Jacob, Dejice (2020) *Opportunistic acceleration of array-centric Python computation in heterogeneous environments.* PhD thesis.

http://theses.gla.ac.uk/82011/

Reproduced under a Creative Commons license. Please refer to the terms of the licence: https://creativecommons.org/licenses/by/4.0/

Enlighten: Theses
https://theses.gla.ac.uk/
research-enlighten@glasgow.ac.uk

# Opportunistic Acceleration of Array-centric Python Computation in Heterogeneous Environments

Dejice Jacob

Submitted in fulfilment of the requirements for the degree of
*Doctor of Philosophy*

School of Computing Science

College of Science and Engineering
University of Glasgow

August 2020

© Dejice Jacob

## Abstract

Dynamic scripting languages, like Python, are growing in popularity and increasingly used by non-expert programmers. These languages provide high level abstractions such as safe memory management, dynamic type handling and array bounds checking. The reduction in boilerplate code enables the concise expression of computation compared to statically typed and compiled languages. This improves programmer productivity. Increasingly, scripting languages are used by domain experts to write numerically intensive code in a variety of domains (e.g. Economics, Zoology, Archaeology and Physics). These programs are often used not just for prototyping but also in deployment. However, such managed program execution comes with a significant performance penalty arising from the interpreter having to decode and dispatch based on dynamic type checking.

Modern computer systems are increasingly equipped with accelerators such as GPUs. However, the massive speedups that can be achieved by GPU accelerators come at the cost of program complexity. Directly programming a GPU requires a deep understanding of the computational model of the underlying hardware architecture. While the complexity of such devices is abstracted by programming languages specialised for heterogeneous devices such as CUDA and OpenCL, these are dialects of the low-level C systems programming language used primarily by expert programmers.

This thesis presents the design and implementation of ALPyNA, a loop parallelisation and GPU code generation framework. A novel staged parallelisation approach is used to aggressively parallelise each execution instance of a loop nest. Loop dependence relationships that cannot be inferred statically are deferred for runtime analysis. At runtime, these dependences are augmented with runtime information obtained by introspection and the loop nest is parallelised. Parallel GPU kernels are customised to the runtime dependence graph, JIT compiled and executed.

A systematic analysis of the execution speed of loop nests is performed using 12 standard loop intensive benchmarks. The evaluation is performed on two CPU–GPU machines. One is a server grade machine while the other is a typical desktop. ALPyNA's GPU kernels achieve orders of magnitude speedup over the baseline interpreter execution time (up to 16435x) and large speedups (up to 179.55x) over JIT compiled CPU code.

The varied performance of JIT compiled GPU code motivates the need for a sophisticated cost model to select the device providing the best speedups at runtime for varying domain sizes. This thesis describes a novel lightweight analytical cost model to determine the fastest device to execute a loop nest at runtime. The ALPyNA Cost Model (ACM) adapts to runtime dependence analysis and is parameterised on the hardware characteristics of the underlying target CPU or GPU. The cost model also takes into account the relative rate at which the interpreter is able to supply the GPU with computational work. ACM is re-targetable to other accelerator devices and only requires minimal install time profiling.

**Acknowledgements**

I would like to acknowledge my supervisors without whose support this thesis would not be possible. My utmost thanks goes to Jeremy Singer for his constant enthusiasm, encouragement and ideas; also for tolerating my penchant to burst into his office at all hours. I have learnt a lot and had lots of fun along the way. My sincere gratitude to Phil Trinder for his persistence and belief in me. The diligence and effort he put into developing my research and writing skills have hopefully borne fruit.

Thanks go to all my colleagues with whom I have worked. They have reviewed my work, made suggestions, and at the very least pretended to be interested in what I was doing. Thanks go to Kristian Hentschel, AnnaLito Michala, Cristian Urlea, Stephen McQuistin, Colin Perkins, Blair Archibald and Syed Waqar Nabi.

I am grateful to Alistair Murray for supervising me during my internship with Codeplay, Edinburgh. The crazy train journey into Edinburgh every day was worth it.

I am especially grateful to my parents, for their inspiration, encouragement and continuous support to undertake this endeavour.

Finally, I would like to thank my wife Ruby for trusting me along this journey. Her tireless support, encouragement and occasional nagging have helped me to the finishing line.

This work was funded from EPSRC grants EP/L000725/1 and EP/M508056/1.

# Dedication

*"Trust in the LORD with all thine heart; and lean not unto thine own understanding. In all thy ways acknowledge him, and he shall direct thy paths."*

Proverbs ch3 v5,6

To Daddy and Mummy,

*for going without so that I can go with. You have made me the man I am today.*

To Abigail, Rebekah and Samuel,

*you give meaning to my life. I will forever endeavour to earn your love and respect.*

To my beloved Ammu,

*I have found a virtuous woman. Your price is far above rubies. My heart safely trusts in you.*

To the Lord Jesus Christ,

*who found my life and soul valuable enough to save. Thank you!*

# Table of Contents


**1 Introduction** **1**

1.1 Hypotheses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.2 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Publications and Authorship . . . . . . . . . . . . . . . . . . . . . 4

1.4 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

**2 Background** **7**

2.1 Language Runtimes and JIT Compilers . . . . . . . . . . . . . . . . 8

2.1.1 Python . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.1.2 CPython Virtual Machine . . . . . . . . . . . . . . . . . . 9

2.1.3 JIT compilation . . . . . . . . . . . . . . . . . . . . . . . 10

2.1.4 Numba – A JIT compiler for Python . . . . . . . . . . . . . 12

2.1.5 Section Summary . . . . . . . . . . . . . . . . . . . . . . . 14

2.2 Dependence analysis . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.3 Heterogeneous Compute Architectures . . . . . . . . . . . . . . . . 17

2.3.1 GPU hardware execution model . . . . . . . . . . . . . . . 19

2.3.2 GPU Programming Model . . . . . . . . . . . . . . . . . . 21

2.4 Importance of Loop Parallelisation . . . . . . . . . . . . . . . . . . 24

2.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

**3 Literature Survey** **27**

3.1 Python JIT compilation frameworks . . . . . . . . . . . . . . . . . 29

3.2 Library bindings and Directive based JIT acceleration . . . . . . . . 31

3.3 Loop parallelisation . . . . . . . . . . . . . . . . . . . . . . . . . . 33

3.3.1 Loop parallelisation in managed language runtimes . . . . . . . . . 34
3.4 Accelerating Algorithmic Skeletons . . . . . . . . . . . . . . . . . . . . 38
3.5 Parallel Domain Specific Languages . . . . . . . . . . . . . . . . . . . . 43
3.6 Intermediate Representations for Parallel Computation . . . . . . . . . . 45
3.7 Parallel Cost Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.7.1 Analytical Cost Models . . . . . . . . . . . . . . . . . . . . . . 46
3.7.2 Machine Learning Models . . . . . . . . . . . . . . . . . . . . . 49
3.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

**4 ALPyNA : System Architecture 51**
4.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.1.1 Design Principles . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.1.2 Application Programming Interface . . . . . . . . . . . . . . . . 53
4.2 Benefits of Deferring Analysis to Runtime . . . . . . . . . . . . . . . . . 54
4.3 Staged Dependence Analysis . . . . . . . . . . . . . . . . . . . . . . . . 56
4.3.1 Static Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.3.2 Runtime Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.3.3 Hardware Abstraction Layer . . . . . . . . . . . . . . . . . . . . 67
4.4 Code Generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.4.1 GPU Code Generation . . . . . . . . . . . . . . . . . . . . . . . 69
4.4.2 Runtime Type Patching . . . . . . . . . . . . . . . . . . . . . . . 74
4.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

**5 Performance Evaluation of ALPyNA 77**
5.1 Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
5.2 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.2.1 Hardware . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.2.2 Software . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.3 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.4 Performance Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.4.1 Comparison of ALPyNA CPU Variant with CPython Interpreter Execution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

5.4.2 Comparison of ALPyNA GPU Variant with CPython Interpreter Execution . . . 85
5.4.3 Comparison of ALPyNA GPU Variant with JIT compiled CPU Variant 87
5.4.4 Comparison of ALPyNA GPU code with hand-written GPU code . 88
5.4.5 Comparison of ALPyNA GPU code with other solutions . . . . . . 89
5.5 Analysis and Compilation Overhead . . . 90
5.5.1 Static Analysis overhead . . . 91
5.6 Summary . . . 93

**6 The ALPyNA Cost Model 95**

6.1 Motivation . . . 96
6.2 ALPyNA Cost Model . . . 97
6.2.1 Modelling Interpreter Execution . . . 98
6.2.2 Modelling JIT Compiled CPU Execution . . . 99
6.2.3 Modelling GPU Execution . . . 99
6.3 Evaluation . . . 107
6.3.1 Experimental Setup . . . 107
6.3.2 Methodology . . . 107
6.3.3 Comparative Baselines . . . 108
6.3.4 Cost Model Performance . . . 108
6.4 Summary . . . 113

**7 Conclusion 115**

7.1 The Problem . . . 115
7.2 Contributions . . . 116
7.2.1 Staged Automatic Loop Parallelisation . . . 116
7.2.2 Systematic Analysis of ALPyNA . . . 117
7.2.3 Lightweight analytical cost model for ALPyNA . . . 118
7.3 Limitations . . . 119
7.3.1 ALPyNA . . . 119
7.3.2 ALPyNA Cost Model . . . 120
7.4 Future Work . . . 120

**A Evaluation of ALPyNA on platform *T2*** **123**

**B Evaluation of ACM on platform *T2* and *T3*** **129**

**Bibliography** **134**

# List of Tables


3.1 Overview of heterogenous compilation technologies . . . . . . . . . . . . . 28

5.1 Loop nest characteristics of benchmarks. . . . . . . . . . . . . . . . . . . . . 78

5.2 Speedup of CPU relative to CPython on *T1*. . . . . . . . . . . . . . . . . . . 85

5.3 Speedup of GPU relative to CPython and CPU JIT on *T1*. . . . . . . . . . . 86

5.4 Analysis and code generation time taken by ALPyNA for CPU and GPU on Platform *T1*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91

6.1 ALPyNA GPU speedups. Some benchmarks are always faster on the CPU. 110

6.2 Misprediction penalties for all platforms. . . . . . . . . . . . . . . . . . . . 112

6.3 Ratio of mispredicted/correct ranges on platforms *T1*, *T2* and *T3* using ACM and SVM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

A.1 Speedup of CPU relative to CPython on *T2*. . . . . . . . . . . . . . . . . . . 125

A.2 Speedup of GPU relative to CPython and CPU JIT on *T2*. . . . . . . . . . . 126

A.3 Analysis and code generation time taken by ALPyNA for CPU and GPU on platform *T2*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128

# List of Figures


2.1 Conceptual overview of topics covered in Chapter 2 . . . . . . . . . . . . . 8

2.2 CPython : Compilation and Execution Overview . . . . . . . . . . . . . . 9

2.3 JIT compilation of managed languages . . . . . . . . . . . . . . . . . . . . 11

2.4 Numba's JIT compilation of annotated functions. . . . . . . . . . . . . . . 12

2.5 Basic dependence types . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.6 Example of loop carried dependences. . . . . . . . . . . . . . . . . . . . . 16

2.7 Simplified block diagram of a heterogeneous system. . . . . . . . . . . . . 20

2.8 Two-tier thread hierarchy in CUDA. . . . . . . . . . . . . . . . . . . . . . 22

2.9 CUDA work-flow between host and accelerator device. . . . . . . . . . . . 23

2.10 Histogram showing number of `for` loops per notebook in a sample of end-user code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

2.11 Histogram showing the maximum `for` loop nesting depth per notebook . . 25

4.1 Dependence graph of loop nest in Listing 4.4 with iteration domain $(i,j) \leftarrow (32,1024)$ and $(k) \leftarrow 64$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

4.2 Dependence graph of loop nest in Listing 4.4 with iteration domain $(i,j) \leftarrow (32,1024)$ and $(k) \leftarrow (8)$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

4.3 Dependence graph of loop nest in Listing 4.4 with iteration domain $(i,j) \leftarrow (16,1024)$ and $(k) \leftarrow (16)$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

4.4 Staged system architecture of ALPyNA. . . . . . . . . . . . . . . . . . . . 58

4.5 ALPyNA preprocessing, static loop analysis and skeleton generation. . . . 59

4.6 Loop landmark IR structure used by ALPyNA. . . . . . . . . . . . . . . . 62

4.7 ALPyNA runtime dependence analysis and kernel generation. . . . . . . . 66

4.8 Interaction between VM and GPU. . . . . . . . . . . . . . . . . . . . . . . 70

5.1 Comparison of total execution time of ALPyNA generated code on *T1*. . . . 83

5.2 Execution time of ALPyNA CPU vs GPU on Platform *T1*. . . . . . . . . . 87

5.3 ALPyNA GPU vs hand written GPU code on Platform *T1*. . . . . . . . . . 89

5.4 Proportion of time spent by ALPyNA for analysis, compilation and execution on Platform *T1*. . . . . . . . . . . . . . . . . . . . . . . . . . . 90

5.5 ALPyNA runtime analysis and compilation overhead (Platform *T1*). . . . . 92

5.6 ALPyNA static analysis times for each benchmark on Platform *T2*. . . . . . . . 92

6.1 ALPyNA system architecture with cost model. . . . . . . . . . . . . . . . . 96

6.2 Simplified block diagram of a GTX-1060 (GP104). . . . . . . . . . . . . . 101

6.3 ALPyNA: profiling for GPU starvation point. . . . . . . . . . . . . . . . 103

6.4 ACM misprediction penalties on Platform *T1*. . . . . . . . . . . . . . . . . 109

6.5 ACM and SVM misprediction ranges on platform *T1*. . . . . . . . . . . . . 111

A.1 Comparison of total execution time of ALPyNA generated code on *T2*. . . . 124

A.2 Proportion of time spent by ALPyNA for analysis, compilation and execution on Platform *T2*. . . . . . . . . . . . . . . . . . . . . . . . . . . 127

A.3 Execution time of ALPyNA CPU vs GPU on Platform *T2*. . . . . . . . . . 127

A.4 ALPyNA runtime analysis and compilation overhead (Platform T2). . . . . 128

B.1 ACM misprediction penalties on Platform *T2*. . . . . . . . . . . . . . . . . 130

B.2 ACM and SVM misprediction ranges on platform *T2*. . . . . . . . . . . . . 131

B.3 ACM misprediction penalties on Platform *T3*. . . . . . . . . . . . . . . . . 132

B.4 ACM and SVM misprediction ranges on platform *T3*. . . . . . . . . . . . . 133

# Acronyms

**API** Application Programming Interface.

**AST** Abstract Syntax Tree.

**CPU** Central Processing Unit.

**CUDA** Compute Unified Device Architecture.

**CWP** Computation Warp Parallelism.

**DDR** Double Data Rate.

**DLP** Data Level Parallelism.

**DSL** Domain Specific Language.

**FFI** Foreign Function Interface.

**FPGA** Field Programmable Gate Arrays.

**FPS** Frames per Second.

**GC** Garbage Collector.

**GDDR** Graphics Double Data Rate.

**GIL** Global Interpreter Lock.

**GPU** Graphics Processing Unit.

**HAL** Hardware Abstraction Layer.

**IO** Input / Output.

**IPC** Inter-Process Communication.

**IR** Intermediate Representation.

**ISA** Instruction Set Architecture.

**JIT** Just-in-Time.

**JVM** Java Virtual Machine.

**LGPL** GNU Lesser General Public License.

**LLVM** Low Level Virtual Machine.

**LOOCV** Leave-One-Out Cross Validation.

**MIMD** Multiple Instruction Multiple Data.

**MIV** Multiple-Index-Variable.

**MWP** Memory Warp Parallelism.

**NVVM** NVIDIA Virtual Machine.

**OpenCL** Open Computing Language.

**OS** Operating System.

**PC** Program Counter.

**PCIe** Peripheral Component Interconnect Express.

**PE** Processing Element.

**PTX** Parallel Thread Execution ISA.

**RAM** Random Access Memory.

**RAW** Read–after–Write.

**SEJITS** Selective Embedded Just-in-Time Specialisation.

**SIMD** Single Instruction Multiple Data.

**SIMT** Single Instruction Multiple Threads.

**SISD** Single Instruction Single Data.

**SIV** Single-Index-Variable.

**SM** Streaming Multiprocessor.

**SoC** System-on-Chip.

**SPIR** Standard Portable Intermediate Representation.

**SVM** Support Vector Machine.

**VM** Virtual Machine.

**WAR** Write–after–Read.

**WAW** Write–after–Write.

**ZIV** Zero-Index-Variable.

# Chapter 1

# Introduction

Dynamic scripting languages, like Python, are growing in popularity and increasingly used by non-expert programmers [30, 144]. These languages provide high level abstractions such as safe memory management and dynamic type handling. The reduction in boilerplate code enables the concise expression of computation compared to statically typed languages. This improves programmer productivity. Increasingly, scripting languages are used by domain experts to write numerically intensive code. These programs are often used not just for prototyping but also in deployment [87, 116, 147].

Many programmers who write numerically intensive code come from a variety of domains (e.g. Economics, Zoology, Archaeology and Physics). Dynamic languages provide programmers with automatic memory management, array bounds checking and better portability. However such managed program execution comes with a significant performance penalty arising from the runtime having to decode and dispatch based on dynamic type checking.

Physical limitations [41,151] mean that it is no longer possible to increase the clock speeds of general purpose CPUs following Moore's law [97] and Dennard scaling [38]. Consequently, general purpose compute devices include accelerators [141] such as GPUs.

The massive speedups that can be achieved by GPU accelerators come at the cost of program complexity. Directly programming a GPU requires a deep understanding of the computational model of the underlying hardware architecture. While the complexity of such devices is abstracted by programming languages specialised for heterogeneous devices such as CUDA and OpenCL, these are dialects of the low-level C systems programming language used primarily by expert programmers.

Dynamic scripting languages provide bindings to CUDA and OpenCL for programming GPUs, but the programming model is static and low-level. Many libraries and programming frameworks exist to ease the burden on programming hardware accelerators e.g. Loopy [74], Parakeet [124] and River Trail [56]. However, they still require a knowledge of the under-

lying hardware characteristics and domain experts are typically not proficient accelerator programmers.

Just-in-Time (JIT) compilers increase the performance of managed languages by compiling frequently executed code paths to machine code at execution time. In general, JIT compilation targetting a CPU has been widely studied [16]. In the case of dynamic languages, compilation is complicated by type uncertainty until runtime. Current JIT compilers are very efficient at generating machine code for a CPU, e.g. HotSpot [105] and PyPy [119].

Research into automatic JIT compilers that take into account devices with diverse execution models (like GPUs) in a heterogeneous environment is still at a nascent stage. This thesis describes novel research into automatically JIT compiling Python loop nests for CPU–GPU platforms. The additional information available at runtime is exploited to optimise code, structure the thread hierarchy for a GPU, and to parameterise a novel cost model that predicts which device (CPU or GPU) will minimise the execution time of the loop nest.

## 1.1 Hypotheses

Dependence analysis produces a valid execution order constraints between sequential computational statements [70]. When applied to loops, dependence analysis enables a compiler to reason about automatically parallelising them.

The motivation for the thesis research arises from the following hypotheses :

H1. Staged static and dynamic dependence analysis on array-centric loop nests, in a general purpose dynamic scripting language, will yield higher performance parallel code than static dependence analysis alone.

H2. Code can be automatically synthesised to target heterogeneous architectures, with minimal user intervention. When such code is executed on a resource-aware adaptive Virtual Machine (VM), it will

- almost never degrade performance despite the overheads of dynamically re-targeting code
- in many instances significantly reduce execution time.

H3. A staged static and dynamic analytical cost-model can accurately determine the quicker device that will execute a given instance of an array-centric loop nest written in a dynamic scripting language in a heterogeneous CPU–GPU environment. Such a cost model need only be parameterised on the hardware characteristics of the CPU and the GPU, and requires only installation time profiling of the relevant compute devices using a simple pre-determined kernel.

## 1.2 Contributions

This thesis makes the following five key research contributions:

1. **A critical review of existing approaches to loop parallelism and GPU acceleration**: In Chapter 3, existing approaches to Just-in-Time (JIT) parallelism that focus on parallelising well known higher order functions and on parallelising loop nests speculatively with guard conditions is reviewed. A taxonomy of compiler systems is built based on resource awareness, target device, parameters to optimise and parallelised lanaguage control structures. While many parallel compilation techniques exist, we motivate the need for a staged loop parallelisation framework for dynamic languages.

2. **A novel automatic loop parallelisation framework for Python loop nests on heterogeneous CPU/GPU platforms that exploits staged analysis and and JIT compilation**: Chapter 4 presents the design and implementation of ALPyNA, a loop parallelisation and GPU code generation framework. A novel hybrid staged parallelisation approach is used to aggressively parallelise each execution instance of a loop nest. To support staged parallelisation, a static compilation phase parses the loop nest Abstract Syntax Tree (AST) and an in-memory data structure representing the loop is made available to ALPyNA's runtime analysis. The dependence graph is built and evaluated at runtime using introspection; the CPU and GPU kernels are generated specific to the dependence relationships that emerge for that specific execution instance of the loop nest. Automatic Just-in-Time (JIT) parallelisation hides complexity from the programmer while generating GPU kernels tailored to the exact dependence graph and iteration domain sizes that emerge at runtime. Augmented runtime information for kernel generation enables effective mapping of iteration domains to parallel GPU axes and chunking of thread-blocks.

3. **A systematic comparative analysis of parallel performance of ALPyNA**. Chapter 5 presents a detailed performance analysis of generated GPU code relative to execution time in the CPython interpreter and JIT compiled CPU code. The evaluation is performed on two CPU–GPU machines. One is a server grade machine while the other is a typical desktop. The execution time and speed up are compared using twelve standard loop intensive benchmarks, measuring each benchmark at 5 iteration domain sizes and with sequential interpreter runtimes varying from 0.21s – 5329s. ALPyNA's GPU code achieves orders of magnitude speedup over the baseline interpreter execution time (up to 16435x measured). We also show speed-ups (up to 179.55x) over the JIT compiled CPU variants for six of the 12 benchmarks. The varied performance results of ALPyNA motivate the need for a sophisticated cost model to select the device providing the best speed-ups for varying iteration domain sizes.

4. **A novel, lightweight, staged and extensible cost model for optimal device selection to execute loop nests**: Chapter 6 derives a novel lightweight predictive cost model to determine the optimal device in a CPU/GPU environment to JIT compile for every loop nest instance. This model is designed to adapt to the dependence analysis and custom kernels generated by ALPyNA at runtime. ALPyNA's cost model is parameterised on the hardware characteristics of the underlying target CPU or GPU and requires minimal install time profiling. The cost model takes into account relative rates at which an interpreter is able to supply the GPU with computational work. The cost model is re-targetable to other accelerator devices. The accuracy of the cost model is checked using the same twelve benchmarks used to characterise the performance of ALPyNA. We show a misprediction penalty of 14.6% (geometric mean) and a misprediction range of 14.66% (geometric mean) over a wide range of iteration domain sizes in 360 experiments performed on three different heterogeneous machines. The performance of the costmodel also performs better than a trained Support Vector Machine (SVM) model.

These contributions help to improve the accessibility of accelerator performance to developers working in dynamic languages. We aim to achieve this by implicit automatic parallelisation to ease the burden of reasoning about parallelism. The ALPyNA framework is written primarily for programming in the Python language, but the principles espoused are transferrable to other dynamic interpreted languages that have a JIT compiler and we believe to a range of accelerators.

## 1.3 Publications and Authorship

The source code for ALPyNA is available under the terms of the GNU Lesser General Public License (LGPL) at `https://bitbucket.org/djichthys/alpyna`. Some of the material in this dissertation has already been published in peer-reviewed venues. This dissertation unifies and expands work in the following three publications :

1. JACOB, D., AND SINGER, J. ALPyNA: Acceleration of Loops in Python for Novel Architectures. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (2019), Association for Computing Machinery, pp. 25–34

2. JACOB, D., TRINDER, P., AND SINGER, J. Python programmers have GPUs too: Automatic Python loop parallelization with staged dependence analysis. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (2019), pp. 42–54

3. JACOB, D., TRINDER, P., AND SINGER, J. Pricing Python Parallelism: A Dynamic Language Cost Model for Heterogeneous Platforms. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (New York, NY, USA, 2020), DLS 2020, Association for Computing Machinery, p. 29–42

The work reported in this dissertation is primarily my own with the following exceptions :

- The empirical study of the prevalence of loop nest control structures within end-user code (published in Jacob et al [61]) was performed by Jeremy Singer (cf. Section 2.4).
- The definition of loop header dominance and the duality function to elucidate the derivation of the cost model in Section 6.2 was primarily developed by Jeremy Singer and reviewed by Phil Trinder.
- The mathematical representations of Equations 6.6, 6.7, 6.9, 6.13 and 6.14 were re-written and simplified by Jeremy Singer and Phil Trinder.
- The SVM training model used as a comparison baseline for ACM (Section 6.3.3) was developed by Jeremy Singer.

## 1.4 Thesis Outline

The remainder of this dissertation is organised as follows:

- Chapter 2 – Background : This chapter presents technical concepts foundational to the rest of the thesis. It first introduces Virtual Machines (VM) and JIT compilers. It then introduces the idea of dependence analysis and parallelisation of loops. The chapter also introduces the execution model and the programming model of a GPU. The last part of this chapter is a study of the prevalence of loops in a large corpus of publically available Python programs.
- Chapter 3 – Literature Survey: This chapter examines current relevant and related work in the field of JIT compiling dynamic scripting languages for heterogeneous systems. A particular emphasis is placed on exploring related JIT compiler technologies for accelerators in dynamic languages.
- Chapter 4 – System architecture : This chapter introduces ALPyNA a staged dependence analysis and runtime parallelisation framework for Python. The simple Application Programming Interface (API), the motivations, capabilities and the implementation of ALPyNA are described here.

- Chapter 5 – Performance Evaluation of ALPyNA : This chapter evaluates the performance of ALPyNA's code generation capabilities using 12 loop intensive benchmarks over a wide variety of loop domain sizes. The overheads that affect the overall performance gain of JIT compilation is presented to motivate the development of a cost model.
- Chapter 6 – Cost model description and evaluation: This chapter derives a light-weight cost model that is integrated into ALPyNA and is capable of adapting to its staged analysis and runtime code generation capabilities. It predicts the faster device in a CPU/GPU heterogeneous environment. This is used to guide runtime code generation and JIT compilation. The prediction accuracy of the cost model is evaluated on three platforms and compared to a trained SVM model.
- Chapter 7 – Conclusion: The concluding chapter summarises the main contributions of this thesis, the limitations of the implementation and possible solutions to overcome such limitations. Further avenues to develop this work are also discussed.

# Chapter 2

# Background

This chapter reviews the relevant background material for the dissertation and sets out the main concepts related to JIT compilers, dependence analysis, GPU hardware architecture and the hierarchical parallel thread computation model for programming a GPU. The material in this chapter discusses the underlying concepts and technologies that underpin the design of ALPyNA (Chapter 4). Figure 2.1 gives a summary of the background topics in this chapter and how they relate to each other.

