Autotuning under Tight Budget Constraints: A Transparent Design of Experiments Approach

Pedro Bruel, Arnaud Legrand

March 20, 2019

Contents

1 Introduction

The use of heterogeneous programming models and computer architectures in High-Performance Computing (HPC) has increased despite the difficulty of optimizing and configuring legacy code, and of developing new solutions for heterogeneous computing. Due to the great diversity of programming models and architectures, there is no single optimization strategy that fits all problems and architectures.

An optimization solution tailored for a specific problem requires time and expert knowledge. Using general optimizations can be faster and cheaper, often at the cost of application-specific performance improvements. A possible solution to this trade-off is the automation of the program optimization process.

The automated selection of algorithms and configuration of programs, or *autotuning*, casts the program optimization problem as a search problem. The possible configurations and optimizations of a program are used to compose a *search space*, and search is performed by evaluating the impact of each configuration on the initial program. Using the increasingly available computing power to measure different versions of a program, an *autotuner* searches for good selections, configurations, and optimizations. Each measurement provides a value for a meaningful program metric, such as execution time and memory or power consumption.

From the implementation in a high-level programming language to the instruction selection in code generation, it is possible to expose optimization and configuration opportunities in various stages of program development and execution. The time to measure the results of an optimization choice depends on the selected stage and on the metric to be optimized. Some stages are more expensive to measure and therefore to optimize. For example, access to certain architectures might be costly and limited, or it may take a long time to solve certain problem instances.

The initial objective of this thesis was to study the effectiveness of search heuristics, such as simulated annealing, on autotuning problems. Our first target autotuning domain was the set of parameters of a compiler for GPU programs. The search heuristics for this case study were implemented using the OpenTuner framework [1], and consisted of an ensemble of search heuristics coordinated by a Multi-Armed Bandit algorithm. The autotuner searched for a set of compilation parameters that optimized 17 heterogeneous GPU kernels, from a set of approximately 10^{23} possible combinations of all parameters. With 1.5h autotuning runs we have achieved up to $4\times$ speedup in comparison with the CUDA compiler's high-level optimizations. The compilation and execution times of programs in this autotuning domain are relatively fast, and were in the order of a few seconds to a minute. Since measurement costs are relatively small, search heuristics could find good optimizations using as many measurements as needed. A detailed description of this work is available in our paper [4] published in the *Concurrency and Computation: Practice and Experience* journal, which is reproduced in Appendix B.

Our next case study was developed in collaboration with *Hewlett-Packard Enterprise*, and consisted of applying the same heuristics-based autotuning approach to the configuration of parameters involved in the generation of FPGA hardware specification from source code in the C language, a process called *High-Level Synthesis* (HLS). The main difference from our work with GPU compiler parameters was the time to obtain the hardware specification, which could be in the order of hours for a single kernel.

In this more complex scenario, we achieved up to $2\times$ improvements for different hardware metrics using conventional search algorithms. These results were obtained in a simple HLS benchmark, for which compilation times were in the order of minutes. The search space was composed of approximately 10^{123} possible configurations, which is much larger than the search space in our previous work with GPUs. Search space size and the larger measurement cost meant that we did not expect the heuristics-based approach to have the same effectiveness as in the GPU compiler case study. This work was published [5] at the 2017 *IEEE International Conference on ReConFigurable Computing and FPGAs*, and is reproduced in Appendix C.

Approaches using classical machine learning and optimization techniques would not scale to industrial-level HLS, where each compilation can take hours to complete. Search space properties also increase the complexity of the problem, in particular its structure composed of binary, factorial and continuous variables with potentially complex interactions. Our results on autotuning HLS for FPGAs corroborate the conclusion that the empirical autotuning of expensive-to-evaluate functions, such as those that appear on the autotuning of HLS, require a more *parsimonious* and *transparent* approach.

The next step taken on this work was study the Design of Experiments (DoE) methodology, with the goal of developing a parsimonious and transparent approach to autotuning. One of the first detailed descriptions and mathematical treatment of DoE was presented by Ronald Fisher [7] in his 1937 book *The Design of Experiments*, where he discussed principles of experimentation, latin square sampling and factorial designs. Later books such as the ones from Jain [6], Montgomery [8] and Box *et al.* [3] present comprehensive and detailed foundations. Techniques based on DoE are parsimonious because they allow decreasing the number of measurements required to determine certain relationships between parameters and metrics, and are transparent because each choice of parameter value can be justified by the results of statistical tests.

In DoE terminology, a *design* is a plan for executing a series of measurements, or *experiments*, whose objective is to identify relationships between *factors* and *responses*. While factors and responses can refer to different concrete entities in other domains, in computer experiments factors can be configuration parameters

for algorithms and compilers, for example, and responses can be the execution time or memory consumption of a program.

Designs can serve diverse purposes, from identifying the most significant factors for performance, to fitting analytical performance models for the response. The field of DoE encompasses the mathematical formalization of the construction of experimental designs. More practical works in the field present algorithms to generate designs with different objectives and restrictions.

Our application of the DoE methodology requires support for factors of different types and number of possible values, such as binary flags, integer and floating point numerical values and unordered enumerations of abstract values. We also need designs that minimize the number of experiments needed for identifying the most relevant factors in a given problem, since at this moment we are not interesting in a precise analytical model.

The design construction techniques that fit these requirements are limited. Considering flexibility of application and effectiveness, the best candidate we have found so far are *D-Optimal* designs. Considering that we are going to analyse the results of an experiments plan, the *D-Efficiency* of a design is inversely proportional to the *geometric mean* of the *eigenvalues* of the plan's *covariance matrix*. A D-Optimal design has the best D-Efficiency. Our current approach is based on D-Optimal designs.

In the first part of the stay of the student in the *Laboratoire d'Informatique de Grenoble* (LIG), we performed a comprehensive study of the area of *Design of Experiments* and developed an initial approach that applies this knowledge to the autotuning of computer programs. We obtained promising results on the autotuning of a Laplacian kernel for GPUs where the entire search space was available and the performance model was known. We are now evaluating the performance of our method on the SPAPT autotuning benchmark [2], which contains a set of parametrized High-Performance Computing kernels and applications presenting large search spaces and difficult search problems. We are using computational resources from Grid5000 to run experiments. We intend to submit the results of the initial promising application of our approach and of the experiments using SPAPT to the 33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS).

The rest of this document is organized as follows. Section 2 presents and discusses our objectives. Section 3 presents the work plan and schedule. Annex A.

2 Objectives

An effective autotuning strategy for expensive-to-evaluate functions should be semi-automatic and strongly build on previous knowledge of FPGA compiling experts. We believe techniques inspired from Design of Experiments and from the "sequential approach" are likely to be effective in this context.

We are interested in extending this approach to FPGAs using real applications and industry partners who are able to guide the process. We expect that devising new robust and cheap experimental designs that enable mixing binary, factorial and continuous variables will present an interesting challenge. This collaboration is therefore an interesting opportunity. The stay of 18 months of Pedro Henrique Rocha Bruel at Communauté Université Grenoble Alpes (ComUGA) is collaborating in joining efforts and achieving progress.

3 Schedule

Planned Research Activities	**Periods**		
	04/19-08/19	09/19-01/20	02/20-05/20
Sampling for D-Optimal Designs			
User-Centric Optimization			
Extended Paper			
Case Study on FPGAs			
Thesis Writing			

References

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316. ACM, 2014.

[2] Prasanna Balaprakash, Stefan M Wild, and Boyana Norris. SPAPT: Search problems in automatic performance tuning. *Procedia Computer Science*, 9:1959–1968, 2012.

[3] George EP Box, J Stuart Hunter, and William Gordon Hunter. *Statistics for experimenters: design, innovation, and discovery*, volume 2. Wiley-Interscience New York, 2005.

[4] Pedro Bruel, Marcos Amarís, and Alfredo Goldman. Autotuning cuda compiler parameters for heterogeneous applications using the opentuner framework. *Concurrency and Computation: Practice and Experience*, pages e3973–n/a, 2017.

[5] Pedro Bruel, Alfredo Goldman, Sai Rahul Chalamalasetti, and Dejan Milojicic. Autotuning high-level synthesis for fpgas using opentuner and legup. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2017.

[6] Per Nikolaj D Bukh. *The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling*. JSTOR, 1992.

[7] Ronald Aylmer Fisher. *The design of experiments*. Oliver And Boyd; Edinburgh; London, 1937.

[8] Douglas C Montgomery. *Design and analysis of experiments*. John wiley & sons, 2017.

Pedro Bruel | Researcher & Software Engineer

Performance Tuning & Modeling • Optimal Experimental Design

2b Rue Charles Gounod, 38000 Grenoble, France +33 07 68 33 24 38
pedro.bruel@gmail.com ime.usp.br/~phrb pedro-bruel phrb

Experience

2017 – May 2020 **PhD Researcher**
Grenoble Informatics Laboratory
University of Grenoble Alpes, France
Developing Design of Experiments Techniques for autotuning High-Performance Computing kernels and compilers on CPUs, GPUs and FPGAs

2015 – May 2020 **PhD Researcher**
Software Systems Laboratory
University of São Paulo, Brazil
Developed autotuners for High-Level Synthesis compilers for FPGAs and for the CUDA Compiler using Search Heuristics

2015 – 2016 **PhD Research Collaborator**
Hewllett-Packard Enterprise
University of São Paulo, Brazil
Developed an autotuner for the LegUp High-Level Synthesis compiler for Altera FPGAs

2012 – 2014 **Research Intern**
Computer Music Research Group
University of São Paulo, Brazil
Maintained and developed a multiagent system for music composition via agent interaction

Education

2015 – 2020 **PhD in Computer Science**
University of Grenoble Alpes, France
University of São Paulo, Brazil
High-Performance Computing, Autotuning, Design of Experiments, Search Heuristics, Data Analysis

2010 – 2014 **BsC in Molecular Sciences**
University of São Paulo, Brazil
Multiagent Systems, Digital Signal Processing

Selected Publications

Bruel, P., Goldman, A., Chalamalasetti, S.R. and Milojicic, D., **2017**. *Autotuning high-level synthesis for FPGAs using OpenTuner and LegUp*. In ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on (pp. 1-6). IEEE.

Bruel, P., Chalamalasetti, S.R., Dalton, C., El Hajj, I., Goldman, A., Graves, C., Hwu, W.M., Laplante, P., Milojicic, D., Ndu, G. and Strachan, J.P., **2017**. *Generalize or Die: Operating Systems Support for Memristor-based Accelerators*. In 2017 IEEE International Conference on Rebooting Computing (ICRC) (pp. 1-8). IEEE.

Bruel, P., Amarís, M. and Goldman, A., **2017**. *Autotuning CUDA compiler parameters for heterogeneous applications using the OpenTuner framework*. Concurrency and Computation: Practice and Experience, 29(22), p.e3973.

Bruel, P., Meirelles, P., Cobe, R., Goldman, A., **2017**. *OpenMP or Pthreads: Which is Better for Beginners?*. In 8th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU).

Gonçalves, R., Amaris, M., Okada, T., **Bruel, P.** and Goldman, A., **2016**. *Openmp is not as Easy as it Appears*. In System Sciences (HICSS), 2016 49th Hawaii International Conference on (pp. 5742-5751). IEEE.

Bruel, P. and Queiroz, M., **2014**. *A Protocol for creating Multiagent Systems in Ensemble with Pure Data*. In International Computer Music Conference (ICMC).

Languages

Portuguese	Native
English	*CEFR C2*
French	*CEFR B2*
Spanish	*CEFR A2*

Skills

Performance Tuning
Stochastic Search Heuristics
Design of Experiments
Optimal Experimental Design
Performance Modeling
Data Science

Software Engineering
Python Julia R Bash
C/C++ OpenMP MPI
CUDA C Java

Tools and Infrastructure
GNU/Linux Git Grid5000
GCE/AWS Automated Testing
Continuous Integration LaTeX

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **00**:1–13
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Autotuning CUDA Compiler Parameters for Heterogeneous Applications using the OpenTuner Framework

Pedro Bruel, Marcos Amarís and Alfredo Goldman

Instituto de Matemática e Estatística (IME), Universidade de São Paulo (USP), R. do Matão, 1010 – Cidade Universitária, São Paulo – SP, 05508-090

SUMMARY

A Graphics Processing Unit (GPU) is a parallel computing coprocessor specialized in accelerating vector operations. The enormous heterogeneity of parallel computing platforms justifies and motivates the development of automated optimization tools and techniques. The Algorithm Selection Problem consists in finding a combination of algorithms, or a configuration of an algorithm, that optimizes the solution of a set of problem instances. An autotuner solves the Algorithm Selection Problem using search and optimization techniques.
In this paper we implement an autotuner for the CUDA compiler's parameters using the OpenTuner framework. The autotuner searches for a set of compilation parameters that optimizes the time to solve a problem. We analyse the performance speedups, in comparison with high-level compiler optimizations, achieved in three different GPU devices, for 17 heterogeneous GPU applications, 12 of which are from the Rodinia Benchmark Suite. The autotuner often beat the compiler's high-level optimizations, but underperformed for some problems. We achieved over 2x speedup for *Gaussian Elimination* and almost 2x speedup for *Heart Wall*, both problems from the Rodinia Benchmark, and over 4x speedup for a matrix multiplication algorithm. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Autotuning, GPUs, Compilers, CUDA, OpenTuner

1. INTRODUCTION

The usage of and reliance on parallel computing models and architectures became increasingly predominant since their emergence. The enormous heterogeneity of these platforms complicates the task of hand-optimizing parallel computing programs, justifying and motivating the development of automated tools and techniques for program optimization.

