Dash Database: Structured Kernel Data for the Machine Understanding of Computation

by

Benjamin R. Willis

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2020 by the
Graduate Supervisory Committee:

John Brunhaver, Chair
Chaitali Chakrabarti
Aviral Shrivastava

ARIZONA STATE UNIVERSITY

December 2020

ABSTRACT

As device and voltage scaling cease, ever-increasing performance targets can only be achieved through the design of parallel, heterogeneous architectures. The workloads targeted by these domain-specific architectures must be designed to leverage the strengths of the platform: a task that has proven to be extremely difficult and expensive. Machine learning has the potential to automate this process by understanding the features of computation that optimize device utilization and throughput. Unfortunately, applications of this technique have utilized small data-sets and specific feature extraction, limiting the impact of their contributions.

To address this problem I present Dash-Database; a repository of C and C++ programs for software-defined radio applications and its neighboring fields; a methodology for structuring the features of computation using kernels, and a set of evaluation metrics to standardize computation data sets. Dash-Database contributes a general data set that supports machine understanding of computation and standardizes the input corpus utilized for machine learning of computation; currently only a small set of benchmarks and features are being used. I present an evaluation of Dash-Database using three novel metrics: breadth, depth and richness; and compare its results to a data set largely representative of those used in prior work, indicating a 5x increase in breadth, 40x increase in depth, and a rich set of sample features. Using Dash-Database, the broader community can work toward a general machine understanding of computation that can automate the design of workloads for domain-specific computation.

ACKNOWLEDGMENTS

I first want to thank my advisor, Dr. John S. Brunhaver, for putting his time and effort into me and my work. Without him I would not be where I am today, both professionally and personally. I would also like to thank my parents, Dr. Wayne Willis and Patty Jones, for supporting me in more ways than one throughout my graduate term.

I would like to thank Richard Uhrie for proposing the ideas behind and building the tools for TraceAtlas. I would also like to thank Mukul Gupta, Vamsikrishna Lanka, Sriharsha Uppu, Connor Harris, Jacob Holtom, Alex Chiriyath, Yukang Fu, Christian Chapman, Owen Ma, Sharanya Srinivas, Matthew Kinsinger, Saquib Siddiqui, Yang Li, Anish Nallan, Hanguang Yu, Shun Yao, and Sean Bryan for writing applications and developing build flows within Dash-Corpus.

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7860. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusion contained herein are those of the authors and should not be interpreted as necessarily representing the social policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

TABLE OF CONTENTS

Page

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER

1 INTRODUCTION 1

2 BACKGROUND 4

2.1 Machine Learning on Computation 4

2.1.1 Machine Learning of Compilation 4

2.1.2 Machine Learning of Scheduling 6

2.1.3 Machine Learning of Programs 6

2.1.4 Machine Learning of Performance Prediction 7

2.1.5 Machine Learning of Design Space Exploration 7

2.2 Data Sets of Neighboring Fields 8

2.3 Kernels for Structuring Computation 9

3 CORPUS 11

3.1 Program Selection 11

3.2 Building for Depth 13

3.3 Data Labels 14

4 TRACEATLAS 17

4.1 Kernel Definition 17

4.2 Compilation and Tracing 18

4.3 Kernel Extraction 19

4.4 Type 1 Kernels 19

4.4.1 Type 2 Kernels 20

4.4.2 Type 3 and 4 Kernels 21

CHAPTER Page

4.5 Kernel Labeling 22

4.6 Structured Data 24

5 CORPUS PROCESSING 25

5.1 Dash-Automate 25

6 EVALUATION METRICS 28

6.1 Sample Transformation 28

6.2 Breadth 29

6.3 Depth 30

6.4 Richness 30

7 CASE STUDIES 33

7.1 Test Data Set 33

7.2 Breadth and Depth 34

7.3 Richness 35

7.4 Mutual Information and Supervised Learning 37

8 CONCLUSION 40

REFERENCES 41

LIST OF TABLES

Table Page

3.1 Dash-Corpus Directory Description 12

3.2 Corpus Labels 15

3.3 Cross Product Table 16

7.1 Test Corpus 34

7.2 Breadth 35

7.3 Linear Regression | Logistic Regression | Naive Bayes | SVM 39

LIST OF FIGURES

Figure Page

2.1 Fields of Computer Systems That Have Used Machine Learning. 5

4.1 TraceAtlas Processing Pipeline 17

4.2 Diagram of the Kernels Extracted from the BubbleSort Program in Figure 4.1. 23

5.1 Dash-Automate Processing Diagram 26

7.1 Change as the Number of PCA Components Are Increased. 36

7.2 Change as the Number of Univariate Features Increases. 37

Chapter 1

INTRODUCTION

As Moore's Law slows, domain-specific architectures are the only performant paths forward for continued performance scaling [1]. Even as higher-performance technology shows promise for the near future, Dennard Scaling has ended [2], leading computer architects with little choice but to better exploit technology that is already available. This is being done by utilizing fixed-function hardware called accelerators [3]. While this appears to be an attractive method [4], the high risk of designing such hardware is forcing chip designers towards more configurability, a characteristic that generally hurts performance.

Building and designing domain-specific systems-on-chip (DSSOC) has proven to be extremely difficult [5]. Specialized hardware tailored specifically for a given task requires low-level understanding and specialization. Furthermore, designing custom hardware designs is highly labor-intensive and carries substantial fabrication risks.

Given such a risky and complicated process, designing DSSOCs for future workloads will be difficult. Emerging workloads show no sign of becoming simpler, in fact they are likely to be more complex [6], hence the challenge won't go away by itself. Hardware-software co-designers have attempted to meet this challenge by factoring code from the wild into their essential parts, optimizing them, and designing an architecture tailored to their needs [7]. But only domain experts can do this factorization [8], a characteristic that makes this design process expensive and time-consuming. If hardware-software co-design will bring about performance gains via domain-specific architectures, automating this process will be paramount going forward.

Machine learning is an obvious candidate to tackle this challenge. Using a small corpus of computer programs, prior studies have aimed to understand the features of their corpus that solve a computational problem about compilers, scheduling, characterization, power and performance, and design space exploration. However, these studies have lacked scalability by constraining their input corpus and feature selection. Often, these experiments have utilized a handful of benchmark programs and hand-picked features. This limits the amount of knowledge that can be learned by statistical modeling. For machine learning to solve the automation challenge for the broader community, a large corpus of computer programs whose data is structured to represent features from a broad range of computation domains is needed.

The benefits of broad data sets applied to general engineering problems have a history of success. Neighboring fields of research have exploited large data sets to benefit themselves as a whole [9, 10, 11]. By applying machine learning models to a generalized input data set, contributions within these fileds have solved problems for the broader community [12]. Their results were both reproducible and robust to changes in the input domain, thus achieving scalability. Once these input data sets were widely adopted, competitions and community participation fostered unprecedented growth in the research fields involved [13].

To provide a general data set for the machine understanding of computation, I present Dash-Database. Dash-Database has three parts: a large corpus of computer programs, a methodology of structuring the information within these programs, and a data-set of structured computation data suitable for machine learning. I built Dash-Corpus to represent the workloads found most commonly on domain-specific processors for software-defined radio and its neighboring fields: linear algebra, signal and image processing, and cryptography. I utilize TraceAtlas [14], a toolchain for dynamic tracing and feature extraction of computer programs using kernels, to

structure the information in Dash-Corpus. Finally, I evaluate the utility of my data set by proposing three novel metrics: breadth, depth and richness. I show the advantages Dash-Databases has over a corpus of benchmarks that represent those from prior work, and improvements using the three proposed metrics.

Dash-Database contributes three things:

1. A corpus of computer programs that includes applications with parallel kernels from a broad context.

2. A methodology for transforming a corpus of source code into a quantitative data set suitable for machine understanding using TraceAtlas.

3. A set of evaluation metrics for the breadth, depth, richness and utility of a data set for computation.

The rest of the paper is structured as follows: section two gives a background of how prior contributions to computer systems have exploited machine learning for software optimization; section three presents Dash-Corpus, a broad, deep and rich repository of source code aimed at parallel computation; section four described a set of techniques to turn a corpus of source code into a data set suitable for machine learning; section five presents a set of evaluation metrics to show the breadth, depth, richness and utility of a data set; section six presents a set of case studies to show the utility of the data set for machine learning and present potential future work using the

Chapter 2

BACKGROUND

2.1 Machine Learning on Computation

Computer systems research has applied machine learning to five categories: compilation, scheduling, program characterization, performance prediction and design space exploration. These works have two things in common. They use a small corpus of programs, typically from benchmarks, and they define a unique set of features that are used to structure their corpus into a statistical form that serves as the data set for model training and testing. Seldom do these two critical steps of machine learning contributions have any overlap between contributions, making the results of these studies difficult to reproduce. Figure 2.1 shows five different fields of computer systems that exploit machine learning for design research.