Section 2.1 introduces the Python language and then describes Virtual Machine (VM) environments for dynamic interpreted languages and accelerating programs written in managed languages using Just-in-Time (JIT) compilation. While Chapter 3 discusses broader efforts in Python and other dynamic languages to optimise performance, we will focus on the CPython interpreter [122, 123] and the supporting Numba compiler [78] used to accelerate Python code.

The core concepts that underpin dependence analysis and loop parallelisation are introduced in Section 2.2. These concepts enable ALPyNA to parallelise loop nest computation safely and effectively. Section 2.3 provides an overview of heterogeneous computer systems which combine general purpose CPU devices with other accelerators. The work done in this thesis focusses on GPU accelerators. To this end, Section 2.3.1 describes the hardware architecture and scheduling mechanisms within a GPU while Section 2.3.2 describes the CUDA/OpenCL programming model.

Finally, to motivate the need for loop parallelisation, Section 2.4 presents statistical analysis to quantify the prevalence of Python loop nests in a large body of publicly available Python software.

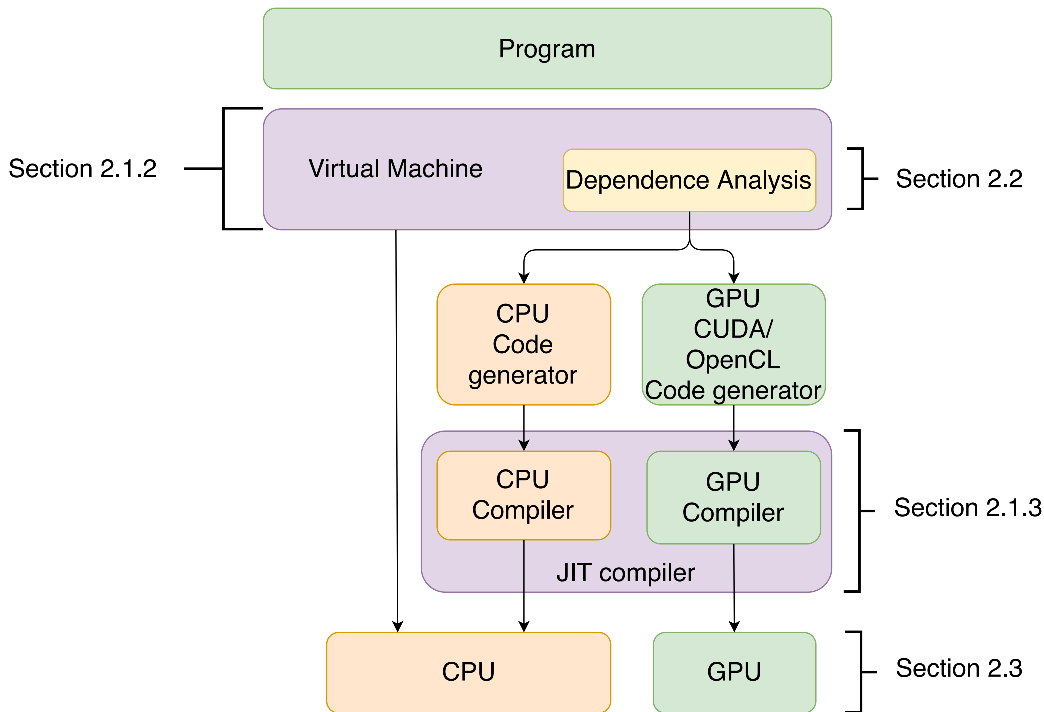



Figure 2.1: Conceptual overview of topics covered in Chapter 2

## 2.1 Language Runtimes and JIT Compilers

Python is a dynamic language [14] commonly executed on a Virtual Machine (VM). The CPython VM is the reference implementation of Python as well as the most widely used. This section describes various relevant features of a VM and the design choices as implemented in CPython.

### 2.1.1 Python

Python is a general purpose dynamic programming language that was designed by Guido van Rossum in 1989 [122]. Its popularity and adoption rate have been steadily rising over many years and it ranks highly in many surveys, e.g. [30, 144]. Python's simplicity and dynamic nature have contributed to its increased adoption for general purpose programming. Python is platform independent and is widely deployed in many different types of systems from tiny embedded platforms [28] to server nodes [87, 116].

Python developers program in various application domains such as archaeology [112], zoology [42], astronomy [120], bio-informatics [139], and meteorology [84]. It is widely adopted in applications that utilise data science [93] and machine learning [107].

Features that contribute to the popularity of Python include dynamic typing, a reference

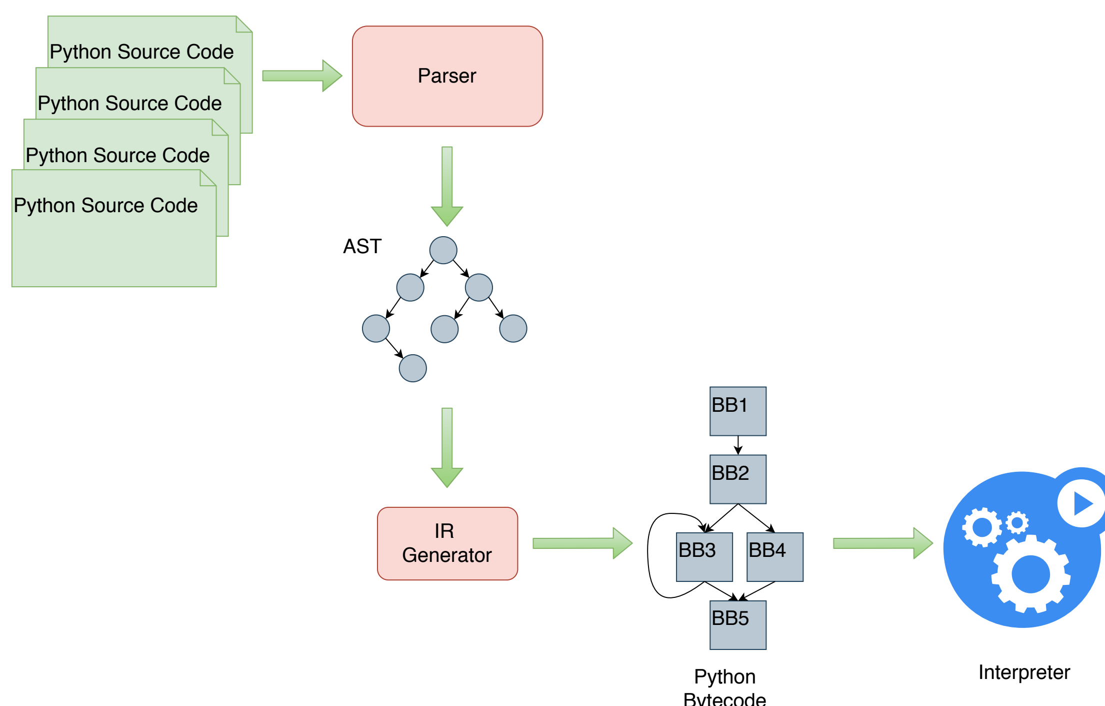



Figure 2.2: CPython : Compilation and Execution Overview

counting and cycle detecting garbage collector, and an easily extensible Foreign Function Interface (FFI). Python also provides limited functional programming support [77] using `map`, `filter` and `reduce` higher order functions. Python program execution by the CPython Virtual Machine (VM) makes it platform independent and portable.

Python code is executed largely in a userspace context. System calls to the Operating System (OS) happen through calls to the FFI which in turn dispatch the system call to the OS. Error causing exceptions are caught in the VM (if not explicitly handled by the user) and are not propagated to the wider system.

### 2.1.2 CPython Virtual Machine

> Formally, virtualisation involves the construction of an isomorphism that maps a virtual *guest* system to a real *host*. – Smith and Nair [132].

The VM is a program execution environment that abstracts away the underlying execution model of the CPU and the OS on which it executes. A VM directly executing a High Level Language without first compiling it into machine instructions is called an interpreter.

Figure 2.2 provides an overview of the steps involved in the compilation and execution of an program interpreted by a VM. An interpreter will typically use the same front-end com-

ponents as a static compiler such as lexical, syntax and semantic analysis phases to generate an Abstract Syntax Tree (AST). The AST is further lowered into an Intermediate Representation (IR) (Python *bytecode*) that is executed by the interpreter. Unlike compilers that generate machine code, the IR representation is much more abstract. When the predominant computational model is a CPU, the IR is considered hardware agnostic. However, this assumption breaks down in a heterogeneous compute environment as explained in Section 2.3.

A Python program can be compiled from Python source code to Python bytecode at program startup. Precompiled Python bytecode representations are stored in `.pyc` files and can be directly interpreted by the VM. The CPython interpreter starts running the compiled Python bytecode. The interpreter follows a *decode* and *dispatch* execution model. It sequentially fetches each bytecode, evaluates and then executes it. The CPython VM interpreter is implemented as a stack machine and maintains an execution stack. Each function call is maintained on a call stack.

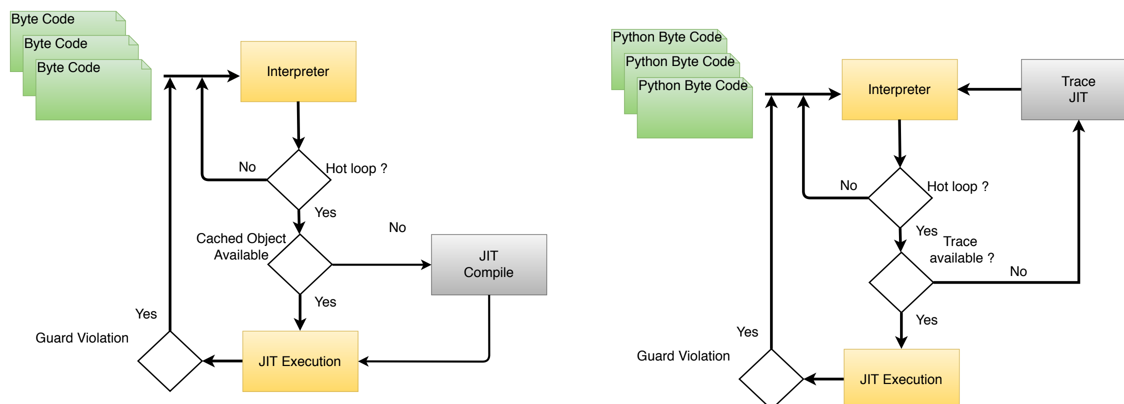
The use of a VM lends itself well to the dynamic type system of the Python language. The data types of the objects are only known at runtime. As each bytecode instruction is dispatched, the types are evaluated from the objects popped from the execution stack. At runtime the interpreter can also check for memory bounds violations when accessing vectors and throw exceptions when necessary.

The Garbage Collected (GC) memory model (Jones et al [65]) of Python helps reduce boilerplate code required to manage dynamically allocated memory within a program. The CPython VM uses a heap at runtime to allocate data objects. The memory is automatically managed using a *reference counting* GC. The GC automatically frees data objects that are no longer required when no references to the data objects remain or when the data object goes out of scope.

### 2.1.3 JIT compilation

CPython executes bytecodes sequentially in a *decode* and *dispatch* manner. To simplify synchronisation, a Global Interpreter Lock (GIL) is used in interpreter execution [22]. The GIL guarantees atomicity during the execution of any bytecode within the instruction stream. This ensures that the programs can be written without requiring explicit language level semantics to guarantee data consistency during execution. However, this comes at the cost of not being able to take advantage of parallel execution performance gains.

CPython supports multi-threading superficially by serialising execution of the threads and uses time-slicing to distribute execution over a single core. Threads can be pre-empted when Input / Output (IO) events force a thread to go to sleep. This design accommodates the use




(a) Adaptive Optimising JIT Compilation and Execution

(b) Tracing JIT Compilation and Execution

Figure 2.3: JIT compilation of managed languages

of a GIL in the interpreter.

JIT compilation has been widely used to speed up execution of interpreted languages (Aycock et al [16]). Modern JIT compilers tend to have the following main components to speed up interpreted code:

1. *Hot path detection*: Hot paths in a VM can be detected in software by checking for code with backward branches (denoting loops) or by maintaining counters to record repeated execution paths. Hardware support for such features could include instrumenting the Program Counter (PC) to check for repeated execution.

2. *Code generation and optimisation*:

   - Hot execution paths are translated and optimised from bytecode to machine code for the target device e.g. Graal [40]. Figure 2.3a shows an overview of the adaptive JIT compilation process.

   - Tracing JIT compilers trace, optimise and memoise hot execution paths rather than translating bytecode. Upon detection of a hot path, the VM stores all the binary instructions dispatched by the interpreter for execution. This instruction stream is optimised and guard conditions are inserted at points where control flow diverges. Tracing JIT compilers are used in systems like Dynamo (Bala et al [18]) and in the PyPy Python VM (Bolz et al [27]). Figure 2.3b provides a high level overview of a tracing JIT compiler.

ALPyNA (Chapter 4) targets loop nests for JIT compilation. Code is generated for CPU and GPU targets. Loop nests are analysed for parallel execution and compiled using the Numba compiler [78].

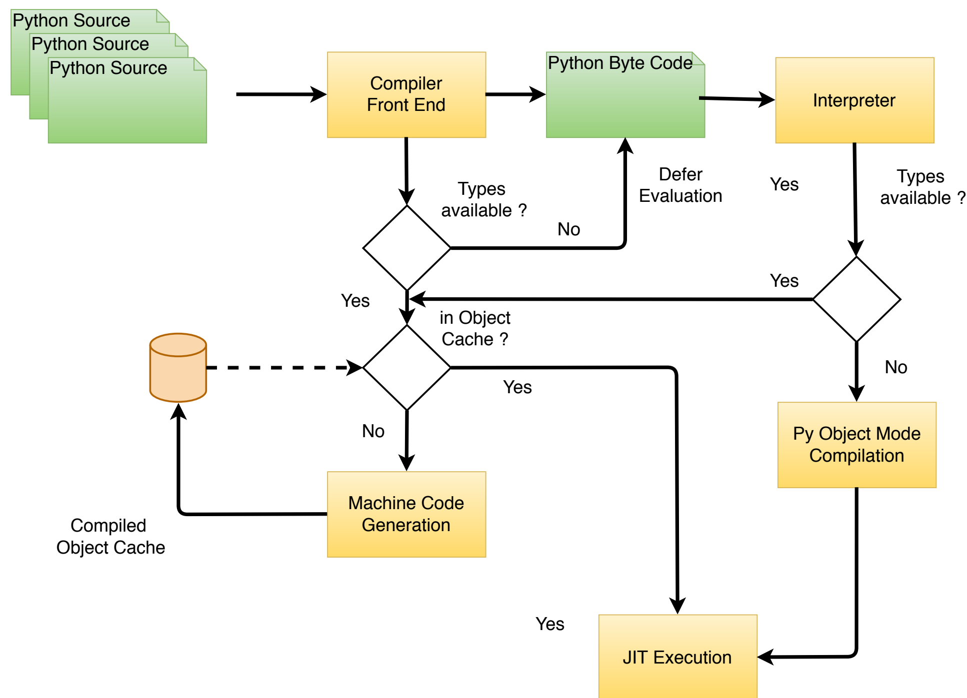



Figure 2.4: Numba's JIT compilation of annotated functions.

### 2.1.4 Numba – A JIT compiler for Python

**Numpy** Scientific calculations written in Python mandate performance oriented libraries and tools. Numpy [148] is a popular module used by Python programmers to perform statistical analysis and other numerically intensive computation. The primary data structure used by Numpy is a homogeneous multi-dimensional object (*ndarray*). The *ndarray* can store data of any fixed size numerical types like *uint32, int16, float32, float64* etc. Data is stored in contiguous memory. This property allows the underlying memory of the *ndarray* object to be accelerated in high performance computation libraries. These libraries are called from the interpreter using Python's Foreign Function Interface (FFI) bindings. Numpy functions are normally written using statically compiled high performance languages like C.

**Numba** is a high performance LLVM [79] based JIT compiler (Lam et al [78]). It supports JIT compilation of functions accessing elements of a Numpy *ndarray* with non-trivial indexing. Only functions marked for compilation using Python's decorator syntax are compiled.

Built-in FFI library functions that operate on Numpy arrays are optimised and fast. However, computation on Numpy arrays with complex indexing of *ndarray* elements is slow due to the heavy overhead of executing looping and control flow structures through the interpreter. Accelerating this code could be done by writing custom Python FFI functions. However this is arduous and error prone because reference counting for each object that is passed through

the Python Foreign Function Interface (FFI) is the responsibility of the developer. Numba eases this burden by directly analysing Python code and compiling it.

Numba is not a tracing JIT compiler. It is not intended to be used as a whole program compiler. It is instead intended to be used as an optional accelerating compiler for functions annotated by the developer. Ideally these functions are leaf functions because Numba does not recursively compile functions called from a decorated function. Numba uses the LLVM compiler tool-chain to compile to machine code. The LLVM IR requires information about the types of the data objects on which the computation is performed. Numba performs type inference on the data objects within a function before lowering it into LLVM IR. As code written in Python is dynamically typed, this inference occurs at runtime for each call to the function, unless the function is decorated with the type parameterised *@jit* call. Listings 2.1 and 2.2 show Numba annotations applied to compile functions for CPU and GPU targets respectively.

Listing 2.1: Numba JIT compilation of *saxpy* (Section 5.1) targeting the CPU

```
@jit('(float32[:], float32[:],
   float32[:], float32, int32)')
def saxpy(out, a, b, alpha,lim) :
 for i in range(0,lim):
   out[i] = alpha * a[i] + b[i]
```

Listing 2.2: Numba JIT compilation of *saxpy* (Section 5.1) targeting the GPU

```
@cuda.jit('(float32[:],
   float32[:], float32[:],
   float32, int32)')
def saxpy(out, a, b, alpha,lim) :
  tx = cuda.threadIdx.x
  bx = cuda.blockIdx.x
  bw = cuda.blockDim.x

  x = (bw * bx) + tx
  if x >= lim :
    return
  out[x] = alpha * a[x] + b[x]
```

Numba utilises LLVM's MCJIT functionality to compile and link JIT compiled machine code for the CPU. LLVM MCJIT objects allow very basic caching for compiled code. Numba exploits this property and memoises machine code compiled for each type signature of a method that the compiler encounters at runtime. To maintain API stability, Numba uses LLVM's C API bindings using a tiny wrapper library (*llvmlite*) rather than the C++ API.

Numba also supports JIT compilation targeting the GPU. Computational kernels written in a restricted form of Python, that exposes the underlying CUDA programming model, are compiled into NVVM IR [103]. This is then optimised and lowered into machine code targeting NVIDIA GPUs [1]. However, these compute kernels have to be expressed in GPU

[1]OpenCL backend was work in progress at the time of performing experiments. It has recently been merged into a stable version of Numba

programming language idioms (Section 2.3.2). ALPyNA relies on Numba's CUDA interface bindings to compile and execute auto-generated GPU kernels. Numba also exposes GPU intrinsics within compute kernels.

### 2.1.5 Section Summary

This section has introduced the basic working principles of VMs for dynamic langaguages with a particular emphasis on Python. A brief description of the inner workings of the CPython VM is provided. As the CPython VM only interprets a program, various JIT compilation techniques are used to speed up program execution. The work in this thesis is done within the context of a JIT compilation environment. Loop nests are, by definition, a repetitive control structure and are very often detected as hot-paths (Ardö et al [10]). Dependence analysis of loop nests inform an optimising compiler about the extent of parallel execution possible within a loop nest. Section 2.2 reviews dependence analysis theory and the techniques used to extract loop level parallelism within a loop nest.

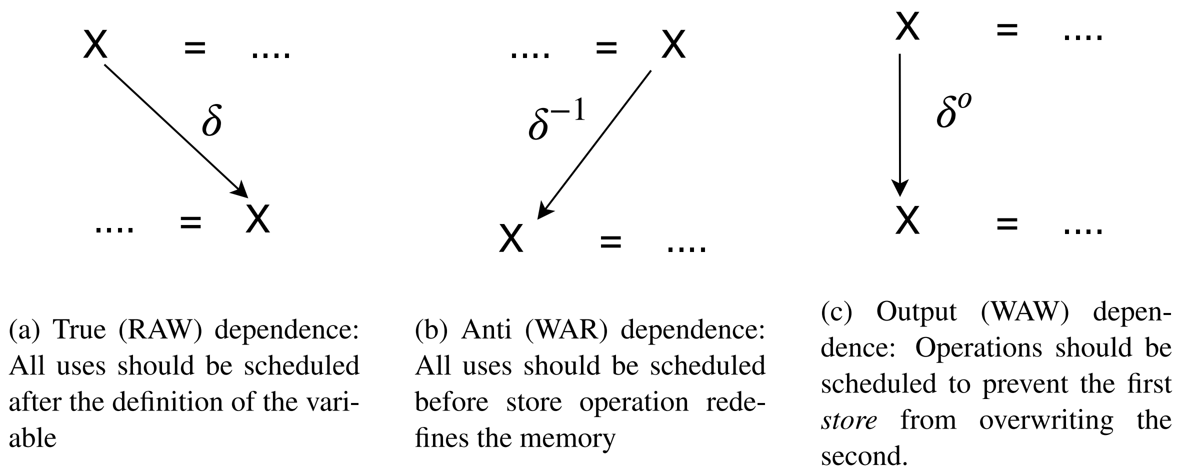
## 2.2 Dependence analysis

Reasoning about the correct execution schedule of statements in a loop nest requires analysis of the order of *writes to* and *reads from* the same memory location. ALPyNA's loop parallelisation analyses dependences to determine which loops in a loop nest can safely be executed in parallel (Section 4.2).

Allen and Kennedy [70], define the existence of a data dependence relationship between two statements in a region of code *iff* both statements access the same memory location and at least one of the operations is a write. A dependence relationship is defined to be established between a *source* and a *sink*. There are three basic dependence relationships based on the temporal order of the *Load–Store* classification of the source and sink.

1. *True* dependence ($\delta$): In any pair of *load–store* operations on a memory location, the earlier operation stores into memory (*definition*) followed by the load operation (*use*). The dependence relationship ensures that the value *used* by the load operation is not an older stale value. This dependence type is also known as a Read–after–Write (RAW) dependence (Figure 2.5a).

2. *Anti* dependence ($\delta^{-1}$): In a pair of *load–store* operations on a memory location, the earlier operation loads from memory followed by a store. This dependence relationship prevents a memory location being clobbered before every valid *use*. It is also known as a Write–after–Read (WAR) dependence (Figure 2.5b).




(a) True (RAW) dependence: All uses should be scheduled after the definition of the variable

(b) Anti (WAR) dependence: All uses should be scheduled before store operation redefines the memory

(c) Output (WAW) dependence: Operations should be scheduled to prevent the first *store* from overwriting the second.

Figure 2.5: Basic dependence types

3. *Output* dependence ($\delta^o$): If both operations on a memory location are *stores*, an output dependence captures the temporal order of these operations to prevent the later *store* operation from being clobbered by the earlier *store* operation due to a re-ordering of instructions. This dependence type is also known as a Write–after–Write (WAW) dependence (Figure 2.5c).

A *loop-carried* dependence occurs when a dependence source occurs on an earlier execution instance of a loop nest than the dependence sink. If both the source and the sink occur within the same execution instance of a loop nest, the dependence is classified as *loop independent*. A loop independent dependence determines the order of load–store operations within each loop execution instance. Theoretically, there is a dependence from each statement *instance* (identified as a dependence source) to another statement *instance* (identified as a dependence sink). However, the memory and time constraints required to capture every instance of a dependence is impractical.

A large category of numerical loop nest computation consists of the dereferencing of arrays, and this is the focus of ALPyNA's auto parallelisation. When array subscripts are linear functions of loop nest iterators, the dependences between the source and sink in each loop execution instance can be modelled precisely using formal mathematical and geometric representations [70, 76]. A dependence vector, indexed by the ordering of the loops in the loop nest, represents the dependences for each *source – sink* pair. The vector is indexed from the outermost to the innermost loop. If the source of a dependence occurs on iteration $i$ and the sink occurs on iteration $j$ of a loop, then the dependence between the source and sink is denoted as shown in Equation 2.1 [70, 158].

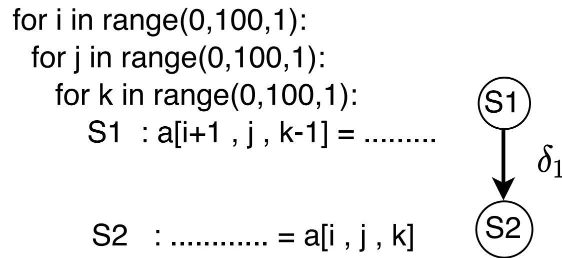



Figure 2.6: Example of loop carried dependences. Dependence vector for this loop nest is $[<,=,>]$

$$Dependence = \begin{cases} < & \text{if } (i \prec\ j) \\ = & \text{if } (i = j) \\ > & \text{if } (i \succ j) \end{cases} \tag{2.1}$$

The *level* of a loop carried dependence is denoted by the index of the leftmost non-'$=$' symbol in the dependence vector. The level of a dependence carried by loop '$p$' is denoted by $(\delta_p, \delta_p^{-1}, \delta_p^o)$ for *true,anti* and *output* dependences respectively. A loop independent dependence is denoted by $\delta_\infty$. Dependence analysis is an effective technique to extract loop level parallelism [52] when applied to loop nests and parameterised by such dependence vectors and loop levels. A dependence graph between statements in a loop nest helps to reason about the correctness of any scheduling transformations within loop nests. It determines the order in which loops should be executed to maintain the correct ordering of computation.

Consider the loop nest in Figure 2.6. The source of the dependence for array '*a*' is statement *S1* and the sink is statement *S2*. The dependence vector for relationship $S1 \rightarrow S2$ contributed by the array variable '*a*' is $(i, j, k) \mapsto [<,=,>]$. Intuitively, every memory location dereferenced by the first subscript at the source $(a[i+1])$ is used in the subsequent loop iteration at the sink $(a[i])$. Here the $i$-loop contributes a loop-carried *true* dependence $[<]$ between statement $S1$ and $S2$. The second subscript contributes a loop independent *true* dependence $[=]$ between $S1$ and $S2$. The third subscript contributes a loop-carried *anti* dependence $[>]$ between $S1$ and $S2$. The left-most non-"$=$" dependence is the loop-carried *true* dependence contributed by the $i$-loop. As this is the outermost loop, this dependence is deemed to be a level-1 loop-carried dependence from $S1$ to $S2$.

In general, a linear function of loop iterators within a subscript pair can be tested using Multiple-Index-Variable (MIV) tests. Banerjee's equations [20] provide a systematic MIV test for dependences between a pair of subscripts. Wolfe et al [158] provide an algorithm to

use these tests to hierarchically determine which dependences are valid and how these should be combined together for each variable pair. However, these tests are expensive [70]. The most common categories of subscript dependence tests are Single-Index-Variable (SIV) (both source and sink are referenced by the same single loop iterator) and Zero-Index-Variable (ZIV) tests. Simplified tests for SIV and ZIV subscripts cater to a large majority of numerical loop nest computations and are faster [52, 70]. This is especially useful in JIT compilation environments, which are the focus of this thesis.

