy efficiency is hard and requires expert knowledge. High-Level Synthesis (HLS) tools enable hardware design for FPGAs to be done in high-level languages reducing the cost and time needed but still requiring configuration. This paper presents an open-source, flexible and virtualized autotuner for LegUp High-Level Synthesis parameters. Our optimization target was the *Weighted Normalized Sum* (WNS) of 8 hardware metrics. Weights were used to define 3 optimization scenarios targeting *Area*, *Performance & Latency* and *Performance*, plus a *Balanced* scenario. The autotuner found optimized HLS parameters that decreased WNS by up to 16% in the *Balanced* scenario, 23% in the *Area* scenario, 23% in the *Performance* scenario and 24% in the *Performance & Latency* scenario. This approach enables autotuning High-Level Synthesis parameters for different objectives by selecting weights for hardware metrics.**

I. Introduction

The slowing of Moore's law and the breakdown of Dennard's scaling made it harder for homogeneous processors to deliver the performance and energy efficiency needed by current and future applications. This motivated a ubiquitous effort to adapt and use heterogeneous architectures such as GPUs and FPGAs as hardware accelerators. GPUs have been widely used for High-Performance Computing tasks but their architecture has limited use in latency sensitive applications, where FPGAs excel. FPGAs have been traditionally programmed using Hardware Description Languages, such as Verilog and VHDL, and were used to emulate ASICs and high-end network switches.

It is a challenge for software engineers to leverage FPGA capabilities. This became increasingly relevant by the adaptation of FPGAs for data centers [1], [2], [3], a field dominated by software engineers. Essential support for software engineers can be provided by High-Level Synthesis (HLS), the process of generating hardware descriptions from high-level code. HLS lowers the complexity of hardware design from the point of view of software engineering and has become increasingly viable as part of the FPGA design methodology, with support from vendor HLS tools [4], [5] for C/C++ and OpenCL. The benefits of higher-level abstractions come with the cost of decreased application performance, making FPGAs less viable as accelerators. Thus, optimizing HLS still requires domain expertise and exhaustive or manual exploration of design spaces and configurations.

High-Level Synthesis is an $\mathbf{NP}$-Complete problem [6] and a common strategy for its solution involves the divide-and-conquer approach [7]. The most important sub-problems to solve are *scheduling*, where operations are assigned to specific clock cycles, and *binding*, where operations are assigned to specific hardware functional units, which can be shared between operations. LegUp [8] is an open-source HLS tool implemented as a compiler pass for the LLVM Compiler Infrastructure [9]. It receives code in LLVM's intermediate representation as input and produces as output a hardware description in Verilog. LegUp exposes configuration parameters of its HLS process, set with a configuration file.

Autotuning treats the process of optimizing or configuring programs as a search problem. In this paper the program to be configured is LegUp and the search space is composed of approximately 10^{126} possible combinations of HLS parameters. Such a large search space is unfeasible to search exhaustively or by hand. OpenTuner [10] is an autotuning framework that combines search techniques to explore complex search spaces. It determines computational resources to each technique based on its previous results.

In this paper we present an autotuner for LegUp HLS parameters implemented using the OpenTuner framework. The autotuner targeted 8 hardware metrics obtained from Quartus [11], for applications of the CHStone HLS benchmark suite [12] in the Intel StratixV FPGA. One of the obstacles we faced was making accurate predictions for hardware metrics from a set of HLS parameters. We decided to run our autotuner with the metrics reported by the lengthier process of hardware synthesis instead.

Our main contribution is an open-source-based, flexible and virtualized autotuning methodology for High-Level Synthesis for FPGAs. Using this methodology software engineers can write code in high-level languages and obtain an optimized hardware description that optimizes different objectives by assigning relative weights to hardware metrics. Our autotuner's virtualized implementation enables deployment in distributed environments. We present data showing that our autotuner found optimized HLS parameters for CHStone applications that decreased the *Weighted Normalized Sum* (**WNS**) of hardware metrics by up to 21.5% on average, and 10% on average.

The paper is organized as follows. Section II presents a background of works related to autotuning, HLS tools and autotuning for FPGAs. Section III describes our autotuner implementation, LegUp's HLS parameters and our optimization metrics. Section IV presents our experiments and the benchmark we used to validate our approach. Section V discusses the results we obtained with our autotuner. Section VI summarizes the results and discusses future work.