2.1.1 Machine Learning of Compilation

Compiler researchers use machine learning for optimization and code transformations, from compiler flags to compilation for heterogeneous architectures. Static program optimization was initially done by permuting the space of legal transformations until an optimal configuration is found, like that proposed by Sharma and Aiken, which used Markov chain Monte Carlo [15] to optimize loop-free assembly programs. They used a benchmark of 25 programs proposed by [16]. Without a restriction on loops in the input programs, the design space can become too large to practically permute in a reasonable timeframe. To solve this, compiler developers utilize direct-prediction models. Stephenson and Amarasinghe proposed a supervised model that

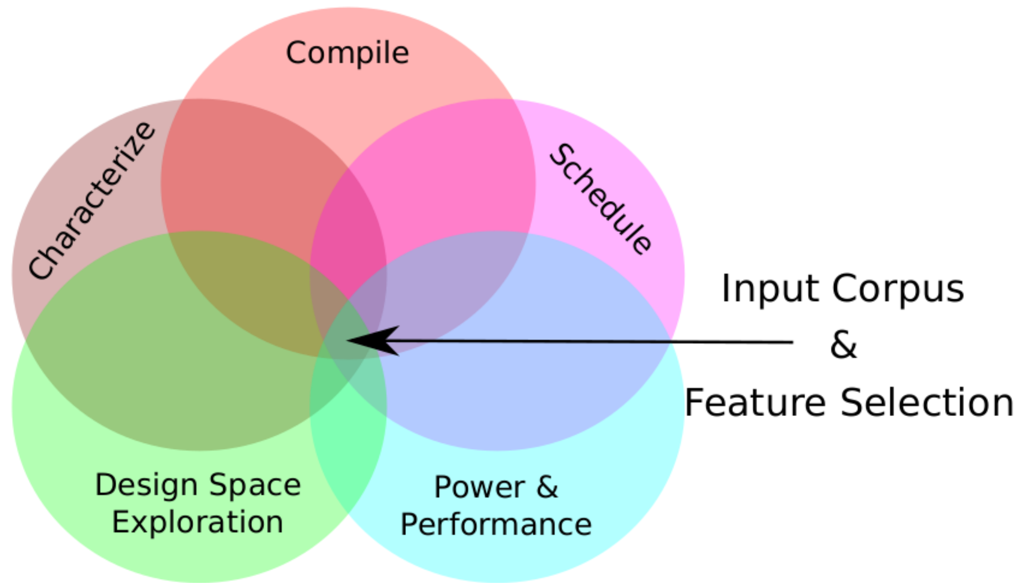


Figure 2.1: Fields of Computer Systems That Have Used Machine Learning.

directly predicted the loop unrolling factors of 2,500 loops from 72 different benchmarks [17]. Grewe, Wang and O'Boyle [18] proposed a compiler that automatically factored OpenMP programs into data-parallel GPU applications using OpenCL and a decision tree model. They trained their model with 47 parallel kernels extracted from 4 different benchmarks and tested it with the NAS benchmark suite. The ARIA compiler and runtime framework [19] optimizes the performance of parallel compute kernels written in OpenMP on heterogeneous systems. They used an input corpus of 12 programs from the Rodinia and Parboil benchmark suites. For parallel compilation, Magni, Dubach and O'Boyle developed a machine learning model that predicts the optimal level of thread coarsening for GPU workloads [20]. They used 17 OpenCL programs to train a cascade of neural networks that accepted a feature vector of static

program characteristics, and showed speedups in most of the evaluations. None of these works had any overlap in the programs constituting their input corpus.

2.1.2 Machine Learning of Scheduling

Gupta, Patil, Bhat, Mishra and Ogras contributed DyPO [21], a Pareto-optimal dynamic scheduling model for a heterogeneous system using logistic regression and an input corpus of 18 programs from the MI-Bench, Cortex and PARSEC benchmark suites, structuring the data using performance monitor counts. Adams, Ma, Anderson, Baghdadi, Li, Gharbi, Steiner, Johnson, Fatahalian, Durand and Ragan-Kelley created an optimal Halide scheduler [22] using a decision tree and logistic regression. They chose their features for the model by evaluating the granularity and tilings of each pipe stage in a typical Halide processing pipeline to create an optimal configuration. They used randomly generated Halide schedules as the training set, and a set of Halide benchmarks [23]. These two works do not overlap in their input corpus or feature set.

2.1.3 Machine Learning of Programs

Hashimoto, Terai, Maeda and Minami trained a supervised learning classifier [24] recognize Fortran loops as kernels using over 2000 Fortran loops with human labels and 6 defined computation parameters. Stephenson and Amarasinghe [17] created a regression model to predict the unroll factor of novel loops from 72 programs belonging to the SPEC 2000 benchmark suite and 38 features. Liu, Lin and Chen proposed a method called the Ensemble Workload Prediction and an input feature called the Cloud Workload Correction Rate to predict the workloads of server-based cloud computing services at Google [25]. The study used performance data from over

690,000 data points and 7 features about the timing and resource usage of each job. No overlap in input corpus or feature selection.

2.1.4 Machine Learning of Performance Prediction

Many works have applied machine learning to power and performance prediction of computer programs [26, 27, 28, 29, 30, 31]. These models generally use performance monitor data across architectures to train a classifier. Interestingly, these works frequently overlap in the features selected for model training and testing, but their impact has been limited by simulations as their input data.

2.1.5 Machine Learning of Design Space Exploration

Hardware-software co-design researchers attempt to solve the complexity problem in their design spaces through the use of supervised learning models. Azizi, Marhesri, Stevenson, Patel and Horowitz created a machine learning model to characterize the trade-offs between circuit and architectural design choices [32]. Their test corpus was the SPECint2000 benchmark, using which they generated 500 architectural simulations per program. Liu and Carloni presented several supervised learning algorithms and transductive experimental design [33] to learn the design space of target programs from high-level synthesis (HLS). Ipek, Mckee, Supinski, Caruana and Schulz used samples from architectural simulations of the SPEC2000 benchmark to train an artificial neural network to predict an optimal memory hierarchy for a target application [34]. Here I see another overlap in the input corpus of programs in [32, 34].

Out of all the work cited above, only four of them have any overlap in their input corpus and feature selection. While this may seem like a general feature of machine learning contributions, there was no clear justification for the input corpus programs or the features used. As Wang and O'Boyle pointed out in their study

of machine learning applied to compilation [35], the lack of abundant, high-quality training data, and local feature optimization are the most immediate limitations to machine learning models for compilation. While their contribution focused on compilers, the above study shows a general proliferation of this limitation in all facets of computer systems.

2.2 Data Sets of Neighboring Fields

Neighboring research fields have been role models for computer systems when applying machine learning to research questions. Two research fields have particularly interesting histories: computational linguistics and image processing. After large data sets with sufficient structure, annotations and open-source communities were introduced, machine learning algorithms trained on their data led to massive contributions in their respective fields. Participants contributed methods to structure the input data into statistical features that became widely adopted and accepted for their success.

WordNet [11] is an English lexical data set of over 100,000 words and 200,000 word-sense pairs. After its introduction in 1995, Finkelstein, Gabrilovich , Matias, Rivlin, Solan, Wolfman and Ruppin proposed IntelliZap, a novel framework for interpreting text for search queries [36]. IntelliZap went on to become a "gold standard" [37] in computational linguistics, and was widely adopted by internet search engines to better interpret the searches of users. The combination of a large input data set and a methodology for structuring its data, though they did not come at the same time, led to contributions that benefitted the entire research field.

Imagenet [9], inspired by WordNet, is a data set of over 14 million images and over 1,000 image classifications, about 140,000 images per category. After being introduced in 2009, its creators started the ImageNet Image Recognition Challenge

[38], an annual competition to see who could build the world's best image classifier. In 2012, Krizhevky, Sutskever and Hinton won the competition with a convolutional neural network [12]. Their contribution revolutionized the image classification field by structuring the input data features using convolutional filters. Just like WordNet and IntelliZap, ImageNet and "AlexNet" introduced a large input data set and a method for structuring the input data that became widely utilized in the entire field, leading to a machine understanding of images that today beats even human classifiers [38].

If computer systems is to achieve the level of success brought on by data sets and data structuring for the machine learning of computational linguistics and image classification, a large input data set and a methodology for structuring the data is necessary.

2.3 Kernels for Structuring Computation

Kernels are an excellent candidate for structuring computational data. Asonovic, [39] first proposed a set of 13 kernel archetypes that describe every type of computation. These kernel archetypes were supposed to formally define the challenges that faced architects and program designers in an age of chip multiprocessors and the power wall. Image processing pipelines and graphics processing designers have contributed their interpretations of kernels [40, 41, 42]. TACO also uses kernels to optimize sparse matrix operations [43]. All of these works define kernels according to stages within a processing pipeline, commonly referred to as computation kernels.

Kernels are the most important parts of programs to optimize [14]. They represent the parts of a program that account for a large percent of the runtime. Structuring programs by the kernels they contain was also done by ARIA [19] for mapping parts of an input program to the optimal module on a heterogeneous architecture. Collecting characteristics about these kernels leads to a structuring of the computation in a

statistical form, allowing the characteristics of the kernel to be learned by statistical models.

Chapter 3

CORPUS

Dash-Corpus is a collection of programs from the application domain of software-defined radio. Here, a computational domain refers to a field of computer systems that focuses on applications to a specific problem, e.g. SDR, finite element analysis, climate modeling, web browsing to name a few. The computational domain defines an arena of algorithms that are applied to problems within an application of the domain. These applications and their algorithms ultimately define the information that can be learned from a corpus of source code.