A Graphics Processing Unit (GPU) is a parallel computing coprocessor specialized in accelerating vector operations such as graphics rendering. The General Purpose computing on Graphics Processing Unit methodology, or GPGPU, consists in providing accessible programming interfaces for languages such as C and Python that enable the use of GPUs in different parallel computing domains. The Compute Unified Device Architecture (CUDA) is a GPGPU platform introduced by the NVIDIA corporation.

The program autotuning problem fits in the framework of the Algorithm Selection Problem, introduced by Rice in 1976 [1]. The objective of an autotuner is to select the best algorithm, or algorithm configuration, for each instance of a problem. Algorithms or configurations are selected according to performance metrics such as the time to solve the problem instance, the accuracy of the

*Correspondence to: Instituto de Matemática e Estatística (IME), - CCSL - Universidade de São Paulo (USP), R. do Matão, 1010 – Cidade Universitária, São Paulo – SP, 05508-090

solution and the energy consumed. The set of all possible algorithms and configurations that solve a problem defines a *search space*. Various optimization techniques search this space, guided by the performance metrics, for the algorithm or configuration that best solve the problem.

There are specialized autotuners for domains such as matrix multiplication [2], dense [3] or sparse [4] matrix linear algebra, and parallel programming [5]. Other autotuning frameworks provide more general tools for the representation and search of program configurations, enabling the implementation of autotuners for different problem domains [6, 7].

The OpenTuner framework [6] provides tools for the implementation of autotuners for various problem domains. It implements different search techniques that explore the same search space for program optimizations. Running and measuring program execution time — that is, the empirical exploration of the search space — is done sequentially. The framework also provides support for parallel compilation.

In this paper we implemented an autotuner for the CUDA compiler using the use the OpenTuner framework [6] and used it to search for the compilation parameters that optimize the performance of 17 heterogeneous GPU applications, 12 of which are from the Rodinia Benchmark Suite [8]. We used 3 different NVIDIA GPUs in the experiments, the Tesla K40, the GTX 980 and the GTX 750.

Our main contribution is to show that it is possible to optimize code written for GPUs by automatically tuning just the parameters of the CUDA compiler. We propose a thorough methodology for analysing result correctness and checking for invalid flag combinations and compilation errors. The optimization achieved by autotuning often beat the compiler high-level optimization options, such as `-O1`, `-O2` and `-O3`. The autotuner found compilation options that achieved over 2x speedup for the *Gaussian Elimination* problem from the Rodinia Benchmark Suite, almost 2x speedup for *Heart Wall* problem, also from Rodinia, and over 4x speedup for one of the matrix multiplication optimizations in our benchmark, in comparison with the high-level compiler optimizations. We also show that the compilation parameters that optimize an algorithm for a given GPU architecture will not always achieve the same performance in different hardware.

The rest of this paper is organized as follows. Section 2 presents work related to autotuning and the optimization of GPU programs, discusses NVIDIA GPU architectures and the OpenTuner framework. Section 3 describes the GPU testbed, the algorithm benchmark, the compiler parameters that define search space and the implementation of the autotuner. Section 4 presents the results of the autotuning experiments, and discusses the performance of the autotuner and the selection of compiler flags. Section 5 concludes the paper.

2. RELATED WORK AND BACKGROUND

Rice's conceptual framework [1] formed the foundation of autotuners in various problem domains. In 1997, the PHiPAC system [2] 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. Whaley *et al.* [3] introduced the ATLAS project, that optimizes dense matrix multiplication routines. The OSKI [4] library provides automatically tuned kernels for sparse matrices. The FFTW [9] library provides tuned C subroutines for computing the Discrete Fourier Transform. Periscope [10] is a distributed online autotuner for parallel systems and single-node performance. In an effort to provide a common representation of multiple parallel programming models, the INSIEME compiler project [5] 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 [11] is a language, compiler and autotuner that introduces abstractions, such as the `either...or` construct, that enable programmers to define multiple algorithms for the same problem. The ParamILS framework [7] applies stochastic local search methods for algorithm configuration and parameter tuning. The OpenTuner framework [6] provides ensembles of techniques that search spaces of program configurations. Bosboom *et al.* and Eliahu use OpenTuner to implement a domain specific language for data-flow programming [12] and a framework for recursive parallel algorithm optimization [13].

Concurrency Computat.: Pract. Exper. (2016)
DOI: 10.1002/cpe

The accuracy of a GPU performance model is subject to low level elements such as instruction pipeline usage and small cache hierarchies. A GPU's performance approaches its peak when the instruction pipeline is saturated, but becomes unpredictable when the pipeline is under-utilized [14, 15]. Considering the effects of small cache hierarchies [16, 17] and memory-access divergence [18, 19] is also critical to a GPU performance model.

Guo and Wang [20] introduce a framework for autotuning the number of threads and the sizes of blocks and warps used by the CUDA compiler for sparse matrix and vector multiplication GPU applications. Li *et al.* [21] discuss the performance of autotuning techniques in highly-tuned GPU General Matrix to Matrix Multiplication (GEMMs) routines, highlighting the difficulty in developing optimized code for new GPU architectures. Grauer-Gray *et al.* [22] autotune an optimization space of GPU kernels focusing on tiling, loop permutation, unrolling, and parallelization. Chaparala *et al.* [23] autotune GPU-accelerated Quadratic Assignment Problem solvers. Sedaghati *et al.* [24] build a decision model for the selection of sparse matrix representations in GPUs.

To the best of our knowledge, this is the first work that tackles the autotuning problem in GPUs by tuning the parameters of the CUDA compiler, using the OpenTuner framework. This approach has the potential to improve the performance of legacy code in a variety of parallel computing platforms and heterogeneous parallel applications.

2.1. The OpenTuner Framework

OpenTuner search spaces are defined by *Configuration*s, that are composed of *Parameter*s of various types. Each type has restricted bounds and manipulation functions that enable the exploration of the search space. OpenTuner implements ensembles of optimization techniques that perform well in different problem domains. The framework uses *meta-techniques* to coordinate the distribution of resources among techniques. Results found during search are shared through a database. An OpenTuner application can implement its own search techniques and meta-techniques, making the ensemble more robust. OpenTuner's source code is available[†] under the MIT License.

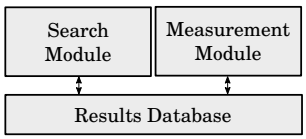

Figure 1. Simplified OpenTuner Architecture.

Figure 1 shows a high-level view of OpenTuner's architecture. Measurement and searching are done in separate modules, whose main classes are called *drivers*. The search driver requests measurements by registering configurations to the database. The measurement driver reads those configurations and writes back the desired results. Using the measurement driver available in the framework, the measurements are performed sequentially. Before running the user-defined measurement function, the OpenTuner measurement driver calls compilation hooks that can be run in parallel using Python threads.

OpenTuner implements optimization techniques such as the Nelder-Mead [25] simplex method and Simulated Annealing [26]. A resource sharing mechanism, called *meta-technique*, aims to take advantage of the strengths of each technique by balancing the exploitation of a technique that has produced good results in the past and the exploration of unused and possibly better ones.

2.2. NVIDIA GPU Microarchitecture

NVIDIA GPU architectures have multiple asynchronous and parallel Streaming Multiprocessors (SMs) which contain Scalar Processors (SPs), Special Function Units (SFUs) and load/store units.

[†]Hosted at GitHub: `github.com/jansel/opentuner` [Accessed on 10 February 2015]

Concurrency Computat.: Pract. Exper. (2016)
DOI: 10.1002/cpe

Each group of 32 parallel threads scheduled by and SM, or *warp*, is able to read from memory concurrently [27]. The Tesla, Fermi, Kepler and Maxwell NVIDIA architectures vary in a large number of features, such as number of cores, registers, SFUs, load/store units, on-chip and cache memory sizes, processor clock frequency, memory bandwidth, unified memory spaces and dynamic kernel launches. Those differences are summarized in the Compute Capability (C.C.) of an NVIDIA GPU.

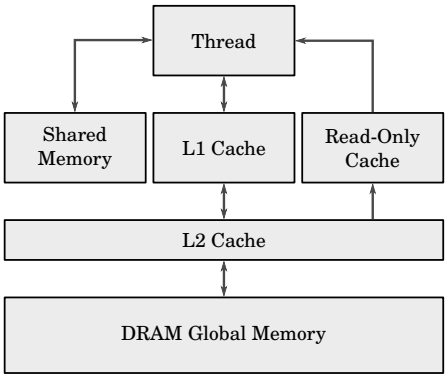

Figure 2. Memory hierarchy of threads in a kernel executed in Kepler architectures, adapted from [28]

The hierarchical memory of an NVIDIA GPU contains global and shared portions. Global memory is big, off-chip, has a high latency and can be accessed by all threads of the kernel. Shared memory is small, on-chip, has a low-latency and can be accessed only by threads in a same SM. Each SM has its own shared L1 cache, and new architectures have coherent global L2 caches. Optimizing thread accesses to different memory levels is essential to achieve good performance. Figure 2 shows the memory access hierarchy in a Kepler architecture.

2.3. Compute Unified Device Architecture (CUDA)

The CUDA programming model and platform enables the use of NVIDIA GPUs for scientific and general purpose computation. A single *master* thread runs in the CPU, launching and managing computations on the GPU. Data for the computations has to be transferred from the main memory to the GPU's memory. Multiple computations launched by the master thread, or *kernels*, can run asynchronously and concurrently. If the threads from a same warp must execute different instructions the CUDA compiler must generate code to branch the execution correctly, making the program lose performance due to this *warp divergence*.

The CUDA language extends C and provides a multi-step compiler, called *NVCC*, that translates CUDA code to Parallel Thread Execution code, or *PTX*. *NVCC* uses the host's C++ compiler in several compilation steps, and also to generate code to be executed in the host. The final binary generated by *NVCC* contains code for the GPU and the host[29]. When *PTX* code is loaded by an application at run-time, it is compiled to binary code by the host's device driver. This binary code can be executed in the device's processing cores, and is architecture-specific. The targeted architecture can be specified using *NVCC* parameters[30].

3. EXPERIMENTS

In this section we present the GPU testbed, the algorithm benchmark and the autotuner implementation and its search space.

Concurrency Computat.: Pract. Exper. (2016)
DOI: 10.1002/cpe

3.1. GPU Testbed

To be able to show that different GPUs require different options to improve performance, and that it is possible to achieve speedups in different hardware, we wanted to tune our benchmark for different NVIDIA microarchitectures. We selected the Tesla K40, the GTX 750 and the GTX 980. Using the GK110B chip, the Tesla K40 is a Kepler microarchitecture GPU, the oldest GPU of the testbed. The GTX 750 is the first Maxwell microarchitecture GPU and uses the GM107 chip. The GTX 980 is the latest Maxwell GPU and uses the GM204 chip. Table I summarizes the hardware characteristics of the three GPUs.

Model	C.C.	Global Memory	Bus	Bandwidth	L2	Cores/SM	Clock
Tesla-K40	3.5	12 GB	384-bit	276.5 GB/s	1.5 MB	2880/15	745 Mhz
GTX-750	5.0	1 GB	128-bit	86.4 GB/s	2 MB	512/4	1110 Mhz
GTX-980	5.2	4 GB	256-bit	224.3 GB/s	2 MB	2048/16	1216 Mhz

Table I. Hardware specifications of the GPUs in the testbed

3.2. Algorithm Benchmark

We composed a benchmark with 17 heterogeneous GPU applications. The benchmark contains 4 optimization strategies for *matrix multiplication* (counted as a single application) [30], 1 vector addition problem [30], 1 solution for the *maximum sub-array problem* [31], 2 *sorting* algorithms [32] and 12 applications from the Rodinia Benchmark Suite [33]. The remainder of this section discusses some details of these algorithms, and introduces a three-letter code for each application, used in Section 4.

Matrix Multiplication We used four different memory access optimizations: global memory with non-coalesced accesses (MMU); global memory with coalesced accesses (MMG); shared memory with non-coalesced accesses to global memory (MSU); and shared memory with coalesced accesses to global memory (MMS). All performance measurements for these optimizations used square matrices, with $N = 8192$. The run-time complexity for a sequential matrix multiplication algorithm using two matrices of size $N \times N$ is $O(N^3)$. In a CUDA application with N^2 threads, the run-time complexity is $O(N)$

Vector Addition Algorithm For two vectors A and B, the Vector Addition (VAD) $C = A + B$ is obtained by adding the corresponding components. In a GPU algorithm each thread performs an addition of a position of the vectors A and B and stores the result in the vector C. The number of GPU threads used in this problem is equal to the size of the vectors. All performance measurements for this problem used arrays of size $N = 4194304$.