II. Background

In this section we discuss background work related to autotuning, HLS tools, and autotuning for FPGAs.

a) Autotuning: Rice's conceptual framework [13] supports autotuners in various problem domains. In 1997, the PHiPAC system [14] used code generators and search scripts to automatically generate high performance code for matrix multiplication. Since then, systems tackled different domains with a diversity of strategies. In an effort to provide a common representation of multiple parallel programming models, the INSIEME compiler project [15] implements abstractions for OpenMP, MPI and OpenCL, and generates optimized parallel code for heterogeneous multi-core architectures.

Some systems provide generic tools that enable the implementation of autotuners in various domains. PetaBricks [16] is a language, compiler and autotuner that introduces abstractions that enable programmers to define multiple algorithms for the same problem. The ParamILS framework [17] applies stochastic local search methods for algorithm configuration and parameter tuning. The OpenTuner framework [10] provides ensembles of techniques that search spaces of program configurations.

b) Tools for HLS: Research tools for High-Level Synthesis have been developed as well as vendor tools [4], [5]. Villareal *et al.* [18] implemented extensions to the Riverside Optimizing Compiler for Configurable Circuits (ROCCC), which also uses the LLVM compiler infrastructure, to add support for generating VHDL from C code. Implemented within GCC, GAUT [19] is an open-source HLS tool for generating VHDL from C/C++ code. Other HLS tools such as Mitrion [20], Impulse [21] and Handel [22] also generate hardware descriptions from C code. We refer the reader to the survey from Nane *et al.* [23] for a comprehensive analysis of recent approaches to HLS.

c) Autotuning for FPGAs: Recent works study autotuning approaches for FPGA compilation. Xu et al. [24] used distributed OpenTuner instances to optimize the compilation flow from hardware description to bitstream. They optimize configuration parameters from the Verilog-to-Routing (VTR) toolflow [25] and target frequency, wall-clock time and logic utilization. Huang et al. [26] study the effect of LLVM pass ordering and application in LegUp's HLS process, demonstrating the complexity of the search space and the difficulty of its exhaustive exploration. They exhaustively explore a subset of LLVM passes and target logic utilization, execution cycles, frequency, and wall-clock time. Mametjanov *et al.* [27] propose a machine-learning-based approach to tune design parameters for performance and power consumption. Nabi and Vanderbauwhede [28] present a model for performance and resource utilization for designs based on an intermediate representation.

III. Autotuner

This section describes our autotuner implementation, the LegUp HLS parameters selected for tuning, and the autotuning metrics used to measure the quality of HLS configurations.

A. Autotuner Design

We implemented our autotuner with OpenTuner [10], using ensembles of search techniques to find an optimized selection of LegUp [6] HLS parameters, according to our cost function, for 11 of the CHStone [12] applications.

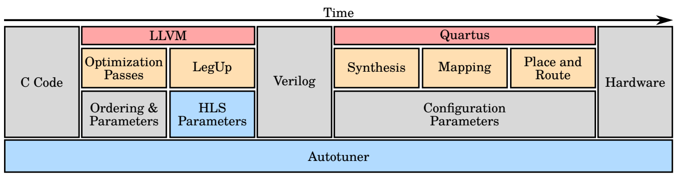


Fig. 1. High-Level Synthesis compilation process. The autotuner search space at the HLS stage is highlighted in blue

The autotuner used a combination of four techniques: two variations of greedy genetic algorithms, a differential evolution algorithm and an implementation of the Nelder-Mead method. Whenever a good set of parameters is found it is shared between techniques and they update their local values. Computational resources were shared between search techniques according to their past results. The algorithm used to share resources was the *Multi-Armed Bandit with sliding window, Area Under the Curve credit assignment* (MAB AUC). We refer the reader to the OpenTuner paper [10] for a detailed description.

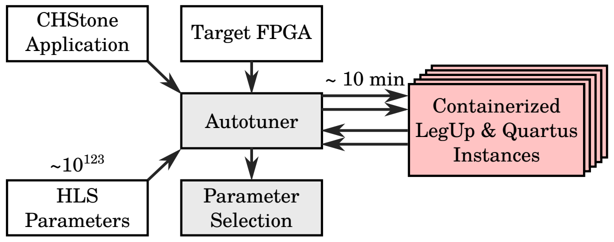


Fig. 2. Autotuner Setup

Figure 1 shows the steps to generate a hardware description from C code. It also shows the Quartus steps to generate bitstreams from hardware descriptions and to obtain the hardware metrics we targeted. Our autotuner used LegUp's HLS parameters as the search space, but it completed the hardware generation process to obtain metrics from Quartus, as represented by blue highlights in Figure 1.