3.1 Program Selection

Dash-Corpus is focused on C and C++ libraries for software-defined radio applications and its neighboring domains. Table 3.1 summarizes the libraries and projects in Dash-Corpus. Each directory represents a library, research project or application targeting specific kernel archetypes. Each category, image signal processing (ISP), digital signal processing (DSP), linear algebra (LA) and cryptography (crypt) represent neighboring computational domains that each project in Dash-Corpus contains.

I felt that building the corpus using these libraries would thoroughly perturb the parallel kernel computing space. SDR applications have ever-increasing energy and performance metrics, forcing architects to exploit heterogeneous, parallel architectures [66]. The movement toward handsets, wearable electronic devices and autonomous machines has driven the adoption of this computational domain to devices for both consumers and industry. SDR enables these devices to be connected to the internet and other communication networks, use global positioning systems and even

Table 3.1: Dash-Corpus Directory Description

Directory	ISP	DSP	LA	Crypt
Artisan		x		
Armadillo [44]	x	x	x	
BLAS [45]			x	
Benchmarks [46, 47, 48]		x		
CortexSuite [49]	x	x	x	
Dash-RadioCorpus	x	x	x	x
Eigen [50]			x	
FEC [51]		x		x
FFmpeg [52]		x		x
FFTW [53]		x		
GSL [54]		x	x	
Halide [55]	x	x	x	
Kestrel		x	x	x
LiquidSDR [56]		x	x	x
mbed_TLS [57]				x
MiBench [58]		x	x	x
OpenCV [59]	x		x	
PERFECT [60]	x	x	x	
SDH			x	
SHOC [61]		x	x	
SigPack [62]	x	x	x	
SPUCE [63]		x		
Streamit [64]		x	x	
VOLK [65]			x	

radars. Libraries like SigPack [62], LiquidSDR [56], and srsLTE [67] are built for these purposes. Also present on these devices is the need for visual and audio features, including music, image capture and display. Architects of these devices utilize accelerators, application-specific integrated circuits, and other dedicated hardware to meet strict power and performance constraints for these workloads using algorithms from libraries like OpenCV [59], Forward Error Correcting [51], FFmpeg [52] and others.

Dash-Corpus was also designed to include programs from the input data-sets of prior work. CortexSuite [49], Halide benchmarks [23], SHOC [68], and Streamit [64] have all been used as the input corpus of prior applications of machine learning to computation.

Dash-Corpus was built with a focus on C and C++ applications and libraries. High-performance libraries for software-defined radio and its neighboring fields are frequently available in C/C++. There is a significant portion of libraries that are written in Fortran, but a lack of Fortran competency among the application designers effectively eliminated the use of this language in the corpus. My processing pipeline relies on code that can be compiled with LLVM [69]. While a variety of languages including Fortran are supported by LLVM, I found the representation of high-performance parallels in C and C++ libraries sufficient to justify utilizing only those two languages.

3.2 Building for Depth

Applications driving libraries in Dash-Corpus often have different configurations. These parameters can include API call parameters, data type and precision, and data size. Changing the parameters of an API call can have a significant impact on the implementation of the algorithm being called. Thus, I captured as many

configurations as possible from computer programs in the corpus. Included in several of the driver and application programs of Dash-Corpus are build flows that sweep these parameters, both at compile-time and runtime, providing multiple flavors of each algorithm.

3.3 Data Labels

Dash-Corpus includes a labeling mechanism for source code sections that are likely to define a parallel kernel. Table 3.2 describes these labels. They were selected for their prevalence to SDR applications, including kernels that can be found in its neighboring three fields. In essence, these labels are descriptions of algorithms, and the kernels that may contain these labels are quantitative descriptions of the computational implementation of this algorithm.

By labeling the algorithms that are deemed important in Dash-Corpus, the resulting data contains user annotations. User annotations are critical parts of any data set, and are often the most difficult aspects of large data sets, requiring crowd-sourcing efforts in many cases [9, 10]. Labels serve to support supervised learning algorithms, sample categorization and a representation of ground truth. The implementation and capturing of these labels are described in section 4.

The entries in table 3.2 represent general names that can have a broad range of implementations. To better specify what a kernel is doing, I add modifiers to each label. These are pieces of metadata that specify what configuration the kernel had when it was executing. For example, if the kernel was given the label GEMM, the user may attach the metadata to the label to include the dimensions of the matrices, the precision of the data, and whether the data was real or complex. These pieces of metadata can be useful later for when kernels are mapped to fixed-function hardware.

Table 3.2: Corpus Labels

Abbr.	Label	Description
	FFT	Fast Fourier Transform.
	GEMM	General Matrix-Matrix Multiply
	GEMV	General Matrix-Vector Multiply
	SPMM	Sparse Matrix-Matrix Multiply
	SPMV	Sparse Matrix-Vector Multiply
	ZIP	Signal Processing Pipeline
	FIR	Finite Impulse Response
T	transpose	Matrix transpose
QL	QR/LU	QR/LU matrix operations
C	Correlator	Should be FIR
I	matrixInverse	Matrix inversion operation
	IIR	Infinite Impulse Response
RI	randomInit	Random input generation
FL	fileLoad	Load input data from a file
FE	FEC_ENC	Forward Error Correction Encoding
FD	FEC_DEC	Forward Error Correction Decoding
	EVD	Eigenvalue Decomposition
	SVD	Singular Value Decomposition
MR	MapReduce	Parallel Pipeline Algorithm
	Map	Filter and Sort
R	Reduce	Summary Algorithm
DP	DynamicProgram	Result Reuse Algorithm
GT	GraphTraversal	State Machine
NB	NBody	Finite Element Algorithm
SBS	SplatBlurSlice	Image processing filters
M	Multi	Multiple Kernel Structures

Table 3.3 shows the sample contribution of each directory to each label category. Totals for all categories and labels are along the bottom row and right-most column, respectively, with a grand total of all structured samples of Dash-Corpus at the bottom right corner.

Table 3.3: Cross Product Table

Directory	FL	GEMV	SBS	M	FE	FFT	FIR	GT	T	NB	DP	ZIP	GEMM	I	SVD	MR	RI	FD	QL	IIR	C	UL	Total
OpenCV	15952		4303	11422		99	9778	243				2458				59						23875	68189
Eigen		29							7				1017				39					1143	2235
RadioCorpus		6		8	168	150	38					46	54	24	22			151	7		13	755	1442
GSL		4		196		14	24	3				24					186					970	1421
Artisan		30				354	70	91		19	7					18	342					788	1719
Armadillo						104																410	514
Benchmarks						440							44				111					737	1332
SigPack						149	117													4		634	904
FFTW						5136																11780	16916
MatrixOps							30		24				89				152				30	694	1019
SPUCE							12													279		87	378
Halide						8																1020	1028
FEC																	8					55	63
LiquidSDR																						527	527
SHOC																						166	166
FFmpeg																						132	132
CortexSuite																						117	117
mbed_TLS																						969	969
VOLK																						129	129
Kestrel																						281	281
Total	15952	69	4303	11626	168	6454	10069	337	31	19	7	2528	1204	24	22	77	838	151	7	283	43	45269	99481

Chapter 4

TRACEATLAS

TraceAtlas is a repository of software tools that facilitate the processing of computer programs into kernels. It was originally proposed as a dynamic tracing backend intended as a foundation for a machine understanding of computation [70]. TraceAtlas creates an approximation for the kernels present in a program by identifying hot-code, or segments of the program that run many times. Its toolchain has three parts: a compilation stage for injection of the dynamic tracing backend, a kernel parsing stage and a data structuring stage. Dynamic tracing uses LLVM IR bitcode, a custom annotation pass and zlib [71] to facilitate low-overhead dynamic tracing. Kernels are approximated from the dynamic trace and the original source program bitcode using a multi-stage parsing algorithm. Finally, the static and dynamic instruction counts of each parsed kernel are captured. The result is a structured, statistical data sample for each kernel, providing information about its function and implementation.

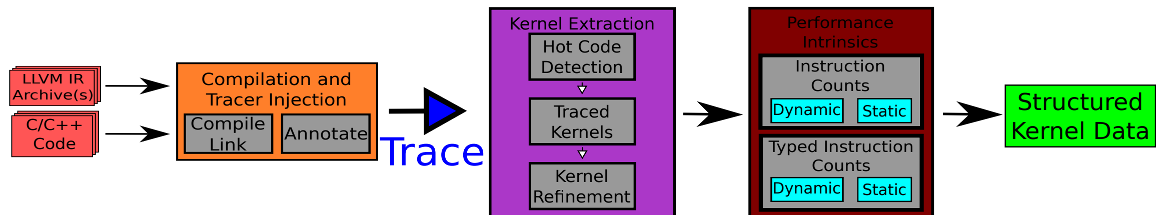


Figure 4.1: TraceAtlas Processing Pipeline

4.1 Kernel Definition

TraceAtlas defines a kernel as a temporally related set of basic blocks that recur many times and contain a semantic purpose [14]. In practice, these kernels are loops or recursive function calls. Using the dynamic trace history, the cartographer parses

the loops of the program that accounted for the majority of the basic blocks executed, 99.6% on average [14], into sets of basic blocks that may overlap with other sets of blocks, thus potentially forming a hierarchy of kernels. By partitioning programs into these cycles, the cartographer captures the computation that accounted for most of the real-time performance of the program.