Maximum Sub-Array Problem Let X be a sequence of N integer numbers $(x_1, ..., x_N)$. The Maximum Sub-Array Problem (MSA) consists of finding the contiguous sub-array within X which has the largest sum of elements. The solution for this problem is frequently used in computational biology for gene identification, analysis of sequences of protein and DNA, and identification of hydrophobic regions. The maximum sub-array problem can be solved sequentially in $O(N)$ comparisons [34], and in $O(N/t)$ with a parallel solution [35], where t is the number of threads.

The implementation [31] used in this paper creates a kernel with 4096 threads, divided in 32 blocks with 128 threads. The N elements are divided in intervals of N/t, and each block receives a portion of the array. The blocks use the shared memory for storing segments, which are read from the global memory using coalesced accesses. Each interval is reduced to a set of 5 integer variables, which are stored in vector of size $5 \times t$ in global memory. This vector is then transferred to the CPU memory for later processing. All performance measurements for the Maximum Sub-array Problem used arrays with $N = 1073741824$.

Concurrency Computat.: Pract. Exper. (2016)
DOI: 10.1002/cpe

Sorting Algorithms The benchmark contains two sorting algorithms, quicksort (QKS, with $N = 65536$) and bitonicsort (BTN, with $N = 4194304$) [‡]. Both algorithms sort random numbers sampled from a uniform distribution.

Rodinia Benchmark Suite Rodinia's [8] applications have implementations for multi-core CPUs and GPUs using three parallel programming models (OpenMP, CUDA and OpenCL). The benchmark was devised for heterogenous parallel computing research and its applications represent different high-level domains or behaviours, called the Berkeley dwarfs [36, 37]. Table II shows the Rodinia applications contained in our benchmark.

Application	Berkeley Dwarf[37]	Domain
B+Tree (BPT)	Graph Traversal	Search
Back Propagation (BCK)	Unstructured Grid	Pattern Recognition
Breadth-First Search (BFS)	Graph Traversal	Graph Algorithms
Gaussian Elimination (GAU)	Dense Linear Algebra	Linear Algebra
Heart Wall (HWL)	Structured Grid	Medical Imaging
Hot Spot (HOT)	Structured Grid	Physics Simulation
K-Means (KMN)	Dense Linear Algebra	Data Mining
LavaMD (LMD)	N-Body	Molecular Dynamics
LU Decomposition (LUD)	Dense Linear Algebra	Linear Algebra
Myocyte (MYO)	Structured Grid	Biological Simulation
Needleman-Wunsch (NDL)	Dynamic Programming	Bioinformatics
Path Finder (PTF)	Dynamic Programming	Grid Traversal

Table II. Rodinia applications used in the experiments

3.3. The Autotuner

Our implementation uses the OpenTuner framework's native parameter types to represent CUDA compilation options. Flags and multi-valued parameters are represented as single *EnumParameter*s, and parameters that accept numerical values are represented as *IntegerParameter*s. The same encoding was used in Jansel *et al.* [6] in the implementation of an autotuner for GCC compilation parameters.

The autotuner we implemented used the *multi-armed bandit with sliding window, area under the curve credit assignment* meta-technique, simply named *AUC Bandit*. AUC Bandit is OpenTuner's core meta-technique [6], and its ensemble of search techniques is composed by implementations of the Nelder-Mead algorithm and three variations of genetic algorithms. There is no guarantee that any tuning run will find the optimal solution, but the variance of the results found by different tuning runs decreases as tuning time progresses, as is shown in the results from the original OpenTuner paper [6].

The search in the space of CUDA options potentially generates invalid parameter combinations due to incompatible flags or architecture restrictions. To prevent these invalid configurations from misguiding the search techniques we first modified all programs in the benchmark so that they verify the correctness of their output using previously computed correct outputs. Then we made the autotuner check for errors during compilation and execution whenever testing a new configuration and assign a penalty value when finding errors. Errors could be used to acquire knowledge about the interactions and incompatibilities between parameters, which would enable the autotuner to

[‡]Obtained from: `http://digitalcommons.providence.edu/student_scholarship/7/` [Accessed on 10 February 2015]

Concurrency Computat.: Pract. Exper. (2016)
DOI: 10.1002/cpe

prune the search space and find better solutions faster. The final verification step used to validate an autotuned result was to manually check the output of the CUDA toolkit's `nvprof` profiler, certifying that the CUDA kernels were launched and executed properly.

The code for our autotuner and all the experiments and results is available[§] under the GNU General Public License.

3.4. The Search Space

Flag	Description
`no-align-double`	Specifies that `malign-double` should not be passed as a compiler argument on 32-bit platforms. **Step**: NVCC
`use_fast_math`	Uses the fast math library, implies `ftz=true`, `prec-div=false`, `prec-sqrt=false` and `fmad=true`. **Step**: NVCC
`gpu-architecture`	Specifies the NVIDIA virtual GPU architecture for which the CUDA input files must be compiled. **Step**: NVCC **Values**: `sm_20`, `sm_21`, `sm_30`, `sm_32`, `sm_35`, `sm_50`, `sm_52`
`relocatable-device-code`	Enables the generation of relocatable device code. If disabled, executable device code is generated. Relocatable device code must be linked before it can be executed. **Step**: NVCC
`ftz`	Controls single-precision denormals support. `ftz=true` flushes denormal values to zero and `ftz=false` preserves denormal values. **Step**: NVCC
`prec-div`	Controls single-precision floating-point division and reciprocals. `prec-div=true` enables the IEEE round-to-nearest mode and `prec-div=false` enables the fast approximation mode. **Step**: NVCC
`prec-sqrt`	Controls single-precision floating-point squre root. `prec-sqrt=true` enables the IEEE round-to-nearest mode and `prec-sqrt=false` enables the fast approximation mode. **Step**: NVCC
`def-load-cache`	Default cache modifier on global/generic load. **Step**: PTX **Values**: `ca`, `cg`, `cv`, `cs`
`opt-level`	Specifies high-level optimizations. **Step**: PTX **Values**: `0 - 3`
`fmad`	Enables the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations (FMAD, FFMA, or DFMA). **Step**: PTX
`allow-expensive-optimizations`	Enables the compiler to perform expensive optimizations using maximum available resources (memory and compile-time). If unspecified, default behavior is to enable this feature for optimization level $\geqslant$O2. **Step**: PTX
`maxrregcount`	Specifies the maximum number of registers that GPU functions can use. **Step**: PTX **Values**: `16 - 64`
`preserve-relocs`	Makes the `PTX` assembler generate relocatable references for variables and preserve relocations generated for them in the linked executable. **Step**: NVLINK

Table III. Description of flags in the search space

Table III details the subset of the CUDA configuration parameters used in the experiments [¶]. The parameters target different compilation steps: the *PTX* optimizing assembler; the *NVLINK* linker; and the *NVCC* compiler. We compared the performance of programs generated by tuned parameters with the standard compiler optimizations, namely `--opt-level=0,1,2,3`. Different `--opt-levels` could also be selected during tuning. We did not use compiler options that target the host linker or the library manager since they do not affect performance. The size of

[§] Hosted at GitHub: `https://github.com/phrb/gpu-autotuning` [Accessed on 10 February 2015]
[¶] Adapted from: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc [Accessed on 10 February 2015]

DOI: 10.1002/cpe

the search space defined by all possible combinations of the flags in Table III is in the order of 10^6 making hand-optimization or exhaustive searches very time consuming.

4. RESULTS

This section presents the speedups achieved for all algorithms in the application benchmark, highlights the most significant speedups, and discusses the performance and accuracy of the autotuner.

4.1. Performance Improvements

All boxplots presented in this section were made using the standard implementations available for the R language. The black band inside the boxes represents the median of the measurements. The lower and upper bounds of the box represent, respectively, the first and third quartiles of the data. The whiskers represent the third and first quartile plus and minus the *inner quartile range* times 1.5. Finally, the circles represent the outliers.

Figures 3, 4, 5 and 6 compare the distributions of 10 perfomance measurements of binaries compiled with high-level compiler optimizations and with autotuned compiler options. The results for the high-level optimizations `--opt-level=0,1,2,3` were denoted by *-O0*, *-O1*, *-O2* and *-O3*. The results for the autotuned configurations were denoted by the keyword *Tuned*.

Figure 3 shows that the autotuned solution for the Heart Wall problem (HWL) in the Tesla K40 achieved over 2x speedup in comparison with the high-level CUDA optimizations. Figure 4 shows, in the GTX 980, almost 2.5x speedup for the Gaussian Elimination problem (GAU). Figure 5 show 10% speedup of the autotuned solution for the Path Finder problem (PTF) in the Tesla K40. Figure 6 shows over 5% speedup of the autotuned solution, also in the Tesla K40, for the Myocyte problem (MYO).

Figures 7, 8, 9 and 10 present summaries of the results. The autotuner did not find solutions that improved upon the high-level optimizations for the problems BTN and MSU in any of the GPUs of the testbed, but it found solutions that achieved speedups for at least one GPU for the other problems.

We do not know precisely which hardware characteristics impacted performance the most. Although more experiments are needed to confirm the following hypothesis, we believe that the Maxwell GPUs, the GTX 980 and GTX 750, had differing results from the Tesla K40 because they are consumer grade GPUs, producing not so precise results and with different default optimizations. The similarities between the compute capabilities of the GTX GPUs could also explain the observed differences from the K40. Finally, the overall greater computing power of the GTX 980 could explain its differing results from the GTX 750, since the GTX 980 has a greater number of Cores, SMs/Cores, Bandwidth, Bus, Clock and Global Memory.

4.2. Autotuner Performance

This section presents an assessment of the autotuner's performance. Figures 11, 12, 13 and 14 present the performance of the best solution found by the autotuner versus the time these solutions were found, in seconds since the beginning of the tuning process. The Figures show the evolution of the autotuned solution for the best results in our experiments, the Heart Wall problem (HWL) and the Gaussian Elimination problem (GAU) in the GTX 980, and the Path Finder problem (PTF) and the Myocyte problem (MYO) in the Tesla K40.

The points in each graph represent the performance, in the y-axis, of the best configuration found at the corresponding tuning time, shown in the x-axis. The leftmost point in each graph represents the performance of a configuration chosen at random by the autotuner. Each subsequent point represents the performance of the best configuration found so far.

Note that the autotuner is able to quickly improve upon the initial random configuration, but the rate of improvement decays over tuning time. The duration of all tuning runs was two hours, or 7200 seconds. The rightmost point in each graph represents the performance of the last improving

Concurrency Computat.: Pract. Exper. (2016)
DOI: 10.1002/cpe

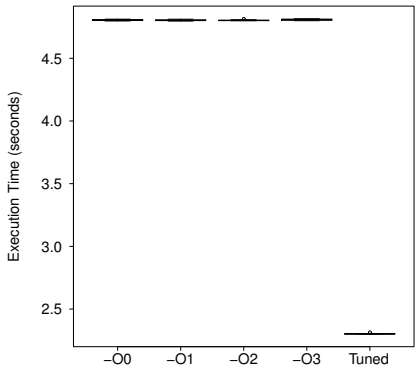

Figure 3. Boxplots for the Tesla K40, comparing autotuned results and high-level compiler optimizations for the Heart Wall problem (HWL)

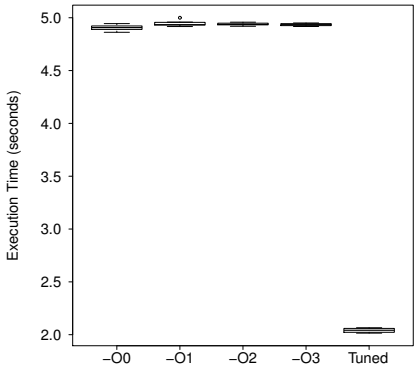

Figure 4. Boxplots for the GTX 980 GPU, comparing autotuned results and high-level compiler optimizations for the Gaussian Elimination problem (GAU)

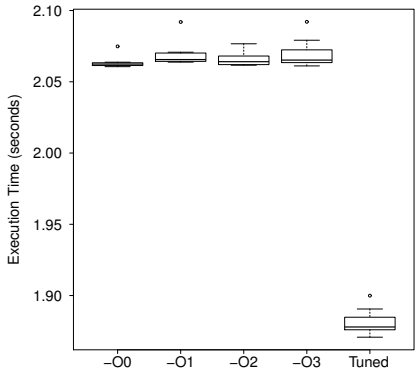

Figure 5. Boxplots for the Tesla K40, comparing autotuned results and high-level compiler optimizations for the Path Finder problem (PTF)

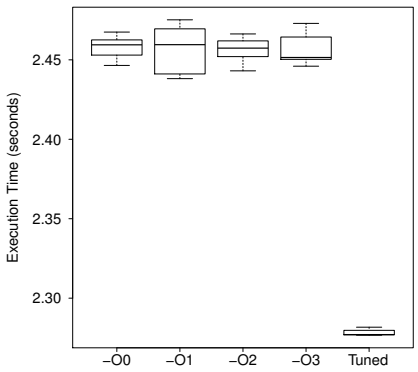

Figure 6. Boxplots for the Tesla K40, comparing autotuned results and high-level compiler optimizations for the Myocyte problem (MYO)

configuration found by the autotuner. After the tuning time of this configuration, the autotuner did not find any improving configuration until the end of its tuning run.