Figure 2 shows our setup using Docker containers running LegUp and Quartus. This virtualized setup enabled the portable installation of dependencies and can be used to run experiments in distributed environments. The arrows coming from the autotuner to the containers represent the flow of new configurations generated by search techniques, and the arrows coming from the containers to the autotuner represent the flow of measurements for a set of parameters. For CHStone applications the measurement process takes approximately 10 minutes to complete, and the majority of this time is spent in Quartus's synthesis, mapping and place & route.

B. *High-Level Synthesis Parameters*

We selected an extensive set of LegUp High-Level Synthesis parameters, shown partially in Table I. Each parameter in the first two rows of Table I has an 8, 16, 32 and 64 bit variant. *Operation Latency* parameters define the number of clock cycles required to complete a given operation when compiled with LegUp. *Zero-latency* operations can be performed in a single clock cycle. *Resource Constraint* parameters define the number of times a given operation can be performed in a clock cycle. *Boolean or Multi-Valued* parameters are used to set various advanced configurations. For example, the **enable_pattern_sharing** parameter can be set to enable resource sharing for patterns of computational operators, as is described by Hadjis *et al.* [29]. For a complete list and description of each parameter, please refer to LegUp's official documentation [30].

TABLE I
SUBSET OF ALL AUTOTUNED LEGUP HLS PARAMETERS

Type	Parameters
Operation Latency	**altfp_**[divide, truncate, fptosi, add, subtract, multiply, extend, sitofp], **unsigned_**[multiply, divide, add, modulus], **signed_**[modulus, divide, multiply, add, **comp_**[o, u]], [local_mem, mem]**_dual_port**, **reg**
Resource Constraint	**signed_**[divide, multiply, modulus, add], **altfp_**[multiply, add, subtract, divide], **unsigned_**[modulus, multiply, add, divide], [shared_mem, mem]**_dual_port**
Boolean or Multi-value	**pattern_share_**[add, shift, sub, bitops], **sdc_**[multipump, no_chaining, priority], **pipeline_**[resource_sharing, all], **ps_**[min_size, min_width, max_size, bit_diff_threshold], **mb_**[minimize_hw, max_back_passes], **no_roms**, **multiplier_no_chain**, **dont_chain_get_elem_ptr**, **clock_period**, **no_loop_pipelining**, **incremental_sdc**, **disable_reg_sharing**, **set_combine_basicblock**, **enable_pattern_sharing**, **multipumping**, **dual_port_binding**, **modulo_scheduler**, **explicit_lpm_mults**

C. *Autotuning Metrics*

LegUp does not provide accurate metric estimates for its HLS process. To obtain values for hardware metrics we needed to perform the synthesis, mapping and place & route steps. We used Quartus to do so, and selected 8 hardware metrics reported by Quartus to compose our cost or fitness function. From the fitter summary we obtained 6 metrics. *Logic Utilization* (*LUT*) measures the number of logic elements and is composed of Adaptive Look-Up Table (ALUTs), memory ALUTs, logic registers or dedicated logic registers. The *Registers* (*Regs.*), *Virtual Pins* (*Pins*), *Block Memory Bits* (*Blocks*), *RAM Blocks* (*BRAM*) and *DSP Blocks* (*DSP*) metrics measure the usage of the resources indicated by their names.

From the timing analysis we obtained the *Cycles* and *FMax* metrics, used to compute the *Wall-Clock Time* metric. This metric composed the cost function, but *Cycles* and *FMax* were not individually used. We chose to do that because all of our hardware metrics needed to be minimized except for *FMax*, and computing *Wall-Clock Time* instead solved that restriction. The *Wall-Clock Time* wct is computed by $wct = \mathit{Cycles} \times (\alpha/\mathit{FMax})$, where $\alpha = 10^6$ because *FMax* is reported in MHz.