In theory, TraceAtlas supports all kernel archetypes proposed by [39], as well as several other kernel abstractions have been proposed [39, 42, 55, 64]. In practice, it has had limited ability to precisely capture kernels proposed by these sources because of its dependence on hot code, as explained later. Rather, the kernels it parses are based on loops in the LLVM IR that tend to miss, blend, or roughly approximate the computational kernels as defined by others. For structuring the computational data in Dash-Corpus, it is an excellent tool, as will be demonstrated in section 7.

4.2 Compilation and Tracing

TraceAtlas programs are compiled using the LLVM backend. First, the original source code is compiled into LLVM IR bitcode. This only includes code that has been compiled into LLVM IR. Operations used in the C and C++ libraries, like ctors, dtors, memory operations, interrupts, containers and others are dynamically linked at runtime, and do not show up in the trace itself. A custom annotation pass injects software into the LLVM IR of the original program to produce a trace of all basic block entrances and exits, and every load and store operation into a text file. Using the compression library zlib, TraceAtlas compresses this dynamic trace at runtime. This reduces memory overheads of dynamic tracing by 500-2000x, and trace time reductions of 25% compared to naive implementations.

4.3 Kernel Extraction

TraceAtlas extracts kernels from the dynamic trace with a tool called the cartographer, the kernel mapper. Once the dynamic trace history of the program is available, the cartographer exploits the dynamic trace history of the program as well as the original static bitcode program to create sets of basic blocks that are recurrent in structure, thus forming a TraceAtlas kernel. The program has two stages, each with two parts.

4.4 Type 1 Kernels

Kernels are built by collecting a set of temporally related basic blocks. First, the cartographer generates two pieces of data for each basic block in the source bitcode: a count for the number of times each basic block occurs, and a vector of affinity scores to neighboring blocks. Next, it picks kernel "seeds", or the origins of kernels. It does this by sorting the set of basic block counts from greatest to least and picking the highest-count block available. The cartographer greedily sums the highest-affinity basic blocks from the affinity vector of the seed until a threshold parameter is met (set to 0.95 for Dash-Database). Once the set of basic blocks have been assembled, the cartographer removes all members of this kernel from the set of possible seeds and continues the algorithm until no seed candidates are left.

An affinity score between two basic blocks is calculated according to equation 4.1, where α is an operator, r is a constant radius parameter (set to 5 for Dash-Database) and k is a frequency count within that radius. A distance between two basic blocks represents the integer number of basic blocks along a path described in the dynamic trace between a target block A and an adjacent block B. An affinity score is calculated for each block that appears within the radius of the target basic block,

forming a probability mass function. Thus, the greedy sum of temporally related blocks described above represents all blocks that occurred 95% of the time with each kernel seed.

$$f_r(A,B) = \frac{1}{2r+1}\sum_{k=1}^{2r+1} P(A\alpha^r B^k) \tag{4.1}$$

4.4.1 Type 2 Kernels

Often, basic blocks that are required for the computation to make sense are missing. The cartographer uses the dynamic trace and an estimation of each kernels' entry block to fill these gaps. To estimate the entry block of each kernel, the cartographer picks the first block it sees that belongs to a kernel set, and assigns this block to be the kernel entrance. This assignment is only done once. Next, the cartographer adds every basic block it sees between occurrences of the kernel entrance to a temporary set of basic blocks, one for each kernel. If the cartographer sees the entrance of a kernel set before it sees the entrance of any other kernel index, it assumes that a kernel has just completed a revolution, and adds the temporary set to the final set of that kernel, clearing the temporary set afterward. By interpolating between instances of a kernel's entrance, the cartographer can add each basic block it may have missed in the first step.

Since the entrance of each kernel is only an estimation, and since the entrance is only assigned once, the entrance estimate may not be correct on the first iteration, leading to a kernel set that is still incomplete. To fix this, the cartographer runs this step twice. With the correct kernel entrance block marking the kernel boundary, blocks that may have been missed are captured in the second iteration.

4.4.2 Type 3 and 4 Kernels

In the second stage, kernels are refined. After filling any potential holes in each kernel block set, blocks that don't participate in the recursive routine of the kernel, or blocks that may join two kernels together could have been erroneously added to the kernel block set. The cartographer attempts to remove these impurities by first finding blocks that cannot reach back to themselves after executing, and second, splitting any kernel block set that has internal loops into multiple kernel block sets. The final product is a set of kernel approximations that represent 99.5% of the original application trace basic blocks on average [14].

A result of the cartographer is shown in figure 4.4.2. After the TraceAtlas pipeline generated a dynamic trace history for the program in listing 4.1, the cartographer parsed its control flow graph, shown in figure 4.4.2, into kernels. There were three resulting kernels. The first kernel on the left of figure 4.4.2 is the first for loop in listing 4.1. The two kernels on the right of the graph represent a parent-child relationship, where kernel k_1 is the parent of kernel k_2. Kernel k_1 represents the outer loop of the sorting algorithm and kernel 2 is the inner loop.

Listing 4.1: Code Segment in the C Language Facilitating the Bubblesort Sorting Algorithm.

```
#include <stdlib.h>

#define SIZE 1024

int main( int argc, char* argv[] )
{
  int in = (int* )malloc( sizeof(int)*SIZE );
  for( int i = 0; i < SIZE; i++ )
```

```
  {
    in[i] = rand()
  }

  int swap;
  for( int i = 0; i < SIZE; i++ )
  {
    for( int j = i; j < SIZE; j++ )
    {
      if( in[j] > in[i] )
      {
        swap = in[j];
        in[i] = in[j];
        in[j] = swap;
      }
    }
  }
  return 0;
}
```

4.5 Kernel Labeling

TraceAtlas supports the labeling of source code segments likely to result in a kernel. This works by calling a kernel entrance and exit function in the TraceAtlas tracing backend. The programmer specifies kernel boundaries, an entry and an exit, for source code segments likely to yield a TraceAtlas kernel. When the program is

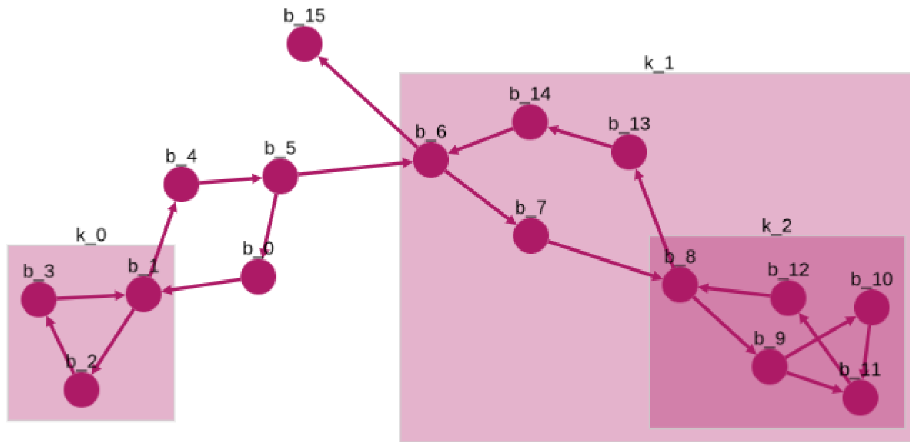


Figure 4.2: Diagram of the Kernels Extracted from the BubbleSort Program in Figure 4.1.

traced, these label entrances and exits are injected into the dynamic trace history. When parsing the dynamic trace for type 2 kernels, the cartographer keeps a last-in first-out buffer of open labels. A label is added to the buffer when its entrance is observed, and a label is removed when its exit is observed. When a kernel boundary is encountered, all labels in the label buffer are added to the label set of that kernel. Therefore, kernels are allowed to have multiple user-defined labels attached to them.

This mechanism has a degree of uncertainty. Since kernels are built from a dynamic trace history, the location of kernels, and any labels intended for those kernels, are not directly defined in the source code. Labels that wrap kernels with hot code are most reliably assigned the label intended for it. Labels that wrap kernels that don't have hot code may fall into one of two common outcomes that have been observed. First, the kernel label is not assigned to a kernel at all, and does not show up in the output of the cartographer. Second, the label is appended to a kernel that

is a conglomeration of multiple kernels, resulting in a kernel that has many different labels. I refer to this phenomenon as "kernel fusion" because of the lack of a hot code segment, or "fusion" of computation, for each kernel label. As a result, kernels may have labels that were not intended to be part of one kernel data point.

4.6 Structured Data

The final stage of the cartographer creates a structured, statistical representation of each kernel extracted from the input trace. Using the origin source bitcode and each kernel's basic block set, the cartographer creates counts of all intrinsic instruction similar to that proposed in [20]. There are two types, each with two flavors. The first type is a static count of what is in the source program. These are just counts of all unique LLVM IR instructions found in the original bitcode, exclusive to each kernel. The second type is dynamic: a count of each unique LLVM IR instruction that occurs within the trace. This is done by multiplying each basic block count with the instructions it contains. Both the static and dynamic counts have two types of counts, creating four different sets of instruction counts. First is the raw instruction count, and second is the "cross-product" count, which is each raw instruction combined with the type of the data being processed, yielding a typed instruction count. All instruction counts are normalized by the total number of instructions relative to that set of counts, creating a normalized breakdown of the entire program for each instruction that was used. The result is a feature vector with 920 members.