Allen and Kennedy [70] prove that when a statement in a level-'$k$' loop nest is executed sequentially at level-$k$, then any re-ordering transformation to the inner loops is valid as long as they remain inner loops relative to loop-$k$. As a corollary, all such inner loops can also be parallelised. Dependence analysis of loop-nest statements provide a framework to reason about the schedule of operations and the amount of parallelism in the loop nest. Cycle detection within a built-up dependence graph of the whole loop nest will reveal the constraints on parallelising code. If no cycles are detected, then the statements can be parallelised. By executing outer level loops sequentially, an optimising algorithm can remove dependences at that level in the dependence graph and attempt to break cycles within the dependence graph. Breaking such cycles in the dependence graph allows for the execution of the remaining inner-level loops in parallel. Further structural loop transformations like loop interchange, loop fusion and loop fission allow code optimisation for various parameters like number of parallel threads, memory locality and execution order.

Since the 1980s these loop optimisation techniques have been researched and incorporated into high-performance parallelising toolchains for static compilation languages like FORTRAN. Generally, numerically intensive scientific code was developed in these languages [109]. As mentioned in Section 2.1.1, developers in various domains are now increasingly working with dynamic scripting languages such as Python. Exploiting the use of parallel accelerators such as GPUs can decrease execution time. However, this requires developers to know the hardware architecture of a GPU as well as the low-level programming semantics of CUDA / OpenCL (Section 2.3). As loop nests are a frequently occuring control structure in code written by such non-expert developers (Section 2.4), automatic loop parallelisation will greatly improve developer productivity. Leveraging dependence analysis enables parallelisation and automatic code generation for GPUs while maintaining data dependences. This motivation underpins the work described in this thesis.

## 2.3 Heterogeneous Compute Architectures

Generally, heterogeneous computer architectures refer to systems that contain different kinds of Computational Processing Elements (PEs) to perform a computational task. General pur-

pose machines are increasingly heterogeneous [162]. ALPyNA is intended to assist non-specialist programmers to exploit the performance available from heterogeneous platforms.

Flynn's taxonomy [44] classifies computer architectures based on the concurrency of *instruction* stream execution and access to *data* streams. Flynn defines an *instruction* stream as a sequence of intructions executed by a machine and a *data* stream as a sequence of data that an instruction works on. According to this taxonomy, the most prevalent[2] architectures are :

- A Single Instruction Single Data (SISD) machine executes each instruction in an instruction stream sequentially. The simplest form of an SISD executes instructions in the same order as the instruction stream is read (*in-order*). Performance due to memory access performance can be improved by instruction *pipelining* and *out-of-order* execution of instruction sequences.
- A Single Instruction Multiple Data (SIMD) machine executes each instruction in an instruction stream and concurrently executes that operation across multiple data points. SIMD machines are good at exploiting Data Level Parallelism (DLP). Vector extensions like SSE/AVX [86] and ARM NEON [11] on CPUs are examples of SIMD execution.
- A Multiple Instruction Multiple Data (MIMD) machine executes independent *instruction* streams and *data* streams concurrently. MIMD machines are suitable for execution of independent instruction streams in parallel. Data access to the same location from multiple instruction streams tend to degrade performance due to data synchronisation mechanisms. Modern multi-core CPUs (e.g. Intel Core–i5, AMD Ryzen Threadripper, Qualcomm Snapdragon, Intel Xeon Phi) are examples of MIMD machines. MIMD mahines, while being the most general case in Flynn's taxonomy are difficult to scale due to the complexity of data synchronisation between cores.

**Single Instruction Multiple Threads (SIMT)** Typical SIMD extensions to CPUs apply a single instruction to multiple data *lanes* concurrently. NVIDIA introduced the SIMT model [100] for their G80 microarchitecture. A SIMT machine is specially designed to exploit Data Level Parallelism (DLP) by applying an instruction to multiple threads that are executed simultaneously. Unlike a SIMD machine where a single instruction is applied to parallel lanes of a vectorised logical unit, in the SIMT model, each thread is executed on a single PE. If there are more threads than SIMT cores, these are executed as batches of threads by a hardware scheduler and time-sliced.

Heterogeneous computer architectures use a combination of a general purpose CPU and specialised compute accelerators. While heterogeneous systems come in a variety of forms,

[2]No practical use has been envisaged for a fourth machine type Multiple Instruction Single Data (MISD)

this thesis is focussed on JIT compilation for CPU–GPU architectures. Figure 2.7 shows a simplified view of a heterogeneous system with a multicore CPU and a discrete GPU.

Modern general purpose CPUs have three cache levels. Each core in a multicore CPU has its own L1 and L2 caches. All these cores in turn share access to an L3 cache to speed up memory access to main memory.

A discrete GPU communicates and transfers data from the main memory (DDR RAM) through the PCIe bus to its own global memory (usually GDDR RAM). Heterogeneous systems that combine a CPU with integrated GPUs on the same System-on-Chip (SoC) [37, 43] eliminate data transfer overheads by sharing main memory RAM. Main memory is reserved for use by the integrated GPU and data is transferred using a memory-to-memory transfer. The hardware architecture of a GPU is discussed in Section 2.3.1.

### 2.3.1 GPU hardware execution model

GPU hardware was originally designed to accelerate for the manipulation and rendering of images in memory. Modern graphics processing algorithms perform computation on hundreds of thousands of pixels in parallel. This stems from the nature of graphics workloads that have to compute and output images at the rate of 24–100 Frames per Second (FPS). A GPU device typically operates at lower frequencies compared to a CPU, but performs many computations in parallel. These performance characteristics make GPUs an attractive computational device not just for graphical applications but also for data parallel general purpose applications.

As the evaluation of the work in this thesis is done on NVIDIA GPUs, NVIDIA and CUDA terminology are employed to describe technical GPU terms. The basic compute unit of an NVIDIA GPU is a SIMT processor called a CUDA core. A group of such SIMT processors are grouped together to form a Streaming Multiprocessor (SM). A GPU can scale up the number of threads executed in parallel by adding SM units.

**Parallel compute** The GPU computational architecture is markedly different to that of a CPU. Consider the simplified GPU architecture shown in Figure 2.7. Lindholm et al [85] describe the basic architecture that is used in almost all modern GPUs. An NVIDIA GPU consists of a number of parallel compute multiprocessors named Streaming Multiprocessors (SMs). Each SM consists of a large number of compute PEs. NVIDIA calls these computational PEs CUDA cores. Each CUDA core executes a single thread at a time and has its own Program Counter (PC) and register set. Unlike traditional CPUs, threads are executed in parallel in batches known as *warps* (Wilt [157]). AMD uses the term *wavefront* [68] for

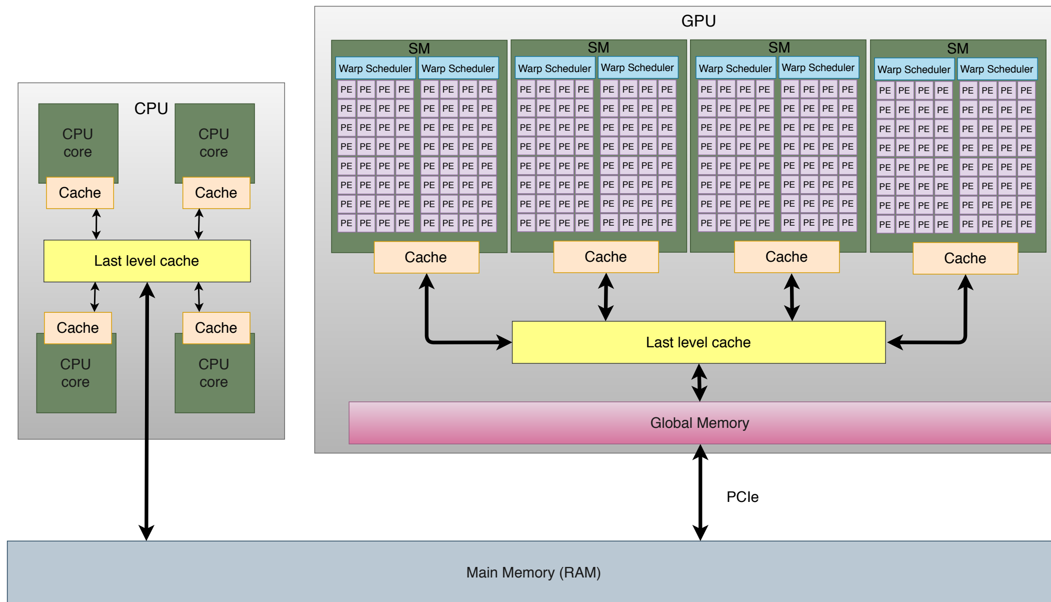



Figure 2.7: Simplified block diagram of a heterogeneous system with a CPU and a discrete GPU. The discrete GPU is connected to the host device through a PCIe bus.

their GPUs. In each warp, the threads executed belong to the same *thread–block* (Section 2.3.2).

Each *warp* executes the same instruction of a compiled GPU kernel per time-cycle in lock-step. To allow for control flow divergence between threads, a mask register is used to determine which threads have taken a branch. Every branch is evaluated for each thread in a warp and the mask register is used to conditionally enable instructions of a branch if that branch was taken in the corresponding thread.

Warps are scheduled to execute by *warp* schedulers. An SM may have multiple warp schedulers depending on how many physical CUDA cores it contains. If the number of threads in a *thread block* (Section 2.3.2) exceed the size of a warp, the hardware scheduler partitions the threads into warps. The threads in a warp are ordered in increasing order of their *thread-id* enumeration. The scheduler may swap the execution of a warp with another warp either from the same thread-block or from a different one altogether in a SIMT manner. Overlapping execution of warps in this manner is done to hide memory latency.

**Memory hierarchy** A discrete GPU is generally connected to the CPU in a heterogeneous system over the PCIe bus [7, 101]. Such GPUs will have access to its own dedicated memory accessible by every SM compute unit. Graphics memory devices (GDDR RAM) are optimised for high bandwidth applications.

Global memory is accessed through an L2 cache that is shared across all SM units within a

Listing 2.3: Matrix Addition in C

```
void MatAdd( float **a, float **b , float **c, int m, int n)
{
    for( i=0 ; i < m ; i++ ) {
        for( j=0 ; j < n ; j++ ) {
            c[i][j] = a[i][j] + b[i][j];
        }
    }
}
```

Listing 2.4: Matrix addition example as a CUDA GPU kernel written in CUDA–C

```
__global__ void MatAdd( float **a, float **b , float **c , int m, int n)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y ;
    int j = blockIdx.x * blockDim.x + threadIdx.x ;

    if( i < m && j < n )
            c[i][j] = a[i][j] + b[i][j];
}
```

GPU. Depending on the design of the GPU microarchitecture, a SIMT processor might have its own L1 cache or it might share its L1 cache with another SIMT processor within the same SM.

### 2.3.2 GPU Programming Model

Programming a GPU using CUDA or OpenCL comprises writing a **host** program that runs on the CPU and compute **kernels** that execute on accelerator devices like a GPU. The host manages the program work-flow and the accelerator resources. Compute kernels are written in a C-like low-level language. Such kernels generally make use of parallel idioms to locally dereference the execution context and to synchronise between parallel threads. These are usually implemented as compiler intrisics.

#### Kernel Programming Model

Parallel computation on the GPU is expressed as a data parallel computation kernel. Instances of a GPU kernel are executed in parallel threads. These parallel threads are mapped to an $N$–dimensional computational plane. In practice, GPUs are limited to three dimensions.

Listing 2.3 shows a function written in C that sequentially computes the addition of two matrices. A nested `for`-loop iterates over each element of input matrices and calculates each element of the output matrix. Listing 2.4 shows a CUDA kernel implementation of

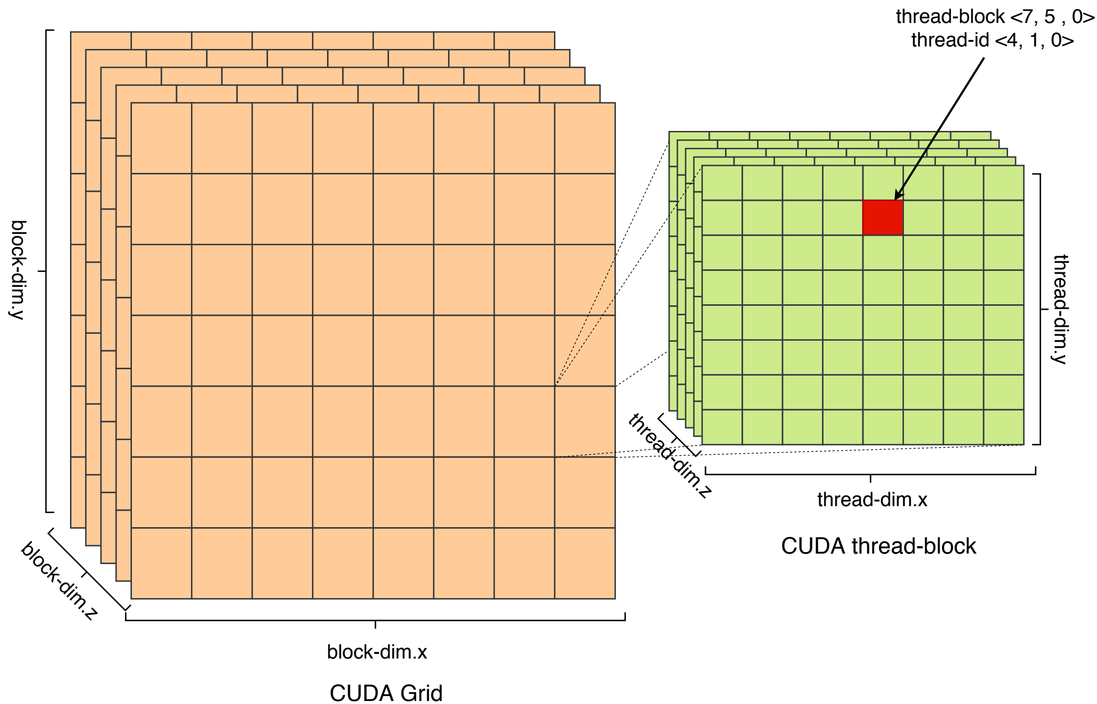



Figure 2.8: Thread hierarchy in CUDA is divided into a two-tier hierarchy of *grids* and *thread-blocks*. These are represented as vectors of up to three dimensions *(x,y,z)*.

the same computation. The kernel is written in CUDA-C and represents a single execution instance of the sequential loop nest in Listing 2.3.

Each element of the output matrix is computed by a unique thread. Threads are arranged into a two-tier hierarchical schema of **grids** and **thread–blocks**. A thread-block is comprised of an *N*–dimensional group of threads. Resource constraints of the actual SM unit dictate a maximum overall number of threads that can be executed on it. This varies between GPU architectures. Multiple thread– blocks of a homogenous size then make up a grid.

Figure 2.8 shows a 3–dimensional execution space denoted by its thread hierarchy tuple $(grid),(thread-block) > \mapsto (8,8,5),(8,8,4)$. The unique thread context executing a kernel instance is the unique combination of *thread–id*s along each dimension. A thread–id along any particular dimension can be dereferenced in each thread using $(blockid_{axis} * blocksize_{axis} + threadid)$ semantics. A thread-block is executed together on the same SM unit. This allows threads within a thread–block to access shared memory and use synchronisation primitives between them. However, threads in different thread-blocks cannot share memory.

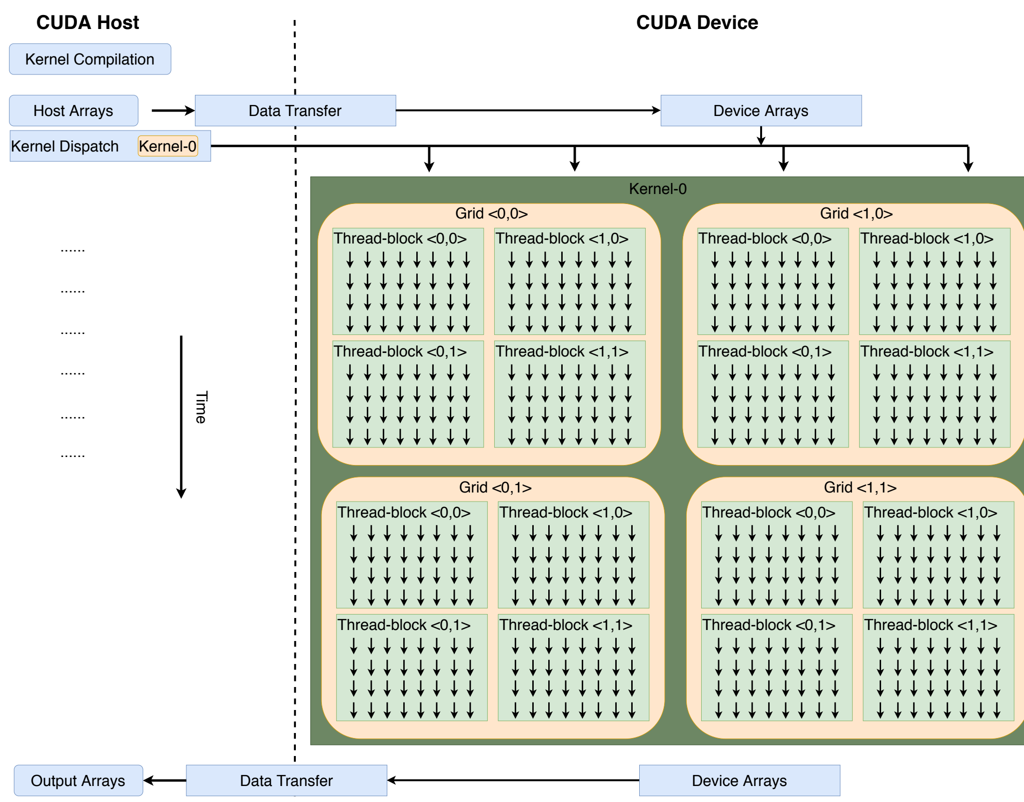



Figure 2.9: CUDA work-flow between host and accelerator device.

### Host Programming Model

CUDA kernels are compiled, setup and managed from the **host** device in a *master–slave* configuration. An example of such a setup would be a host CPU connected to a GPU device as shown in Figure 2.9. A program is initiated on the host. Memory arrays are allocated on the host and transferred by the host to the accelerator device.

Computational kernels written in CUDA are compiled to machine code targetted at the GPU device. In the CUDA and OpenCL programming models, there are two types of compilation processes. An **online** compiler generates machine code during the execution of the host program on host device. Unlike JIT compilers, this is only done when the host program explicitly invokes the compilation of a kernel. The OpenCL framework allows kernels to be compiled ahead-of-time by an **offline** compiler and invoked at runtime by the host program. In CUDA, such a mechanism is enforced by dynamically linking to a shared library.

The host program dispatches a compiled kernel to the accelerator device with its corresponding thread hierarchy (Section 2.3.2). Kernel computation on the GPU is executed asynchronously. If multiple kernels are invoked, they are queued for execution in the same order as they are invoked. After the kernel is executed on the GPU, the output data arrays are transferred back to the host. This programming model selectively accelerates data-parallel parts of the program on a GPU device while executing sequential code on the host CPU.

## 2.4 Importance of Loop Parallelisation

The majority of the execution time of many programs is spent executing loops (e.g. Ardö *et al* [10]). Widespread use of loops (particularly nested loops) as control-flow contructs in code point to their suitability for optimisation. This section shows that programmers write Python code in a manner amenable to loop dependence analysis and parallelisation.

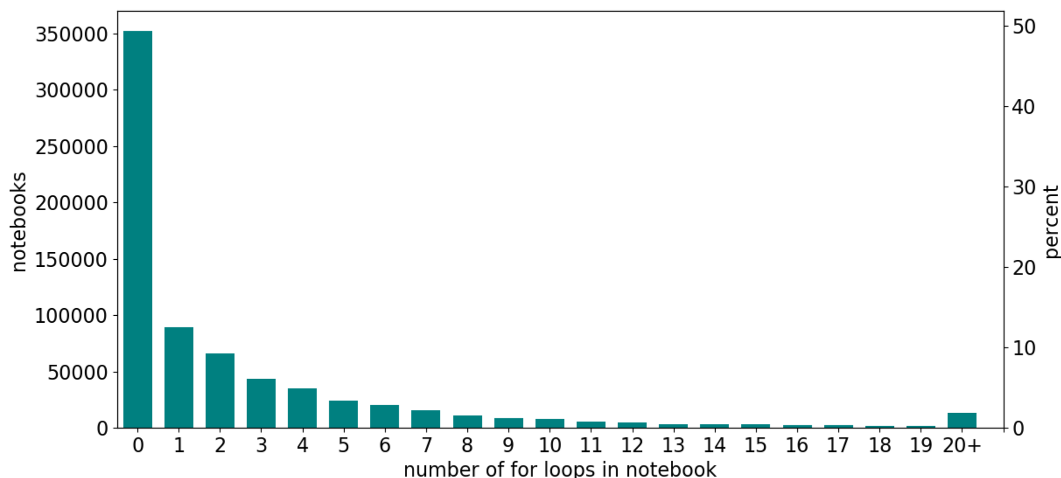



Figure 2.10: Histogram showing number of `for` loops per notebook in a sample of end-user code

To study the potential impact on developer productivity, a large corpus of Jupyter notebooks, downloaded from github by Rule et al [125], is studied. The whole corpus contains over 1.25 million Jupyter notebooks. comprising Python code snippets. Of these, 713745 could be parsed using a Python 3 parser (the rest were written in Python 2). This parser is written using Python's `ast` module. The `ast` Python module allows for the sub-classing of each node within the Python AST. The analysed code samples are representative of developer programming style, perhaps skewed to the scientific domain.

Each notebook is parsed to obtain the number of distinct `for` loops in each notebook. The depth of each loop nest is also identified. The depth of a loop nest provides the possible number of loops which may be parallelised. Figure 2.10 shows that approximately half the notebooks had loop nests within them. Figure 2.11 shows the maximum nesting depth that is found in each Jupyter notebook, and the median depth is 1 while the maximum depth is 14.

This motivating study shows the potential to widely apply loop parallelisation techniques in end-user Python code. Automatic parallelisation and appropriate JIT compilation of loop nests provide a transparent method of accelerating code in a heterogeneous environment. Chapter 4 describes the design of ALPyNA which addresses automatic parallelisation of such loop nests.

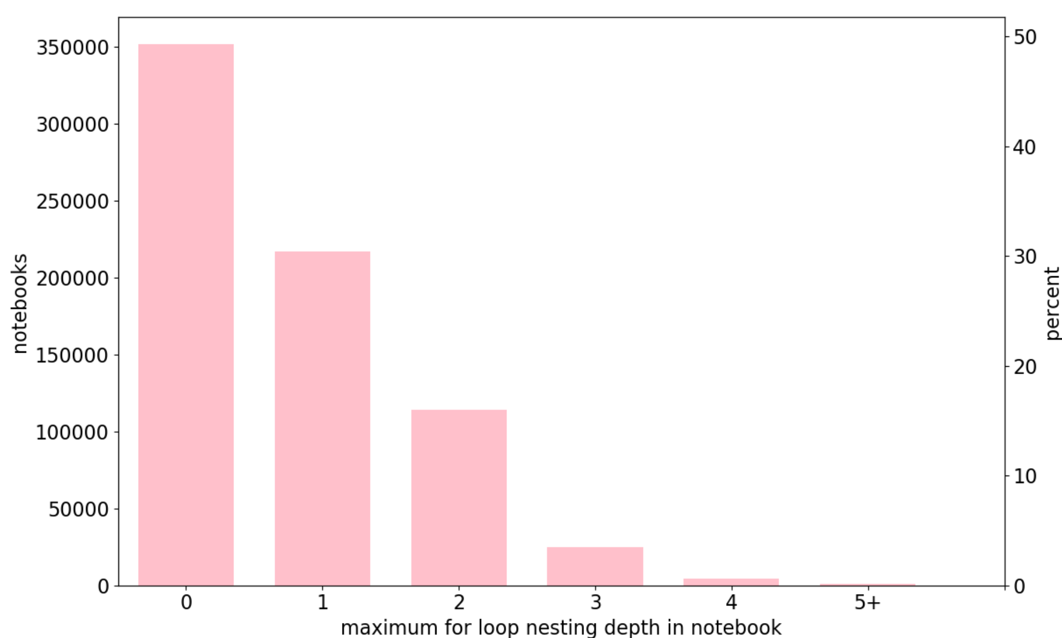



Figure 2.11: Histogram showing the maximum `for` loop nesting depth per notebook

## 2.5 Summary

This chapter has presented the theory and terminology required to understand the rest of the material in this thesis. It brings together disparate areas of compiler and runtime technologies and serves as a foundation for the research.

Section 2.1 introduced the execution of a managed dynamic language in a VM and speeding up of such code using various JIT compilation techniques. In the context of Python, the runtime features of the CPython VM is explored. The Numba compiler which uses developer insight to accelerate specific parts of code within a program is introduced. The Numba compiler is used to expose the underlying GPU CUDA framework to the ALPyNA code generator as well as to JIT compile and execute CPU and GPU variants of hot code fragments.

The theory and terminology used for dependence analysis of loop nests and the basis of parallelisation is introduced in Section 2.2. These have been used in compilers for static languages for many years. Applying this framework to a dynamic runtime environment has different challenges and opportunities as we show in the design of ALPyNA (Chapter 4). Section 2.3 introduced the core ideas and the programming model of GPU programming. Understanding the SIMT parallel hardware architecture and the CUDA based execution model of data parallel programs is necessary to exploit the performance available from modern GPUs.

This chapter ends with a study analysing the popularity of loop nests in the programming styles of various developers. We show that roughly half the code fragments of a very large

corpus of publicly available Python notebooks contain code that can be targetted by the contributions in this thesis. The next chapter takes a critical look at the state-of-the-art technologies as used in this thesis.

# Chapter 3

# Literature Survey

The proliferation of heterogeneous compute architectures has led to a varied set of tools and approaches to enable programmers to harness them. Developing performant code for accelerators remains complex and presents a steep learning curve, especially for non-expert programmers. The rising popularity of managed languages and runtimes such as Python (Bissyandé *et al* [25]) and libraries (Virtanen *et al* [153]) for compute intensive calculations points to the need for easy programmability of heterogeneous architectures. The primary aim of this dissertation is to aid non-expert programmers to access accelerator performance. This chapter discusses current research into heterogeneous programming and the various technologies involved to automatically accelerate code in dynamic languages.