Equation 1 describes the cost or fitness function used by our autotuner to evaluate the sets of HLS parameters generated during tuning. The function $f(M, W)$ computes a *Weighted Normalized Sum* (**WNS**) of the measured metrics $m_i \in M$, where M was described previously. The weights $w_i \in W$ correspond to one of the scenarios in Table II. A value is computed for each metric m_i in relation to an initial value m_i^0 measured for each metric. For a given set of measurements M_t, a value of $f(M_t, W) = 1.0$ means that there was no improvement relative to the starting HLS set. The objective of the autotuner is to minimize $f(M, W)$.

$$f(M, W) = \frac{\displaystyle\sum_{\substack{m_i \in M \\ w_i \in W}} w_i \left(\frac{m_i}{m_i^0}\right)}{\displaystyle\sum_{w_i \in W} w_i} \tag{1}$$

IV. EXPERIMENTS

This section describes the optimization scenarios, the CHStone applications, and the experimental settings.

A. *Optimization Scenarios*

Table II shows the assigned weights in our 4 optimization scenarios. The *Area*-targeting scenario assigns low weights to wall-clock time metrics. The *Performance & Latency* scenario assigns high weights to wall-clock time metrics and also to the number of registers used. The *Performance* scenario assigns low weights to area metrics and cycles, assigning a high weight only to frequency. The balanced scenario assigns the same weight to every metric. The weights assigned to the metrics that do not appear on Table II are always 1. The weights are integers and powers of 2.

TABLE II
WEIGHTS FOR OPTIMIZATION SCENARIOS
(*High* = 8, *Medium* = 4, *Low* = 2)

Metric	*Area*	*Perf. & Lat*	*Performance*	*Balanced*
LUT	High	Low	Low	Medium
Registers	High	High	Medium	Medium
BRAMs	High	Low	Low	Medium
DSPs	High	Low	Low	Medium
FMax	Low	High	High	Medium
Cycles	Low	High	Low	Medium

We compared results when starting from a *default* configuration with the results when starting at a *random* set of parameters. The default configuration for the StratixV was provided by LegUp and the comparison was performed in the *Balanced* optimization scenario.

B. *Applications*

To test and validate our autotuner we used 11 applications from the CHStone HLS benchmark suite [12]. CHStone applications are implemented in the C language and contain inputs

TABLE III
AUTOTUNED CHSTONE APPLICATIONS

Application	Short Description
blowfish	Symmetric-key block cypher
aes	Advanced Encryption Algorithm (AES)
adpcm	Adaptive Differential Pulse Code Modulation dec. and enc.
sha	Secure Hash Algorithm (SHA)
motion	Motion vector decoding from MPEG-2
mips	Simplified MIPS processor
gsm	Predictive coding analysis of systems for mobile comms.
dfsin	Sine function for double-precision floating-point numbers
dfmul	Double-precision floating-point multiplication
dfdiv	Double-precision floating-point division
dfadd	Double-precision floating-point addition

and previously computed outputs, allowing for correctness checks to be performed for all applications.

Table III provides short descriptions of the 11 CHStone applications we used. We were not able to compile the *jpeg* CHStone application, so did not use it. All experiments targeted the *Intel StratixV 5SGXEA7N2F45C2* FPGA.

C. *Experimental Design and Settings*

We performed 10 tuning runs of $1.5h$ for each application. Section V presents the mean relative improvements for each application and individual metric. The code needed to run the experiments and generate the figures, as well as the implementation of the autotuner and all data we generated, is open and hosted at GitHub [31].

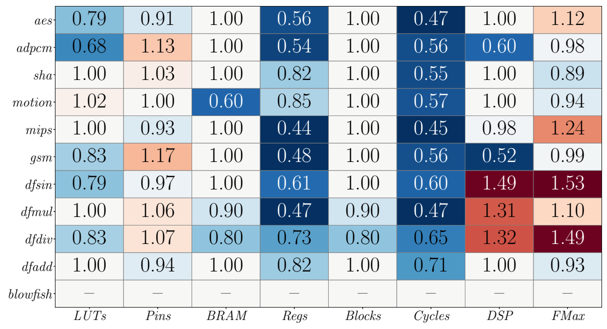
The experimental settings included Docker for virtualization and reproducibility, LegUp $v4.0$, Quartus Prime Standard Edition $v16.0$, and CHStone. All experiments were performed on a machine with two Intel(R) Xeon(R) CPU E5-2699 v3 with 18 x86_64 cores each, and 503GB of RAM. The instructions and the code to reproduce the software experimental environment are open and hosted at GitHub[32].