Chapter 5

CORPUS PROCESSING

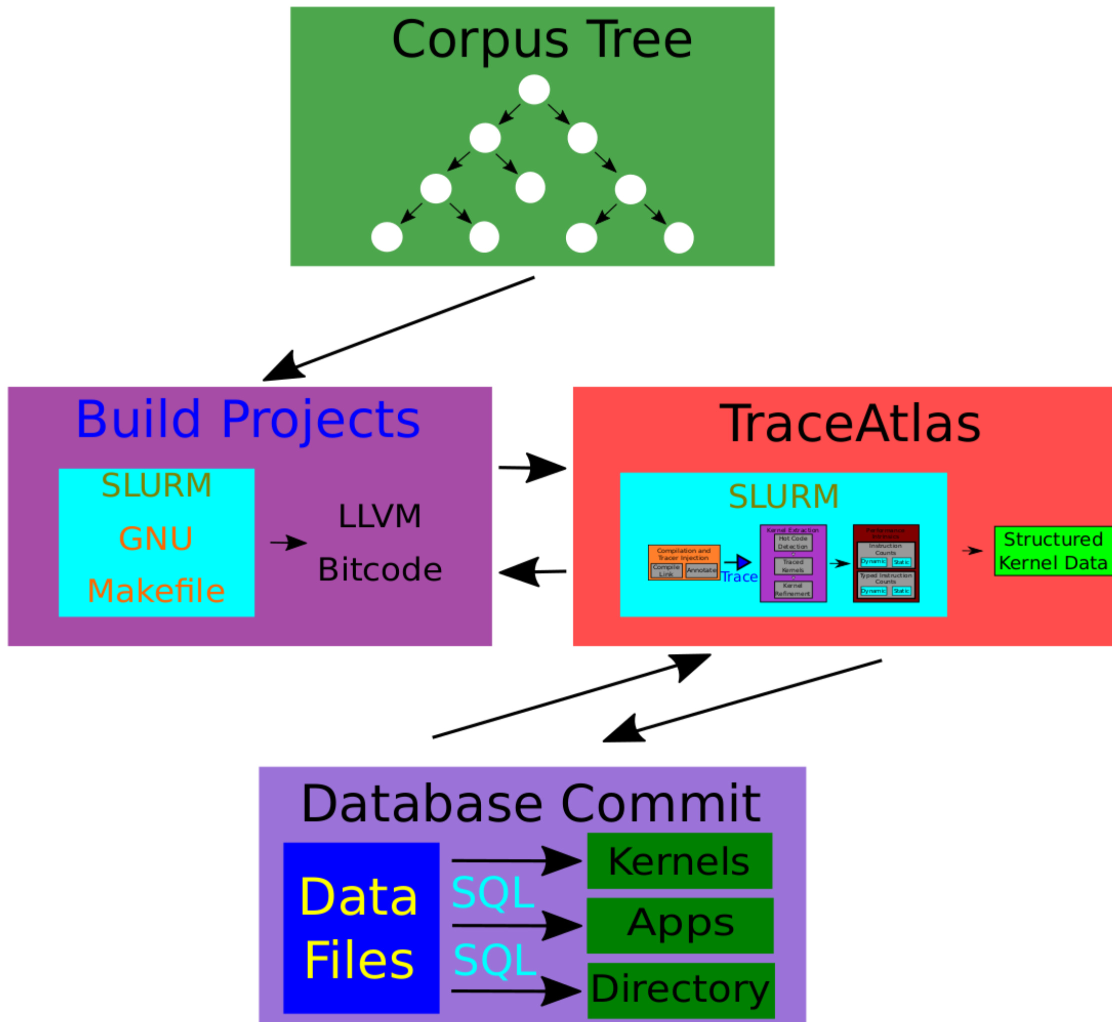
Dash-Corpus is structured using Dash-Automate, an automation script that facilitates the TraceAtlas pipeline, a collection of LLVM bitcode libraries and GNU Makefiles. To store each valid data point, Dash-Automate uses a SQL database. Since Dash-Corpus is built to produce over 2,200 dynamic traces, Dash-Automate uses the SLURM workload manager to efficiently build the corpus across a server cluster. Figure 5 shows the processing flow of Dash-Automate.

5.1 Dash-Automate

Dash-Automate begins by building each project into LLVM IR bitcode. Using a tree of subdirectories, where nodes are projects and edges are directory pointers, Dash-Automate calls each project's build flow, implemented as GNU Makefiles, that are defined specifically for the project being built. These build flows often include multiple compile-time configurations that will capture different algorithm settings, precision, and assumptions (for example, a double-precision, 2-dimensional, real-to-real fast Fourier transform).

Each Makefile uses a set of environment variables that facilitate the generation of LLVM IR. The source code of the project is built into bitcode and any external libraries being targeted are linked to the output bitcode via external bitcode archives. This produces a bitcode file that contains all program elements, namely LLVM basic blocks and memory transactions, that will be traced. Also, a separate runtime data repository is pointed to by an environment variable and used in each program for data

Figure 5.1: Dash-Automate Processing Diagram



input and output. Put together, this environment makes Dash-Corpus a portable and reliable build environment that can be scaled to other machines and platforms.

After the bitcode file is generated, Dash-Automate facilitates the TraceAtlas pipeline as described in figure 4.1. Each individual bitcode program produced in Dash-Corpus is run through the TraceAtlas pipeline. For each bitcode, an array of scripts are batched to SLURM, where each script has specific parameters and error checks to ensure that the generation of data runs smoothly and without inconsistency.

This array facilitates the building of the bitcode into executable programs with any user-defined args, generated dynamic traces using ay user-defined runtime arguments, and processes these traces using the TraceAtlas toolchain. If an error is encountered, the pipeline for that bitcode is haulted, and Dash-Automate highlights the problematic project in a report file. Once the generation of data is complete, Dash-Automate structures all data generated by each bitcode into a form suitable for pushing into the SQL database and commits the data.

Chapter 6

EVALUATION METRICS

Data sets require an evaluation of their quality. When computational data is structured to support machine learning, there is a need to evaluate the information this structuring contains. I propose a series of metrics to conduct this evaluation. I define three metrics: breadth, depth and richness.

6.1 Sample Transformation

My evaluations use a custom transformation to create a discrete probability space of 1 feature for each data sample. Let S be a set of data samples with N sample in an input data set, M be the number of features in the feature space F, δ be a transformed feature onto its kth quantile, and Π be a concatenation operator on a set of characters to form a string. Then

$$X = \{x_i = \delta_0\delta_1...\delta_{M-1}; \forall s_i \in S\}, 0 \leq i \leq N-1 \tag{6.1}$$

where

$$\delta_m = Q_m(s_{i,m}) \tag{6.2}$$

is a feature quantizer function, one for each feature, whose bins are the lower, middle and upper median values of all samples of the feature m, and

$$\delta_m \in [0,1,2,3], 0 \leq m \leq M-1 \tag{6.3}$$

For each feature f_m on the feature space F, a set of quantile boundaries is found by finding the median values of the lower quarter, lower-middle quarter, upper-middle quarter, and upper quarter among every sample of that feature. Then, for each

transformed data sample, x_i, each of its features are projected onto their respective quartile, and each feature's quartile number (0-3 where 0 is the lower quartile) is concatenated together to form a single string. This results in a discrete data sample of dimension 1.

I use this transformation for two reasons. First, it largely reduces the complexity of the sample space, which makes evaluations on results easier to interpret without losing information. Second, a discrete space of dimension 1 is required for the measurements I use to define my data set metrics detailed below.

6.2 Breadth

Breadth measures the variety of implementations in the input corpus. When a corpus is used for machine learning in prior work, often these corpuses only include code that came from a handful of places - mostly benchmarks. Breadth punishes these input corpuses with a low score for their lack of sampling from the entire computational community, and rewards corpuses who include code from more varieties of implementations.

There are two characteristics about breadth. First, it does not increase as the size of the data base increases. The breadth of a computation space is determined by the types of computation that is within it, therefore moving in a direction orthogonal to depth. Second, breadth is captured by computation kernels. These kernels represent the actual implementation of recurring operations that will be implemented on an architecture, leaving out "scheduled kernel" archetypes, as proposed by [39]. I view every domain-specific computer program with parallel kernels as a pipeline, and the information contained within each pipe stage is interchangeable, thus the scheduling of these pipe stages is orthogonal to the kernel.

I measure breadth as the size of the event space Λas proposed in equation 6.1. This transformation does two things. First, it represents the number of features that were of non-zero variance in the feature space Δ. This shows how many unique instructions and instruction-type cross products were used in the input corpus. Second, it represents all unique configurations of computation kernels as described by its instructions. This filters the noise of scheduling and platform-specific implementation. Third, it removes outliers from the data set by projecting each data sample onto a space generated by the distribution of the features. By ignoring the tails of the distributions of each feature, the unique strings present in the data set after transformation indicate which computational configurations were observed.

6.3 Depth

Depth is the body of the database. It represents the sheer size of the input data set. Sufficient sampling is required to provide enough material to train machine learning models. It also provides a sufficient amount of testing when the data set is split to test, train and validate these models. I measure depth by raw data sample count.