This dissertation primarily targets acceleration of loop nests in Python on CPU–GPU systems. Section 3.1 discusses alternate VMs and techniques used to speed-up Python applications. Section 3.2 discusses directive based JIT compilation and bindings to heterogeneous programming languages in dynamic languages. Section 3.3 introduces the theory behind loop parallelisation and explores contemporary loop parallelisation frameworks in runtimes and JIT compilers for heterogeneous devices. Section 3.4 outlines functional programming and acceleration of algorithmic skeletons programming on GPU hardware. Section 3.5 discusses current research into parallelising Domain Specific Languages (DSLs) that enable parallelisation. These enable expert programmers working with high level abstractions to guide compiler optimisation. Section 3.6 introduces various Intermediate Representations (IR) where parallelism is built in by design. Section 3.7 describes various analytical and machine-learning approaches to predicting the optimal device, load-balancing and deriving the cost of executing computation in a heterogeneous environment. Finally Section 3.8 concludes and summarises this chapter.

To provide a systematic overview of all the work that is discussed in this chapter, we have characterised each work based on the primary language targeted, the type system, compilation type, cost awareness, the level of manual intervention required and the accelerator

| System | Language | Type System | Compilation Type | Cost Awareness | Programmer Intervention | Parallelism Resolution | Target Devices |
|---|---|---|---|---|---|---|---|
| **Virtual Machines** (Section 3.1) | | | | | | | |
| PyPy [10, 27, 94, 110] | Python | Dynamic | JIT | Hot path detection | Automatic | Runtime | CPU–SIMD |
| Pyston [96] | Python | Dynamic | JIT | Hot path detection | Automatic | Runtime | CPU–SIMD |
| Cython [23] | Python | Static | Static | – | Explicit | Compile time | CPU–SIMD |
| Jython [67] | Python | Dynamic | JIT | Hot path detection | Automatic | Runtime | CPU–SIMD |
| Truffle [161] | Ruby, R, Python | Dynamic | JIT | – | Automatic | Runtime | CPU, GPU, FPGA |
| **Library bindings and Directive based JIT acceleration** (Section 3.2) | | | | | | | |
| Numba [78] | Python | Static / Dynamic | JIT | – | Explicit | Compile time | CPU, GPU |
| PyCUDA / PyOpenCL [75] | Python | Static | JIT | – | Explicit | Compile time | CPU, GPU |
| Jibaja *et al* [64] | JavaScript | Dynamic | JIT | – | Explicit | Runtime | CPU–SIMD |
| **Loop parallelisation** (Section 3.3) | | | | | | | |
| Megaguards [115] | Python | Dynamic | JIT | – | Automatic | Runtime | CPU, GPU |
| Three Fingered Jack TFJ [127, 128] | Python | Dynamic | JIT | – | Automatic | Compile time | FPGA |
| ALPyNA [60, 61] | Python | Dynamic | JIT | Analytical | Automatic | Runtime | CPU, GPU |
| Paragon [126] | C/C++ | Static | Static | – | Automatic | Runtime | GPU |
| Wang *et al* [156] | C/C++ | Static | Static | – | Automatic | Runtime | GPU |
| Leung *et al* [82] | Java | Static | JIT | Analytical | Automatic | Runtime | GPU |
| Tornado [36, 46] | Java | Static | JIT | – | Explicit | Runtime | CPU, GPU, FPGA |
| Apollo [29, 140] | LLVM IR | Static | JIT | Profiling | Automatic | Runtime | CPU |
| **Algorithmic skeletons** ( Section 3.4) | | | | | | | |
| Copperhead [31]) | Python | Dynamic | JIT | – | Explicit | Compile time | GPU |
| Parakeet [124] | Python | Dynamic | JIT | – | Explicit | Compile time | GPU |
| Ishizaki *et al* [59] | Java Streams | Static | JIT | – | Implicit | Runtime | GPU |
| Ikra [92] | Ruby | Dynamic | JIT | – | Explicit | Compile time | GPU |
| Fumero *et al* [48, 49] | Java | Static | JIT | – | Explicit | Compile time | CPU, GPU |
| Lime compiler [39]) | Java | Static | JIT | – | Explicit | Compile time | CPU, GPU |
| Dandelion [121] | .NET LINQ | Static | JIT | – | Explicit | Compile time | CPU, GPU, distributed |
| RiverTrail [56] | JavaScript | Dynamic | JIT | – | Explicit | Compile time | CPU, GPU |
| ParallelJS [154] | JavaScript | Dynamic | JIT | – | Explicit | Runtime | CPU, GPU |
| Fumero *et al* [47] | JavaScript | Dynamic | JIT | Hot path detection | Implicit | Runtime | CPU, GPU |
| Lift [136, 137] | Lift | Static | Static | – | Implicit | Compile time | CPU, GPU |
| Futhark [55] | Futhark | Static | Static | – | Implicit | Compile time | CPU, GPU |
| LambdaJIT [89]) | C++ | Static | JIT | – | Automatic | Compile time | CPU, GPU |
| **Embedded DSL** (Section 3.5) | | | | | | | |
| SEJITS / Asp [32, 69] | Python (Host) | Dynamic | JIT | – | Semi-auto | Compile time | CPU, GPU |
| Delite [138] | Scala (Host) | Static | JIT | – | Semi-auto | Runtime | CPU, GPU |
| SYCL [117, 118] | C++ (Host) | Static | Static | – | Explicit | Compile time | CPU, GPU, FPGA |
| LooPy [74] | Python (Host) | Dynamic | JIT | – | Semi-auto | Compile time | CPU, GPU |
| **Intermediate Representation (IR)** (Section 3.6) | | | | | | | |
| SPIR-V [72] | SPIR-V IR | Static | Static / JIT | – | Semi-auto | Compile time | CPU, GPU, FPGA |
| INSPIRE [66] | INSPIRE IR | Static | Static / JIT | – | Semi-auto | Compile time | CPU, GPU |
| HSAIL [6] | HSA IR | Static | Static / JIT | – | Semi-auto | Compile time | CPU, GPU |
| **Cost Models** (Section 3.7) | | | | | | | |
| Armih *et al* [12, 13] | C/C++ | Static | Static | Analytical | Automatic | Compile time | CPU, GPU |
| AJITPar [91] | Racket | Dynamic | JIT | Analytical | Automatic | Runtime | CPU |
| MWP–CWP [57, 130] | – | Static | Static / JIT | Analytical | Automatic | Runtime | GPU |
| Chikin [34, 35] | C/C++ | Static | Static | Analytical | Automatic | Runtime | GPU |
| 3DPP [163] | CUDA, OpenCL | Static | Static | Analytical | Automatic | Compile time | GPU |
| Kennedy *et al* [71] | FORTRAN | Static | Static | Analytical | Automatic | Compile time | CPU |
| Wang *et al* [155] | OpenCL | Static | Static | Machine learning | Automatic | Runtime | CPU, GPU |
| XAPP [9] | – | Static | Static | Machine learning | Automatic | Runtime | CPU, GPU |
| Wu *et al* [160] | – | Static | Static | Machine learning | Automatic | Runtime | GPU |

Table 3.1: Overview of heterogenous compilation technologies

devices supported. This characterisation is presented in Table 3.1. Only accelerator devices connected to a host device by a hardware bus are taken into consideration and hence distributed computational models are not discussed. The level of intervention required from the programmer to identify parallelism and exploit acceleration is based on the classification of parallel tools made by Vandierendonck and Mens [149], as follows :

- **Explicit parallelisation** : The programmer explicitly states the code constructs that are targeted for analysis and optimisation.
- **Implicit parallelisation** : The features of constructs used within a tool, framework or language can be implicitly targeted to analyse for parallelism.
- **Automatic parallelisation** : Parallel processing is extracted from sequential statements in the source language.
- **Semi-automatic parallelisation** : Tools guide an expert programmer to aid parallelisation and optimisation.

'*Parallelism resolution*' indicates whether the decision to use a target device is taken at '*compile time*' or '*runtime*'. A *compile time* resolution does not necessarily mean that the compilation of a parallel kernel happens at compile time but rather that the decision was taken ahead of time. The term '*static*' is overloaded when used to classify the systems in Table 3.1. When referring to typing systems, programming languages may be statically or dynamically typed. When referring to compilation, these systems may be statically compiled or JIT compiled.

## 3.1 Python JIT compilation frameworks

The most popular implementation of Python is the CPython reference implementation (Section 2.1.1). However, CPython only uses an interpreter. PyPy [119] is an alternative VM for the execution of Python programs with an emphasis on performance. Bolz *et al* [27] first proposed the use of a tracing JIT compiler to speed up the PyPy implementation of Python. A light-weight counter keeps track of the number of times a backward edge is taken in a loop. When a hot-loop is identified, the interpreter traces and memoises all the instructions that are executed in an instance of the loop. Guard conditions are placed at all locations where control-flow may diverge. Crucially the tracing JIT compiler traces the bytecode interpreter and not the actual program itself. To ensure that the user program rather than the interpreter is being traced, the interpreter is provided with hints about when a hot-loop has occurred in the user program. The traced assembly instructions are optimised and executed until the loop terminates or a guard condition has been violated.

Meier *et al* [94] have proposed various solutions such as fine-grained locking, and transactional memory to remove the limitation of the GIL in PyPy. Loop optimisations within the PyPy tracing JIT have been proposed by Ardö *et al* [10]. These optimisations centre around loop-invariant code motion being applied with the clever use of loop-peeling in the JIT trace. When combined with other loop optimisations such as redundant guard condition removal and common subexpression elimination, significant improvement is shown in small loop bodies. Plangger *et al* [110] implemented loop vectorisation by unrolling loops, tracing the dependences between instructions and searching for groups of instructions to combine into SIMD instructions on the target machine. This produced a speed-up of between 0.96x–2.1x over the non-vector version of the benchmarks.

Pyston (Modzelewski *et al* [96]) is a performance oriented Python VM initially developed by DropBox. It uses a two-tier JIT compilation process to execute hot paths. A two-tier JIT compiler keeps track of the number of times a hot-path is executed and uses two thresholds to trigger each compilation phase. A baseline JIT compilation phase (BJIT) is executed when a hot-path detection counter crosses a lower threshold. Pyston's BJIT compiler is a lightly optimising compiler that is strongly coupled to the interpreter. In this phase all objects are boxed as Python objects and no type speculation is performed for the code. During execution by the BJIT compiler, type information is gathered for each object. When the hot-path detection counter crosses the higher threshold, the types inferred by the BJIT compiler are passed on to an LLVM [79] based JIT compiler. Many variants of code are generated with each type gathered from the previous BJIT compilation phase. A guard condition violation in this tier will check for another variant that was compiled and cached. If such a variant is not available, execution will fall back to the BJIT execution variant. As LLVM has a large number of optimisation passes and the IR generated is typed, aggressive optimisation can be performed. The two tier JIT compilation process enables the VM to utilise fast compilation for shorter execution paths while performing heavy optimisations on the most frequently executed hot-paths.

Cython (Behnel *et al* [23]) is a typed language extension to Python. Cython compiles Python code annotated with C-types to C. The C code is further compiled and linked against the Python runtime. Cython performs type evaluation on each object in a function from a set of types initially declared for some of the variables. If this is not possible, object types fall back to dynamic Python types. The compiled code integrates the C and Python runtimes natively within the language constructs itself. Runtime calls between the C-runtime and Python are integrated into the language itself and requires no manual intervention from the programmer. Typical loop computational patterns are recognised and converted to their equivalent C/C++ versions after being aggressively optimised if the type system allows it. Scientific and numeric computation libraries such as SciPy, pandas and scikit-learn [108] use Cython for performance enhancement. If the programmer explicitly releases the GIL and guaran-

tees that no boxed Python object is accessed within these regions of code, multi-core CPU parallel code is generated using OpenMP.

Python has also been implemented using the VMs of other languages. Jython (Juneau *et al* [67]) is an implementation of Python that uses the Java VM. It can extend the Python runtime with either the Java or the C runtimes using the Java FFI. As the Python interpreter is written in Java, the JVM acts as a meta-interpreter and accelerates the hotpaths within the Python interpreter. The underlying Java VM is truly multi-threaded. This is an advantage over CPython where multi-threaded performance is limited by the GIL. The underlying Java VM also has the advantage of using the more sophisticated garbage collection algorithms that come as standard with the JVM.

Building optimising managed runtimes for dynamic languages is complicated and error prone. The OpenJDK project provides a rich feature set to programmers developing language VMs. Truffle (Würthinger *et al* [161]) optimises interpreter execution by partial evaluation of the program AST. When a hot path is identified within the interpreter, Truffle partially specialises the AST to reduce the overhead of execution. Truffle is paired with Graal [40], an optimising JIT compiler. Graal transforms the partially specialised code to a specialised IR. Truffle annotates nodes that enable control divergence with their execution probabilities and frequency to tailor the number of compiler optimisation passes on-demand. Truffle and Graal are widely used as a framework to accelerate a number of dynamic languages such as Ruby, R, JavaScript and Python.

**Limitations** Tracing JIT compilers such as PyPy, Pyston and Jython accelerate dynamic languages by tracing the binary instructions that have already been executed on the CPU and optimising the hot paths identified. However, the parallel SIMT execution model (Section 2.3) of GPUs requires computation to be explicitly written with parallel thread semantics. Implementing a tracing JIT in the presence of architecturally different devices such as GPUs is difficult due to the complexity of translating a binary trace of CPU instructions to the SIMT model of GPUs. Other approaches such as Cython require the programmer to utilise C-types to compile code statically. The compiler can only parallelise simple loops targeting multi-core CPUs and does not take into consideration optimisation opportunities that arise from runtime dependence analysis.

## 3.2 Library bindings and Directive based JIT acceleration

Numba (Lam *et al* [78]) is a JIT compiler for Python. Numba compiles and executes Python code specifically identified by the programmer using decorator syntax. Unlike Copperhead

and Parakeet (Section 3.4), which concentrate on accelerating higher order functions, Numba is capable of compiling a larger subset of a Python program. Numba uses the LLVM compiler toolchain to compile code targeting the CPU. Numba can automatically deduce the types required to compile a function at runtime or it can defer to the programmer to compile a function with its types provided. It can also compile and vectorise `ufuncs` [1].

Listing 3.1: Numba JIT compilation of *saxpy* (Section 5.1) targeting the CPU

```
@jit('(float32[:], float32[:],
   float32[:], float32, int32)')
def saxpy(out, a, b, alpha,lim) :
 for i in range(0,lim):
   out[i] = alpha * a[i] + b[i]
```

Listing 3.2: Numba JIT compilation of *saxpy* (Section 5.1) targeting the GPU

```
@cuda.jit('(float32[:],
   float32[:], float32[:],
   float32, int32)')
def saxpy(out, a, b, alpha,lim) :
  tx = cuda.threadIdx.x
  bx = cuda.blockIdx.x
  bw = cuda.blockDim.x

  x = (bw * bx) + tx
  if x >= lim :
    return
  out[x] = alpha * a[x] + b[x]
```

Numba also supports GPU compute capabilities. Numba enables the exploitation of GPUs by exposing GPU thread and synchronisation primitives to the programmer. This enables programmers to write GPU kernels in idiomatic CUDA. Therefore programmers must be aware of the low level SIMT programming model. Listing 3.1 and 3.2 show examples using JIT compilation directives to compile the *saxpy* benchmark targeting the CPU and GPU respectively. ALPyNA uses the Numba compiler tool-chain to JIT compile and execute the code generated at runtime for the CPU and GPU targets (Section 4.4). Further details about Numba are explained in Section 2.1.4.

PyCUDA and PyOpenCL (Klöckner *et al* [75]) provide a thin abstraction layer for Python programmers to execute code on accelerators such as GPUs. The compilation, scheduling of kernels and data transfer can be finely controlled in the host using Python syntax. The kernels themselves are written in CUDA and OpenCL, and passed as strings to the online compiler. The numerical arrays used with these kernels are *numpy* arrays, i.e. typed arrays with data stored in contiguous aligned memory locations. This makes it easier to inter-operate with the CUDA and OpenCL runtimes and to transfer memory to a GPU without the need for marshalling. Listing 3.3 shows the *saxpy* kernel passed to the CUDA compiler as a C-string and called from the Python interpreter.

[1] A Python `ufunc` (*U*niversal *Func*tion) executes the body of the computation on Numpy *ndarrays* (n-dimensional arrays) in an element-by-element manner.

Listing 3.3: CUDA implementation of *saxpy* in PyCUDA

```
# out,a,b  are numpy arrays
a_gpu = cuda.mem_alloc(a.nbytes)
b_gpu = cuda.mem_alloc(b.nbytes)
out_gpu = cuda.mem_alloc(out.nbytes)
cuda.memcpy_htod(a_gpu,a)
cuda.memcpy_htod(b_gpu,b)
cuda.memcpy_htod(out_gpu,out)

mod = cuda.SourceModule("""
           __global__ void saxpy(float *out, float *a, float *b,
                                              float alpha, int size)
           {
             int tid = blockIdx.x * blockDim.x + threadIdx.x ;
             if( tid >= size )
                 return ;
             out[tid] = alpha * a[tid] + b[tid];
           }
           """)
func =  mode.get_function(saxpy)
func(out_gpu,a_gpu,b_gpu,alpha, a.size),
     block=blk_size,thread_block_size=tblk_size)
cuda.memcpy_dtoh(out, out_gpu)
```

Jibaja *et al* [64] proposed new SIMD vector data types in JavaScript. Code is type specialised at runtime and SIMD method calls on these data types are speculatively called with a guard condition for the deoptimisation case. Frequently executed code is then inlined aggressively. These SIMD types are implemented for use within the SpiderMonkey and V8, the JavaScript engines of the Firefox and Chromium browsers respectively.

**Limitations** Although low level bindings maintain most of the low-level flexibility of programming an accelerator, programming such devices is intricate and error prone and expert knowledge of the underlying hardware is required. These directive based compiler approaches also require type annotations although the host language is dynamically typed.

## 3.3 Loop parallelisation

Automatic loop parallelisation seeks to extract and exploit parallelism within loop nests while maintaining all valid schedules of statement execution as intended by the programmer. Dependence analysis is done by expressing the memory *load–store* access patterns in

a loop nest. In the case of arrays, memory access patterns that are linear relationships of the loop nest iterators are classified as Single-Index-Variable (SIV), Zero-Index-Variable (ZIV), Multiple-Index-Variable (MIV) or non-linear (Section 2.2). Such tests were popularised by Wolfe *et al* [158], Banerjee *et al* [20] and Pugh [114]. Allen and Kennedy [70] pioneered the notion of parallelising loop nests by detecting cycles in a dependence graph and mapping them to the loops causing these dependences.

The Polyhedral model [21,106] represents the loop domain ranges and dependences between statements in a compact matrix like structure. The abstraction is based on the representation of all memory accesses as a polytope. Loop transformations can be succinctly explored within the polyhedral model where the domain ranges are known. Well known polyhedral tools to analyse dependences and optimise loop nests include GRAPHITE (Pop *et al* [111]) for GCC and Polly (Grosser *et al* [53]) for the LLVM compiler.

### 3.3.1 Loop parallelisation in managed language runtimes

Loop parallelism for GPUs and other accelerators has been researched in the static compilers extensively. Their use in dynamic languages and JIT compilation environments presents both difficulties and opportunities. Loop parallelisation and compilation of GPU kernels requires a type system supported by accelerators. This is only available at runtime for dynamic languages. However, dynamic instrospection and runtime dependence resolution in managed languages allows greater parallelisation opportunities.

Megaguards (Qunaibit *et al* [115]) is a loop parallelisation framework implemented on ZipPy [164]. ZipPy is a Python3 implementation built on top of a Java Virtual Machine which uses Graal and Truffle (Würthinger *et al* [161]) as the JIT compilation and runtime optimisation framework. Megaguards treats a loop nest as a static region with no mis-speculations. To ensure the correctness of this approach, Megaguards attempts to lift and combine all type related guard conditions outside the loop nest. If the type-stability test fails, the loop nest is executed in the interpreter. However, if type-stability tests and bounds checks for each array succeed within the scope of the loop nest, these checks are lifted out of the loop nest body. Bounds checks can only be checked for linear indexing of array elements. Megaguards performs inter-procedural analysis on each function call within a loop nest and compiles a type specialised variant for each function. When the safety of lifting these guard conditions is ascertained, the loop nest is analysed for dependences. If no cross-iteration dependences are detected, OpenCL kernels representing the loop nest are generated to execute on a GPU. If cross-iteration dependences are detected, sequential code is executed on the CPU using Graal. Dependence testing is done leveraging Polyhedral Extraction Tool (PET) [152]. Alias analysis is performed at runtime on array references to ensure that dependence testing using

PET is not erroneous. By not parallelising loop nests with cross-iteration dependences, opportunities to optimise loop nests where the outer loop carries a dependence will be missed.

Three Fingered Jack – TFJ (Sheffield *et al* [127, 128]) is a loop parallelisation framework for Python. It is an embedded DSL supporting a subset of the Python language constructs written as loop nests. It is not a whole program compiler and uses decorator syntax on user defined functions to optimise and generate code. TFJ supports a simple type system on loop nests with array semantics and no control flow divergence. At compile time, dependence analysis is performed on a TFJ decorated function with loop nests and transformed into an XML based intermediate representation. A greedy approach is taken towards parallelisation as popularised by Allen and Kennedy [70]. The algorithm is tuned towards reordering loops so that the outer loop has a large iteration domain while trying to maintain unit-stride memory access. At runtime, function arguments are checked again. If a compiled variant is already cached it is executed. However, if a valid combination is not in the code cache, it is compiled with the correct types. Otherwise execution falls back onto the interpreter for unsupported type constructs. For CPU variants, C++ code using SIMD vector intrinsics are generated. TFJ can also perform hardware synthesis to execute code on an FPGA.

Paragon (Samadi *et al* [126]) speculatively generates parallel CUDA GPU kernels from potentially parallelisable loop nests. Such loop nests are identified by profiling before compiling the code. For each unit of work that is potentially parallelisable, Paragon maintains checkpoints to be able to roll back computations if the guard conditions in the parallel computational variant are violated. Correctness is checked by maintaining a log of load and store accesses for each memory location accessed in parallel on the GPU. At the end of kernel execution, a supplementary kernel checks each memory location in the log. More than one *store* operation or a *store* and *load* operation to any memory location triggers the conflict flag indicating that the sequential execution of the loop nest will be chosen.

Wang *et al* [156] improve upon the speculative loop parallelisation approach of Paragon by profiling loop execution and being more sophisticated in detection of dependences. Static analysis separates loops that are definitely sequential or parallel from loops that are potentially parallel but conservatively marked as sequential. The code is profiled to obtain a level of confidence regarding such loops. Parallel kernels are generated to execute on a GPU along with guard conditions. At runtime, both the parallel and sequential program are simultaneously executed. The computational results from the variant that successfully completes the execution first is utilised. Dependence checking is done in-place during execution to check for dependence violation of the speculatively parallelised loop. A dependence violation triggers the killing of the parallel execution variant and the result from the sequential variant is committed. This system is more precise in the detection of dependence violations than Paragon by checking the order of memory accesses as opposed to just naively checking whether more than one access for a memory location has occurred. Unlike Paragon, par-

allelisation is permitted even in the presence of cross iteration dependences as long as the dependence constraint is not violated. It is also less memory intensive as it does not maintain a log of memory locations that could cause potential dependence violations.

JikesRVM [4] is a managed Java runtime with a two tier JIT compilation process. It does not have an interpreter stage and compiles all code. Leung *et al* [82] implement automatic loop parallelisation in the second JIT compilation stage. The second stage targets code that is computationally heavy. Parallelism that is identified in such a loop nest is used to generate the driving code on the host CPU and parallel kernels on the GPU. A simple parameterised cost model based on profiling is proposed to analyse the cost of offloading computation to a GPU (discussed further in Section 3.7). Loops within a loop nest are identified as being CPU bound, *implicitly* GPU bound or explicitly GPU bound. Loops identified as implicit GPU loops are parallelised as threads while explicit GPU loops are executed sequentially within a kernel. Implicit GPU loop nests are constrained to be perfect loop nests. Any loop carried dependences between instructions in a loop body are not parallelised even if the dependence is carried by an outer loop. As dependence analysis is performed on the byte code representation of a loop nest, control flow has to be recovered from the byte code before the analysis stage to correctly generate GPU kernels. Runtime array bounds verification is performed before offloading computation to ensure no exceptions are raised by GPU execution. All array subscripts are constrained to be of the ZIV or SIV form (Section 2.2). Any bounds check violation triggers the execution of the loop nest computation on the CPU. The limitation of parallelising only perfectly nested loops with no loop carried dependences may hinder loop parallelisation opportunities for execution on GPUs.

Listing 3.4: Tornado JIT compilation of *saxpy* targeting the GPU

```
public class Compute {
    public void saxpy(int[] out, int[] a, int[] b, int alpha) {
        for (@Parallel int i = 0; i < out.length; i++) {
            out[i] = alpha * a[i] + b[i];
        }
    }

    public void compute(int[] out, int[] a, int[] b, int alpha) {
        TaskSchedule s = new TaskSchedule("s0")
                              .task("t0", this::saxpy, out, a, b, alpha)
                              .streamOut(out).execute();
    }
}
```

Tornado (Clarkson *et al* [36]) transforms and compiles data parallel code written in Java to execute on accelerator devices in a heterogeneous platform. The Tornado system is composed of *(i)* an API and decorator, *(ii)* an optimising runtime and *(iii)* a JIT compiler. Pro-

grammers identify parallel loops using the decorator syntax (`@Parallel`). These decorators are hints to the compiler to generate parallel code if supported by the target device. The Tornado API helps a programmer to express dependences between computational kernels as a graph of task-nodes. Listing 3.4 shows the parallel *saxpy* kernel being expressed as a single task in Tornado. Each task-node encapsulates potential data parallel code, the data used for computation and meta data such as compiler and runtime configurations. Metadata for each task may be manually specified by the programmer or automatically decided by the runtime. Tornado considers a task as a basic unit of computation. At the task level, the Tornado task-graph model is a data-flow model of computation with each task a data-parallel model. The Tornado Runtime performs task level dependence analysis and generates the orchestration code for each accelerator and data transfer code at runtime. To reduce data transfer overheads, data is maintained locally on each accelerator. The programmer explicitly transfers output data back to the host after computation. At runtime the whole task graph is optimised, redundant data transfers are removed and the execution time of the computational critical path is optimised. After optimisation the JIT compiler is invoked for each task. and caches compiled code for future use. The optimisation phase chooses the target accelerator to execute a task. The JIT compiler utilises Graal [40] for dependence analysis and augments Graal with information to generate OpenCL kernels for parallel execution. Tornado's JIT compiler optimises the loop structure of the explicitly parallel loop nest for each accelerator device.

Tornado was extended by Fumero *et al* [45] to perform parallel reductions by decorating the output variable of the reduction computation with the decorator `@Reduce`. Such a decorator is considered as a hint to the compiler. At runtime, data flow dependencies are checked to detect a reduction operation. A simple reduction can be detected if a store operation happens to the same location from which a load operation occurred. Tornado then replaces instructions within the IR representation of the loop structure. These specialised instructions can be compiled to OpenCL implementations of parallel reduction skeletons.