V. RESULTS

This section presents summaries of the results from 10 autotuning runs of $1.5h$ in the scenarios from Table II. Results are presented in *heatmaps* where each row has one of the 11 CHStone applications in Table III and each column has one of the 8 hardware metrics and their *Weighted Normalized Sum* (**WNS**) as described in Section III-C.

Cells on heatmaps show the ratio of tuned to initial values of a hardware metric in a CHStone application, averaged over 10 autotuning runs. The objective of the autotuner is to minimize all hardware metrics except for *FMax*. To create a consistent presentation of our data in the heatmaps we inverted the ratios for *FMax* so that cell values less than 1.0 always mean a better metric value. Heatmaps are also colored so that darker blues mean better values and darker reds mean worse values in relation to the starting point.

Figure 3 compares the ratios of absolute values for each hardware metric for *Default* and *Random* starts, in the *Balanced* scenario. Cell values less than 1.0 mean that the *Default* start achieved smaller absolute values that the *Random* start.



	LUTs	*Pins*	*BRAM*	*Regs*	*Blocks*	*Cycles*	*DSP*	*FMax*
aes	0.79	0.91	1.00	0.56	1.00	0.47	1.00	1.12
adpcm	0.68	1.13	1.00	0.54	1.00	0.56	0.60	0.98
sha	1.00	1.03	1.00	0.82	1.00	0.55	1.00	0.89
motion	1.02	1.00	0.60	0.85	1.00	0.57	1.00	0.94
mips	1.00	0.93	1.00	0.44	1.00	0.45	0.98	1.24
gsm	0.83	1.17	1.00	0.48	1.00	0.56	0.52	0.99
dfsin	0.79	0.97	1.00	0.61	1.00	0.60	1.49	1.53
dfmul	1.00	1.06	0.90	0.47	0.90	0.47	1.31	1.10
dfdiv	0.83	1.07	0.80	0.73	0.80	0.65	1.32	1.49
dfadd	1.00	0.94	1.00	0.82	1.00	0.71	1.00	0.93
blowfish	–	–	–	–	–	–	–	–

Fig. 3. Comparison of the absolute values for Random and Default starting points in the Balanced scenario

Values of "–" mean that the *Default* start could not find a set of HLS parameters that produced a valid output during any of the $1.5h$ tuning runs. The *Default* start found better values for most metrics.

The *Random* start found better values for *DSP*, *Pins* and *FMax* for some applications. For example, it found values 49% smaller, 6% smaller and 53% larger for *DSP*, *Pins* and *FMax*, respectively, for the *dfsin* application. The *Default* start found better values for *Regs* and *Cycles* for all applications. For example, it found values 53% smaller for *Regs* and *Cycles* for the *dfmul* application, and 56% and 55% smaller for *Regs* and *Cycles*, respectively, for the *mips* application.

We believe that the *Random* start found worst values in most cases because of the size of the search space. We recommend that autotuners use default starting points specific to a given application and board. The remaining results in this Section used the *Default* starting point provided by LegUp.

Figure 4 shows the results for the *Balanced* scenario. These results are the baseline for evaluating the autotuner in other scenarios, since all metrics had the same weight. The optimization target was the *Weighted Normalized Sum* (**WNS**) of hardware metrics, but we were also interested in the changes in other metrics as their relative weights changed. In the *Balanced* scenario we expected to see smaller improvements of **WNS** due to the competition of concurrent improvements on every metric.

The autotuner found values of **WNS** 16% smaller for *adpcm* and *dfdiv*, and 15% smaller for *dfmul*. Even on the *Balanced* scenario it is possible to see that some metrics decreased while others decreased consistently over the 10 tuning runs. *FMax* and *DSP* had the larger improvements for most applications, for example, 51% greater *FMax* in *adpcm* and 69% smaller *DSP* in *dfmul*. *Cycles*, *Regs* and *Pins* had the worst results in this scenario, with 34% larger *Cycles* in *dfdiv*, 15% larger *Regs* in *dfdiv* and 17% larger *Pins* in *gsm*. Other metrics had smaller improvements or no improvements at all in most applications.