6.4 Richness

Richness measures the ability of structured data to represent the source code it came from. If a data set is rich, each data sample will contain useful information within its features about the kernel it represents.

To evaluate the richness of a feature space, I utilize explained variance and information entropy. Both measurements are applied in a two-step process that shows the contribution of each individual feature to both measurements using equation 6.1. First, each dimension of the feature space is separated, evaluated, and reordered according to its contribution to the explained variance of the entire data set. Then the

feature space is reconstructed using that ordering and each feature's contribution to the two metrics is shown.

Explained variance can be thought of as the amount of information that has be learned. By ordering the features of the data set by their contribution to explained variance, the feature space can be evaluated for the features that are holding the most information, or richness, about the data set.

Information entropy is a measure of the "surprise" one should expect when observing a sample of a random variable. The more entropy a feature has, the more difficult it will be to make an accurate prediction about that feature. High-entropy, or "low-level", features don't provide much information for a machine learning model to train with. Low-entropy, or "high-level", features that have little contribution to entropy, provide lots of information to a machine learning model to learn from.

I use two methods to separate the dimensions of the feature space. The first is PCA, where explained variance is each eigenvalue's contribution to the total eigenvalue sum. For PCA analysis, explained variance for each eigenvector is the amount of the total eigenvalue sum its corresponding eigenvalue accounts for. The second is an evaluation of the features themselves, which I will refer to as univariate analysis. For univariate analysis, explained variance is defined as each feature's contribution to the sum of all feature's population variance.

The implementation of this evaluation will now be described. An input data set is parsed into its most significant features using principle components and univariate features in decreasing order of explained variance. A set of trials are conducted, starting at the top of the sorted list of features, where 1 additional sorted feature is added for each trial. Within each trial, the input data set with only the selected features is transformed as described in equation 6.1. A histogram of X is generated and normalized, forming a probability mass function. The entropy of this discrete

random variable is calculated using 6.4. The results of this experiment shows each feature's contribution to explained variance and information entropy.

$$H(X) = -\sum_{n=0}^{N-1} P(x_n) log_2(P(x_n)) \tag{6.4}$$

Mutual information is a measure of shared entropy between two random variables. Equation 6.5 defines the mutual information score between two random variables X and Y. When two features of a feature vector have a high mutual information score, they share lots of entropy between each other, thus indicating a strong correlation. Features whose mutual information is 0 are said to be independent. Thus, gauging the richness of a sample space by the amount of information it provides to the labels makes information relationships is straightforward.

I measure the richness of a data set by calculating, for each sample in the label space Y, its contribution to the mutual information between a sample space X and the label space Y. First, I transform the sample space as described by 6.1. Then three probability mass functions are calculated separately for the transformed sample space X, the label space Y, and the joint probability mass function of X and Y. Finally, equation 6.5 is applied to each sample in the label space to yield a contribution score.

$$I(X, Y) = -\sum_{n=0}^{N-1} \sum_{m=0}^{N-1} p_{(X,Y)}(x_n, y_m) log_2\left(\frac{p_{(X,Y)}(x_n, y_m)}{p_X(x_n) p_Y(y_m)}\right) \tag{6.5}$$

Chapter 7

CASE STUDIES

I applied the metrics defined in section 6 to Dash-Database and a data set modeling those of prior work. The results show that Dash-Database is superior to the smaller data set in breadth and depth, and that the data structuring techniques described in section 5 provides a rich data set.

7.1 Test Data Set

The test data set was built from a subset of Dash-Corpus. It includes Halide "benchmarks" from the test suite of Halide's Github repository [72], a subset of the Scalable Heterogeneous Computing benchmark [68], and a subset of the CortexSuite benchmark [49]. I used the same methodology described in section 5 to construct this data set. Some programs were traced multiple times with different compile-time and runtime arguments. Table 7.1 details each program. In total, 56 traced programs produced 1,458 data samples.

I constructed the test corpus to represent the input corpus of prior work cited in section 2. The Halide benchmarks were used as the test corpus in [22], the SHOC benchmark was used in [29, 31, 73], and CortexSuite in [21].

The programs in the test corpus were modified in two ways according to the Dash-Automate build flow described in section 5. First, all programs were evaluated with a single thread. Second, their file input/output were adapted to the Dash-Automate environment, but this did not make a difference to the program kernels.

Table 7.1: Test Corpus

Halide	CortexSuite	SHOC
bilateral	stitch	bb_gemm
blur	sift	bm_small x 2
camera	spectral	md
MaxPool x 11	lda x 2	pp_scan
harris	disparity	qsort x 2
hist_eq	tracking	stencil
interp	svm	ss_sort
lensblur	localization	triad
local_lap	sphinx	
unsharp	mser	
AveragePool x 11	liblinear	
	texture_synthesis	
	motion_estimation	
	kmeans	
	rbm	

7.2 Breadth and Depth

The breadth and depth scores of each corpus are shown in table 7.2. I see a 2-fold magnitude difference in depth, indicating how a small data set lacks ours in raw statistical strength. For breadth, the full data set is about 5 times larger, which is not a linear factor of depth. Even though the test corpus produced about 1.5% of the kernels, it had about 20% of the breadth. To help explain this, each input corpus needs to be considered. The test corpus had 3 libraries in it, while the full corpus had

Table 7.2: Breadth

	Breadth	Data Samples
Dash-Database	3,387	99,481
Test Data Set	737	1,458

about 20, or about 15%. This tracks closely with the difference between the breadth measurement. Thus, the first property of breadth is observed.

7.3 Richness

Since Dash-Automate was used to generate both Dash-Database and the test data set, the results of the richness evaluation showed similarities between them. Figure 7.1 shows, for both data sets, a log-linear scatter of the change in explained variance and the change in information entropy for increasing PCA vector counts. The explained variance curves are similar to each other for the majority of the plot, indicating a similar eigenvalue decomposition for both Dash-Database and the Test Data Set. Toward the right of the plot, the change in explained variance for the test data set begins to fall below the curve of Dash-Database. This is because of the lack of breadth in the test corpus. A lesser breadth requires fewer features to represent its information, therefore the contributions of higher-dimension features will approach zero more quickly than that of a more broad data set.

The information entropy scatters, while showing a high degree of variance relative to the explained variance scatters, generally trend downward with the same shape as the explained variance scatters. This follows with the expectation of contributions to explained entropy: the highest entropy features contribute the most to the explained variance of the data set. Another observation worth making is a slight tendency for the data points to cluster together. When features cluster together in information entropy, this means those features are related in their ability to identify a kernel

group or type. Since TraceAtlas built both the test data set and Dash-Database, the clusters can be seen acting together in some parts, albeit on different levels.

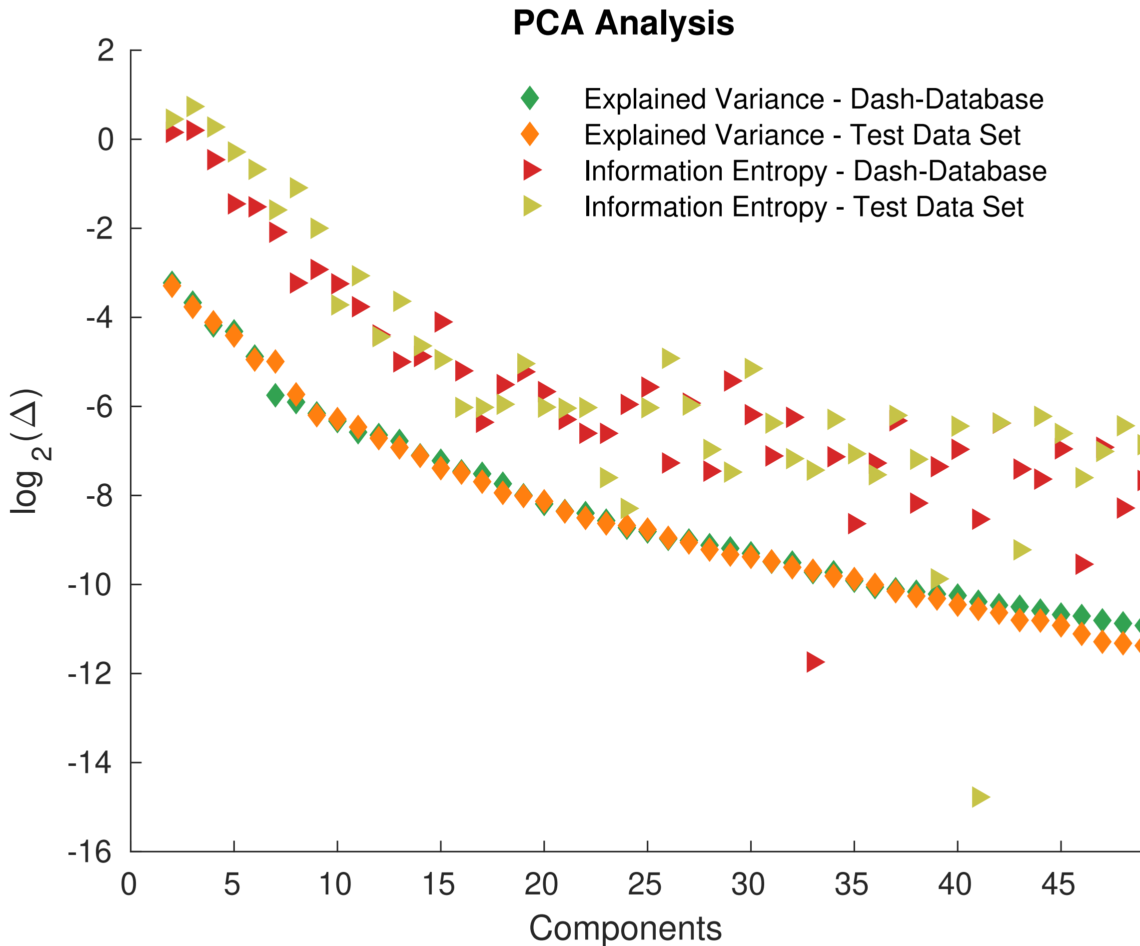


Figure 7.1: Change as the Number of PCA Components Are Increased.