Fumero *et al* [46] further extend Tornado by lowering a task graph into specialised byte code that dynamically target accelerator devices. These byte codes cater to parallel execution and data transfer semantics. The Tornado byte-code interpreter is itself interpreted in the context of a standard Java VM. The virtualisation layer that this approach introduces makes computation migration transparent to the user. JIT compilation is performed on demand and compiled code is cached for later reuse. Tornado can dynamically target multiple accelerator devices in a heterogeneous environment. The task graph is optimised for execution by profiling multiple variants of each task on each device and remembering the devices selected for a particular schedule of tasks. The Tornado VM uses a memory manager to optimise data transfer between devices. Memory is pre-allocated on each accelerator device. While read-only variables are safely mirrored, writable memory locations on each accelerator de-

vice are mirrored as versioned memory spaces on the host. The correct version of memory is committed based on which device's output is determined to be valid by the Tornado runtime.

Apollo (Sukumaran-Rajan *et al* [29, 140]) is a runtime optimising polyhedral compiler. It partially transforms source code snippets of loop nests into LLVM IR called code-bones. During static compilation, these code-bones are characterised as linear, potentially linear and non-linear loop nests. During execution, the Apollo runtime launches an online profiling phase for potentially linear loop nests and linear inequalities are built up from the profiling results to build up a speculative polyhedral model of the loop nest. At runtime, speculatively parallelised code-bones are optimised and executed on parallel hardware. To mask the execution latency of polyhedral analysis, code generation and compilation, a sequential version of the code is executed on the host along with a parallelised version. If any guard conditions are violated, results from the parallel execution are invalidated and the results from sequential execution are committed. The finer granularity of dependence analysis that arises from using polyhedral analysis at the IR level spreads dependence analysis between memory references in a single statement over multiple IR instructions. This could potentially lead to more parallel code. However the size of code-bones over which analysis is performed is restricted because polyhedral analysis time rises quadratically [146] over the number of statements.

**Limitations** Loops are a basic computational construct in almost every imperative language and are amenable to parallel execution. Loop parallelisation for heterogeneous architectures in a managed runtime has the potential to provide transparent use of GPUs to non-expert programmers. The work in this thesis takes this approach. Many current works presented above miss optimisation opportunities while doing runtime analysis. Megaguards [115], Leung *et al* [82] parallelise only perfect loop nests and do not parallelise loops that may contain cross iteration dependences. Also Leung *et al* only allow simple linear relationships of iterators in subscript pairs. Three Fingered Jack [127, 128] does not support control flow divergence within loop bodies. The Tornado [36, 46] system is capable of handling multiple target devices. The cost model is based on heavy profiling on all available devices. Samadi *et al* [126] and Wang *et al* [156] use speculative kernel execution on GPUs and require post-execution correctness checking before committing the results.

## 3.4 Accelerating Algorithmic Skeletons

Copperhead (Catanzaro *et al* [31]) is a parallelising source-to-source compiler for Python. It extracts nested parallelism from data parallel higher level functions in Python and generates C++ CUDA code for GPU execution. It is developer directed and relies on decorator syntax

(@cu) to parallelise Python higher-order functions such as map, zip and reduce. Listing 3.5 shows the application of the @cu directive to JIT compile the *saxpy* kernel for a GPU. Nested parallel computation formed by the composition of such data parallel functions is mapped to the thread organisation hierarchy of the GPU. The programmer can also specify data dependences to the compiler by providing arguments within the decorator syntax. This allows the compiler to analyse data dependences and optimise the compiled GPU kernel code. The Copperhead compiler disallows the use of standard loop-like syntax within the body of any kernel.

Listing 3.5: Copperhead GPU JIT compilation of *saxpy*

```
@cu
def saxpy(a, b, alpha):
    return map((lambda x, y: alpha * x + y), a, b)
```

Parakeet (Rubinsteyn *et al* [124]) is another directive based JIT compiler focussing on accelerating numerical Python. Similar to Copperhead, it targets higher order functions for acceleration and generates code for CPU and GPU devices. Parakeet specialises each higher order function call within a target code region with its types. Function calls within these code regions are recursively specialised and type evaluation is propagated up the call chain of the function being accelerated. In addition to this, standard compiler optimisations such as constant folding, function in-lining and common sub-expression elimination are also employed. Parakeet also supports looping structures within the body of the function that it seeks to compile unlike Copperhead. However, no attempt is made to parallelise any nested loops. Parakeet compiles higher order functions directly to NVIDIA PTX rather than generating CUDA code.

Java Streams was introduced in Java 8 to enable programmers to program using higher order functions for computation. Ishizaki *et al* [59] designed a JIT compiler that targets foreach primitives that have been explicitly parallelised by composition with the parallel primitive. The computational kernel is extracted, optimised and a CUDA variant is generated and JIT compiled. The domain size of the parallel computation is derived from the limits of the foreach statement. Virtual methods within the objects are handled by direct or guarded virtualisation. In direct virtualisation, the target of the virtual call is replaced by a non-virtual method if it is determined to be loop-invariant during JIT compilation. Otherwise, runtime profiling is used to replace the virtual method with the best non-virtual method with a guard condition. This system does not parallelise nested calls to the parallel skeleton function and thus could potentially miss out on opportunities for parallelism.

Ikra (Masuhara *et al* [92]) is a library based data-parallel extension to Ruby. The primary data-parallel array class is the PArray class. Its class methods such as new, map and inject are provided as algorithmic skeletons which can be executed in parallel on a GPU.

The computational kernel is written in a restricted subset of Ruby. Ikra originally used an offline compiler to generate GPU executable variants with a simple type inference system. Springer *et al* [134] changed the original design to a JIT kernel compilation model of the parallel methods of the `PArray` object. To improve performance, loops surrounding calls to parallel methods are translated to C++ and compiled to execute on the host.

Fumero *et al* [48,49] implement a similar approach towards accelerating array programming in Java. An `ArrayFunction` class with parallel higher order class functions is exposed to the programmer. These functions are extended from the Java function interface to allow composition of these methods. Auto generated code for accelerators is optimised taking the data flow through the composition of functions into consideration. The functions to be accelerated are optimised using Graal [40] and converted from Graal's internal IR to OpenCL kernels.

The Lime compiler (Dubach *et al* [39]) is a high level compiler that extends the Java language with operators such as `@` and `!` to express higher order parallelisable functions such as *map* and *reduce* respectively. The => operator is provided to pipeline data flow between computational kernels. The Lime compiler auto-generates OpenCL code, data transfer directives as well as orchestration code. Dependence between kernels is represented in a task-graph representation. A relatively simple pattern matching memory optimisation is performed to allocate the most efficient memory type within the OpenCL memory hierarchy (`global,private` or `local` memory).

Dandelion (Rossbach *et al* [121]) aims to provide a unified programming model for heterogeneous systems. It seeks to transparently and automatically distribute data-parallel portions of a program to different execution devices. Programs are manually represented as a data-flow-graph and are written in the .NET LINQ programming language. Dandelion internally uses a multi-tiered data-flow graph representation to distribute work to each node. The top data-flow graph layer is the cluster level and manages all the machines within the cluster. At each machine node, the computation is further expanded to a machine level data flow graph. Dandelion's node level runtime then decides whether the CPU or the GPU executes the computation. Dandelion handles dynamic memory allocation by converting to stack allocation of the memory where the size of the object can be unambiguously inferred. If this cannot be inferred, execution falls back to the CPU variant. Dandelion requires the usage of extended types to exploit the performance of GPUs within the program representation.

RiverTrail (Herhut *et al* [56]) accelerates JavaScript programs on both the CPU and GPU. Parallel computation is centred around the `ParallelArray`, a data structure that enables parallelism. Well known parallel algorithmic skeletons provide a high level abstraction layer to the programmer while transparently enabling parallel execution on multicore CPUs and GPUs. Listing 3.6 shows the invocation of the `ParallelArray` *map* skeleton to JIT

compile the *saxpy* kernel for a GPU. Type inference is done by RiverTrail to translate untyped JavaScript to typed OpenCL code. Other arbitrary side-effect free functions that operate on each element of the parallel data structure can also be transformed into OpenCL and compiled for execution on different devices in a heterogeneous environment. Each `ParallelArray` object is immutable thus enabling optimised memory layouts during execution. RiverTrail can also distribute computational load between a host and an accelerator using an offload factor specified by the programmer at runtime.

Listing 3.6: RiverTrail GPU JIT compilation of *saxpy*

```
function saxpy(a,b,alpha) {
 var ones = new ParallelArray(a.length,
                             function(i) {return 1;});
 return ones.map( function(e1,e2 ) { return alpha * e1 + e2 ;}, b )
}
```

ParallelJS (Wang *et al* [154]), like RiverTrail, is a JavaScript JIT compiler that exposes a data structure upon which programmers can apply higher order functional primitives such as `map`, `reduce` and `filter`. Functions that are side-effect free and have no order restrictions can be parallelised. It performs type inference at runtime and propagates this to each expression within a kernel. ParallelJS transforms code at runtime to LLVM IR. When the type system cannot resolve all types to ones that are supported by a GPU, the higher order functions and the kernels are compiled to a sequential CPU version and executed. ParallelJS does not perform any cost analysis before offloading computation to the GPU.

Fumero *et al* [47] use the *partial evaluation* capabilities of the FastR implementation to accelerate parallel algorithmic skeletons in the numerical computing language R. The FastR [135] VM uses Truffle [161] to type specialise execution paths and Graal [40] for OpenCL kernel compilation. Type specialisation by Truffle at the AST level before JIT compilation removes the typical overhead of execution by an interpreter. In this system, a specialised AST node replaces the `mapply` parallel skeleton function (i.e. a *map* function) which transforms the type specialised execution path to an OpenCL kernel. The assumption that any execution order is permitted is exploited to execute the kernel in parallel on a GPU. In the event of control flow divergence, the hot path is type specialised, optimised and compiled. To model the deoptimisation, generated OpenCL kernels set a guard violation flag. At the end of kernel execution on the GPU, the flag is checked before committing any results. A guard violation will result in the code being re-executed in the interpreter.

Automatically optimising an algorithm for a particular compute device such as a CPU or GPU is an intricate process that applies sequences of transformations. Some of these transformations are device-specific. While OpenCL allows programmers to make an application portable across devices, the optimisation strategies to enable efficient computation are not

portable across devices. Lift (Stuewer *et al* [136, 137]) is a functional DSL that combines high level algorithmic programming optimisation with lower level performance portability. The transformation from the high-level functional primitives to the lower level optimisations are written once by an expert system designer. Composition of algorithmic primitives is taken into consideration to optimise data flow. The high abstraction layer transformations are lowered to a functional parallel IR. Rewrite rules allow the expert tuner to express algorithmic, data layout and address space patterns that map to OpenCL primitives. These rewrite rules produce a searchable space of potential optimisations. A Monte-Carlo tree search of the optimisation space is performed to select an optimal set of transformations for a target device such as a GPU.

Futhark (Henrikson *et al* [55]) is a statically compiled functional language that supports loops and parallelises code to execute on a GPU. Dependence analysis is elevated to a higher abstraction level compared to an array's subscript index analysis. Parallel operators are provided to programmers to maximise optimisation opportunities and ease code generation. Futhark provides parallel operators specialised for parallel execution. Unlike a pure functional language, Futhark allows in-place updates to arrays. These arrays allow explicit indexing within parallel operators and higher order functions. When safe to do so, in-place array updates reduce the cost of copying arrays. The type system and alias analysis guarantee the safety of in-place array updates. `For/While` loop constructs are permitted within the scope of a function. Loop analysis is done by considering loop constructs as a simple form of *tail recursion*. Excessive data transfer between the GPU and CPU is optimised by aggressively hoisting memory allocations out of parallel code. Only regular arrays are supported for GPU execution.

LambdaJIT (Lutz *et al* [89]) accelerates C++ *lambda* functions on GPUs. Lambdas are first transformed to standalone classes. Although C++ is a statically compiled language, the LamdaJIT compiler lowers and serialises the computation of the lambda into byte code and injects it into a separate section of the executable. Every invocation of the lambda is then replaced with a call to the LambdaJIT runtime. When a lambda call site is activated, the LambdaJIT runtime evaluates type traits embedded into the bytecode and safety of concurrent execution. If it is safe to do so, CUDA kernels are generated for execution on the GPU.

**Summary** In the functional programming paradigm, parallelism is often implicit in the algorithmic skeletons such as *map*. These algorithmic skeletons serve to abstract the higher level logic of the algorithm away from its implementation details. This enables automatic parallelisation systems to reason about device specific parallelism optimisations. A large body of recent research is actively seeking to accelerate programs written using higher-order functions on parallel hardware like GPUs.

## 3.5 Parallel Domain Specific Languages

A DSL allows programmers to focus on the algorithm being developed. DSLs are developed and optimised for a particular application domain. The programmer can then delegate the implementation and performance optimisations to the DSL compiler. These compilers can perform effective optimisations on program patterns that are more prevalent in the specific domain.

Selective Embedded Just-in-Time Specialisation (SEJITS) (Catanzaro *et al* [32]) takes into consideration that domain experts are most comfortable programming in high level languages despite the higher execution overhead of their managed runtimes. To ease the steep learning curve required to produce high performance code in heterogeneous environments, it provides a set of class libraries with abstractions that are appropriate for a non-expert programmer. These abstractions are selectively specialised and JIT compiled targeting available hardware. By using meta-programming techniques, these abstraction layers are presented to the programmer as extensions of the language or as embedded DSLs. By accelerating only the computationally intensive code identified by the programmer, SEJITS can perform optimisations at the algorithmic level specific to a domain while still providing a non-expert programmer with the comfort and ease of use of a higher level programming language.

Asp (Kamil *et al* [69]) is a SEJITS based approach to acceleration in Python. It provides base classes from which computational patterns can be extended and optimised using templates or using specialised AST walkers. If the AST walker determines that all the code within the computational kernel conforms to the computational pattern being expressed, code targeting a specific device is auto-generated and optimised. In this case both the host language as well as the language used to transform the computational pattern is Python. While this eases development effort for non-expert programmers, it still requires a programmer to identify computational patterns and apply the correct transformations to generate efficient code.

Delite (Sujeeth *et al* [138]) is a framework of compiler tools to aid in the quick development of a Domain Specific Language (DSL). It provides programmers of DSLs with parallel variants of higher order functions such as `map`, `reduce` and `filter`. Delite DSL code is written in Scala and is initially compiled into Java bytecode. The Java bytecode is then transformed into the Delite DSL's own IR which is optimised and converted to the target parallel platform like CUDA, OpenCL, C++ or Scala. The Delite IR is expressed in a sea-of-nodes representation [105]. This representation removes artificial constraints on program order and helps better represent parallel operations. The Delite DSL compiler performs common compiler optimisations such as common subexpression elimination, dead code elimination and code motion. Since these optimisations are performed at the level of objects, the overall performance gain can be substantial. Delite also performs Array-of-Struct to Struct-of-Array

data layout conversions to optimally match the target execution device.

SYCL [117, 118] is a single source API designed as a a higher level of abstraction for OpenCL programming. It has been adopted as a Khronos programming standard [159]. It has also been adopted to be the foundation of Intel's OneAPI [58] framework for heterogeneous programming. SYCL is written in standard templated C++ programming. It simplifies OpenCL's model of separating kernel programming from host programming by providing single source programming. The linked program contains both host and device code. It can automatically select an appropriate device within a heterogeneous environment while also enabling programmer directed device selection. Automatic type-checking between host and accelerator is performed by the runtime. The SYCL language specification is designed to implicitly perform compilaton and data transfer between host and accelerator devices. This is unlike OpenCL where the programmer has to explcitly perform data transfer and compilation steps before execution. Data transfer and kernel execution which is explicitly programmed in OpenCL is implicitly managed as part of the SYCL language specification. The program is statically compiled to SPIR-V [72], the Khronos standard IR for heterogeneous environments. Silva *et al* [129] report that SYCL programs use less memory (0.2x–0.39x) while being 2.35x–2.77x slower than the equivalent OpenCL program.

LooPy [74] is an embedded DSL in Python that describes an execution model for array style computations. An expert programmer is required to express loop domains in the polyhedral model and encode a sequence of array computations to be executed. Any dependences between statements should be described within the DSL. LooPy exposes various loop optimisation transformations to the programmer. These optimisations must be manually chosen to generate and compile parallel code. LooPy generates OpenCL code from the polyhedral representation of the computation to target various devices within a heterogeneous platform. If the developer does not specify the types of an array, LooPy can infer the types and generate OpenCL kernels at runtime. LooPy does not automatically parallelise or optimise loops. Instead it is intended to aid an expert programmer to optimise code while maintaining a high level abstraction.

**Summary** A DSL can perform aggressive optimisations for specific domains. However, it increases the burden on novice programmers who must learn a language specialised for a particular task rather than program in a general purpose language. Morover in some DSLs, such as LooPy, execution schedules and device specific optimisations must be explicitly written by the programmer, requiring considerable understanding of the hardware.

## 3.6 Intermediate Representations for Parallel Computation

An Intermediate Representation (IR) is a program representation that transforms a program into a standard format that is fairly independent of the source language and the target binary. It enables the easy retargeting of code to multiple target languages or hardware binaries.

SPIR-V [72] is an Intermediate Representation (IR) used in compilers for graphics shaders and parallel compute languages such as OpenCL and OpenGL. SPIR-V is originally based on the LLVM IR representation. The maturity of the LLVM compiler stack and almost all its static optimisation passes can be used to optimise kernels. As this is a Khronos standard, the IR can be distributed, rather than source code. This allows specific target devices to apply target specific optimisations on the IR before final execution.

The Heterogeneous System Architecture (HSA) [6] is a specification that describes the integration of data-parallel and task-parallel devices on a single chip. It describes the hardware integration requirements as well as a software programming model that is required to coherently program each compute device in the chip. Such devices are designed to share global memory with all other devices to reduce latency and data transfer times. In this respect, the HSA model is different from the traditional heterogeneous device model where discrete accelerator devices are connected over a high-speed data transfer bus such as PCIe.

HSA compliant compilers transform high level programs into a new IR called Heterogeneous System Architecture – Intermediate Language (HSAIL). HSAIL is a data parallel IR that can operate on multiple HSA compliant devices. It has explicit support for devices that follow a data parallel model such as synchronisation primitives, a tiered data-parallel thread hierarchy and a memory hierarchy. Each device will have its own backend finaliser to translate HSAIL to its own ISA to execute on a particular device. The HSA runtime is heavily used in AMD's ROCm [8] to provide a unified heterogeneous programming model using C/C++. Numba [78] also provides an HSAIL target enabling Python language support to target AMD GPUs.

INSPIRE (Jordan *et al* [66]) is an Intermediate Representation (IR) for modelling heterogeneous parallel programs. Specific directives such as *fork-join*, Inter-Process Communication (IPC), data transfer and thread identification, formally and concisely represent parallel computational models directly within the IR thus aiding analysis and optimisation. The concept of a thread hierarchy, which is important in modern GPU systems, is represented using recursively nested *thread-groups*. INSPIRE IR can be translated into a number of backends such as OpenMP and OpenCL.

**Summary** Intermediate representations standardise the representation of different control structures in a higher level language into a small number of well recognised patterns. These IRs are generated by compilers and not normally exposed to end-user programmers. The higher level information lost while lowering into a standard IR inhibits optimisations to enable coarse-grained parallelism. Parallel IRs aim to encode parallelism within the computation directly into the IR. ALPyNA maintains loop dependence relationships in an in-memory IR format to dynamically determine loop dependence relationships (Section 4.3.1). These dependence relationships are evaluated at runtime for each loop nest execution instance.

## 3.7 Parallel Cost Models

Cost models to determine parallel execution time are dependent on computational size, amount of parallelism identified, hardware characteristics and data access patterns. Trinder *et al* present a wide-ranging survey on resource analysis and the analytical techniques used to model parallelism on manycore systems [145]. Mittal *et al* [95] survey and categorise compiler and runtime cost modelling of CPU–GPU systems based on (i) when the cost calculation is scheduled (i.e. static or dynamic) and (ii) the relative performance of features within the target device. Determination of costs for statically compiled programs will either rely on profiling programs or symbolic resolution of the attributes that determine the amount of parallelism that can be extracted from a program. In the case of static symbolic resolution, this implies that the cost can only be determined if the domain size of the problem is hard-coded within the program. However, this expectation is not practical. For example, in the case of imperative loop nest parallelism, Peterson *et al* [109] report that roughly 11.9%, 71% and 3.2% of the lower bound, upper bound and stride values respectively of loop constructs in loop nests are unknown at compile time and in many cases inhibit parallelisation of loop nests. In this section, efforts to determine the cost of parallel execution of a program are discussed.

### 3.7.1 Analytical Cost Models

Armih *et al* [12, 13] describe a simple analytical cost model to load balance the distribution of computation for parallel algorithmic skeletons. The *CM1* model is parameterised only on hardware characteristics to distribute the data amongst distributed multicore CPU devices. The more generic *CM2* model builds a relative cost model to distribute computation on nodes with both a multicore CPU and a GPU. Computation is load-balanced across multiple nodes by using the notion of relative speed which is derived from hardware parameters as well as profiling time on each device within the heterogeneous environment. Belicov *et al* [24] extend this approach to determine the cost of computation using an analytical cost model

to weight the distribution of parallel computation between devices of a heterogeneous environment. The computational capability of each device is calculated as a weighted product of hardware characteristics such as frequency, cache-size , memory and communication latency.

The AJITPar project (Maier *et al* [91]) parallelises higher order functions written in Racket and executed using the Pycket VM. Pycket is built using the PyPy tracing JIT compiler and runtime (Section 3.1). Dependences between tasks are represented by task graphs. The runtime tunes and transforms parallel skeletons to distribute the parallelism across many cores. Morton *et al* [98] introduce a tuneable online cost model to AJITPar which uses constant feedback and lightweight runtime profiling to tune the granularity of parallelism to be distributed amongst parallel execution nodes. Bytecodes are classified and each bytecode class is assigned a performance cost weight. These weights are learned by profiling a large number of benchmark programs to find their relative execution costs. The guard conditions at each control flow divergence point mark the end-points of a trace. The cost of a trace is a weighted sum of its bytecode. Counters at each point of control flow divergence in a program are used to weight each trace to calculate the overall program cost. At runtime, the relationship between abstract cost and actual runtime (assuming a linear relationship) is established. For any new execution time, the relative cost factor is used to derive a new tuning factor to speed up execution.

The Memory Warp Parallelism (MWP)–Computation Warp Parallelism (CWP) model introduced by Hong *et al* [57] is a detailed static analytical model to predict performance of computational kernels on GPUs. It attempts to predict whether computational time in the GPU is dominated by memory access time or vice-versa. This cost model uses memory bandwidth, compiled instructions and pipelining effects within a kernel to compute which of MWP and CWP effects dominate the execution time. Sim *et al* [130] extend this model to add cache effects by profiling for the average memory access latency.

Chikin [34, 35] describe a dynamic cost model for parallelising OpenMP loops. The static compiler stores a list of array subscript relationships which may be determined statically or at runtime. At runtime, parameters such as loop stride are inferred from these relationships to determine memory coalescing patterns in the loop nest. These are plugged into the *fork-join* cost model of the CPU execution proposed by Liao *et al* [83] and the GPU execution MWP–CWP cost model proposed by Hong *et al* [57]. These cost models are used at runtime to predict the runtime on each device. The parameters used in these models utilise a large number of micro-benchmarks on each machine to determine absolute values for memory latency and cycles per iteration. Chikin's model resolves memory coalescing and memory bandwidth values at runtime to determine execution costs. The kernels are compiled based on statically determined dependence structure and does not take context and kernel initialisation into consideration. Like the original MWP–CWP cost model that it is based on, it also does not take a cache memory hierarchy into consideration for cost analysis.

Leung *et al* [82] introduced a cost analysis function into an automatic loop parallelisation system for JikesRVM (discussed in Section 3.3). This is a simple parameterised analytical model that assigns an absolute cost to each bytecode in the representation of a loop nest. The cost for each byte code is calculated by profiling a range of micro-benchmarks on both the CPU and GPU at installation time and averaging them. During compilation, the estimated number of instructions becomes known and at runtime, the domain sizes and data transfer costs can also be ascertained. Conditional branches are assumed to be taken $50\%$ of the time and that each nested loops execute 10 times unless the domain range is known at compile time. The absolute costs are calculated by using the simple relations. The cost model in Equation 3.1 shows the relationship between the average time ($t_{cpu}, t_{gpu}$) to execute a byte code on each device, the number of byte code instructions in a loop ($insts$) and the size of the iteration domain. Loops are assumed to be iterating over each item in the output array and the size of the array ($A_{out}.size$) is used as the overall iteration domain size. A profiled initialisation cost ($init$) and data transfer cost ($copy$) is taken into consideration for GPU execution. However, this method does not take GPU starvation effects due to relative speed difference with the host or how the a specific thread hierarchy for a GPU is mapped and executed on a GPU's hardware execution units.

$$\begin{aligned} Cost_{cpu} &= t_{cpu} \times insts \times A_{out}.size \\ Cost_{gpu} &= init + (t_{gpu} \times insts \times A_{out}.size) + (copy \times \sum_{A \in A_{inout}} A.size) \end{aligned} \tag{3.1}$$

Zefreh *et al* [163] propose a cost model to determine a data partitioning scheme for loop nests that are nested up to three levels deep. The cost model (3DDP) is designed to distribute data across a heterogeneous cluster of GPUs for scientific computing. The algorithm load balances the workload normalised to the computational power of the GPU. The objective function aims to minimise data communication costs while maintaining load balancing between each node. This cost model assumes that no cross-iteration loop dependences exist within the loop nest and can only handle up to three loops.

Kennedy and McKinley [71] calculate the cost incurred using different loop optimisation strategies in static languages like FORTRAN where a linear relationship exists between subscripts of each dependence pair. This cost model calculates the distance between each pair of dependences in a loop nest. Costs are assigned for each loop execution instance depending on the size of the cache line to simulate a cache-miss. For simple subscript relationships such as ZIV and SIV (Section 2.2) these abstract costs can be easily established. However the worst case scenario is assumed for more complicated memory access patterns such as MIV relationships.

**Limitations** Analytical cost models by Armih *et al* [12, 13] and AJITPar [91] require extensive profiling for each compute workload to accurately determine execution costs. The Memory Warp Parallelism (MWP)–Computation Warp Parallelism (CWP) model and its use by Chikin *et al* [34, 35] show accurate cost predictions for individual kernels on a GPU. However, it does not take into consideration the relative performance difference of the interpreter and GPU starvation. The 3DPP approach only takes perfectly nested loops up to 3 loops deep into consideration. Leung *et al* [82] predicate their model on the approximate execution time to execute the computation equivalent to a byte code. Extensive profiling of benchmarks on each GPU is required to learn this value. The ALPyNA Cost Model (Chapter 6) is a lightweight analytical cost model parameterised on the hardware characteristics of each target device in a heterogeneous environment. It also considers GPU starvation effects arising from the slower execution speed of the interpreter.

### 3.7.2 Machine Learning Models

The use of machine learning is becoming increasingly popular to predict execution performance of computation as well as the appropriate optimisations to perform on the computation. Machine learning models are trained by an expert on the performance benefits of compiler optimisations and factors that affect performance. During execution, computational patterns and their costs are inferred from the model which guides the compiler optimisations or the devices selected for executing the computation. Machine learning approaches to compiler optimisation was pioneered by O'Boyle et al [50], in the Milepost compiler.