Figure 5 shows the results for the *Area* scenario. We believe that the greater coherence of optimization objectives is responsible for the greater improvements of **WNS** in the following scenarios. The autotuner found values of **WNS** 23% smaller for *dfdiv*, 18% smaller for *dfmul*, and smaller values overall in comparison with the *Balanced* scenario. Regarding individual metrics, the values for *FMax* were worse overall,

	WNS	*LUTs*	*Pins*	*BRAM*	*Regs*	*Blocks*	*Cycles*	*DSP*	*FMax*
aes	0.94	0.90	1.00	1.00	0.97	1.00	0.98	1.00	0.76
adpcm	0.84	0.73	1.13	1.00	0.91	1.00	1.05	0.63	0.49
sha	0.98	1.00	1.03	1.00	0.96	1.00	0.86	1.00	0.97
motion	0.98	0.97	1.00	1.00	0.99	1.00	0.95	1.00	0.95
mips	0.95	1.00	1.07	1.00	0.90	1.00	1.01	0.80	0.89
gsm	0.95	0.95	1.17	1.00	0.99	1.00	0.95	0.49	1.08
dfsin	0.93	1.03	1.03	1.00	1.05	1.00	1.07	0.65	0.73
dfmul	0.85	1.00	1.13	0.90	0.88	0.90	1.06	0.31	0.76
dfdiv	0.84	1.00	1.07	0.80	1.15	0.80	1.34	0.41	0.57
dfadd	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97
blowfish	0.99	1.00	1.00	1.00	0.98	1.00	0.98	1.00	0.99

Fig. 4. Relative improvement for all metrics in the *Balanced* scenario

with 14% smaller *FMax* in *gsm* and 62% greater *Cycles*, for example. As expected for this scenario, metrics related to area had better improvements than in the *Balanced* scenario, with 73% and 72% smaller *DSP* for *dfmul* and *dfdiv* respectively, 33% smaller *Blocks* and *BRAM* in *dfdiv* and smaller values overall for *Regs* and *LUTs*.

	WNS	*LUTs*	*Pins*	*BRAM*	*Regs*	*Blocks*	*Cycles*	*DSP*	*FMax*
aes	0.96	0.92	1.00	1.00	0.93	1.00	0.97	1.00	0.86
adpcm	0.83	0.78	1.11	1.00	0.96	1.00	1.04	0.63	0.52
sha	0.96	1.00	1.06	1.00	0.84	1.00	0.83	1.00	1.08
motion	0.97	0.94	1.00	1.00	0.97	1.00	0.92	1.00	0.95
mips	0.91	1.00	1.06	1.00	0.89	1.00	1.00	0.72	0.84
gsm	0.89	0.83	1.06	1.00	0.86	1.00	0.89	0.82	1.14
dfsin	0.94	1.00	1.06	1.00	1.00	1.00	1.02	0.76	0.80
dfmul	0.82	1.00	1.06	1.00	0.90	1.00	1.21	0.27	0.86
dfdiv	0.77	1.00	1.22	0.67	1.03	0.67	1.62	0.28	0.77
dfadd	0.99	1.00	1.11	1.00	0.94	1.00	1.00	1.00	1.04
blowfish	0.99	1.00	1.00	1.00	0.97	1.00	0.91	1.00	1.01

Fig. 5. Relative improvement for all metrics in the *Area* scenario

Figure 6 shows the results for the *Performance* scenario. The autotuner found values of **WNS** 23% smaller for *dfmul*, 19% smaller for *dfdiv*, and smaller values overall than in the *Balanced* scenario. *FMax* was the only metric with a *High* weight in this scenario, so most metrics had improvements close overall to the *Balanced* scenario. The values for *FMax* were best overall, with better improvements in most applications. For example, 41%, 30%, 44% and 37% greater *FMax* in *dfdiv*, *dfmul*, *dfsin* and *aes* respectively.

	WNS	*LUTs*	*Pins*	*BRAM*	*Regs*	*Blocks*	*Cycles*	*DSP*	*FMax*
aes	0.82	0.75	1.00	1.00	0.92	1.00	1.05	1.00	0.63
adpcm	0.84	0.94	1.17	1.00	0.96	1.00	0.94	0.69	0.74
sha	0.92	1.00	1.06	1.00	0.92	1.00	0.88	1.00	0.96
motion	0.99	1.00	1.00	1.00	1.01	1.00	1.00	1.00	0.98
mips	0.90	1.00	1.06	1.00	0.93	1.00	1.03	0.83	0.80
gsm	0.95	0.92	1.00	1.00	0.95	1.00	0.90	1.00	1.02
dfsin	0.87	1.11	1.06	1.00	1.27	1.00	1.25	0.53	0.56
dfmul	0.77	1.00	1.17	0.83	0.85	0.83	1.04	0.21	0.70
dfdiv	0.81	1.00	1.06	1.00	1.03	1.00	1.25	0.50	0.59
dfadd	0.97	1.00	1.00	1.00	0.97	1.00	1.00	1.00	0.94
blowfish	0.94	1.00	1.00	1.00	0.98	1.00	0.91	1.00	0.95