The autotuner used the mean of 10 performance measurements of a given configuration as the fitness value, reducing fluctuations and improving the measurement's accuracy. This is reflected in the proximity of the measurements reported by the autotuner during tuning, shown in figures 11, 12, 13 and 14 and the mean values of 10 measurements of the final solution, shown in the corresponding figures 3, 4, 5 and 6.

4.3. Parameter Selection

We attempted to associate compilation parameters to applications and GPUs using WEKA's [38] clustering algorithms. Although we could not find significant relations for most applications we detected that the `ftz=true` in MMS and the Compute Capabilities 3.0, 5.0 and 5.2 in GAU caused the speedups observed in the GTX 980 for these applications. Table IV shows clusters obtained with the K-means WEKA algorithm for autotuned parameter sets for the Rodinia Benchmark in the GTX 750. Unlike most clusters found for all GPUs and problems, these clusters did not contain an equal number of instances. The difficulty of finding associations between compiler optimizations, applications and GPUs justifies the autotuning of compiler parameters in the general case.

Concurrency Computat.: Pract. Exper. (2016)
DOI: 10.1002/cpe

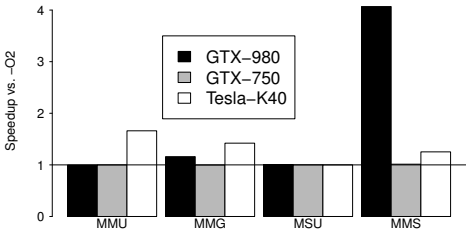

Figure 7. Summary of the speedups achieved versus *-O2* in the matrix multiplication optimizations

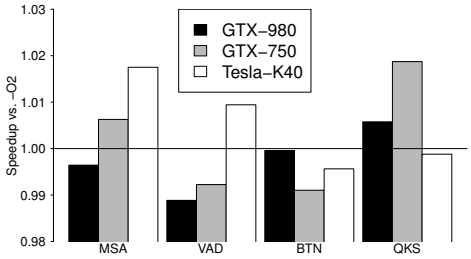

Figure 8. Summary of the speedups achieved versus *-O2* in the other independent applications

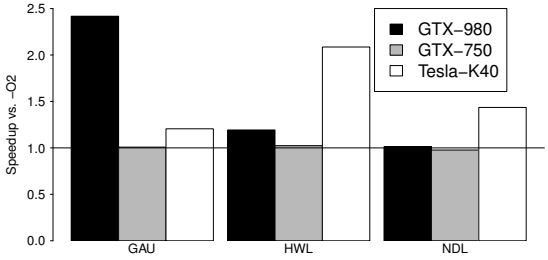

Figure 9. Summary of the biggest speedups achieved versus *-O2* in the Rodinia applications

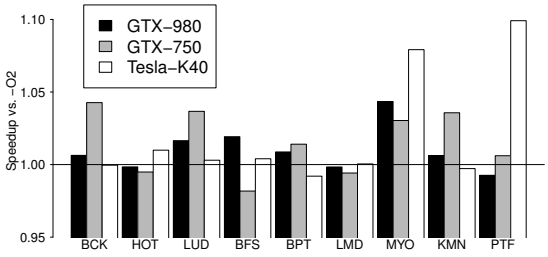

Figure 10. Summary of the smaller speedups achieved versus *-O2* in the Rodinia applications

Flag	Cluster 0 (17%)	Cluster 1 (83%)
no-align-double	on	on
use_fast_math	on	on
preserve-relocs	off	off
relocatable-device-code	true	false
ftz	true	true
prec-div	true	false
prec-sqrt	true	true
fmad	false	false
allow-expensive-optimizations	false	true
gpu-architecture	sm_20	sm_50
def-load-cache	cv	ca
opt-level	1	3
maxrregcount	42	44.6

Table IV. Parameter clusters for all Rodinia problems in the GTX 750

Concurrency Computat.: Pract. Exper. (2016)
DOI: 10.1002/cpe

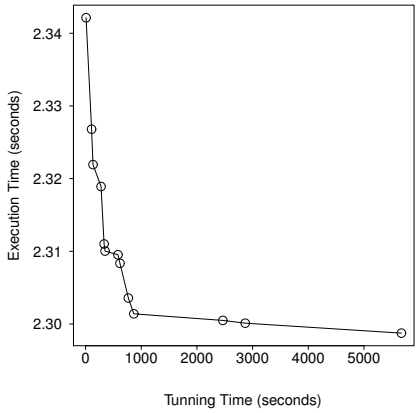

Figure 11. Best solutions found by the autotuner over time for the Heart Wall problem (HWL) in the Tesla K40

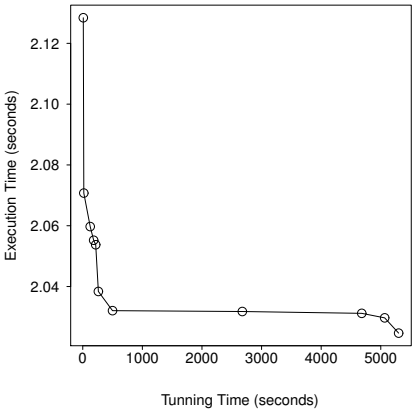

Figure 12. Best solutions found by the autotuner over time for the Gaussian Elimination problem (GAU) in the GTX 980

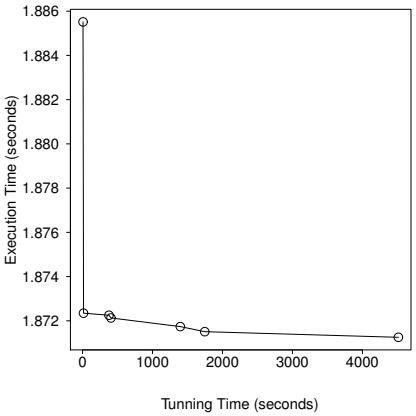

Figure 13. Best solutions found by the autotuner over time for the Path Finder problem (PTF) in the Tesla K40

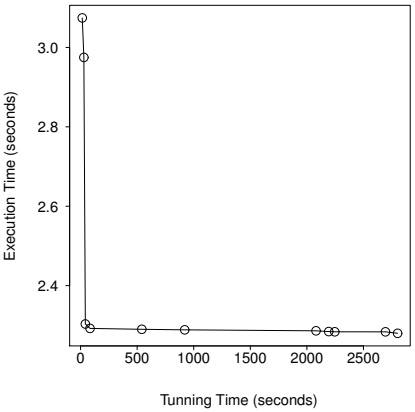

Figure 14. Best solutions found by the autotuner over time for the Myocyte problem (MYO) in the Tesla K40

5. CONCLUSION

We used the OpenTuner framework to implement an autotuner for the search space defined by the parameters of the CUDA compiler. We composed a benchmark of 17 heterogeneous applications and compared their performance in one Kepler and two Maxwell microarchitecture GPUs. Although the autotuner often beat the compiler's high-level optimizations it still underperformed for some problems. We achieved over 2x speedup for Gaussian Elimination (GAU) and almost 2x speedup for Heart Wall (HWL), and over 4x speedup for a matrix multiplication algorithm (MMS).

This paper showed that it is possible to improve the performance of GPU applications applying empirical and automatic tuning techniques. Our results and clustering attempts emphasize the importance of automatic optimization techniques by showing that different compilation options have to be selected in order to achieve performance improvements in different GPUs and that it is difficult to associate compiler optimization parameters, applications and GPUs.

Future work will include application parameters and options for the GCC compiler, which also composes the CUDA compilation chain. We would also like to apply the Programming by Optimization [39] design paradigm for GPU and parallel programming. We will continue to apply

Concurrency Computat.: Pract. Exper. (2016)
DOI: 10.1002/cpe

clustering algorithms to future experiments and we will perform more comprehensive experiments to explore the search space of CUDA compiler parameters. We hope these approaches will enable us to provide valuable guidelines for CUDA parameter selection in the future.

To the best of our knowledge, this is the first work that applied autotuning techniques to CUDA compiler parameters for GPU applications using the OpenTuner framework, comparing the speedup achieved in different GPU architectures for heterogeneous applications.

ACKNOWLEDGEMENTS

This work was supported by Hewlett-Packard, FAPESP (São Paulo Research Foundation, grant number #2012/23300-7), CAPES and CNPq. We also thank NVIDIA for the donation of a Tesla K40 GPU.

REFERENCES

1. J. R. Rice, "The algorithm selection problem," in *Advances in Computers 15*, 1976, pp. 65–118.
2. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. New York, NY, USA: ACM, 2014, pp. 253–260.
3. R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–27.
4. R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.
5. 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.
6. 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.
7. 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.
8. S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, Dec 2010, pp. 1–11.
9. M. Frigo and S. G. Johnson, "Fftw: An adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 1381–1384.
10. M. Gerndt and M. Ott, "Automatic performance analysis with periscope," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 736–748, Apr. 2010.
11. J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," *SIGPLAN Not.*, vol. 44, no. 6, pp. 38–49, Jun. 2009.
12. J. Bosboom, S. Rajadurai, W.-F. Wong, and S. Amarasinghe, "Streamjit: a commensal compiler for high-performance stream programming," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 2014, pp. 177–195.
13. D. Eliahu, O. Spillinger, A. Fox, and J. Demmel, "Frpa: A framework for recursive parallel algorithms," Master's thesis, EECS Department, University of California, Berkeley, May 2015.
14. Y. Zhang and J. Owens, "A quantitative performance analysis model for GPU architectures," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, Feb 2011, pp. 382–393.
15. M. Amarís, D. Cordeiro, A. Goldman, and R. Y. Camargo, "A simple bsp-based model to predict execution time in gpu applications," in *High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on*, Dec 2015, pp. 285–294.
16. T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee, "A performance model for gpus with caches," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 7, pp. 1800–1813, July 2015.
17. J. Picchi and W. Zhang, "Impact of l2 cache locking on gpu performance," in *SoutheastCon 2015*, April 2015, pp. 1–4.
18. D. Sampaio, R. M. d. Souza, S. Collange, and F. M. Q. a. Pereira, "Divergence analysis," *ACM Transactions on Programming Languages and Systems*, vol. 35, no. 4, pp. 13:1–13:36, Jan. 2014.
19. S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," *SIGPLAN Not.*, vol. 45, no. 5, pp. 105–114, Jan. 2010.
20. P. Guo and L. Wang, "Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus," in *Computational and Information Sciences (ICCIS), 2010 International Conference on*, Dec 2010, pp. 1154–1157.
21. Y. Li, J. Dongarra, and S. Tomov, "A note on auto-tuning gemm for gpus," in *Computational Science–ICCS 2009*. Springer, 2009, pp. 884–892.
22. S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *Innovative Parallel Computing (InPar), 2012*, May 2012, pp. 1–10.
23. A. Chaparala, C. Novoa, and A. Qasem, "Autotuning gpu-accelerated qap solvers for power and performance," in *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on*

Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on. IEEE, 2015, pp. 78–83.
24. N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 99–108.
25. J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.
26. S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi *et al.*, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.
27. NVIDIA Corporation, *CUDA C Best Practices Guide*, August 2014.
28. N. Corporation, "Web site: [Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110] Visited on Nov, 2015."
29. NVIDIA, *CUDA COMPILER DRIVER NVCC*, September 2015.
30. ——, *CUDA C: Programming Guide, Version 7.*, March 2015.
31. C. Silva, S. Song, and R. Camargo, "A parallel maximum subarray algorithm on gpus," in *5th Workshop on Applications for Multi-Core Architectures (WAMCA 2014). IEEE Int. Symp. on Computer Architecture and High Performance Computing Workshops*, Paris, 2014, pp. 12–17.
32. A. Sinha and K. Agrawa, "[Web site: Digital Commons at Providence College. Sorting in CUDA: by Ayushi Sinha] Visited on Jan, 2016."
33. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54.
34. J. L. Bates and R. L. Constable, "Proofs As Programs," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 113–136, Jan. 1985.
35. C. E. R. Alves, E. Cáceres, and S. W. Song, "BSP/CGM Algorithms for Maximum Subsequence and Maximum Subarray." in *PVM/MPI*, ser. Lecture Notes in Computer Science, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241. Springer, 2004, pp. 139–146.
36. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," TECHNICAL REPORT, UC BERKELEY, Tech. Rep., 2006.
37. K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009.
38. G. Holmes, A. Donkin, and I. H. Witten, "Weka: A machine learning workbench," in *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*. IEEE, 1994, pp. 357–361.
39. H. H. Hoos, "Programming by optimization," *Communications of the ACM*, vol. 55, no. 2, pp. 70–80, 2012.