Wang *et al* [155] translate OpenMP programs to OpenCL and perform loop optimisations and data layout optimisation. In the training phase, feature extraction is performed from the AST of a large number of benchmarks. Relevant features are selected by an expert. From the profiling runs, performance on both the CPU and GPU are correlated with the extracted feature set. At runtime, the feature set of a program is extracted from its AST and a decision tree classifier is used to predict the best performing device and the kernel variant to choose for the target device. When loop bounds are unknown at the time of prediction an average value is estimated.

XAPP (Ardlani *et al* [9]) predicts the performance of code executing on a GPU by studying the features of the same program's binary on the CPU variant. During the training phase of the machine learning algorithm, architecture independent feature sets are extracted from a large number of programs and their execution times on each of the target GPUs. A unique cost function is then generated for each GPU and correlated to the architecture independent feature set. A two level machine learning technique is used to train the model. At the first level, regression models are built with a smaller set of features. The predictions from the smaller models are combined to correlate the feature set to execution time on each GPU.

During execution, the model extracts the relevant feature set from the CPU binary and makes the relevant execution time prediction for each hardware GPU.

Wu *et al* [160] train a machine learning model on one GPU to predict performance scaling on other GPUs. A large number of GPU benchmarks are executed on a single GPU and performance counters during execution time are collected. To create the model, training kernels are first clustered according to the feature set using the *K-means* algorithm. A *neural-net* classifier is then built correlating the feature sets in the training kernel clusters to the scaling factor for each performance counters. To predict the performance of a kernel on a new GPU, the kernel cluster is first identified from the feature set and the amount of scaling that can be achieved with the hardware resources is predicted.

**Summary** While a machine learning model may achieve good predictions regarding the cost of computation on heterogeneous hardware, Amaris *et al* [5] conclude that in general, an analytical model tends to be more accurate. Machine learning models also require a large amount of training data and profiling to train the model. At runtime, the machine learning cost models are comparatively heavier. Analytical models fit better into a JIT compilation environment where such latency is not acceptable. In Chapter 6, ALPyNA's cost model (ACM) predictions are compared against a trained SVM model to determine the performant device in a heterogeneous environment.

## 3.8 Summary

This chapter has presented and characterised the research literature related to the work presented in this thesis. It has brought together work in the domains of language VMs, loop parallelisation in managed languages, GPU related parallelisation in the functional paradigm, parallel IRs and various analytical and machine learning cost models in CPU–GPU systems.

Exploitation of GPU performance in high level dynamic languages is an area of ongoing active interest. A large body of research has been published on JIT compilation targeting GPUs in functional languages and paradigms. To the best of our knowledge, relatively little work has been done for automatic parallelisation in managed languages. Our extensive survey (Section 3.3.1) has uncovered Megaguards [115], Tornado [36, 45, 46], TFJ [127, 128], Paragon [126] and Wang *et al* [156]. Chapter 4 describes a staged dependence analysis approach to automatically parallelise loops for GPUs and optimise execution using runtime information. Chapter 6 describes a novel analytical cost model that predicts the optimal device to execute a loop nest in a CPU–GPU environment.

# Chapter 4

# ALPyNA : System Architecture

Dynamically typed, high-level languages such as Python, R, Ruby and JavaScript are increasingly being used as general purpose computing languages in a wide range of application domains (Chapter 2). Python is particularly attractive to end-user developers given its simplicity and accessibility. These characteristics help developers express computation in a compact style with reduced boilerplate code. This increases readability enabling rapid prototyping and iterative software programming. The complexity of supporting all the advantageous features of a dynamic language is abstracted away from the programmer by the VM. However, the brevity of dynamic languages comes at the expense of performance. The overhead of program execution by a VM makes a program slower than a similar program written in a statically compiled language. This overhead is due to the various features such as dynamic dispatch and memory management.

Python programs are executed sequentially in the standard CPython interpreter. The Global Interpreter Lock (GIL) in the CPython implementation [22] constrains execution to being single-threaded. Parallel resources are now available as commodity hardware, whether as multicore processors or GPUs. Given the ubiquity of GPUs and the performance improvement they offer, it is reasonable to assess whether we can take advantage of them when executing Python programs.

This thesis focusses on automatic acceleration of loop nests written in Python in a heterogeneous environment. This chapter presents the system architecture of ALPyNA, a novel hybrid loop parallelisation framework for Python. ALPyNA enables developers to parallelise Python loop nests on GPUs transparently. A staged approach to loop dependence analysis is described to optimise and execute computation depending on runtime parameters.

This chapter is structured as follows. Section 4.1 presents the motivation behind the design framework. Section 4.2 describes the parallelisation advantages derived by deferring loop nest dependence analysis to runtime. Section 4.3 describes ALPyNA's static and runtime staged loop dependence resolution framework. Section 4.4 describes the design of the GPU

code generator that generates kernels from the exact dependence relationships derived at runtime. Finally, Section 4.5 summarises the overall design of ALPyNA.

## 4.1 Motivation

Commodity GPUs offer a large numbers of cores (of the order of thousands) for minimal cost and are often extremely effective for data parallel tasks. Depending on the workload, such accelerators can provide orders of magnitude better performance. The most commonly used programming languages to target GPUs are CUDA and OpenCL (Chapter 2). These are C-like dialects with low level access to memory

The PyCUDA and PyOpenCL frameworks [75] are C-like eDSLs that enable GPU programming using idiomatic CUDA and OpenCL. However, programming GPUs is highly complex as it exposes the programmer to the physical realities of the GPU being used. Numba [78] makes it slightly easier for programmers to write computational kernels in Python, but expects idiomatic CUDA/ OpenCL compute kernels. The process of reasoning gets progressively harder for complex code and imposes high cognitive burdens on the developer [39, 115, 136].

ALPyNA's design goal is to dynamically maximise parallelism of loop nest execution in a heterogeneous environment containing multiple accelerators. The remainder of this thesis focusses on JIT compilation for CPU–GPU systems. However, ALPyNA is designed to be able to easily extend this to other accelerators. Extensible features include *(i)* the Hardware Abstraction Layer (HAL) (Section 4.3.3), *(ii)* parallelising each statement independently (Section 4.4.1) and *(iii)* a cost model to guide the selection of the performant device (Chapter 6).

### 4.1.1 Design Principles

ALPyNA is designed with the goal of automatically parallelising loop nests in Python in a heterogeneous environment. It is targeted as a tool to be used by non-expert developers. The adjective 'Pythonic' describes elegant Python code, as agreed by the community of practice. This philosophy is encapsulated in the PEP-20 document [143]. The principles espoused in PEP-20 have heavily influenced the design of ALPyNA's parallelisation system for Python, which targets commodity GPUs to execute loop nests. A summary of these 'Pythonic' principles are available inside the Python interpreter as an 'easter egg'—simply enter `import this` to see the complete list of guidelines. This section explains how we interpreted PEP-20 in our context.

**Beautiful is better than ugly. Simple is better than complex. Readability counts.** Acceleration frameworks that aim to automatically parallelise end-user code should not require excessive refactoring or library calls that make the code 'ugly' or 'complex'. ALPyNA analyses sequential code and transforms it into GPU code without requiring the user to explicitly rewrite loop nests. While ALPyNA performs '*implicit*' i.e. automatic parallelisation, it is not in general, an automatic framework. The user should explicitly identify and invoke dependence analysis on functions with loop nests.

**Explicit is better than implicit.** Automatic parallelisation is done transparently and implicitly to aid non-expert developers to write idiomatic Python code. However, ALPyNA only parallelises loop nests with known co-routines that yield integer iteration spaces with a constant stride pattern (currently the `range` function). The user (who should be aware of this semantic restriction) knows which regions of code may be appropriate for parallel execution. Further, the user directly identifies each potential target for parallelisation with a simple call to the ALPyNA framework. We treat parallelisation like memory management, as a runtime concern that the developer does not need to handle explicitly; instead the developer devolves responsibility to the ALPyNA execution engine. This is a case where **Practicality beats purity**.

**Errors should never pass silently.** If a loop nest cannot be parallelised, then the system reports the error. There may be a dependence violation, or the presence of Python structures we cannot handle (e.g. function calls in loop bodies). Hardware constraints of GPU accelerators such as memory constraints could prevent effective parallelisation. In each case, at the appropriate stage, the framework should report an error to the user.

**Now is better than never.** This is the rationale for our just-in-time resolution of loop nest dependences and value types. While there is some runtime overhead to this deferred analysis, it provides more accurate dependence resolution and more efficient GPU code.

**Namespaces are one honking great idea.** ALPyNA uses namespaces to manage multiple runtime variants of a single loop nest source fragment, perhaps targeting different backends including GPUs.

### 4.1.2 Application Programming Interface

ALPyNA is a dynamic loop parallelisation framework for Python aimed at non-expert programmers. The API is designed to reduce the amount of boilerplate code while enabling the analysis of conventional loop nests at runtime. It dynamically performs dependence analysis on functions containing loop nests for each function invocation.

ALPyNA is not designed as a whole program compiler. The programmer writes numerical kernels using nested *for*-loops with a constant stride value using the Python `range`

function. Restricting the analysis to linear loops allows the analysis engine to reason about dependences carried by the loops, and apply loop level optimisations.

Listing 4.1: Example invocation of ALPyNA API. Staged analysis is performed on all functions on which `static_analyse` is explicitly called on.

```
1 import numpy as np
2 import Static_Analysis_Driver as alp
3 ...
4 ...
5 alpyna_ex_engine = alp.static_analyse(func_list)
6 alpyna_ex_engine.loopy_kern_1(arr_a, arr_b, lims)
```

Listing 4.1 shows a typical example of the ALPyNA API as invoked by a user. Static analysis (Section 4.3.1) is performed once at the beginning on all computational kernels containing loop nests. The static analysis phase (Section 4.3.1) builds up the in-memory dependence relationship data structures required for runtime dependence analysis. ALPyNA's static analysis returns a specialised `module` object containing a dictionary of callable functions which the programmer can dereference and invoke with relevant arguments. In this example, a function `loopy_kern_1` (shown in Listing 4.2) will be called. When such functions are called at runtime, ALPyNA's dynamic analysis and introspection system intercepts each call; it then generates, compiles, and executes relevant CPU or GPU kernels with appropriate cost modelling, data marshalling and transfer. The transfer of control from the Python VM to ALPyNA's runtime is transparent to the user.

## 4.2 Benefits of Deferring Analysis to Runtime

Consider the *loop* in function `loopy_kern_1` shown in Listing 4.2. The dependence relationship between the *load* and *store* operations is classified as a SIV relationship [52]. Dependence analysis informs us that the statement in the *for*-loop can be parallelised as long as the loop iteration domain is within the range $[0, 1024)$ in order to be correct. Allen and Kennedy [70], define the number of iterations between the *load* and the *store* as the *distance*.

Listing 4.2: Benefit of runtime parallelisation

```
def loopy_kern_1( arg_a, arg_b, arr_len ):
 for i in range(arr_len):
  arg_a[i+1024] = arg_a[i] + arg_b
```

In a static language like Fortran, symbolic resolution of the limits would result in the generation of a speculative parallel variant of the kernel nested within a guard condition that checks if the loop iteration domain (*distance*) is less than $1024$. If so, a parallel version of

the loop would be invoked for execution. Otherwise the loop would be executed sequentially. Smarter compilers would do *strip mining* to tile all iterations that can be run in parallel and execute them with SIMD instructions.

Listing 4.3: Runtime parallelisation with loop invariant in subscript

```
def loopy_kern_2(arg_a, arg_b, al ,alpha) :
 k = alpha + arg_b
 for i in range(al):
     arg_a[i + k] = arg_a[i] + arg_b
```

Listing 4.3 shows a similar loop nest with the *distance* becoming unresolvable at compile time due to the presence of a loop-invariant variable $k$. To ensure correctness, compile time analysis has to conservatively assume that dependences exist and execute the loops sequentially or execute the parallel version when all the guard conditions have been met. Equation 4.1 shows the influence of variable $k$, the iteration domain size ($al$), and the size of the array ($size$) on the distance and direction of cross-iteration dependences. The cross-iteration dependences (depicted using notation detailed in Section 2.2) reverses direction depending on whether $k$ is positive or negative.

$$Dependence = \begin{cases} (k \geq al) \vee (-(size - al) \leq k \leq -al < 0) & no\ dependence \\ (k = 0) \vee (k = -size) & \delta_\infty \\ (0 < k < al) & \delta_i \\ all\ other\ cases & \delta_i\ and/or\ \delta_i^{-1} \end{cases} \tag{4.1}$$

When the number of data dependences becomes larger, the dependence relationships make generating all the required guard conditions and code variants *NP*-hard. By deferring this analysis to runtime, we can infer how much parallelism can be extracted from the loop-nest depending on the loop size and generate code to satisfy the dependence constraints while still executing in parallel.

Listing 4.4: Example loop nest dynamically parallelised by ALPyNA

```
def ln_func(arg_a,k,limits) :
 im, jm = limits
 for i in range(0,im,1):
   for j in range(0,jm,1):
     arg_a[i+k,j] = arg_a[i,j] + 4        # Statement - S1
     arg_a[i+16,j] = arg_a[i,j]           # Statement - S2
```

To demonstrate the potential optimisations that can be unlocked due to a dynamic knowledge of loop bounds and/or subscript values, consider the loop nest in Listing 4.4. The dependence

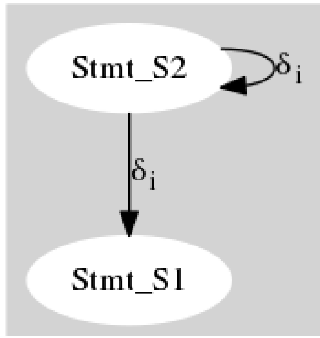



Figure 4.1: Dependence graph of loop nest in Listing 4.4 with iteration domain (*i,j*) ← (32,1024) and (*k*) ← 64

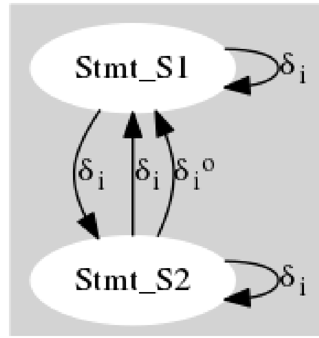



Figure 4.2: Dependence graph of loop nest in Listing 4.4 with iteration domain (*i,j*) ← (32,1024) and (*k*) ← (8)

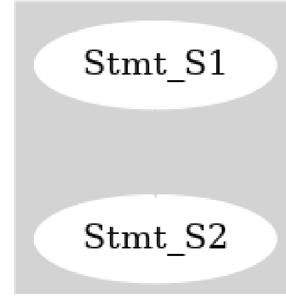



Figure 4.3: Dependence graph of loop nest in Listing 4.4 with iteration domain (*i,j*) ← (16,1024) and (*k*) ← (16)

relationships between the two statements *S1* and *S2* are determined by loop bounds $(im, jm)$, the loop invariant variable '$k$' and the coefficients and constants in each array subscript. Due to the unresolved loop domain sizes and non-iterator variable within the array subscript on the LHS of *S1*, a purely static compiler must conservatively generate sequential code.

Figure 4.1 shows the dependence graph for an instance of the nested loops in Listing 4.4 having limits $(im, jm) \leftarrow (32, 1024)$ and $(k) \leftarrow (64)$. A true dependence carried by the outer loop $\mathcal{F}_i$ exists from Statement *S2* to *S1* as well as from Statement *S2* to itself. The latter dependence is a *cyclical* dependence which requires sequential execution of the loop that carries the dependence ($\mathcal{F}_i$). All $32 \times 1024$ execution instances of Statement *S1* can safely be executed in parallel because $k \geq im$. Executing the outer loop ($\mathcal{F}_i$) sequentially $(im \leftarrow 32)$ allows 1024 execution instances (corresponding to the inner loop $\mathcal{F}_j$) to be executed in parallel. Figure 4.2 shows the dependence graph for the same loop nest with domain limits $(im, jm) \leftarrow (32, 1024)$ and variable $(k) \leftarrow (8)$. As $k < im$, dependences carried by the outer loop ($\mathcal{F}_i$) can cause cross iteration dependences. The dependences carried by ($\mathcal{F}_i$) cause cyclical dependences on *S1* and *S2* to themselves, as well as between *S1* and *S2*. To prevent the violation of these dependence constraints, the outer loop carrying the dependence ($\mathcal{F}_i$) is executed sequentially. This enables parallel execution of the inner loop ($\mathcal{F}_j$). If the loop limits have the values $(im, jm) \leftarrow (16, 1024)$ and $(k) \leftarrow (16)$, there are no cyclical dependences within the dependence graph. Therefore, all instances of statements *S1* and *S2* can be executed in parallel (Figure 4.3).

## 4.3 Staged Dependence Analysis

Often, the key control structure for parallelism in imperative languages is the *loop*, particularly hot loops (where most of the execution time is concentrated). The key data structure in

dependence analysis for parallelising loop nests is often the *array*.

A large number of loop parallelisation techniques have been developed for use in optimising compilers. Chapter 2 has already referred to the fundamental aspects of loop parallelisation encapsulated in the work of Allen and Kennedy [70]. For the majority of use-cases, Goff *et al* [52] present simplified fast dependence tests. By computing the cyclic dependences between statements carried by various loops, we can identify which loops can be executed in parallel for the overall set of nested loops, without changing the computation. These techniques generally apply to imperative, numerical computation. Code is usually written in static languages such as Fortran and C/C++ high-performance computing code although the variable aliasing problem is more acute for C-style languages.

The challenges of general auto-parallelisation derive from the following root causes:

1. complexity of analysis (for both aliasing and dependence).

2. conservative nature of static analysis, since runtime values like loop bounds are usually unavailable.

3. difficulty of mapping parallel tasks to available hardware resources to achieve significant speedup.

ALPyNA overcomes the above difficulties by using a hybrid analysis technique, combining static and runtime dependence analysis. It benefits from the Python language's relative simplicity, in terms of structured control flow (no C-style `goto`) and loop iterator guarantees provided by the `range` function semantics. These features make analysis much less complex. The Python execution is interpreter-based and relatively slow, and the performance gains from parallel execution make the significant analysis overhead affordable. Dependence analysis of the loops reveals opportunities for parallelism while still maintaining the ordering constraints expressed in the original computation.

ALPyNA targets linear loop nests as the unit of analysis in functions designated by the programmer rather than being a whole program compiler. The rich nature of the Python runtime environment enables the memoisation of analysis artifacts throughout program execution. It refines the knowledge base as information about data types, loop bounds and runtime dependences become available. The ALPyNA loop parallelisation framework is designed to generate and JIT compile CPU and GPU kernels from Python loop nests. The decision to compile and execute a particular loop nest instance on either the CPU or the GPU will depend on the iteration domain sizes and the loop carried dependences that arise at runtime. ALPyNA optimises kernel generation corresponding to the dependences that are resolved at runtime. This avoids the code bloat that occurs when statically generating kernel variants that may emerge at runtime. ALPyNA's runtime code generation scales better than static

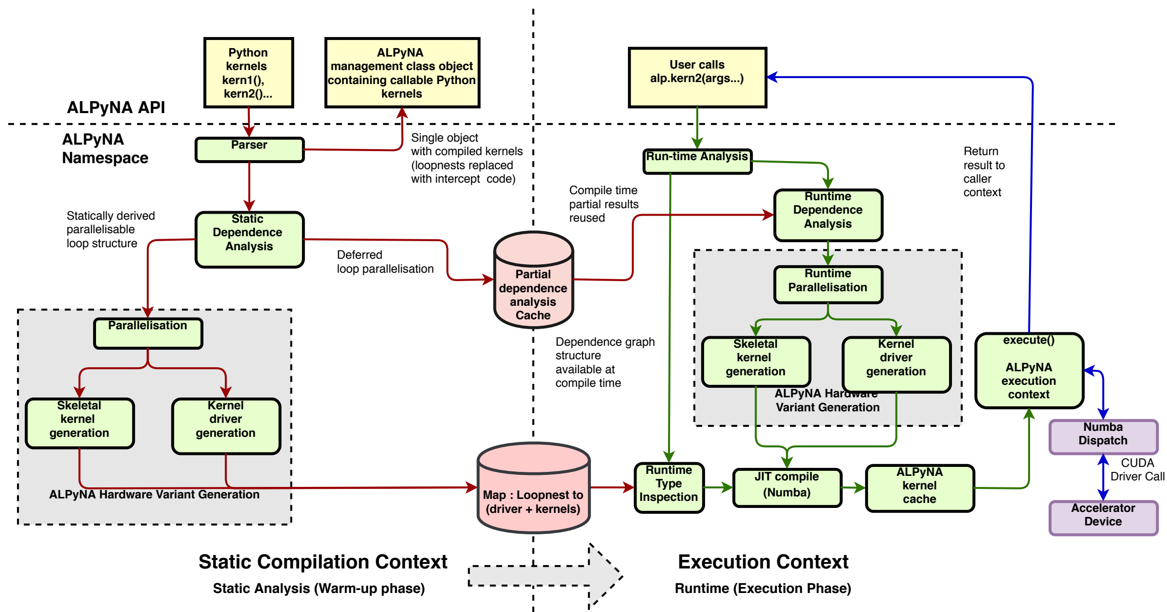



Figure 4.4: The ALPyNA system architecture is staged, with an ahead-of-time static analysis and a near-identical structure for the lazy dynamic analysis; both resolved and deferred dependence relationships are preserved in memory from the initial stage and utilised at runtime.

generation as the number of variants grow exponentially as a function of the dependence relationships (Section 4.2) and data types of arrays. A cost model (Chapter 6) is deployed to automatically and transparently determine the optimal device for each execution instance of the loop nest.

Chapter 3 has already discussed runtime frameworks that speculatively generate kernels at compile time based on cost models. Instead of speculative generation of variants depending on a combination of iterator, subscript and hardware properties, ALPyNA uses a *staged* approach to parallelisation. Figure 4.4 illustrates ALPyNA's staged compilation architecture. The ALPyNA API (Section 4.1.2) is the interface used by developers to invoke static analysis of loop nests. The left hand side outlines the static analysis and compilation while the right hand side outlines the runtime compilation. The static analysis phase (Section 4.3.1) preprocesses the loop nests (§4.3.1: Normalisation and §4.3.1: If–Conversion). If all the dependences can be obtained by static analysis, skeletal kernel variants are generated (§4.3.1: Partial Dependence Analysis). This builds up in-memory data structures to enable runtime analysis and kernel generation. The static compilation context then transforms replaces loop nests with nested functions (§4.3.1: Nested Function Generation).

The right hand side of Figure 4.4 shows the 'execution context' which performs dependence analysis on loop nests with dependences marked for runtime analysis (Section 4.3.2). The Hardware Abstraction Layer (HAL) (Section 4.3.3) is the interface between the analysis framework and the hardware specific code generation. Code generation for loop nests after

runtime analysis is discussed in Section 4.4. Runtime type patching (Section 4.4.2) is then performed on these kernels before JIT compilation and execution.

### 4.3.1 Static Analysis

ALPyNA takes, as its input, functions written in ordinary Python. An AST parser scans for numerical loop nests within these functions and analyses for potential parallelism. All other regular Python code constructs will be executed in the CPython interpreter as normal. This allows developers to interleave loop nests with standard Python code, e.g. conditional execution constructs (*if / else* constructs that are not inside loop bodies). Figure 4.5 provides an overview of ALPyNA's static analysis phase which is described in this Section.

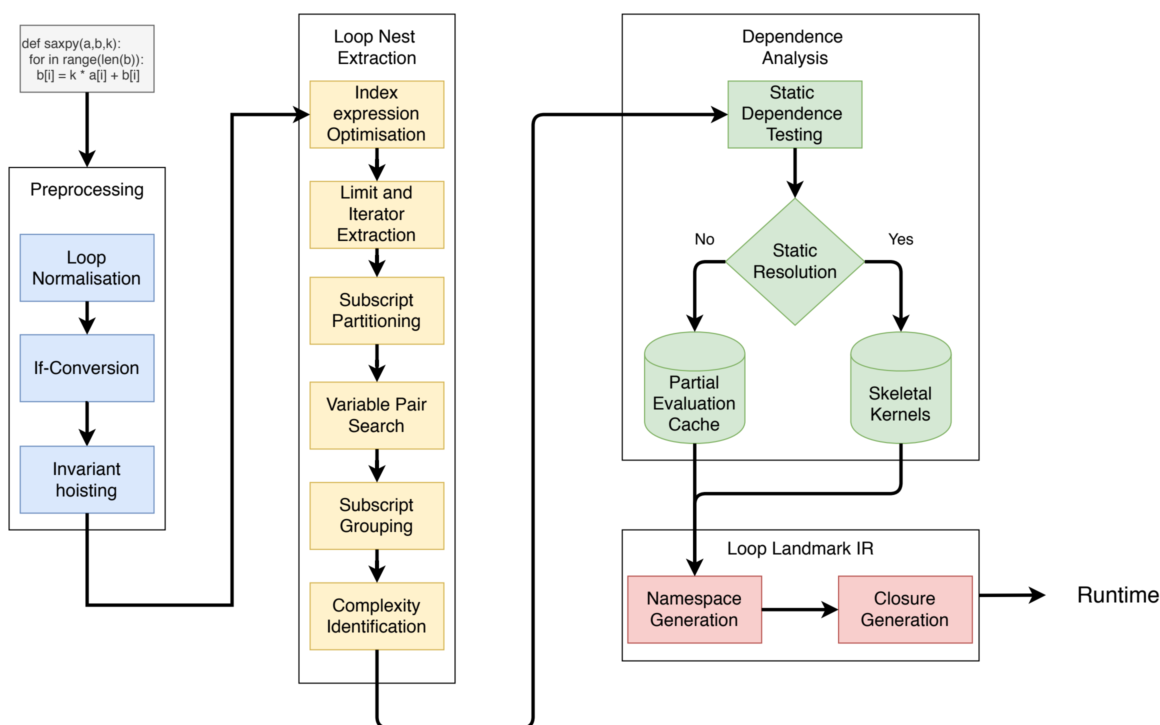



Figure 4.5: ALPyNA preprocessing, static loop analysis and skeleton generation.

#### Normalisation

ALPyNA's static analysis phase converts loops into a normalised form. To do this, ALPyNA's AST parser transforms a reference to the loop iterator within array subscripts to the form $(start + (iterator \times stride))$. The loop limits are normalised to achieve a unit loop stride. ALPyNA performs symbolic evaluation to simplify the normalised subscript and limit expressions.

Listing 4.5: Source loop nest before preprocessing.

```
1 def mfunc(arg_a, arg_b , test, limits):
2   ylim, xlim  = limits
3   for i in range(ylim):
4     for j in range(2,xlim,4):
5       if test :
6         arg_a[i+1, j] = arg_a[i,j] ** 2 + arg_b[j]
7       else :
8         arg_a[i+1, j] = arg_a[i,j] ** 3 - arg_b[j+1]
```

Normalisation of linear subscript expressions in loop nests is a standard optimisation pass performed in static compilers such as LLVM [79] and JIT compilers such as HotSpot [105]. Standard dependence tests, such as SIV, ZIV and MIV, are performed on loops with a unit loop stride. By transforming loops to a normal form, dependence testing can be simplified.