Fig. 6. Relative improvement for all metrics in the *Performance* scenario

Figure 7 shows the results for the *Performance & Latency* scenario. The autotuner found values of **WNS** 24% smaller for *adpcm*, 18% smaller for *dfdiv*, and smaller values overall than in the *Balanced* scenario. *Regs*, *Cycles* and *FMax* had higher weights in this scenario, and also better improvements overall. For example, 16% and 15% smaller *Regs* in *dfdiv* and *sha* respectively, 23% and 11% smaller *Cycles* in *sha* and *aes* respectively, and 53% greater *FMax* in *adpcm*. Although *FMax* had the worst improvements in relation to the *Balanced* scenario, the *Wall-Clock Time* was still decreased by the smaller values of *Cycles*.

	WNS	*LUTs*	*Pins*	*BRAM*	*Regs*	*Blocks*	*Cycles*	*DSP*	*FMax*
aes	0.83	0.83	1.00	1.00	0.88	1.00	0.89	1.00	0.73
adpcm	0.76	0.78	1.11	1.00	0.95	1.00	0.98	0.62	0.47
sha	0.88	1.00	1.06	1.00	0.85	1.00	0.77	1.00	1.00
motion	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.95
mips	0.95	1.00	1.11	1.00	0.91	1.00	0.99	0.89	0.96
gsm	0.92	0.92	1.11	1.00	0.87	1.00	0.90	0.67	1.11
dfsin	0.97	1.00	1.17	1.00	0.99	1.00	0.99	0.61	1.00
dfmul	0.86	1.00	1.22	1.00	0.89	1.00	1.01	0.33	0.84
dfdiv	0.82	1.00	1.11	1.00	0.84	1.00	0.96	0.34	0.83
dfadd	0.98	1.00	1.17	1.00	0.94	1.00	0.98	1.00	1.01
blowfish	0.96	1.00	1.00	1.00	0.97	1.00	0.93	1.00	0.97

Fig. 7. Relative improvement for all metrics in the *Performance & Latency* scenario

Figure 8 summarizes the average improvements on **WNS** in the 4 scenarios over 10 runs. Only the *Weighted Normalized Sum* of metrics directly guided optimization. With the exception of *dfadd* in the *Balanced* scenario, the autotuner decreased **WNS** for all applications in all scenarios by 10% on average, and up to 24% for *adpcm* in the *Performance & Latency* scenario. The figure also shows the average decreases for each scenario.

	Balanced	*Area*	*Performance*	*Perf. & Lat.*
aes	0.94	0.96	0.82	0.83
adpcm	0.84	0.83	0.84	0.76
sha	0.98	0.96	0.92	0.88
motion	0.98	0.97	0.99	0.98
mips	0.95	0.91	0.90	0.95
gsm	0.95	0.89	0.95	0.92
dfsin	0.93	0.94	0.87	0.97
dfmul	0.85	0.82	0.77	0.86
dfdiv	0.84	0.77	0.81	0.82
dfadd	1.00	0.99	0.97	0.98
blowfish	0.99	0.99	0.94	0.96
Average	0.93	0.91	0.89	0.90

Fig. 8. Relative improvement for **WNS** in all scenarios

VI. Conclusion

We used the OpenTuner framework to implement an autotuner for LegUp HLS parameters in 11 applications from the CHStone benchmark suite and targeting the StratixV FPGA. We evaluated the improvements achieved by the autotuner in 8 hardware metrics and 4 optimization scenarios, using as optimization target the weighted normalized sum of the hardware metrics.

Our results show that it is always valuable to have a sensible starting position for an autotuner, and this becomes more relevant as the size of the search space and the number of targeted metrics increase. The flexibility of our approach is evidenced by the results for different optimization scenarios. Improvements in **WNS** increased when higher weights were assigned to metrics that express a coherent objective such as area, performance and latency. The improvements of metrics related to those objectives also increased.