Figure 7.2 shows the analysis for univariate features. The results are notably different than that of the PCA results. First, the explained variance of Dash-Database continues making about the same levels of contributions after about 30 features, whereas the test data set starts to fall further lower. This indicates that the test data set contains slightly less information than Dash-Database, which is most likely due to the increased breadth of Dash-Database that allowed it to capture rich features not in the test corpus. Also notable in this plot is the higher presence of clustering in both the information entropy and explained variance curves. These features likely

point to the same types of kernels, indicating that although differences exist in the richness of the test data set and Dash-Database, the same feature clusters are still present in each one.

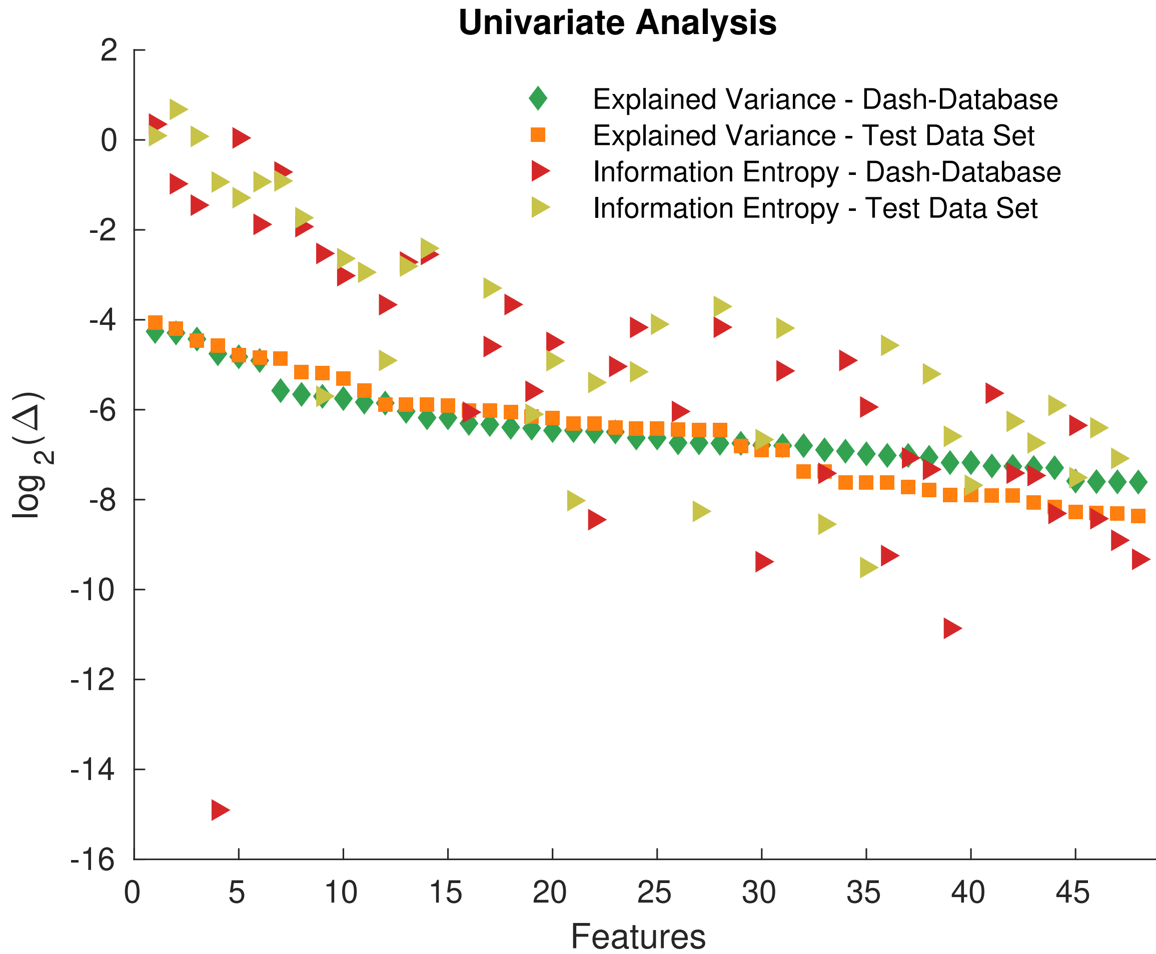


Figure 7.2: Change as the Number of Univariate Features Increases.

7.4 Mutual Information and Supervised Learning

I conducted a supervised learning evaluation of Dash-Database coupled with the mutual information experiment presented in section 6. These evaluations required the presence of labels for kernel data points, which the test corpus did not have. As a result, the following evaluations were only conducted on Dash-Corpus.

I evaluated the contribution of each label to the mutual information between transformed data samples and the label space. Then I trained four regression classifiers to predict this label space: linear regression using ordinary least-squares, logistic regression using a sigmoid function, a gaussian naive Bayes classifier and a support vector machine. The input data for training and testing these classifiers included 20 univariate features that has the highest population variances. Ten trials were conducted on a random 80/20 split, where all four models saw the same data for each trial. The results were averaged over these trials and are presented in Table 7.3. Three metrics were used to evaluate the performance of each classifier: precision, which is a measure of the ratio between the amount of correct predictions for the target label divided by the total number of predictions for that label; recall, a ratio between the number of correct predictions for the target label and the number of total samples that were present for that label; and F1, the harmonic mean of precision and recall.

The results show a relationship between the mutual information contribution, the number of samples, and the F1 score for each label. Generally, the F1 score of each label was highest when both its mutual information and sample count was high. This is a striking result, as the mutual information was almost perfectly predicting which labels the learning models understood the most. The richness measurement therefore indicates how learnable each kernel is in an input data set. It is notable to point out that the "Multi" label was one of the most represented labels, yet had lackluster scores. This was because of the types of kernels this label was supposed to represent, as explained in section 3. Any kernel represented in source code that was deemed likely to be a kernel, but did not fit easily into the labels enumerated in table 3.2 were given the label Multi. Thus, predicting data samples that did not have a clear definition hurt the results of this label.

Table 7.3: Linear Regression | Logistic Regression | Naive Bayes | SVM

Label	Samples	MI	Precision				Recall				F1 Score			
C	20	0.007	0.0	0.0	0.006	0.0	0.0	0.0	0.345	0.0	0.0	0.0	0.012	0.0
DP	4	0.002	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0
FD	75	0.022	0.0	0.0	0.0	0.233	0.0	0.0	0.0	0.036	0.0	0.0	0.0	0.058
FE	85	0.022	0.0	0.0	0.081	0.0	0.0	0.0	0.029	0.0	0.0	0.0	0.041	0.0
FFT	3249	0.359	0.214	0.756	0.937	0.834	0.027	0.923	0.217	0.971	0.048	0.831	0.337	0.897
FIR	5035	0.302	0.069	0.289	0.16	0.527	0.023	0.128	0.002	0.673	0.034	0.177	0.003	0.571
FL	7982	0.515	0.346	0.666	0.379	0.81	0.118	0.844	0.842	0.983	0.176	0.744	0.523	0.888
GEMM	605	0.116	0.056	0.745	0.884	0.903	0.293	0.396	0.591	0.756	0.095	0.516	0.708	0.823
GEMV	34	0.011	0.001	0.0	0.012	0.0	0.065	0.0	0.894	0.0	0.001	0.0	0.024	0.0
GT	168	0.032	0.0	0.0	0.0	0.06	0.007	0.0	0.0	0.002	0.001	0.0	0.0	0.004
I	12	0.004	0.0	0.0	0.185	0.0	0.0	0.0	0.491	0.0	0.0	0.0	0.234	0.0
IIR	142	0.039	0.0	0.851	0.944	0.942	0.003	0.791	0.957	0.877	0.0	0.819	0.95	0.908
M	5782	0.313	0.425	0.396	0.395	0.637	0.042	0.636	0.037	0.483	0.076	0.488	0.068	0.539
MR	37	0.01	0.0	0.0	0.014	1.0	0.0	0.0	0.044	0.266	0.0	0.0	0.021	0.418
NB	9	0.004	0.0	0.0	1.0	0.5	0.0	0.0	1.0	0.5	0.0	0.0	1.0	0.5
QL	4	0.002	0.0	0.0	0.002	0.0	0.0	0.0	0.125	0.0	0.0	0.0	0.005	0.0
RI	412	0.088	0.0	0.629	0.0	0.741	0.0	0.427	0.0	0.797	0.0	0.508	0.0	0.768
SBS	2161	0.179	0.0	0.726	0.557	0.934	0.0	0.247	0.238	0.378	0.0	0.369	0.334	0.538
SVD	11	0.004	0.0	0.0	0.055	0.0	0.0	0.0	0.433	0.0	0.0	0.0	0.093	0.0
T	15	0.005	0.0	0.0	0.033	0.0	0.0	0.0	0.837	0.0	0.0	0.0	0.062	0.0
ZIP	1264	0.139	0.0	0.061	0.208	0.91	0.0	0.007	0.178	0.335	0.0	0.013	0.191	0.489

Chapter 8

CONCLUSION

Dash-Database can support statistical inference on problems in the field of computer systems. Using the example set by neighboring fields on applying machine learning to research questions, Dash-Database proposes a unifying framework. Researchers can use Dash-Database to build upon the contributions of prior work, a feat that cannot be achieved using current approaches.