Loop bounds expressions, i.e. parameters of linear loop iterator generators that are to be evaluated dynamically are hoisted outside the loop nest and stored in temporary variables when safe to do so. Any such expressions that cannot be definitively determined at runtime are marked for runtime evaluation as these may affect the dependence graph and thereby the potential to parallelise the loop nest.

Consider the loop nest shown in Listing 4.5. Each loop within the loop nest is identifed as $(\mathcal{F}_x)$ where $x$ is the iteration variable. The inner loop nest $(\mathcal{F}_j)$ has a stride length of 4 and the iteration domain starts at $2$. ALPyNA replaces the iteration variable *j* by its corresponding normalised expression (Listing 4.6). In this case, the original loop limit has been replaced and hoisted out of the loop nest as a static optimisation.

### If–Conversion

Control flow dependences within a loop nest cannot be modelled using conventional data dependence techniques. Control flow divergence typically occurs due to the presence of `if` statements. A *forward* branch is a branch for which the target of the control flow jumps to a location within the same loop. A branch transferring control to a target outside the loop body is classified as an *exit* branch. This can typically be mapped onto the Python `break` statement.

To model control flow dependences, all statements within a loop body are transformed to a predicated execution form [3]. Each statement is guarded by the presence of a guard condition. The predicate of the guard condition is a boolean expression of compiler generated conditional variables. Scalar expansion [104] is performed on compiler generated conditional variables to increase opportunities for parallelisation. Predicate variables generated for forward branches are expanded to the dimensions of the loops enveloping its definition.

Listing 4.6: Example of loop normalisation and *if-conversion*.

```
1  def mfunc(arg_a, argb, test, limits):
2    ylim, xlim = limits
3    _i_tfend_00_ = (ylim + 0 + 0) // 1
4    _j_tfend_00_ = (xlim + -2 + 3) // 4
5    _condvar_000 = np.full((_i_tfend_00_, _j_tfend_00_), 1)
6    for i in range(0, _i_tfend_00_, 1):
7      for j in range(0, _j_tfend_00_, 1):
8        _condvar_000[i, j] = test
9        if _condvar_000[i, j]:
10         arg_a[i+1, 2 + j*4] = arg_a[i, 2 + j*4 ] ** 2 + arg_b[2 + j*4]
11       if not _condvar_000[i, j]:
12         arg_a[i+1, 2 + j*4] = arg_a[i, 2 + j*4 ] ** 3 - arg_b[3 + j*4]
```

Consider the loop nest in Listing 4.5. The control flow divergence caused by the *if-else* condition is transformed by predicating each statement with a boolean expression. The instances of statements on lines 6 and 8 which are executed depends on the resolution of the conditional expression for each loop instance. Each computational statement is executed if it's own predicate expression evaluates to `True`. When converted into parallel kernels, these predicate conditions along with the statements can be executed in parallel while maintaining the correct data dependences.

Each compiler generated *forward* and *exit* predicate variable is defined outside the loops in which they are used. To simplify the *scalar expansion* of predicate variables, data flow analysis is not performed on compiler generated variables. This optimisation exploits the fact that every *read* from such predicate variables is dominated by a single definition for *forward* branches and by two definitions in the case of *exit* branches. *Forward* branch predicate variables are *defined* at a single point outside the loop in which they are *used*. Predicate variables used for *exit* branches are defined once outside the loop in which they are used and once immediately preceding their use to determine if the *exit* condition has been triggered. As a memory optimisation, *exit* branch predicate variables are expanded to a vector with the dimensions of size of the innermost loop that references the exit branch predication variable. This is due to the existence of a loop carried dependence between the source and sink of the compiler–introduced exit branch predication variable. This dependence is carried by the innermost loop.

### Partial Dependence Analysis

ALPyNA accelerates loop nests within functions specifically identified by programmers. A single ALPyNA function call (Section 4.1.2) preprocesses loop nests in all such functions to aid analysis (as described above).

Figure 4.5 shows ALPyNA's static analysis phase comprising its parser, 'loop landmark IR'

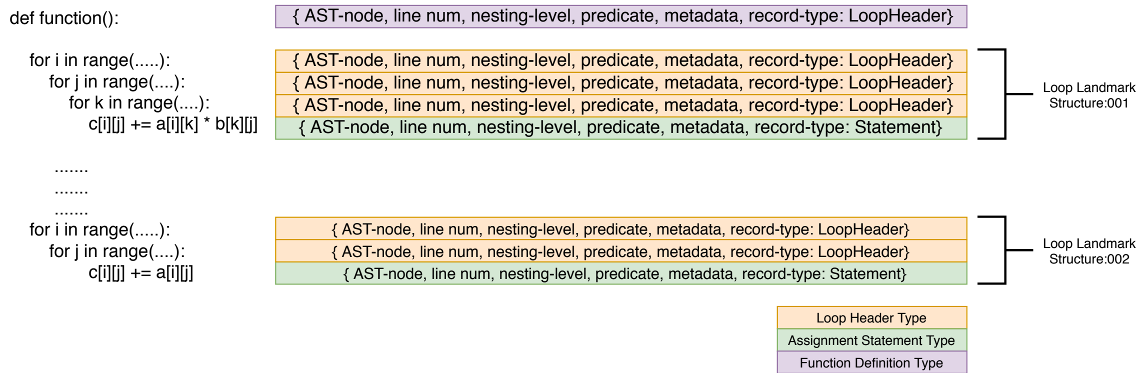



Figure 4.6: Simplified structure of loop landmark structure retained by ALPyNA for runtime analysis.

structure and skeletal kernel generation. Each function is parsed using Python's AST library to create a flat record structure consisting of 'loop landmarks', i.e. fragments of abstract syntax that determine the looping behaviour. This record structure comprises of information required for dependence analysis of the loop. These are grouped together to create 'loop landmarks', as well as group variable and subscript pairs together according to their complexity.

If all the loop bounds and data dependences can be determined statically, ALPyNA can generate the untyped GPU kernels (corresponding to the statements in the loop nest body) at compile time and cache these kernels in memory to reduce dynamic analysis time (Figure 4.5). In such a scenario, only the type information is required to be patched into the cached untyped kernel. This is obtained at runtime by using introspection (Section 4.4.2).

Along with the GPU kernels, the corresponding orchestration code for execution and data management that respects loop carried dependence constraints is also generated and cached. This code will henceforth be referred to as *drivers* in this thesis. Such drivers are only generated if the dependence structure of the loop nest can be determined ahead of time. The auto-generated drivers are responsible for transferring data and marshalling scalars (Section 4.4.2 – GPU Scalar Marshalling) and for executing kernels without violating dependence constraints. If any data dependence cannot be determined statically, the partial evaluation of dependences is cached within the 'loop landmark IR' to use during runtime evaluation of loop nest dependences. Figure 4.6 shows a simplified form of these in-memory data structures used for dependence analysis. Scalar variable writes in loop nests are a special case, requiring runtime analysis and code generation (Section 4.4.2 – GPU Scalar Marshalling).

On the other hand, if the static analysis cannot determine loop bounds at compile time, then it will mark the loop nest for dependence analysis at runtime. The 'landmark' record structure

Listing 4.7: Python loop nest input to ALPyNA – *saxpy*.

```
1 def saxpy( arr_y , arr_x , constval ):
2   for idx_i in range(len(arr_y)):
3     arr_y[idx_i] = constval * arr_x[idx_i] + arr_y[idx_i]
```

containing the original Python AST along with loop nests marked by ALPyNA for deferred analysis are preserved as in-memory data structures. These data structures are carried over to the runtime execution context to aid dynamic dependence analysis. At runtime, parallelisation is performed on loop nests that have been deferred for runtime dependence analysis.

### Nested Function Generation

ALPyNA statically replaces loop nests within the body of marked functions with nested functions. These nested functions initiate dynamic dependence analysis, type inspection and runtime code generation. During parsing, the Python AST node representing the outermost loop is replaced with a nested function that is generated after parsing and analysing the original loop nest.

The standard Python AST library is utilised to replace the original Python function with a nested function. Expressions from the original source code and definitions of compiler generated temporary variables that are hoisted entirely out of the loop nest are marked as `nonlocal` within the nested function. This ensures that during introspection variables are dereferenced from the correct scope.

During construction of the replacement nested functions, ALPyNA generates a namespace within the Python runtime for insertion of JIT compiled kernels. A unique name is generated for each nested function. Loop nest execution sites within the original source code are replaced with calls to the nested functions. Kernels generated at runtime are mapped to the unique nested function name.

A container data structure holding all the variables hoisted out of the loop nest is passed into a *variant selection* function that analyses and compiles code for the loop nest. This data structure acts as a handle to the nested function to introspect and dynamically query loop limits and other variables marked for runtime analysis (such as loop-invariants). This enables runtime analysis to accurately determine loop dependences and parallelise according to the actual runtime dependence. In contrast, a purely static compiler would need to conservatively assume the existence of all dependences that it cannot explicitly disprove with information available ahead-of-time.

Consider the function `saxpy` shown in Listing 4.7. After parsing the loop nest and building the loop parallelisation records (Section 4.3.1) the code is internally transformed to the form

Listing 4.8: ALPyNA – static transformation of *saxpy* with closures.

```
def saxpy(arr_y, arr_x, constval):

  def saxpy_lnest_000():
    nonlocal constval
    nonlocal arr_x
    nonlocal arr_y
    nonlocal _idx_i_tfend_00_
    xfer_container = rt.Closure_Arg_Container()
    xfer_container.constval = constval
    xfer_container.arr_x = arr_x
    xfer_container.arr_y = arr_y
    xfer_container._idx_i_tfend_00_ = _idx_i_tfend_00_
    rt._ufunc_mux(xfer_container, mod_self, 'saxpy_lnest_000', mod_opts)
    constval = xfer_container.constval
    arr_x = xfer_container.arr_x
    arr_y = xfer_container.arr_y
    _idx_i_tfend_00_ = xfer_container._idx_i_tfend_00_
  _idx_i_tfend_00_ = len(arr_y)
  _idx_i_tfend_00_ = (_idx_i_tfend_00_ + 0) // 1
  saxpy_lnest_000()
```

shown in Listing 4.8. The variables identified as within the scope of the loop are `arr_y`, `arr_x` and `constval`. Along with the variables, the expression to determine the iteration domain of the loop is also hoisted out of the loop. Variables that have been taken out of the scope of the loop nest are explicitly marked as `nonlocal` within the nested function (lines 4–7). The nested function marshals all the variables used by the loop nest into a variant selection function along with a reference to the closure (line 13).

The nested function also has access to the 'loop landmark' structure of the AST and the partial evaluation cache (Section 4.3.1). The AST nodes are required to perform runtime dependence analysis on the loop nests. To prevent the AST nodes of the original loop nest from being garbage collected by Python's reference counting GC, the nodes are specifically added to a GC-barrier structure. In addition to maintaining the outermost node of the loop nest, all nodes within the loop records which contain references to child AST nodes of the outermost loop AST node are also not garbage collected. This preserves them for dependence analysis of each variable pair and runtime code generation.

ALPyNA uses the handle to the nested function to introspect vectors and variables for their types and vector dimensions. Variable types discovered at runtime are patched into the generated kernels to create typed variants of kernels that are compiled into binary form. Each compiled kernel is then cached by Numba, which is used as a backend compiler for the CPU and the GPU (Section 2.1.4 and 3.2). The *variant-selection* function also encapsulates the introspection, cost model and device selection logic. ALPyNA also introspects the binding of loop limit expression evaluations to inform runtime dependence analysis.

### ALPyNA Module Generation

The final stage of ALPyNA's static compilation phase is to prepare all data structures required for runtime evaluation, compilation and execution. Each loop nest is replaced with a nested function (Section 4.3.1) which channels all execution into a call that selects a target device to execute a loop nest. Each nested function is given a unique handle for the ALPyNA runtime to dereference on demand.

ALPyNA dynamically generates a Python module and returns a handle to this module as a Python object to any code that calls the ALPyNA static analysis function. Any in-memory data structures generated during the static analysis phase that are required for runtime evaluation are stored in this newly generated module within their own namespaces. A reference to the module is stored within itself to map the function to its corresponding in-memory *partial evaluation cache*. Numeric code very often imports *Numpy* arrays and *math* functions. These modules along with the 'Numba' compiler (Sections 2.1.4 and 3.2) are imported directly into the auto-generated and compiled ALPyNA module to reduce the overhead associated with locating and importing these modules during runtime. Every call to the execution context of a loop nest within a function marked for analysis can now be intercepted by ALPyNA for runtime analysis and code generation. At this stage control is handed over to the ALPyNA runtime.

## 4.3.2 Runtime Analysis

After ALPyNA has completed the static analysis phase of compilation, a reference to the generated ALPyNA module (described in Section 4.3.1) is returned to the caller. When a programmer dereferences a function within this module, the closures representing each loop nest within the original source lazily invoke runtime dependence analysis.

Figure 4.7 shows the overall analysis process at runtime. It starts when a function previously statically analysed and prepared by ALPyNA for runtime dependence analysis is executed. The location of a loop nest invocation is trapped by the closure that statically replaced the loop nest.

If all the dependence relationships in the loop nest were resolved statically, loop nest variable types are inspected, patched into the statically generated kernel, compiled and executed. Any loop nest with dependences unresolved at compile time is analysed again. Dependence relationships that were deduced statically are retrieved from the partial evaluation cache. This is done to reduce analysis time during run time. These dependence relationships are then augmented with dependence relationships that are derived specifically for each execution instance of the loop nest.

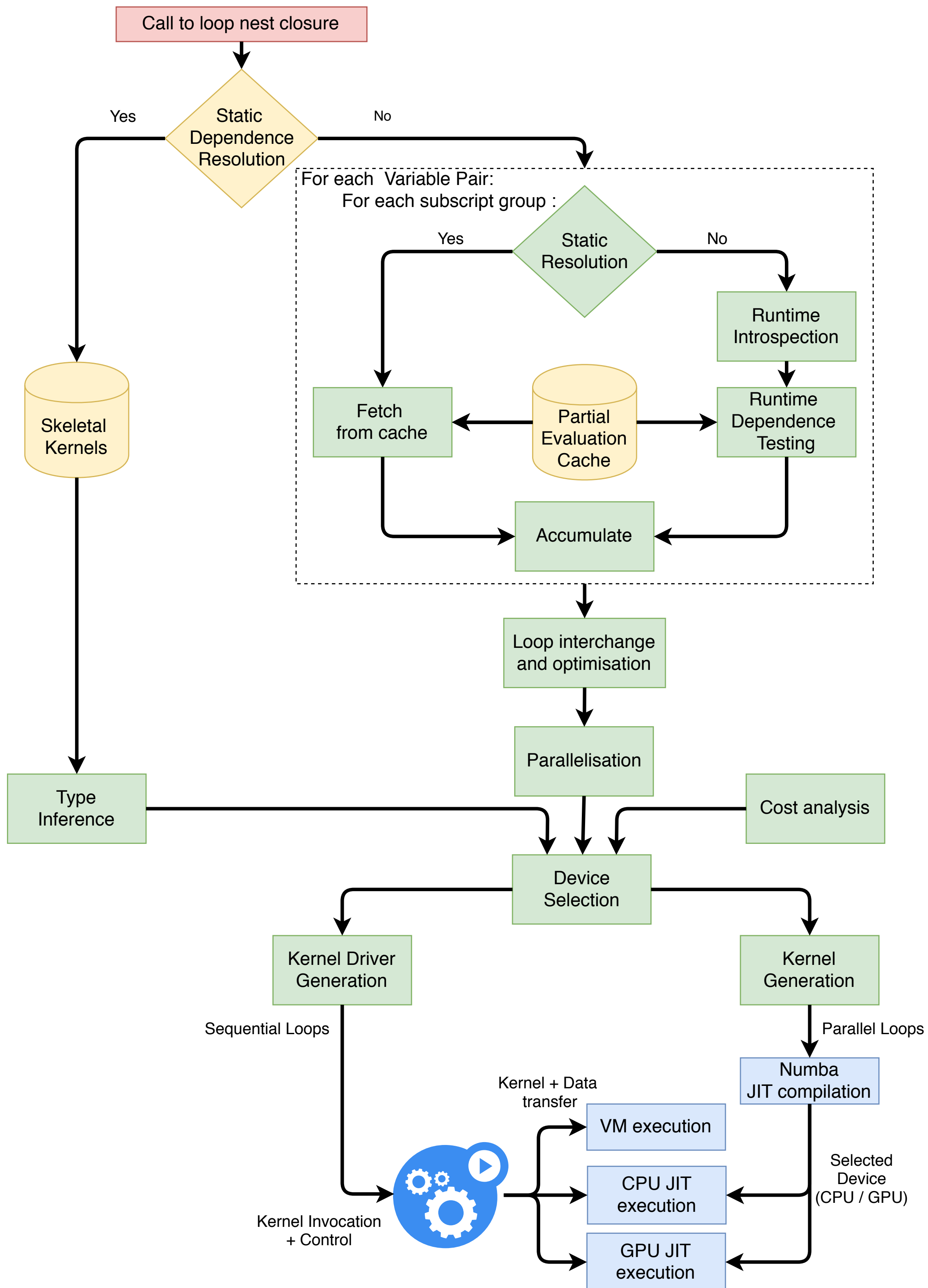



Figure 4.7: ALPyNA runtime dependence analysis and kernel generation. Dependence relationships (resolved and deferred) and untyped statically generated kernels are retrieved from static analysis (yellow) and augmented at runtime with runtime analysis (green).

However, in many instances of numerical loop nests, sufficient information about loop iteration domains and non-iterator variables cannot be obtained statically. To ensure correctness, static loop parallelisation conservatively assumes a dependence exists when fine grained resolution of the dependence relationship cannot be established. This hinders opportunities for optimisation and parallelisation. ALPyNA utilises the Python language's reflection capability [131] to introspect expressions that inhibit such fine resolution of dependence relationships.

The dependence graph is generated for each instantiation of a loop nest. Regenerating the exact dependence graph, where possible, at runtime enables ALPyNA to generate an optimised parallel variant of a loop nest. GPU kernels and their associated drivers are generated with the newly discovered dependence constraints. Resolution of the expressions which make up the loop domain are used to set the GPU *grid* and *block* sizes with suitable padding to conform to the minimum thread-block sizes (Section 4.4.1).

### 4.3.3 Hardware Abstraction Layer

One of the goals of ALPyNA is to be able to extend runtime loop analysis and execute loop nests on a wide range of accelerator devices that may be present in a heterogeneous environment. To enable this, ALPyNA's loop parallelisation and higher level optimisation passes are abstracted away from the code generator for each accelerator by a Hardware Abstraction Layer (HAL). The HAL layer encapsulates all architecture specific code generation to enable portability. ALPyNA currently supports NVIDIA GPUs using CUDA. However, the underlying Numba compiler has support for accelerator devices that have OpenCL compilers. ALPyNA can be extended to generate code for devices with an OpenCL compiler by generating the OpenCL primitives required to implement the API.

The HAL currently enforces the implementation of the following code generation API for each accelerator:

- **Data Transfer**: Code should be generated to transfer all vectors referenced by a compute kernel to the memory space of the target device. All data transfer primitives between the host and target device are hoisted out of the loop nest execution. The HAL API provides a '*prologue*' and '*epilogue*' function which should respectively generate code for data transfer to the accelerator and back.
- **Kernel Type Patching**: Python being a dynamically typed language, ALPyNA introspects the CPython VM for variable types and patches these before JIT compilation.
- **Thread-Hierarchy Calculation**: Modern GPUs arrange groups of parallel threads into a hierarchical structure. CUDA refers to these as *grids* and *thread-blocks*. This

is executed either during the static skeletal code generation phase or at runtime when ALPyNA resolves the domain iteration sizes (Section 4.4.1 – GPU Thread Hierarchy).

- **Kernel Generation**: Untyped kernel code is generated and cached. If these kernels are generated during static analysis, these are cached and preserved to reuse at runtime (Figure 4.7). These functions are only called at runtime if dependence analysis of the loop nest was deferred for runtime analysis.

- **Patch Scalar-Write Kernels**: A statement that *writes* to a scalar and *reads* from it after loop nest execution is marshalled into a compiler generated vector. Marshalling such primitives into type specific vectors determined at runtime (Section 4.4.2 – GPU Scalar Marshalling) reduces transfer overhead. These scalars are then dereferenced in each runtime generated kernel according to their type and position within the compiler generated vector.

- **Kernel Driver Generation**: Host side code that correctly maintains loop carried dependences and invokes parallel kernels is generated during static compilation or at runtime. This code is generated during static analysis if all dependences can be determined at that stage. If any dependence analysis was deferred to runtime (Section 4.3.1 – Partial Dependence Analysis) or if any statements with scalar writes (Section 4.4.2 – GPU Scalar Marshalling) are present within the loop nest, the host side code generation is also deferred to runtime. Any loops that carry dependences are executed sequentially while executing parallel kernels with the correct thread-hierarchy inferred from runtime introspection of loop limits. Runtime kernel driver generation can tailor the number of sequential loops executed on the host according to dependences that are determined at runtime.

- **Cost Model Calculation**: ALPyNA calculates a relative cost for each accelerator in a heterogeneous environment. The execution cost is calculated relative to the performance of the VM. These cost calculation is performed after resolution of dependences and types at runtime. These costs should take execution characteristics of the accelerator device, how many parallel threads are executing, and the relative speed of the VM executing sequential loops on the host device to schedule kernels. The cost model is further discussed in Chapter 6.

## 4.4 Code Generation

If all loop nest dependence relationships can be ascertained statically, the code generator creates and caches untyped skeletal kernels. The corresponding host driver code to execute sequential loops and schedule parallel kernels is also generated and cached.

Runtime optimisations may change which loops are selected to execute within the interpreter. Adapting to these runtime dependences will change the structure of the kernels generated. Statements with writes to a kernel require runtime marshalling into type based vectors (Section 4.4.2 – GPU Scalar Marshalling) which in turn will change the calling convention of accelerator kernels from the host device. Rather than speculatively generating multiple kernel variants for a loop nest, ALPyNA also defers code generation to runtime.

To maintain the API of the HAL layer, patching of types for JIT compilation is performed by ALPyNA. JIT compilation of a loop-nest for the CPU is done by Numba's LLVM based backend. The generated code is single threaded and is not vectorised. A relative execution cost is calculated for the CPU variant (Chapter 6) to decide whether execution of the CPU variant over the GPU variant is quicker. Optional instrumentation is also utilised to generate install-time profiling to initialise the cost model. To reduce overall compilation time, kernels are JIT compiled only for the target device selected by the cost model. After compilation, ALPyNA maps the compiled binary CPU kernel to the CPU namespace of the closure representing the loop nest.

### 4.4.1 GPU Code Generation

ALPyNA generates a single GPU kernel for each statement within a loop nest. This design decision helps ALPyNA to extract the maximum amount of parallelism from a loop nest in the following situations :

- **Imperfect Loops**: the number of loops dominating each statement vary within an imperfect loop nest. Statements that are nested deeper within an imperfect loop nest have a larger number of execution instances relative to statements that are nested to a lesser extent. This in turn automatically implies differing amounts of parallel execution instances for such statements.
- **Differing parallelisation constraints**: Two statements, within perfect or imperfect loop nests, may potentially have different scheduling constraints due to loop carried dependence constraints on each statement. This may result in different loops requiring sequential execution to maintain loop carried dependences between two statements (e.g. dependence graph shown in Figure 4.1 for Listing 4.4).

Figure 4.8 shows the interaction between the interpreter and the generated GPU kernels for the transformed loop nest in Listing 4.6. A kernel GPU is generated for each statement within the original loop nest. Dependence analysis evaluates ordering constraints between statements and determines which loops have to be executed sequentially. Kernels representing each statement are called in the topological order in which the dependence graph is

parsed. The whole set of kernels are executed together on the GPU to avoid data transfer between the interpreter and the GPU within the context of a loop nest. Data access between kernels occurs through GPU global memory.

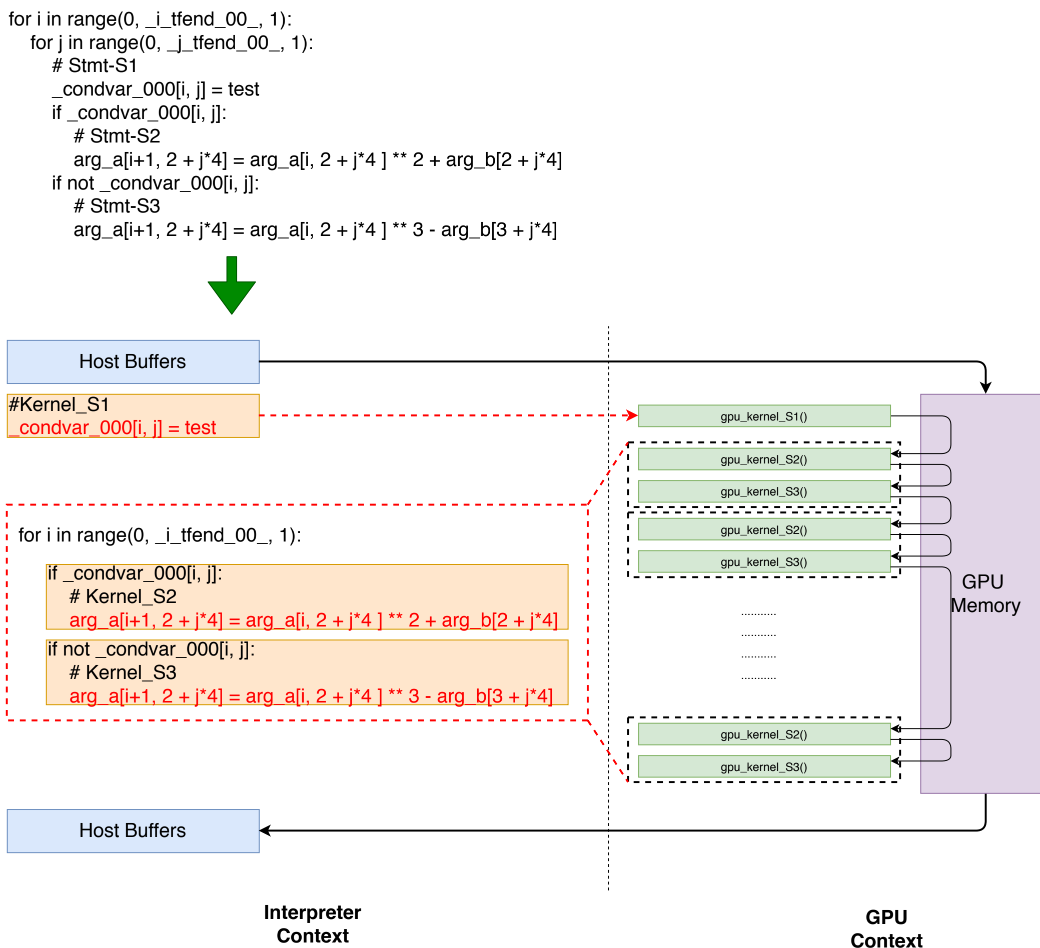



Figure 4.8: Interaction between interpreter and GPU during execution of loop statements. Inter-kernel data access is through global GPU memory.