As shown in Figure 8, the autotuner decreased **WNS** by 7% on average and up to 16% in the *Balanced* scenario, 9% on average and up to 23% for *Area* scenario, 11% on average and up to 23% in the *Performance* scenario and 10% on average and up to 24% in the *Performance & Latency* scenario.

In future work we will study the impact of different starting points on the final tuned values in each optimization scenario, for example we could start tuning for *Performance* at the best autotuned *Area* value. We expected that starting positions tailored for each target application will enable the autotuner to find better **WNS** values faster. We will also apply this autotuning methodology to HLS tools that enable a fast prediction of metric values. These tools will enable the exploration of the trade-off between prediction accuracy and the time to measure an HLS configuration.

Acknowledgment

We thank the Brazilian National Council for Scientific and Technological Development (CNPq) and Hewlett-Packard Enterprise (HPE) for their support.

References

[1] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim *et al.*, "A cloud-scale acceleration architecture," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–13.

[2] "Baidu FPGA Cloud Server," https://cloud.baidu.com/product/fpga.html, accessed: 2017-07-28.

[3] "Amazon AWS FPGA F1 Cloud Instance," https://aws.amazon.com/ec2/instance-types/f1/, accessed: 2017-07-28.

[4] D. Singh, "Implementing fpga design with the opencl standard," *Altera whitepaper*, 2011.

[5] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.

[6] A. C. Canis, "Legup: Open-source high-level synthesis research framework," Ph.D. dissertation, University of Toronto, 2015.

[7] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 24, 2013.

[9] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[10] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 303–316.

[11] "Altera Quartus Prime," https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html, accessed: 2017-07-18.

[12] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1192–1195.

[13] J. R. Rice, "The algorithm selection problem," in *Advances in Computers 15*, 1976, pp. 65–118.

[14] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology," in *Proceedings of International Conference on Supercomputing, Vienna, Austria*, 1997.

[15] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–12.

[16] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, *PetaBricks: a language and compiler for algorithmic choice*. ACM, 2009, vol. 44, no. 6.

[17] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: an automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, no. 1, pp. 267–306, 2009.

[18] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 2010, pp. 127–134.

[19] P. Coussy, G. Lhairech-Lebreton, D. Heller, and E. Martin, "Gaut a free and open source high-level synthesis tool," in *IEEE DATE*, 2010.

[20] V. V. Kindratenko, R. J. Brunner, and A. D. Myers, "Mitrion-c application development on sgi altix 350/rc100," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*. IEEE, 2007, pp. 239–250.

[21] A. Antola, M. D. Santambrogio, M. Fracassi, P. Gotti, and C. Sandionigi, "A novel hardware/software codesign methodology based on dynamic reconfiguration with impulse c and codeveloper," in *Programmable Logic, 2007. SPL'07. 2007 3rd Southern Conference on*. IEEE, 2007, pp. 221–224.

[22] S. Loo, B. E. Wells, N. Freije, and J. Kulick, "Handel-c for rapid prototyping of vlsi coprocessors for real time systems," in *System Theory, 2002. Proceedings of the Thirty-Fourth Southeastern Symposium on*. IEEE, 2002, pp. 6–10.

[23] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, no. 99, pp. 1–1, 2016.

[24] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang, "A parallel bandit-based approach for autotuning fpga compilation," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 157–166.

[25] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed *et al.*, "Vtr 7.0: Next generation architecture and cad system for fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, p. 6, 2014.

[26] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, N. Calagar, S. Brown, and J. Anderson, "The effect of compiler optimizations on high-level synthesis-generated hardware," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 3, p. 14, 2015.

[27] A. Mametjanov, P. Balaprakash, C. Choudary, P. D. Hovland, S. M. Wild, and G. Sabin, "Autotuning fpga design parameters for performance and power," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 84–91.

[28] S. W. Nabi and W. Vanderbauwhede, "A fast and accurate cost model for fpga design space exploration in hpc applications," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 114–123.

[29] S. Hadjis, A. Canis, J. H. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski, "Impact of fpga architecture on resource sharing in high-level synthesis," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 111–114.

[30] "LegUp Documentation," http://legup.eecg.utoronto.ca/docs/4.0, accessed: 2017-07-18.

[31] "Autotuner source code and Data," https://github.com/phrb/legup-tuner, accessed: 2017-07-18.

[32] "Instructions for Reproducing the Environment," https://github.com/phrb/legup-tuner, accessed: 2017-07-18.