DOI: 10.1002/cpe

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.

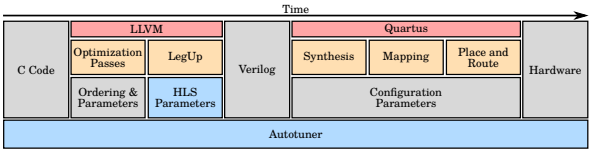

Fig. 1. High-Level Synthesis compilation process. The autotuner search space at the HLS stage is highlighted in blue

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.

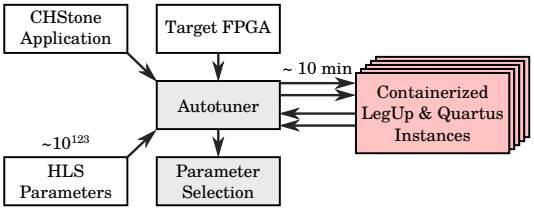

Fig. 2. Autotuner Setup

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 $wct = Cycles \times (\alpha/FMax)$, 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 $f(M, W)$.

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

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.

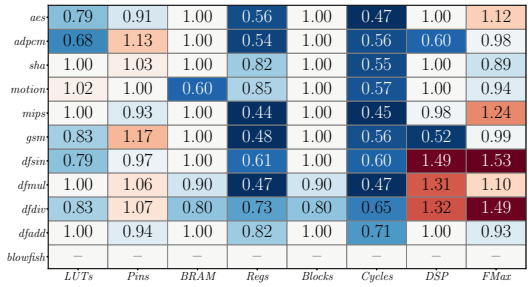

Fig. 3. Comparison of the absolute values for Random and Default starting points in the Balanced scenario

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.

Autotuning under Tight Budget Constraints: A Transparent Design of Experiments Approach

Pedro Bruel*†, Steven Quinito Masnada‡, Brice Videau*, Arnaud Legrand*, Jean-Marc Vincent*, Alfredo Goldman†

†University of São Paulo
São Paulo, Brazil
{phrb, gold}@ime.usp.br

‡University of Grenoble Alpes
Inria, CNRS, Grenoble INP, LJK
38000 Grenoble, France
steven.quinito-masnada@inria.fr

*University of Grenoble Alpes
CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France
{arnaud.legrand, brice.videau, jean-marc.vincent}@imag.fr

***Abstract*—A large amount of resources is spent writing, porting, and optimizing scientific and industrial High Performance Computing applications, which makes autotuning techniques fundamental to lower the cost of leveraging the improvements on execution time and power consumption provided by the latest software and hardware platforms. Despite the need for economy, most autotuning techniques still require a large budget of costly experimental measurements to provide good results, while rarely providing exploitable knowledge after optimization. The contribution of this paper is a user-transparent autotuning technique based on Design of Experiments that operates under tight budget constraints by significantly reducing the measurements needed to find good optimizations. Our approach enables users to make informed decisions on which optimizations to pursue and when to stop. We present an experimental evaluation of our approach and show it is capable of leveraging user decisions to find the best global configuration of a GPU Laplacian kernel using half of the measurement budget used by other common autotuning techniques. We show that our approach is also capable of finding speedups of up to 50×, compared to `gcc`'s `-O3`, for some kernels from the SPAPT benchmark suite, using up to 10× fewer measurements than random sampling.**

I. Introduction

Optimizing code for objectives such as performance and power consumption is fundamental to the success and cost-effectiveness of industrial and scientific endeavors in High Performance Computing (HPC). A considerable amount of highly specialized time and effort is spent in porting and optimizing code for GPUs, FPGAs and other hardware accelerators. Experts are also needed to leverage bleeding edge software improvements in compilers, languages, libraries and frameworks. The objective of techniques for the automatic configuration and optimization of HPC applications, or *autotuning*, is to decrease the cost and time needed to adopt efficient hardware and software. Typical autotuning targets include algorithm selection, source-to-source transformations and compiler configuration.

Autotuning can be studied as a search problem where the objective is to minimize software or hardware metrics. The exploration of the search spaces defined by code and compiler configurations and optimizations presents interesting challenges. Such spaces grow exponentially with the number of parameters, and are also difficult to explore extensively due to the often prohibitive costs of hardware utilization, program compilation and execution times. Developing autotuning strategies capable of producing good optimizations while minimizing resource utilization is therefore essential. The capability of acquiring knowledge about an optimization problem is also crucial in an autotuning strategy, since this knowledge can decrease the cost of subsequent optimizations of the same application or for the same hardware.

It is common and usually effective to use search meta-heuristics such as genetic algorithms and simulated annealing in autotuning. These strategies attempt to exploit local search space properties, but are generally incapable of exploiting global structures. Seymour *et al.* [1], Knijnenburg *et al.* [2], and Balaprakash *et al.* [3], [4] report that these strategies are not more effective than a naive uniform random sample of the search space, and usually rely on a large number of measurements or restarts to achieve performance improvements. Search strategies based on gradient descent are also commonly used in autotuning, and also rely on a large number of measurements, but their effectiveness diminishes significantly in search spaces with complex local structures. Automated machine learning autotuning strategies [5], [6], [7] are promising for building models for predicting important parameters, but still rely on a sizable data set for training.

In summary, search strategies based on meta-heuristics, gradient descent and machine learning require a large number of measurements to be effective, and are usually incapable of providing knowledge about search spaces to users. Since these strategies are not transparent, at the end of each autotuning session it is difficult to decide if and where further exploration is warranted, and often impossible to know which parameters are responsible for the observed improvements. After exploring a search space, deducing any of the space's global properties confidently is impossible, since the space was automatically explored with unknown biases.

The contribution of this paper is an autotuning strategy that leverages existing knowledge about a problem by using an initial performance model that is refined iteratively using performance measurements, statistical analysis, and user input. Our strategy places a heavy weight on decreasing autotuning costs by using a *Design of Experiments* (DoE) methodology to minimize the number of experiments needed to find optimizations. Each iteration uses *Analysis of Variance* (ANOVA) tests and *linear model regressions* to identify promising subspaces and parameter significance. An architecture- and problem-specific performance model is built iteratively and with user

input, which enables making informed decisions about which regions of the search space are worth exploring.

We evaluate the performance of our approach by optimizing a Laplacian Kernel for GPUs, where the search space, the global optimum, and a performance model approximation are known. The budget of measurements was tightly constrained on this experiment. Speedups and budget utilization reduction achieved by our approach on this setting motivated a more comprehensive performance evaluation. We chose the *Search Problems in Automatic Performance Tuning* (SPAPT) [8] benchmark suite for this evaluation, where we obtained diverse results. Out of the 17 SPAPT kernels benchmarked, no speedup could be found for three kernels, but uniform random sampling performed well on all others. For eight of the kernels, our approach found speedups of up to $50\times$, compared to `gcc`'s `-O3` with no code transformations, while using up to $10\times$ fewer measurements than random sampling.

The rest of this paper is organized as follows. Section II presents related work on source-to-source transformation, which is the main optimization target in SPAPT kernels, on autotuning systems and on search space exploration strategies. Section III discusses the implementation of our approach in detail. Section IV discusses the DoE, ANOVA, and linear regression methodology we used to develop our approach. Section V presents the results on the performance evaluation on the GPU Laplacian Kernel and on the SPAPT benchmark suite. Section VI discusses our conclusions and future work.

II. Background

This section presents the background and related work on source-to-source transformation, autotuning systems and search space exploration strategies.

A. Source-to-Source Transformation

Our approach can be applied to any autotuning domain that expresses optimization as a search problem, although the performance evaluations we present in Section V were obtained in the domain of source-to-source transformation. Several frameworks, compilers and autotuners provide tools to generate and optimize architecture-specific code [9], [10], [11], [12], [13]. We used BOAST [10] and Orio [9] to perform source-to-source transformations targeting parallelization on CPUs and GPUs, vectorization, loop transformations such as tiling and unrolling, and data structure size and copying.

B. Autotuning

John Rice's Algorithm Selection framework [14] is the precursor of autotuners in various problem domains. In 1997, the PHiPAC system [15] used code generators and search scripts to automatically generate high performance code for matrix multiplication. Since then, systems approached different domains with a variety of strategies. Dongarra *et al.* [16] introduced the ATLAS project, that optimizes dense matrix multiplication routines. The OSKI [17] library provides automatically tuned kernels for sparse matrices. The FFTW [18] library provides tuned C subroutines for computing the Discrete Fourier Transform. Periscope [19] is a distributed online autotuner for parallel systems and single-node performance. In an effort to provide a common representation of multiple parallel programming models, the INSIEME compiler project [20] implements abstractions for OpenMP, MPI and OpenCL, and generates optimized parallel code for heterogeneous multi-core architectures.

A different approach is to combine generic search algorithms and problem representation data structures in a single system that enables the implementation of autotuners for different domains. The PetaBricks [13] project provides a language, compiler and autotuner, enabling the definition and selection of multiple algorithms for the same problem. The ParamILS framework [21] applies stochastic local search algorithms to algorithm configuration and parameter tuning. The OpenTuner framework [22] provides ensembles of techniques that search the same space in parallel, while exploration is managed by a multi-armed bandit strategy.

C. Search Space Exploration Strategies

Figure 1 shows the contour of a search space defined by a function of the form $z = x^2 + y^2 + \varepsilon$, where ε is a local perturbation, and the exploration of that search space by six different strategies. In such a simple search space, even a uniform random sample can find points close to the optimum, despite not exploiting geometry. A Latin Hypercube [23] sampling strategy covers the search space more evenly, but still does not exploit the space's geometry. Strategies based on neighborhood exploration such as simulated annealing and gradient descent can exploit local structures, but may get trapped in local minima. Their performance is strongly dependent on search starting point. These strategies do not leverage global search space structure, or provide exploitable knowledge after optimization.

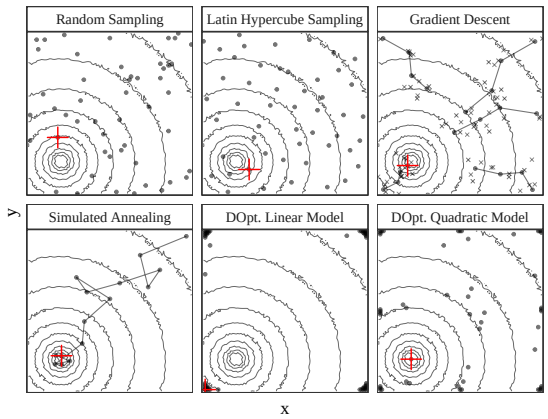

Figure 1: Exploration of the search space, using a fixed budget of 50 points. The red "+" represents the best point found by each strategy, and "×"s denote neighborhood exploration

Measurement of the kernels optimized on the performance evaluations in Section V can exceed 20 minutes, including the time of code transformation, compilation, and execution. Measurements in other problem domains can take much longer to complete. This strengthens the motivation to consider search space exploration strategies capable of operating under tight budget constraints. These strategies have been developed and improved by statisticians for a long time, and can be grouped under the DoE term.

The D-Optimal sampling strategies shown on the two rightmost bottom panels of Figure 1 are based on the DoE methodology, and leverage previous knowledge about search spaces for an efficient exploration. These strategies provide transparent analyses that enable focusing on interesting subspaces. In the next section we describe our approach to autotuning based on the DoE methodology.

III. Autotuning with Design of Experiments

An *experimental design* determines a selection of experiments whose objective is to identify the relationships between *factors* and *responses*. While factors and responses can refer to different concrete entities in other domains, in computer experiments factors can be configuration parameters for algorithms and compilers, and responses can be the execution time or memory consumption of a program. Each possible value of a factor is called a *level*. The *effect* of a factor on the measured response, without the factor's *interactions* with other factors, is the *main effect* of that factor. Experimental designs can be constructed with different goals, such as identifying the main effects or building an analytical model for the response.

In this section we discuss in detail our iterative DoE approach to autotuning. Figure 2 presents an overview of our approach, with numbered steps. In step 1 we define the factors and levels that compose the search space of the target problem, in step 2 we select an initial performance model, and in step 3 we generate an experimental design. We run the experiments in step 4 and then, as we discuss in the next section, we identify significant factors with an ANOVA test in step 5. This enables selecting and fitting a new performance model in steps 6 and 7. The new model is used in step 8 for predicting levels for each significant factor. We then go back to step 3, generating a new design for the new problem subspace with the remaining factors. Informed decisions made by the user at each step guide the outcome of each iteration.