I invite collaboration from other computer architects to fill any gaps left open by the current version of Dash-Corpus. Collection computer programs from domains outside of those proposed in this work can go a long way toward increasing the utility of Dash-Corpus. Thus, by harnessing the contributions of peers, Dash-Database can create a foundation for researchers to create a general understanding of computation: a feat that can solve decades-old challenges like automatic parallelization, optimal architecture configurations, and generalized dynamic scheduling.

REFERENCES

[1] M. Horowitz, E. Alon, D. Patil, S. Naffziger, Rajesh Kumar, and K. Bernstein, “Scaling, power, and the future of cmos,” in *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.*, pp. 7 pp.–15, 2005.

[2] M. Horowitz, “1.1 computing’s energy problem (and what we can do about it),” in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, 2014.

[3] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, “Understanding sources of inefficiency in general-purpose chips,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA ’10, (New York, NY, USA), p. 37–47, Association for Computing Machinery, 2010.

[4] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, “The greendroid mobile application processor: An architecture for silicon’s dark future,” *IEEE Micro*, vol. 31, no. 2, pp. 86–95, 2011.

[5] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, “Darkroom: Compiling high-level image processing code into hardware pipelines,” *ACM Trans. Graph.*, vol. 33, July 2014.

[6] IEEE, “International roadmap for devices and systems,” tech. rep., IEEE, 2020.

[7] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, “Hardware-software co-design of embedded reconfigurable architectures,” in *Proceedings of the 37th Annual Design Automation Conference*, DAC ’00, (New York, NY, USA), p. 507–512, Association for Computing Machinery, 2000.

[8] S. Krishnan, Z. Wan, K. Bhardwaj, P. Whatmough, A. Faust, G. Wei, D. Brooks, and V. J. Reddi, “The sky is not the limit: A visual performance model for cyber-physical co-design in autonomous machines,” *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 38–42, 2020.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[10] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Microsoft coco: Common objects in context,” in *European conference on computer vision*, pp. 740–755, Springer, 2014.

[11] G. A. Miller, “Wordnet: a lexical database for english,” *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[14] R. Uhrie, C. Chakrabarti, and J. Brunhaver, "Automated parallel kernel extraction from dynamic application traces," 2020.

[15] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic program optimization," *Commun. ACM*, vol. 59, p. 114–122, Jan. 2016.

[16] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," *SIGPLAN Not.*, vol. 46, p. 62–73, June 2011.

[17] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *International Symposium on Code Generation and Optimization*, pp. 123–134, 2005.

[18] D. Grewe, Z. Wang, and M. F. P. O'Boyle, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–10, 2013.

[19] R. Lyerly, A. Murray, A. Barbalace, and B. Ravindran, "Aira: A framework for flexible compute kernel execution in heterogeneous platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 269–282, 2018.

[20] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), p. 455–466, Association for Computing Machinery, 2014.

[21] U. Gupta, C. A. Patil, G. Bhat, P. Mishra, and U. Y. Ogras, "Dypo: Dynamic pareto-optimal configuration selection for heterogeneous mpsocs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–20, 2017.

[22] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.

[23] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Trans. Graph.*, vol. 35, July 2016.

[24] M. Hashimoto, M. Terai, T. Maeda, and K. Minami, "An empirical study of computation-intensive loops for identifying and classifying loop kernels: Full research paper," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, (New York, NY, USA), p. 361–372, Association for Computing Machinery, 2017.

[25] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, pp. 1–14, 2011.

[26] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," *ACM SIGOPS operating systems review*, vol. 40, no. 5, pp. 185–194, 2006.

[27] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[28] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 249–258, 2007.

[29] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 564–576, 2015.

[30] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 673–686, 2013.

[31] Y. S. Shao and D. Brooks, "Energy characterization and instruction-level energy model of intel's xeon phi processor," in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 389–394, 2013.

[32] O. Azizi, A. Mahesri, J. Stevenson, S. Patel, and M. Horowitz, "An integrated framework for joint design space exploration of microarchitecture and circuits,ˆ 2010 design," in *Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pp. 250–255, 2010.

[33] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[34] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," *SIGARCH Comput. Archit. News*, vol. 34, p. 195–206, Oct. 2006.

[35] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[36] L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, and E. Ruppin, "Placing search in context: The concept revisited," in *Proceedings of the 10th international conference on World Wide Web*, pp. 406–414, 2001.

[37] F. Hill, R. Reichart, and A. Korhonen, "Simlex-999: Evaluating semantic models with (genuine) similarity estimation," *Computational Linguistics*, vol. 41, no. 4, pp. 665–695, 2015.

[38] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[39] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[40] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), p. 519–530, Association for Computing Machinery, 2013.

[41] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "Gramps: A programming model for graphics pipelines," *ACM Trans. Graph.*, vol. 28, Feb. 2009.

[42] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2012.

[43] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[44] C. Sanderson and R. Curtin, "Armadillo: a template-based c++ library for linear algebra," *Journal of Open Source Software*, vol. 1, no. 2, p. 26, 2016.

[45] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[46] codeplea, "genann." `https://github.com/codeplea/genann`, 2 2019.

[47] alexshi0000, "Tetris-ai." https://github.com/alexshi0000/Tetris-AI, 2 2019.

[48] "DAW, Netlib Library." http://www.netlib.org/benchmark/.

[49] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "Cortexsuite: A synthetic brain benchmark suite," *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 76–79, 2014.

[50] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3." http://eigen.tuxfamily.org/, 2010.

[51] P. Karn, "Fec library version 3.0.1." https://github.com/Venemo/fecmagic, 2007.

[52] S. Tomar, "Converting video formats with ffmpeg," *Linux Journal*, vol. 2006, no. 146, p. 10, 2006.

[53] M. Frigo and S. Johnson, "Fftw. fast fourier transform library," *URL http://www.fftw.org/*, 2005.

[54] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, F. Rossi, and R. Ulerich, *GNU scientific library.* Network Theory Limited, 2002.

[55] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *SIGPLAN Not.*, vol. 48, p. 519–530, June 2013.

[56] J. Gaeddert, "Liquid dsp-software-defined radio digital signal processing library."

[57] A. Ltd, "mbed tls." https://tls.mbed.org/, 2015.

[58] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pp. 3–14, IEEE, 2001.

[59] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[60] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, *et al.*, "The perfect club benchmarks: Effective performance evaluation of supercomputers," *The International Journal of Supercomputing Applications*, vol. 3, no. 3, pp. 5–40, 1989.

[61] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 63–74, ACM, 2010.

[62] sourceforge, "projects/sigpack." `sourceforge.net/projects/sigpack`, 2019.

[63] audiofilter, "spuce." `https://github.com/audiofilter/spuce`, 10 2019. master/cbbc0fe.

[64] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 365–376, IEEE, 2010.

[65] N. West, "Vector optimized library of kernels," *Internet: http://libvolk. org/,[Sep. 10, 2018]*, 2016.

[66] B. Paul, A. R. Chiriyath, and D. W. Bliss, "Survey of rf communications and sensing convergence research," *IEEE Access*, vol. 5, pp. 252–270, 2017.

[67] I. Gomez-Miguelez, A. Garcia-Saavedra, P. D. Sutton, P. Serrano, C. Cano, and D. J. Leith, "Srslte: An open-source platform for lte evolution and experimentation," in *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*, WiNTECH '16, (New York, NY, USA), p. 25–32, Association for Computing Machinery, 2016.

[68] vetter, "shoc." `https://github.com/vetter/shoc/`, 2020. master/0aea03b.

[69] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, (USA), p. 75, IEEE Computer Society, 2004.

[70] R. Uhrie, D. W. Bliss, C. Chakrabarti, U. Y. Ogras, and J. Brunhaver, "Machine understanding of domain computation for domain-specific system-on-chips (dssoc)," in *Open Architecture/Open Business Model Net-Centric Systems and Defense Transformation 2018*, vol. 11015, p. 110150O, International Society for Optics and Photonics, 2019.

[71] J.-l. Gailly and M. Adler, "Zlib compression library," 2004.

[72] halide, "Halide." `https://github.com/halide/Halide/`, 2020. master/2531d11.

[73] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 673–686, 2013.