Step 1 of our approach is to define target factors and which of their levels are worth exploring. Then, the user must select an initial performance model in step 2. Compilers typically expose many 2-level factors in the form of configuration flags, and the performance model for a single flag can only be a linear term, since there are only 2 values to measure. Interactions between flags and numerical factors such as block sizes in CUDA programs or loop unrolling amounts are also common. Deciding which levels to include for these kinds of factors requires more careful analysis. For example, if we suspect the performance model has a quadratic term for a certain factor, the design should include at least three factor levels. The ordering between the levels of other compiler parameters, such as `-O(0,1,2,3)`, is not obviously translated to a number. Factors like these are named *categorical*, and must be treated differently when constructing designs in step 3 and analyzing results in step 5.

We decided to use D-Optimal designs because their construction techniques enable mixing categorical and numerical factors in the same screening design, while biasing sampling according to the performance model. This enables the autotuner to exploit global search space structures if we use the right model. When constructing a D-Optimal design in step 3 the user can require that specific points in the search space are included, or that others are not. Algorithms for constructing D-Optimal designs are capable of adapting to these requirements by optimizing a starting design. Before settling on D-Optimal designs, we explored other design construction techniques such as the Plackett-Burman [24] screening designs shown in the next section, the *contractive replacement* technique of Addelman-Kempthorne [25] and the *direct generation* algorithm by Grömping and Fontana [26]. These techniques have strong requirements on design size and level mixing, so we opted for a more flexible technique that would enable exploring a more comprehensive class of autotuning problems.

After the design is constructed in step 3, we run each selected experiment in step 4. This step can run in parallel since experiments are independent. Not all target programs run successfully in their entire input range, making runtime failures common in this step. The user can decide whether to construct a new design using the successfully completed experiments or to continue to the analysis step if enough experiments succeed.

After running the ANOVA test in step 5, the user should apply domain knowledge to analyze the ANOVA table and determine which factors are significant. Certain factors might not appear significant and should not be included in the regression model. Selecting the model after the ANOVA test in step 6 also benefits from domain knowledge.

A central assumption of ANOVA is the *homoscedasticity* of the response, which can be interpreted as requiring the observed error on measurements to be independent of factor lev-

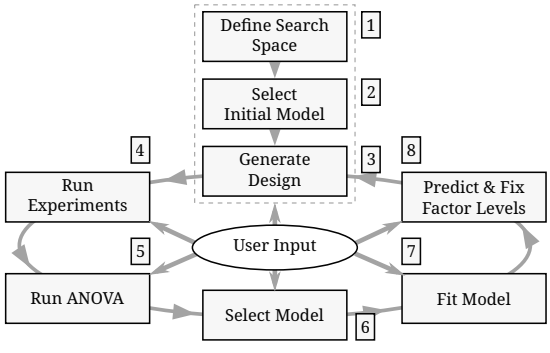

Figure 2: Overview of the DoE approach to autotuning proposed in this paper

els and of the number of measurements. Fortunately, there are statistical tests and corrections for lack of homoscedasticity. Our approach uses the homoscedasticity check and correction by power transformations from the `car` package [27] of the `R` language.

We fit the selected model to our design's data in step 7, and use the fitted model in step 8 to find levels that minimize the response. The choice of the method used to find these levels depends on factor types and on the complexity of the model and search space. If factors have discrete levels, neighborhood exploration might be needed to find levels that minimize the response around predicted levels. Constraints might put predicted levels on an undefined or invalid region on the search space. This presents challenge, because the borders of valid regions would have to be explored.

In step 8 we also fix factor levels to those predicted to achieve best performance. The user can also decide the level of trust placed on the prediction at this step, by keeping other levels available. In step 8 we perform a reduction of problem dimension by eliminating factors and decreasing the size of the search space. If we identified significant parameters correctly, we will have restricted further search to better regions. In the next section we present a simple fictional application our approach that illustrates the fundamentals of the DoE methodology, screening designs and D-Optimal designs.

IV. Design of Experiments

In this section we first present the assumptions of a traditional DoE methodology using an example of *2-level screening designs*, an efficient way to identify main effects. We then discuss techniques for the construction of efficient designs for factors with arbitrary numbers and types of levels, and present *D-Optimal* designs, the technique used in this paper.

A. Screening & Plackett-Burman Designs

Screening designs identify parsimoniously the main effects of 2-level factors in the initial stages of studying a problem. While interactions are not considered at this stage, identifying main effects early enables focusing on a smaller set of factors on subsequent experiments. A specially efficient design construction technique for screening designs was presented by Plackett and Burman [24] in 1946, and is available in the `FrF2` package [28] of the `R` language [29].

Despite having strong restrictions on the number of factors supported, Plackett-Burman designs enable the identification of main effects of n factors with $n+1$ experiments. Factors may have many levels, but Plackett-Burman designs can only be constructed for 2-level factors. Therefore, before constructing a Plackett-Burman design we must identify *high* and *low* levels for each factor.

Assuming a linear relationship between factors and the response is fundamental for running ANOVA tests using a Plackett-Burman design. Consider the linear relationship

$$\mathbf{Y} = \boldsymbol{\beta}\mathbf{X} + \varepsilon, \tag{1}$$

where ε is the error term, $\mathbf{Y}$ is the observed response, $\mathbf{X} = \{1, x_1, \ldots, x_n\}$ is the set of n 2-level factors, and $\boldsymbol{\beta} = \{\beta_0, \ldots, \beta_n\}$ is the set with the *intercept* β_0 and the corresponding *model coefficients*. ANOVA tests can rigorously compute the significance of each factor, we can think of that intuitively by noting that less significant factors will have corresponding values in $\boldsymbol{\beta}$ close to zero.

The next example illustrates the screening methodology. Suppose we wish to minimize a performance metric Y of a problem with factors $x_1, \ldots, x_8$ assuming values in $\{-1, -0.8, -0.6, \ldots, 0.6, 0.8, 1\}$. Each $y_i \in Y$ is defined as

$$\begin{aligned} y_i = &-1.5x_1 + 1.3x_3 + 3.1x_5 + \\ &-1.4x_7 + 1.35x_8^2 + 1.6x_3x_5 + \varepsilon. \end{aligned} \tag{2}$$

Suppose that, for the purpose of this example, the computation is done by a very expensive black-box procedure. Note that factors $\{x_2, x_4, x_6\}$ have no contribution to the response, and we can think of the error term ε as representing not only noise, but our uncertainty regarding the model. Higher amplitudes of ε might make isolating factors with low significance harder to justify.

To study this problem efficiently we decided to construct a Plackett-Burman design, which minimizes the experiments needed to identify significant factors. The analysis of this design will enable decreasing the dimension of the problem. Table I presents the Plackett-Burman design we generated. It contains high and low values, chosen to be -1 and 1, for the factors $x_1, \ldots, x_8$, and the observed response $\mathbf{Y}$. It is a required step to add the 3 "dummy" factors $d_1, \ldots, d_3$ to complete the 12 columns needed to construct a Plackett-Burman design for 8 factors [24].

So far, we have performed steps 1, 2, and 3 from Figure 2. We use our initial assumption in Equation (1) to identify the most significant factors by performing an ANOVA test, which is step 5 from Figure 2. The results are shown in Table II, where the *significance* of each factor is interpreted from the F-test and Pr($>$ F) values. Table II uses "$*$", as is convention in the `R` language, to represent the significance values for each factor.

We see on Table II that factors $\{x_3, x_5, x_7, x_8\}$ have at least one "$*$" of significance. For the purpose of this example, this is sufficient reason to include them in our linear model for the next step. We decide as well to discard factors $\{x_2, x_4, x_6\}$

Table I: Randomized Plackett-Burman design for factors $x_1, \ldots, x_8$, using 12 experiments and "dummy" factors $d_1, \ldots, d_3$, and computed response $\mathbf{Y}$

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	d_1	d_2	d_3	Y
1	-1	1	1	1	-1	-1	-1	1	-1	1	13.74
-1	1	-1	1	1	-1	1	1	1	-1	-1	10.19
-1	1	1	-1	1	1	1	-1	-1	-1	1	9.22
1	1	-1	1	1	1	-1	-1	-1	1	-1	7.64
1	1	1	-1	-1	-1	1	-1	1	1	-1	8.63
-1	1	1	1	-1	-1	-1	1	-1	1	1	11.53
-1	-1	-1	1	-1	1	1	-1	1	1	1	2.09
1	1	-1	-1	-1	1	-1	1	1	-1	1	9.02
1	-1	-1	-1	1	-1	1	1	-1	1	1	10.68
1	-1	1	1	-1	1	1	1	-1	-1	-1	11.23
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	5.33
-1	-1	1	-1	1	1	-1	1	1	1	-1	14.79

Table II: Shortened ANOVA table for the fit of the naive model, with significance intervals from the `R` language

	F value	Pr(> F)	Signif.
x_1	8.382	0.063	·
x_2	0.370	0.586	
x_3	80.902	0.003	**
x_4	0.215	0.675	
x_5	46.848	0.006	**
x_6	5.154	0.108	
x_7	13.831	0.034	*
x_8	59.768	0.004	**

from our model, due to their low significance. We see that factor x_1 has a significance mark of "·", but comparing F-test and Pr(> F) values we decide that they are fairly smaller than the values of factors that had no significance, and we keep this factor.

Moving forward to steps 6, 7, and 8 in Figure 2, we will build a linear model using factors $\{x_1, x_3, x_5, x_7, x_8\}$, fit the model using the values of Y we obtained when running our design, and use model coefficients to predict the levels of each factor that minimize the real response. We can do that because these factors are numerical, even though only discrete values are allowed.

We now proceed to the prediction step, where we wish to identify the levels of factors $\{x_1, x_3, x_5, x_7, x_8\}$ that minimize our fitted model, without running any new experiments. This can be done by, for example, using a gradient descent algorithm or finding the point where the derivative of the function given by the linear regression equals to zero.

Table III compares the prediction for Y from our linear model with the selected factors $\{x_1, x_3, x_5, x_7, x_8\}$ with the actual global minimum Y for this problem. Note that factors $\{x_2, x_4, x_6\}$ are included for the global minimum. This happens here because of the error term ε, but could also be interpreted as due to model uncertainty.

Using 12 measurements and a simple linear model, the predicted best value of Y was around $10\times$ larger than the global optimum. Note that the model predicted the correct levels for x_3 and x_5, and almost for x_7. The linear model predicted wrong levels for x_1, perhaps due to this factor's interaction with x_3, and for x_8. Arguably, it would be impossible to predict the correct level for x_8 using this linear model, since a quadratic term composes the true formula of Y. As we showed in Figure 1, a D-Optimal design using a linear model could detect the significance of a quadratic term, but the resulting regression will often lead to the wrong level.

Table III: Comparison of the response Y predicted by the linear model and the true global minimum. Factors used in the model are bolded

	$\boldsymbol{x_1}$	x_2	$\boldsymbol{x_3}$	x_4	$\boldsymbol{x_5}$	x_6	$\boldsymbol{x_7}$	$\boldsymbol{x_8}$	Y
Lin.	-1.0	–	-1.0	–	-1.0	–	1.0	-1.0	-1.046
Min.	1.0	-0.2	-1.0	0.6	-1.0	0.4	0.8	0.0	-9.934

We can improve upon this result if we introduce some information about the problem and use a more flexible design construction technique. Next, we will discuss the construction of efficient designs using problem-specific formulas and continue the optimization of our example.

B. D-Optimal Designs

The application of DoE to autotuning problems requires design construction techniques that support factors of arbitrary types and number of levels. Autotuning problems typically combine factors such as binary flags, integer and floating point numerical values, and unordered enumerations of abstract values. Because Plackett-Burman designs only support 2-level factors, we had to restrict factor levels to interval extremities in our example. We have seen that this restriction makes it difficult to measure the significance of quadratic terms. We will now show how to optimize our example further by using *D-Optimal designs*, which increase the number of levels we can efficiently screen for and enables detecting the significance of more complex terms.

To construct a D-Optimal design it is necessary to choose an initial model, which can be done based on previous experiments or on expert knowledge of the problem. Once a model is selected, algorithmic construction is performed by searching for the set of experiments that minimizes *D-Optimality*, a measure of the *variance* of the *estimators* for the *regression coefficients* associated with the selected model. This search is usually done by swapping experiments from the current candidate design with experiments from a pool of possible experiments, according to certain rules, until some stopping criterion is met. In the example in this section and in our approach we use Fedorov's algorithm [30] for constructing D-Optimal designs, implemented in `R` in the `AlgDesign` package [31].

In our example, suppose that in addition to using our previous screening results we decide to hire an expert in our problem's domain. The expert confirms our initial assumptions that the factor x_1 should be included in our model since it is usually significant for this kind of problem and has a strong interaction with factor x_3. She also mentions we should replace the linear term for x_8 by a quadratic term for this factor.

Using our previous screening and the domain knowledge provided by our expert, we choose a new performance model and use it to construct a D-Optimal design using Fedorov's algorithm. Since we need enough degrees of freedom to fit our model, we construct the design with 12 experiments shown in Table IV. Note that the design includes levels -1, 0, and 1 for factor x_8. The design will sample from different regions of the search space due to the quadratic term, as was shown in Figure 1.

We now fit this model using the results of the experiments in our design. Table V shows the model fit table and compares the estimated and real model coefficients. This example illustrates that the DoE approach can achieve close model estimations using few resources, provided the approach is able to use user

Table IV: D-Optimal design constructed for the factors $\{x_1, x_3, x_5, x_7, x_8\}$ and computed response Y

x_1	x_3	x_5	x_7	x_8	Y
-1.0	-1.0	-1.0	-1.0	-1.0	2.455
-1.0	1.0	1.0	-1.0	-1.0	6.992
1.0	-1.0	-1.0	1.0	-1.0	-7.776
1.0	1.0	1.0	1.0	-1.0	4.163
1.0	1.0	-1.0	-1.0	0.0	0.862
-1.0	1.0	1.0	-1.0	0.0	5.703
1.0	-1.0	-1.0	1.0	0.0	-9.019
-1.0	-1.0	1.0	1.0	0.0	2.653
-1.0	-1.0	-1.0	-1.0	1.0	1.951
1.0	-1.0	1.0	-1.0	1.0	0.446
-1.0	1.0	-1.0	1.0	1.0	-2.383
1.0	1.0	1.0	1.0	1.0	4.423

input to identify significant factors, and knowledge about the problem domain to tweak the model.

Table VI compares the global minimum of this example with the predictions made by our initial linear model from the screening step, and our improved model. Using screening, D-Optimal designs, and domain knowledge, we found an optimization within 10% of the global optimum while computing Y only 24 times. We were able to do that by first reducing the problem's dimension when we eliminated insignificant factors in the screening step. We then constructed a more careful exploration of this new problem subspace, aided by domain knowledge provided by an expert. Note that we could have reused some of the 12 experiments from the previous step to reduce the size of the new design further.

We are able to explain the performance improvements we obtained in each step of the process, because we finish steps with a performance model and a performance prediction. Each factor is included or removed using information obtained in statistical tests, or expert knowledge. If we need to optimize this problem again, for a different architecture or with larger input, we could start exploring the search space with a less naive model. We could also continue the optimization of this problem by exploring levels of factors $\{x_2, x_4, x_6\}$. The significance of these factors could now be detectable by ANOVA tests since the other factors are now fixed. If we still cannot identify any significant factor, it might be advisable to spend the remaining budget using another exploration strategy such as uniform random or latin hypercube sampling.

Table V: Correct model fit comparing real and estimated coefficients, with significance intervals from the `R` language

	Real	Estimated	t value	Pr($>$ |t|)	Signif.
Intercept	0.000	0.050	0.305	0.776	
x_1	-1.500	-1.452	-14.542	0.000	***
x_3	1.300	1.527	15.292	0.000	***
x_5	3.100	2.682	26.857	0.000	***
x_7	-1.400	-1.712	-17.141	0.000	***
x_8	0.000	-0.175	-1.516	0.204	
x_8^2	1.350	1.234	6.180	0.003	**
x_1x_3	1.600	1.879	19.955	0.000	***

Table VI: Comparison of the response Y predicted by our models and the true global minimum. Factors used in the models are bolded

	$\boldsymbol{x_1}$	x_2	$\boldsymbol{x_3}$	x_4	$\boldsymbol{x_5}$	x_6	$\boldsymbol{x_7}$	$\boldsymbol{x_8}$	Y
Quad.	1.0	–	-1.0	–	-1.0	–	1.0	0.0	-9.019
Lin.	-1.0	–	-1.0	–	-1.0	–	1.0	-1.0	-1.046
Min.	1.0	-0.2	-1.0	0.6	-1.0	0.4	0.8	0.0	-9.934

The process of screening for factor significance using ANOVA and fitting a new model using acquired knowledge is equivalent to steps 5, 6, and 7 in Figure 2. In the next section we evaluate the performance of our DoE approach in two scenarios.

V. Performance Evaluation

In this section we present performance evaluations of our approach in two scenarios that differ on search space size and complexity.

A. GPU Laplacian Kernel

We first evaluated the performance of our approach in a Laplacian Kernel implemented using BOAST [10] and targeting the *Nvidia K40c* GPU. The objective was to minimize the *time to compute each pixel* by finding the best level combination for the factors listed in Table VII. Considering only factors and levels, the size of the search space is 1.9×10^5, but removing points that fail at runtime yields a search space of size 2.3×10^4. The complete search space took 154 hours to be evaluated on *Debian Jessie*, using an *Intel Xeon E5-2630v2* CPU, `gcc` version `4.8.3` and *Nvidia* driver version `340.32`.

We applied domain knowledge to construct the following initial performance model:

$$\begin{aligned}\texttt{time\_per\_pixel} \sim\ & \texttt{y\_component\_number} + \frac{1}{\texttt{y\_component\_number}} + \\ & \texttt{load\_overlap} + \texttt{temporary\_size} + \\ & \texttt{vector\_length} + \texttt{lws\_y} + \frac{1}{\texttt{lws\_y}} + \\ & \texttt{elements\_number} + \texttt{threads\_number} + \\ & \frac{1}{\texttt{elements\_number}} + \frac{1}{\texttt{threads\_number}}.\end{aligned} \tag{3}$$

This performance model was used by the Iterative Linear Model (LM) algorithm and by our D-Optimal Design approach (DLMT). LM is almost identical to our approach, described Section III, but it uses a fixed-size random sample of the search space instead of generating D-Optimal designs. We compared the performance of our approach with the following algorithms: uniform Random Sampling (RS); Latin Hypercube Sampling (LHS); Greedy Search (GS); Greedy Search with Restart (GSR); and Genetic Algorithm (GA). Each algorithm performed *at most* 125 measurements over 1000 repetitions, without user intervention.

Since we measured the entire valid search space, we could use the *slowdown* relative to the *global minimum* to compare algorithm performance. Table VIII shows the mean, minimum and maximum slowdowns in comparison to the global

Table VII: Parameters of the Laplacian Kernel

Factor	Levels	Short Description
`vector_length`	$2^0, \ldots, 2^4$	Size of support arrays
`load_overlap`	*true*, *false*	Load overlaps in vectorization
`temporary_size`	$2, 4$	Byte size of temporary data
`elements_number`	$1, \ldots, 24$	Size of equal data splits
`y_component_number`	$1, \ldots, 6$	Loop tile size
`threads_number`	$2^5, \ldots, 2^{10}$	Size of thread groups
`lws_y`	$2^0, \ldots, 2^{10}$	Block size in y dimension

minimum, for each algorithm. It also shows the mean and maximum budget used by each algorithm. Figure 3 presents histograms with the count of the slowdowns found by each of the 1000 repetitions. Arrows point the maximum slowdown found by each algorithm. Note that GS's maximum slowdown was left out of range to help the comparison between the other algorithms.

All algorithms performed relatively well in this kernel, with only GS not being able to find slowdowns smaller than $4\times$ in some runs. As expected, other search algorithms had results similar to RS. LM was able to find slowdowns close to the global minimum on most runs, but some runs could not find slowdowns smaller than $4\times$. Our approach reached a slowdown of 1% from the global minimum *in all* of the 1000 runs while using *at most* fewer than half of the allotted budget.

We implemented a simple approach for the prediction step in this problem, choosing the best value of our fitted models on the complete set of valid level combinations. This was possible for this problem since all valid combinations were known. For problems were the search space is too large to be generated, we would have to either adapt this step and run the prediction on a sample or minimize the model using the differentiation strategies mentioned in Section IV-A.

This kernel provided ideal conditions for using our approach, where the performance model is approximately known and the complete valid search space is small enough to be used for prediction. The global minimum also appears to not be isolated in a region of points with bad performance, since our approach was able to exploit space geometry. We will now present a performance evaluation of our approach in a larger and more comprehensive benchmark.

B. SPAPT Benchmark Suite

The SPAPT [8] benchmark suite provides parametrized CPU kernels from different HPC domains. The kernels shown in Table IX are implemented using the code annotation and transformation tools provided by Orio [9]. Search space sizes are larger than in the Laplacian Kernel example. Kernel factors are either integers in an interval, such as loop unrolling and register tiling amounts, or binary flags that control parallelization and vectorization.

Table VIII: Slowdown and budget used by 7 optimization methods on the Laplacian Kernel, using a budget of 125 points with 1000 repetitions

	Mean	Min.	Max.	Mean Budget	Max. Budget
RS	1.10	1.00	1.39	120.00	120.00
LHS	1.17	1.00	1.52	98.92	125.00
GS	6.46	1.00	124.76	22.17	106.00
GSR	1.23	1.00	3.16	120.00	120.00
GA	1.12	1.00	1.65	120.00	120.00
LM	1.02	1.01	3.77	119.00	119.00
DLMT	1.01	1.01	1.01	54.84	56.00

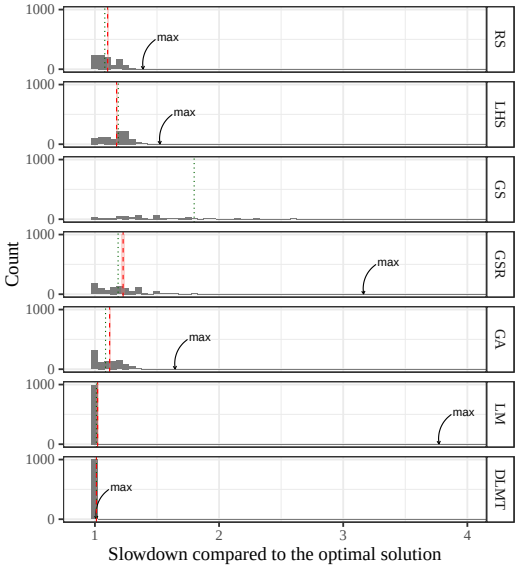

Figure 3: Distribution of slowdowns in relation to the global minimum for 7 optimization methods on the Laplacian Kernel, using a budget of 125 points over 1000 repetitions

We used the Random Sampling (RS) implementation available in Orio and integrated an implementation of our approach (DLMT) to the system. We omitted the other Orio algorithms because other studies using SPAPT kernels [3], [4] showed that their performance is similar to RS regarding budget usage. The global minima are not known for any of the problems, and search spaces are too large to allow complete measurements. Therefore, we used the performance of each application compiled with `gcc`'s `-O3`, with no code transformations, as a *baseline* for computing the *speedups* achieved by each strategy. We performed 10 autotuning repetitions for each kernel using RS and DLMT, using a budget of *at most* 400 measurements. DLMT was allowed to perform only 4 of the iterations shown in Figure 2. Experiments were performed using Grid5000 [32], on *Debian Jessie*, using an *Intel Xeon E5-2630v3* CPU and `gcc` version `6.3.0`.

The time to measure each kernel varied from a few seconds to up to 20 minutes. In testing, some transformations caused the compiler to enter an internal optimization process that did not stop for over 12 hours. We did not study why these cases delayed for so long, and implemented an execution timeout of 20 minutes, considering cases that took longer than that to compile to be runtime failures.

Similar to the previous example, we automated factor elimination based on ANOVA tests so that a comprehensive

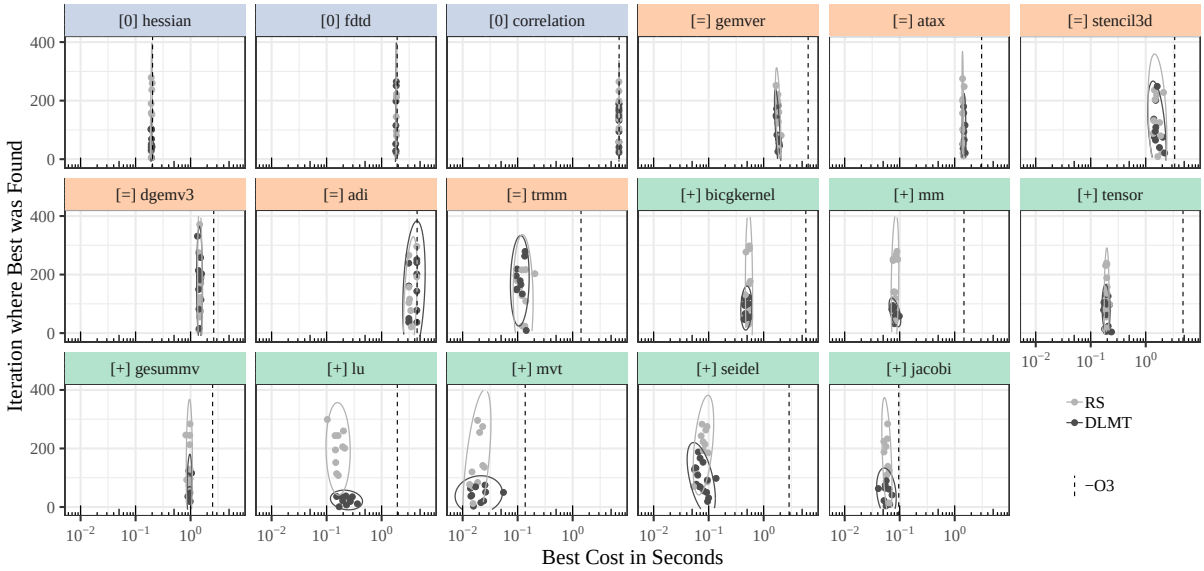

Figure 4: Cost of best points found on each run, and the iteration where they were found. RS and DLMT found no speedups with similar budgets for kernels marked with "[0]" and *blue* headers, and similar speedups with similar budgets for kernels marked with "[=]" and *orange* headers. DLMT found similar speedups using smaller budgets for kernels marked with "[+]" *green* headers. Ellipses delimit an estimate of where 95% of the underlying distribution lies

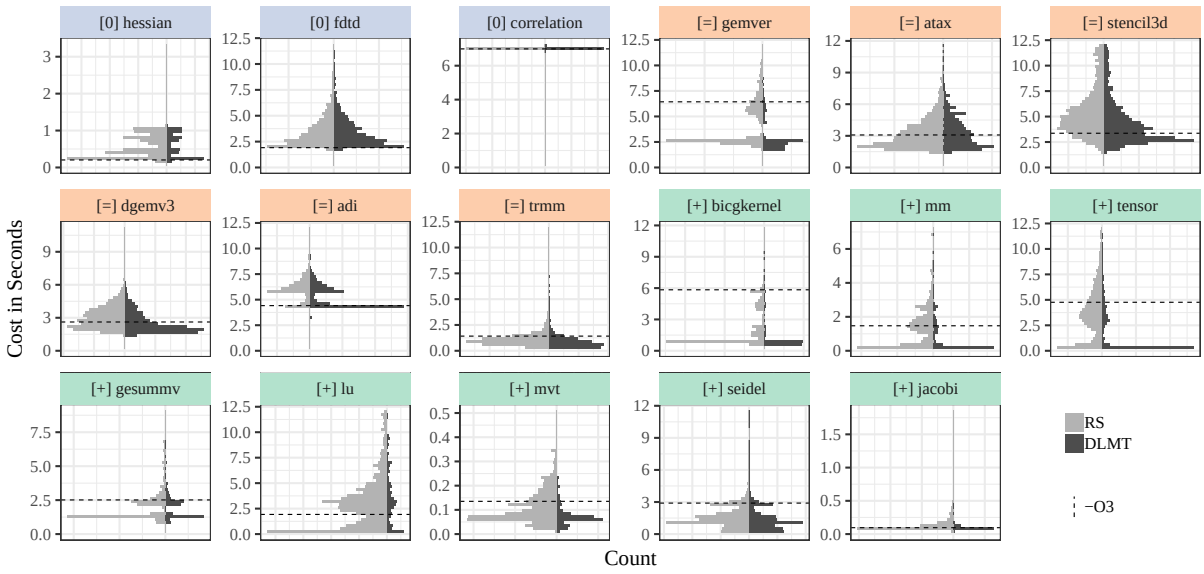

Figure 5: Histograms of explored search spaces, showing the real count of measured configurations. Kernels are grouped in the same way as in Figure 4. DLMT spent fewer measurements than RS in configurations with smaller speedups or with slowdowns, even for kernels in the orange group. DLMT also spent more time exploring configurations with larger speedups

Table IX: Kernels from the SPAPT benchmark used in this evaluation

Kernel	Operation	Factors	Size
atax	Matrix transp. & vector mult.	18	2.6×10^{16}
dgemv3	Scalar, vector & matrix mult.	49	3.8×10^{36}
gemver	Vector mult. & matrix add.	24	2.6×10^{22}
gesummv	Scalar, vector, & matrix mult.	11	5.3×10^{9}
hessian	Hessian computation	9	3.7×10^{7}
mm	Matrix multiplication	13	1.2×10^{12}
mvt	Matrix vector product & transp.	12	1.1×10^{9}
tensor	Tensor matrix mult.	20	1.2×10^{19}
trmm	Triangular matrix operations	25	3.7×10^{23}
bicg	Subkernel of BiCGStab	13	3.2×10^{11}
lu	LU decomposition	14	9.6×10^{12}
adi	Matrix sub., mult., & div.	20	6.0×10^{15}
jacobi	1-D Jacobi computation	11	5.3×10^{9}
seidel	Matrix factorization	15	1.3×10^{14}
stencil3d	3-D stencil computation	29	9.7×10^{27}
correlation	Correlation computation	21	4.5×10^{17}

evaluation could be performed. We also did not tailor initial performance models, which were the same for all kernels. Initial models had a linear term for each factor with two or more levels, plus quadratic and cubic terms for factors with sufficient levels. Although automation and identical initial models might have limited the improvements at each step of our application, our results show that it still succeeded in decreasing the budget needed to find significant speedups for some kernels.

Figure 4 presents the *speedup* found by each run of RS and DLMT, plotted against the algorithm *iteration* where that speedup was found. We divided the kernels into 3 groups according to the results. The group where no algorithm found any speedups contains 3 kernels and is marked with "[0]" and *blue* headers. The group where both algorithms found similar speedups, in similar iterations, contains 6 kernels and is marked with "[=]" and *orange* headers. The group where DLMT found similar speedups using a significantly smaller budget than RS contains 8 kernels and is marked with "[+]" and *green* headers. Ellipses delimit an estimate of where 95% of the underlying distribution lies, and a dashed line marks the `-O3` baseline. In comparison to RS, our approach significantly decreased the average number of iterations needed to find speedups for the 8 kernels in the green group.

Figure 5 shows the search space exploration performed by RS and DLMT. It uses the same color groups as Figure 4, and shows the distribution of the speedups that were found during all repetitions of the experiments. Histogram areas corresponding to DLMT are usually smaller because it always stopped at 4 iterations, while RS always performed 400 measurements. This is particularly visible in `lu`, `mvt`, and `jacobi`. We also observe that the quantity of configurations with high speedups found by DLMT is higher, even for kernels on the orange group. This is noticeable in `gemver`, `bicgkernel`, `mm` and `tensor`, and means that DLMT spent less of the budget exploring configurations with small speedups or slowdowns, in comparison with RS.

Analyzing the significant performance parameters identified by our automated approach for every kernel, we were able to identify interesting relationships between parameters and performance. In `bicgkernel`, for example, DLTM identified a linear relationship for OpenMP and scalar replacement optimizations, and quadratic relationships between register and cache tiling, and loop unrolling. This is an example of the transparency in the optimization process that can be achieved with a DoE approach.

Our approach used a generic initial performance model for all kernels, but since it iteratively eliminates factors and model terms based on ANOVA tests, it was still able to exploit global search space structures for kernels in the orange and green groups. Even in this automated setting, the results with SPAPT kernels illustrate the ability our approach has to reduce the budget needed to find good speedups by efficiently exploring search spaces.

VI. Conclusion

The contribution of this paper is a transparent DoE approach for program autotuning under tight budget constraints. We discussed the underlying concepts that enable our approach to reduce significantly the measurement budget needed to find good optimizations consistently over different kernels exposing configuration parameters of source-to-source transformations. We have made efforts to make our results, figures and analyses reproducible by hosting all our scripts and data publicly [33].

Our approach outperformed six other search heuristics, always finding a slowdown of 1% from the global optimum of the search space defined by the optimization of a Laplacian kernel for GPUs, while using at most half of the allotted budget. In a more comprehensive evaluation, using kernels from the SPAPT benchmark, our approach was able to find the same speedups as RS while using up to $10\times$ fewer measurements. We showed that our approach explored search spaces more efficiently, even for kernels where it performed similarly to random sampling.

We presented a completely automated version of our approach in this paper so that we could perform a thorough performance evaluation on comprehensive benchmarks. Despite using the same generic performance model for all kernels, our approach was able to find good speedups by eliminating insignificant model terms at each iteration. This means that our approach can still improve the performance of applications using unspecialized models that incorporate only general knowledge about algorithm performance. We would incur some budget overhead in this case while insignificant terms are removed.

In future work we will explore the impact of user input and expert knowledge in the selection of the initial performance model and in the subsequent elimination of factors using ANOVA tests. We expect that tailored initial performance models and assisted factor elimination will improve the solutions found by our approach and decrease the budget needed to find them.

Our current strategy eliminates completely from the model the factors with low significance detected by ANOVA tests. In future work we will also explore the effect of adding random experiments with randomized factor levels. We expect this will decrease the impact of removing factors wrongly detected to have low significance.

Decreasing the number of experiments needed to find optimizations is a desirable property for autotuners in problem domains other than source-to-source transformation. We intend to evaluate the performance of our approach in domains such as High-Level Synthesis and compiler configuration for FPGAs, where search spaces can get as large as 10^{126}, and where we already have some experience [34].

ACKNOWLEDGMENT

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations. This work was partly funded by CAPES, *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior*, Brazil, funding code 001.

REFERENCES

[1] K. Seymour, H. You, and J. Dongarra, "A comparison of search heuristics for empirical code optimization." in *CLUSTER*, 2008, pp. 421–429.

[2] P. M. Knijnenburg, T. Kisuki, and M. F. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," *The Journal of Supercomputing*, vol. 24, no. 1, pp. 43–67, 2003.

[3] P. Balaprakash, S. M. Wild, and P. D. Hovland, "Can search algorithms save large-scale automatic performance tuning?" in *ICCS*, 2011, pp. 2136–2145.

[4] ——, "An experimental study of global and local search algorithms in empirical performance tuning," in *International Conference on High Performance Computing for Computational Science*. Springer, 2012, pp. 261–269.

[5] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, "Apollo: Reusable models for fast, dynamic tuning of input-dependent code," in *Parallel & Distributed Processing, 2017. IPDPS 2017. IEEE International Symposium on*. IEEE, 2017, pp. 307–316.

[6] T. L. Falch and A. C. Elster, "Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 8, 2017.

[7] P. Balaprakash, A. Tiwari, S. M. Wild, and P. D. Hovland, "AutoMOMML: Automatic Multi-objective Modeling with Machine Learning," in *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, M. J. Kunkel, P. Balaji, and J. Dongarra, Eds. Springer International Publishing, 2016, pp. 219–239.

[8] P. Balaprakash, S. M. Wild, and B. Norris, "SPAPT: Search problems in automatic performance tuning," *Procedia Computer Science*, vol. 9, pp. 1959–1968, 2012.

[9] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using Orio," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–11.

[10] B. Videau, K. Pouget, L. Genovese, T. Deutsch, D. Komatitsch, F. Desprez, and J.-F. Méhaut, "BOAST: A metaprogramming framework to produce portable and efficient computing kernels for hpc applications," *The International Journal of High Performance Computing Applications*, p. 1094342017718068, 2017.

[11] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[12] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "POET: Parameterized optimizations for empirical tuning," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–8.

[13] 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.

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

[15] 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.

[16] J. J. Dongarra and C. R. Whaley, "Automatically tuned linear algebra software (ATLAS)," *Proceedings of SC*, vol. 98, 1998.

[17] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[18] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 1381–1384.

[19] M. Gerndt and M. Ott, "Automatic performance analysis with Periscope," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 736–748, 2010.

[20] 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.

[21] 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.

[22] 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 Techniques*. ACM, 2014, pp. 303–316.

[23] R. Carnell, *lhs: Latin Hypercube Samples*, 2018, R package version 0.16. [Online]. Available: https://CRAN.R-project.org/package=lhs

[24] R. L. Plackett and J. P. Burman, "The design of optimum multifactorial experiments," *Biometrika*, vol. 33, no. 4, pp. 305–325, 1946.

[25] S. Addelman and O. Kempthorne, "Some main-effect plans and orthogonal arrays of strength two," *The Annals of Mathematical Statistics*, pp. 1167–1176, 1961.

[26] U. Grömping and R. Fontana, "An algorithm for generating good mixed level factorial designs," Beuth University of Applied Sciences, Berlin, Tech. Rep., 2018.

[27] J. Fox and S. Weisberg, *An R Companion to Applied Regression*, 2nd ed. Thousand Oaks CA: Sage, 2011. [Online]. Available: http://socserv.socsci.mcmaster.ca/jfox/Books/Companion

[28] U. Grömping, "R package FrF2 for creating and analyzing fractional factorial 2-level designs," *Journal of Statistical Software*, vol. 56, no. 1, pp. 1–56, 2014. [Online]. Available: http://www.jstatsoft.org/v56/i01/

[29] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2018. [Online]. Available: https://www.R-project.org/

[30] V. V. Fedorov, *Theory of optimal experiments*. Elsevier, 1972.

[31] B. Wheeler, *AlgDesign: Algorithmic Experimental Design*, 2014, R package version 1.1-7.3. [Online]. Available: https://CRAN.R-project.org/package=AlgDesign

[32] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[33] P. Bruel, "Git repository with all scripts and data," https://github.com/phrb/ccgrid19, accessed: 2018-10-14.

[34] P. Bruel, A. Goldman, S. R. Chalamalasetti, and D. Milojicic, "Auto-tuning high-level synthesis for FPGAs using OpenTuner and LegUp," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2017.