Awaiting runtime typing, untyped kernels are generated and cached during static analysis if all the dependence relationships are ascertained at compile time. The corresponding host-side driver code is also generated and cached. At runtime, reflection is used to infer the types of each variable used by the cached kernels, JIT compiled using Numba (Section 2.1.4 and 3.2) and executed. Figure 4.7 shows how ALPyNA re-uses the cached 'skeletal' kernels to save analysis time during dynamic analysis. It also provides a high level view of the runtime loop-to-thread mapping within a kernel and the '*driver*' code maintains the correct GPU kernel invocation order.

If dependence analysis is deferred to runtime, the structure of the kernels and the 'driver' code could potentially change. This happens due to one or more of the following factors.

- A loop which carries a dependence is mapped onto a sequentially executed loop in the 'driver' and executed by the VM. These loops will invoke calls to schedule dynamically generated kernels on the GPU while maintaining dependence constraints. This dependence structure may change due to runtime dependence analysis (Section 4.2).
- Mapping some loops to threads while executing other loops sequentially within a kernel necessitates runtime mapping of parallel *thread-ids* to iterator values within the array subscripts (Section 4.4.1 – GPU Thread Hierarchy).
- Statements with writes to a scalar require pass-by-reference semantics to preserve the value of the scalar for reference within other GPU kernels as well as outside the scope of the loop nest.

ALPyNA selects loops that can be safely parallelised and maps these parallel loops to parallel threads. A loop execution instance of a statement is effectively executed as a GPU thread. In CUDA, these parallel threads are mapped onto a three-dimensional thread domain space (Section 2.3.2). The iterator of the original loop is mapped to a parallel GPU *thread-id*. The *thread-id*s are dereferenced using CUDA $(blockid_{axis} * blocksize_{axis} + threadid)$ semantics. Current GPUs require a minimum number of threads to execute a kernel. This causes the total number of threads to be rounded up to a multiple of the minimum number of threads. To prevent an invalid access of memory by excess threads, the code generator inserts guard conditions to prevent array dereferencing for threads along each parallel axis (Section 4.4.1 – GPU Thread Hierarchy). The iteration domain sizes for each parallel hardware axis is passed *by-value* to each executing kernel to check within the guard condition.

Outer loops that carry dependences are executed sequentially by the interpreter to schedule kernels. To maintain dependence constraints, a kernel instance should know the exact loop execution instance of the sequentially executed loop carrying the dependence. The iterator value representing the loop execution instance of these sequential loops is also passed *by-value* to the kernel function to ensure that dependence constraints imposed by loop-carried dependences are maintained.

Listing 4.9 shows the kernel generated for the *saxpy* benchmark (Listing 4.7). As *saxpy* has only one iteration domain, the size of this domain is passed *by-value* ( `idx_i_iterdom_max` – *line 2*). This size is used in the kernel guard condition (*line 8*) to prevent incorrect memory access to the vector from an excess number of threads that may execute due to thread padding (Section 4.4.1). The array subscripts are a linear function (*line 10*) of the *thread-id* (*line 7*).

ALPyNA generates the host side driver to schedule kernels while maintaining dependence constraints discovered at runtime. Any loops carrying dependences for each statement are transformed into loops invoking GPU kernels. Such sequential loops are executed by the

Listing 4.9: GPU kernel code generated at runtime for *saxpy* (Listing 4.7)

```
1  @cuda.jit('int64,uint32[:],uint32[:],int32')
2  def _gpu_saxpy_lnest_000_206(constval, arr_x, arr_y, idx_i_iterdom_max):
3    bsize_x = cuda.blockDim.x
4    bid_x = cuda.blockIdx.x
5    tid_x = cuda.threadIdx.x
6
7    idx_i = bid_x * bsize_x + tid_x
8    if idx_i >= idx_i_iterdom_max :
9      return
10   arr_y[idx_i * 1] = constval * arr_x[idx_i * 1] + arr_y[idx_i * 1]
```

CPython VM. This enables ALPyNA to generate kernels which do not require the insertion of *synchronisation barriers*.

Listing 4.10 shows the runtime host side driver generated by ALPyNA for the *saxpy* benchmark. The host side code *(i)* marshals and unmarshals data from ALPyNA's auto-generated closure (lines 3–5 and 19–22), *(ii)* transfers data (lines 10–11 and 16–17), *(iii)* sets up the GPU context (line 7) and *(iv)* generates a thread-hierarchy according to the iteration domain size (line 12).

Listing 4.10: GPU driver code generated at runtime for *saxpy* (Listing 4.7)

```
1
2  def _gpu_saxpy_lnest_000(arg_container, _timer):
3    arr_y = arg_container.arr_y
4    constval = arg_container.constval
5    arr_x = arg_container.arr_x
6    _idx_i_tfend_00_ = arg_container._idx_i_tfend_00_
7    stream = cuda.stream()
8    _timer.start_exec()
9    with stream.auto_synchronize():
10     _cvec_arr_y = cuda.to_device(arr_y,stream)
11     _cvec_arr_x = cuda.to_device(arr_x,stream)
12     _gpu_saxpy_lnest_000_206[(32768,), (1024,), stream](constval,
13                                                        _cvec_arr_x,
14                                                        _cvec_arr_y,
15                                                        _idx_i_tfend_00_)
16     _cvec_arr_x.to_host(stream)
17     _cvec_arr_y.to_host(stream)
18   _timer.end_exec()
19   arg_container._idx_i_tfend_00_ = _idx_i_tfend_00_
20   arg_container.arr_x = arr_x
21   arg_container.constval = constval
22   arg_container.arr_y = arr_y
```

While parsing the AST during ALPyNA's static analysis phase, all data variables that are read from or written to are memoised and stored within the loop-landmark data structure (Section 4.3.1). ALPyNA automatically generates commands to create and transfer vectors

to and from the GPU. These commands are hoisted out of the loop nest execution to prevent unnecessary data transfer between each kernel invocation.

### GPU Thread Hierarchy

In CUDA terminology, a group of threads on the GPU constitute a *thread-block* and a group of *thread-blocks* constitute a grid. These thread groupings may be spread over *one, two* or *three* dimensions. Each GPU has minimum and maximum *threads-per-block* and in some cases, the maximum number of threads in one dimension may not be the same as others.

ALPyNA determines loop bounds statically by parsing the original code (Section 4.3.1) or dynamically using introspection within the context of the closure that represents the loop nest (Section 4.3.2). The bounds for each instantiation of a loop nest are determined by introspecting the evaluation of the bounds expression at runtime.

Dependence analysis [70] determines which loops should be executed sequentially for each statement. All other loops that envelope the original statement in the loop nest are executed in parallel within a GPU kernel. Each iteration of a loop executed in parallel is mapped on to a thread that can be calculated using a GPU's $(blockid_{axis} * blocksize_{axis} + threadid)$ semantics.

ALPyNA matches the number of *threads-per-block* and *blocks-per-grid* based on device specific constraints and the iteration domain sizes of a loop nest. Loop bounds for each parallel loop enveloping a statement determine how many threads are required to execute it in parallel. The loops that are determined to be safe to execute in parallel are sorted in descending order of their iteration domain sizes. Each parallel loop is assigned to one of the GPU's *thread axes*, up to the maximum number of axes (three in modern GPUs) supported by the device. Sorting the loops in descending order maximises the parallelisable iteration domain space when the number of parallel loops is greater than the number of *thread axes* supported by the GPU.

If the number of parallel loops is greater than the number of parallel axes allowed by the accelerator, these loops are executed sequentially within each kernel. Correctness is preserved as all the outer loops carrying dependences are executed sequentially in the interpreter. This enables all inner loops to be executed in parallel without causing a dependence violation.

To calculate the CUDA thread hierarchy for each instantiation of a loop nest, an initial block size equal to the iteration domain sizes of each parallel loop is generated. This allocation is chosen if it fits in the maximum block size of the GPU device. If the number of threads is greater than the maximum block size of the GPU, the number of threads along the largest of the allocated parallel axes in a block is iteratively halved and padded to the minimum number of execution threads required by CUDA. On NVIDIA GPUs, this is equal to the

size of a single warp [102] – 32 in modern NVIDIA GPUs. This process is continued until the *block* size is small enough to fit the GPU's maximum *block* size. During each iteration, the *grid* size along the corresponding axes is adjusted to match the reduction in the *block* size. We converge on the *grid* and *block* sizes in $\left\lceil log_2\left(\frac{\prod_{i=1}^{n} D_i}{B}\right)\right\rceil$ iterations where $D_i$ is the the domain size of a loop allocated to execute in parallel, $B$ is the maximum block size supported by the GPU and $n$ is the number of parallel thread axes allocated; e.g. a $1k \times 1k$ domain size for Naïve Matrix multiplication, would converge to a thread hierarchy of $(grid, block) \rightarrow ((32 \times 32), (32 \times 32))$ within 10 iterations.

### 4.4.2 Runtime Type Patching

Dynamic languages such as Python resolve operand types at runtime. This complicates the generation of GPU code as JIT compilation of such kernels require them to be compiled with type information. Python3 provides support for type annotations which can be taken as a hint for compilation; this is not enforced.

ALPyNA uses Numba as the backend compiler to compile kernels to binary form. Numba can perform automatic runtime type evaluation to apply to each kernel immediately before compilation. Automatic type inference occurs on every invocation of a kernel. This feature can be bypassed by a programmer by providing the types for the Numba compiler to generate a binary. Automatic type inference incurs an order of magnitude penalty in compilation time compared to programmer defined types [1].

ALPyNA executes loops carrying dependences sequentially. Kernels in the loop body are executed and dispatched for execution by Numba. When a kernel is invoked multiple times sequentially, Numba's automatic type inference is executed for each invocation. To reduce the compilation overhead of Numba's type inference, ALPyNA hoists type inference outside the loop nest and supplies each generated GPU kernel with the required types. Listing 4.9 shows the typed GPU kernel generated at runtime for the *saxpy* benchmark (Listing 4.7). Type-information is inferred at runtime using introspection and is added to the runtime generated kernel (line-1). Numba can still rely on cached kernels that are dereferenced by a hash of the supplied kernel signature. This is reused in every kernel invocation within a sequentially executed loop.

ALPyNA relies on Numba's CUDA interface to serialise data and to transfer it to/from the GPU. Numba only supports Numpy arrays that are already byte aligned in memory for the array element type. For GPU kernels, Numba exposes bindings to CUDA intrinsics that map to the GPU *grid*, *block* and *synchronise* programming primitives to the programmer. Numba also exposes *pure intrinsic* functions [142] provided by CUDA for numerical computations.

[1] measured on the Desktop platform T2 specified in Chapter 5

ALPyNA uses a one-to-one mapping of such functions to enable programmers to use them within a loop nest.

### GPU Scalar Marshalling

The effect of a write to a scalar variable within a loop body must be preserved for subsequent reads. This is required both for uses of the variable in other kernels while executing on the GPU as well as any use outside the scope of the loop nest.

However, GPUs do not support call-by-reference. Scalar writes within a kernel instance are not visible beyond that kernel instance. To preserve its value for subsequent reads, a scalar can be transferred as a unit-size vector to the GPU. Individual transfers of each such scalar incurs a large cost due primarily to the large execution overhead of the device drivers responsible for setting up any such transfer as well as the latency incurred for each individual transfer across a peripheral bus such as PCIe.

To mitigate this overhead, ALPyNA performs runtime introspection to determine the type required for each scalar variable that is written to. For scalar values that are to be offloaded to the GPU, one composite array per type is dynamically assembled. This technique is similarly employed in Tornado [46]. This composite array can be dereferenced within a runtime generated GPU kernel using $vector + displacement$ semantics. The closure representing the loop nest unmarshalls each composite array back into the appropriate original scalar variables upon the completion of a loop nest execution.

## 4.5 Summary

This chapter presents the ALPyNA framework, a loop parallelisation framework for non-expert programmers. It parallelises numerical computation expressed as dense Python loop nests and automatically JIT compiles them for execution on CPUs and GPUs. The main motivations behind the design are :

- **dynamically maximise parallelisation**: by exploiting dynamic analysis to determine dependences and iteration domain sizes, code generation and and JIT compilation is optimised to achieve greater levels of parallelism than possible in an entirely static system.
- **productivity**: by showing that non-expert programmers need not be aware of the precise low-level programming paradigm that is required to program a GPU.
- **extensibility**: by being able to target multiple accelerators within a heterogeneous environment. ALPyNA is currently able to target CPU and GPU. ALPyNA's HAL layer

enables easy integration of other compilers targeting accelerators such as multicores and FPGAs.

ALPyNA stages compilation by combining static and dynamic analysis for loop nests. The static analysis phase parses loop nests, and decides to generate skeletal kernels or defer analysis to runtime to obtain a more precise understanding of the dependence relationships between memory accesses within statements in a loop nest. During dynamic analysis, loop nests for which skeletal kernels have been generated are patched with types inferred by introspecting the Python runtime for the types and dimensionality. If the dependences cannot be resolved during static analysis, dependence analysis is repeated at runtime. ALPyNA utilises runtime introspection to determine runtime dependences. The precise runtime determination of dependences enhances parallelisation opportunities compared to the conservative assumptions that have to be made during static compilation.

This chapter also describes the challenges and design decisions of generating and JIT compiling CPU and GPU kernels to execute a loop nest in a dynamic language. ALPyNA generates GPU kernels customised to the dependence relationships that emerge at runtime. A detailed performance evaluation of ALPyNA is presented in Chapter 5.

# Chapter 5

# Performance Evaluation of ALPyNA

Where Chapter 4 described the design of ALPyNA's dynamic loop analysis framework, this chapter evaluates the performance of JIT compiled code generated by ALPyNA. ALPyNA's performance is evaluated using 12 loop intensive numerical benchmarks on two CPU – GPU systems; one is a server grade machine while the other is a desktop machine. Experiments performed over a wide range of iteration domain sizes point to differing optimal target execution devices for different domain sizes. ALPyNA's JIT compiled GPU code achieves orders of magnitude speedup relative to interpreter execution across all benchmarks and some speedup (up to 179x) relative to the JIT compiled CPU execution for some benchmarks.

Section 5.1 describes the benchmarks and their characteristics that exercise various features of the ALPyNA framework. Section 5.2 describes the hardware and software setup used to evaluate the benchmarks. Section 5.3 describes the methodology used in the evaluation. Section 5.4 compares the runtime performance of the benchmarks on two CPU–GPU platforms. Performance of the loop nest variant for the CPU and the GPU is compared with the interpreter. The performance of the GPU is also compared with the CPU. The results are tabulated for both platforms. Section 5.5 discusses the overhead required to analyse and compile code at runtime. Finally Section 5.6 summarises the chapter.

## 5.1 Benchmarks

ALPyNA is evaluated using 12 loop-intensive benchmarks taken from the BLAS routines in the Polybench suite [113], the Numba benchmarks, and from domains such as finance (Black-Scholes) and digital signal processing (filterbank correlation – *fbcorr*). The benchmarks represent a variety of characteristics that test ALPyNA's dynamic runtime loop parallelisation capabilities. These benchmarks have been used widely to evaluate parallelisation in runtime systems (e.g. [36, 46, 115, 126]).

| Benchmark | Loop Depth | | Statements | Control Flow Divergence | Pure Intrinsic Functions |
|---|---|---|---|---|---|
| | Total Loops | Parallel Loops | | | |
| black-scholes | 1 | 1 | 12 | ✓ | ✓ |
| conv-2d | 4 | 2 | 1 | ✗ | ✗ |
| conway | 2 | 2 | 2 | ✗ | ✗ |
| gemm | 3 | 2 | 1 | ✗ | ✗ |
| gemver | (2,2,1,2) | (2,1,1,1) | 4 | ✗ | ✗ |
| hilbert | 2 | 2 | 1 | ✗ | ✗ |
| jacobi | 2 | 2 | 2 | ✗ | ✗ |
| mandelbrot | 3 | 2 | 3 | ✓ | ✓ |
| saxpy | 1 | 1 | 1 | ✗ | ✗ |
| syr2k | (2,3) | (2,2) | 2 | ✗ | ✗ |
| vadd | 1 | 1 | 1 | ✗ | ✗ |
| fbcorr | 7 | 4 | 1 | ✗ | ✗ |

Table 5.1: Benchmark characteristics described by the number of loops surrounding each statement. Imperfect loops are depicted by showing the number of loops that surround each statement in the body of the loop nest e.g. syr2k has two loops around the first statement and three around the second. Only some loops are parallelised.

These benchmarks range from extremely simple loops to moderately complex loops. The simplest loops are single loops, containing just a single statement, which are embarrassingly parallel. Moderately complex loop nests contain both perfect and imperfect loop nests. The benchmarks include loop nests containing loop carried dependences. These dependences have to be satisfied by executing loops carrying dependences sequentially while parallelising the other loops. Two benchmarks contain control flow divergence and calls to pure intrinsic functions within the body of the loop nests. Pure intrinsic functions are supported by the hardware which the compiler can inline. Such functions are deterministic functions that *(i)* depend on their own arguments, *(ii)* do not change variables out of their scope and *(iii)* do not produce side effects [142]. Table 5.1 summarises the features of each benchmark by the number of loops that dominate the loop body and how many of these loops have loop carried dependences. Such dependences force sequential execution of the loops that carry the dependence to maintain correctness. On the GPU, this translates to loops carrying dependences executing sequentially in the interpreter. These sequential loops schedule kernels executing inner parallel loops.

When a multidimensional array is de-referenced in the CPython interpreter, Python generates a new object from a sub-slice of the original vector for every subscript dereference. Both Python lists as well as Numpy arrays exhibit this behaviour. Programmers can override this behaviour in Numpy arrays by representing all array dereference expressions as a tuple within a single subscript [99]. To ensure that the additional overhead of object generation does not skew the results in favour of JIT compiled CPU and GPU compiled versions of the

loop nest, all multi-dimensional array subscripts in the benchmarks are re-written using tuple notation.

**black-scholes** implements an option pricing model [26] to calculate prices of derivative financial instruments. It uses various mathematical pure intrinsic functions on each stock price and exhibits control flow divergence with forward branches (cf. Section 4.3.1). The single loop only exhibits loop independent dependences within the body of the loop nest allowing ALPyNA to parallelise the whole loop body.

**conv-2d** convolves an $N * N$ matrix with an $M * M$ kernel represented as

$$y[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j] \times h[m-i, n-j]$$

This is a quadruple nested loop with an $M * M * N * N$ iteration domain. The loops with dimensions $M * M$ must be executed sequentially, while the loops along the $N * N$ domain can be parallelised.

**conway**: “Conway’s Game of Life” [51] is a zero-player game on a 2D board, representing a cellular automaton. Each element is either alive or dead. At each turn, elements are born, survive, or die, based on neighbour elements’ state from the previous turn. This benchmark is the core of the survival calculation, representing a single turn in the game. Effectively, it is a stencil computation across a 2D integer matrix.

**gemm** is an implementation of the standard O($n^3$) dense matrix multiplication algorithm. The loops iterate over the rows and columns of the two-dimensional matrices. The absence of one of the iterators in the access pattern of any subscript of the output matrix generates all three dependence types on the multiply-accumulate statement. This requires the inner *k-loop* to be executed sequentially while allowing the outer *i-* and *j-loops* to be executed in parallel.

**gemver** is a BLAS [1] routine from the Polybench [113] suite. The mathematical calculation is:

$$\hat{A} = A + u1.v1 + u2.v2$$

$$x = \beta \hat{A}^T y + z \ , \ w = \alpha \hat{A} x$$

where the inputs are *A* ($N * N$ matrix), $\alpha, \beta$ (scalars) and *u1, u2, v1, v2, y, z* (vectors each of size $N$). The benchmark consists of four separate loop nests, each with a single statement. All the statements have loop-independent dependences between them. This benchmark has a 1D loop and three 2D loops. One 2D loop can be parallelised across both dimensions. The other two 2D loops do not reference iterators from one of the loops. This causes both a loop carried dependence ($\delta$) and an anti-dependence ($\delta^{-1}$) carried by the loop generating the un-referenced iterator as well as a loop independent dependence ($\delta_\infty$) on itself. In effect, this acts as a reduction along one axis inducing all three dependence types along that axis. These loops can be parallelised across the remaining axes.

**hilbert** computes a 2D matrix used in linear algebra approximation problems. It is calculated as:

$$H_{i,j} = \frac{1}{i + j - 1}$$

The computation is represented as a 2D loop that only writes to a single vector.

When JIT compiled targeting either CPU or GPU, The binary instructions generated by JIT compilation do not require *loads* from memory while performing the compuation. Memory is accessed only for memory *stores*.

**jacobi** is an iterative algorithm to solve a set of linear equations, expressed as a vector product equation. Initial guesses for the solution are plugged into a vector representation. The terms are solved iteratively until the algorithm converges to a solution. This benchmark is the core of the iteration step, a doubly nested loop to compute the next value and the error value for each element in the 2D matrix. The loop nest consists of a perfect loop nest with two loops and two statements. A loop independent *true*-dependence ($\delta_\infty$) between the two statements ensure that every instance of the second statement within the loop will only occur after the corresponding instance of the first statement is executed. Both loops are parallelised and can be executed on the GPU.

**mandelbrot** is a kernel that computes $z_{n+1} = z_n^2 + c$, where $z, c$ are complex numbers with the initial value $z = 0$. The loop nest is implemented as a triply nested loop. A loop carried *true* ($\delta$) and *anti* ($\delta^{-1}$) dependence along with a loop independent *true*-dependence ($\delta_\infty$) exists for all three statements. These dependences are carried by the outer loop enabling parallel execution of the inner loops. A loop carried as well as loop independent output dependence ($\delta^o$) exists between two of the three statements. Two statements are conditionally executed which triggers ALPyNA's *if-conversion* pass.

**saxpy** is single-precision AX plus Y. This benchmark combines scalar multiplication and vector addition on two equally sized arrays of 32-bit floating point values. Mathematically, the computation is represented by $\alpha\vec{x} + \vec{y}$

**syr2k** is a BLAS [1] routine from the Polybench [113] suite. It computes

$$C_{out} = \alpha AB^T + \alpha BA^T + \beta C$$

where *A,B,C* are $N * N$ matrices, and $\alpha, \beta$ are scalars. Dependence analysis generates one statement that can be parallelised along two loops and a second statement with a loop-carried dependence along one axis that runs sequentially and is parallelisable across the other two axes.

**vadd** performs element-wise addition of a pair of 1d vectors. The vector sizes are varied between 1KB to 16MB. The iteration domain size is proportional to the vector length.

**fbcorr** is the filterbank correlation benchmark, used in signal and image processing to classify features. The implementation has seven nested loops of which three must be run in sequential order. The other four loops can be executed in parallel. Current GPU hardware only supports three hardware axes. Since all dependences are carried by the sequentially executing loops, ALPyNA's optimisations will sort the parallel loops in descending order of the iteration domain sizes at runtime. The loop with the smallest loop domain size amongst the parallel loops is executed sequentially within each parallel thread.

## 5.2 Experimental Setup

### 5.2.1 Hardware

ALPyNA's performance is evaluated on two machines; a server grade machine (*M1*) and a typical desktop setup (*M2*). *M1* has a Xeon E5-2620v4 octa core CPU with a 20MB L3 cache and a clock frequency of 2.1GHz that can be 'Turboboost'ed to 3GHz. It has 16GB (2× 8GB) DDR4 RAM with a memory bus speed of 2133MHz.

*M2* has a Core i7-6700 quad core CPU with an 8MB L3 cache clock and a clock frequency of 3.4GHz, that can be boosted to 3.9GHz. It has 16GB DDR4 (2× 8GB) RAM with a memory bus speed of 2133MHz.

*M1*'s GPU is an NVIDIA Titan-XP (GP102) with a clock frequency of 1.4GHz and 12GB of GDDR5 RAM. It has 30 Streaming Multiprocessors (SMs) and a 3MB last level cache (L2). *M2* has an NVIDIA GeForce GTX-1060 (GP104) with a clock frequency of 1.5GHz and 3GB of GDDR5 RAM. It has 9 SMs and a 1.5MB L2 cache. The GP102 and GP104 microarchitectures use SMs that have 128 CUDA cores. These are simple cores that execute together as *warps*. The GP102 and GP104 SMs have a warp size of 32. Each SM has four warp schedulers. This enables four warps of 32 cores to be executed simultaneously. Each SM has two L1 caches [133]. Data transfer occurs over a PCIe 3.0 bus. The GPUs on both *M1* and *M2* have exclusive use their respective PCIe bus. Both GPUs negotiated to use all 16 available channels (x16) on their respective machines.

### 5.2.2 Software

Both machines *M1* and *M2* have similar software stacks. *M1* runs a native x86–64 Linux kernel v4.15. The Python interpreter that executes ALPyNA is CPython 3.6.9. Numpy 1.13.3 is used on *M1* for the array data structures. The CPU and GPU are compiled using Numba v0.34. Numba internally relies on CUDA v8.0.61.

The Linux kernel executing on *M2* is a native x86–64 Linux kernel (v4.9). CPython 3.5.3 is used with Numpy 1.13.3. Compilation and GPU execution is performed by Numba 0.33. Numba in-turn relies on CUDA v8.0.44 for compiling GPU code. We identify the combined hardware and software stacks as target platforms *T1* and *T2* throughout the rest of this thesis.

## 5.3 Methodology

To assess the effectiveness of loop parallelisation, ALPyNA is evaluated on 12 widely used array-intensive benchmarks. These benchmarks are written as nested Python loops in an array-centric format with variable loop domains, to exercise ALPyNA's runtime analysis execution path. ALPyNA's effectiveness is evaluated by measuring the time taken for :

- dependence analysis and CPU and GPU kernel generation.
- CPU and GPU kernel compilation.
- execution time of the CPU and GPU kernel variants.

For the purposes of evaluation, '*execution time*' on an accelerator is the time taken to transfer data back and forth between the host and the accelerator and finish the execution of the entire computation represented by a loop nest. From the programmers point-of-view, a comparison of ALPyNA's performance is made by comparing the effective '*total execution*' time of the JIT compiled CPU and GPU variants *vs* the CPython interpreter. The total execution time includes the time taken for analysis, compilation and execution time.

To profile a wide range of domain sizes for each benchmark the iteration sizes of each loop is doubled (geometrically increasing). Each benchmark has been executed 5 times and the arithmetic mean of the measured values is reported. ALPyNA is capable of automatically choosing a target device to execute an instance of a loop nest using a cost model (Chapter 6). For this evaluation, this automatic selection was disabled and a target device was explicitly chosen to obtain measurements. Each benchmark is executed (i) using the CPython interpreter, (ii) a CPU variant, and (iii) a GPU variant of the loop nest. All benchmark executions are set to timeout after three hours. This resulted in timeouts during the execution of the largest iteration domain sizes of fbcorr and syr2k by the interpreter. The execution times between each experimental run ranged from a maximum of +/-5% for small iteration domain sizes to negligible variances for larger domain sizes.

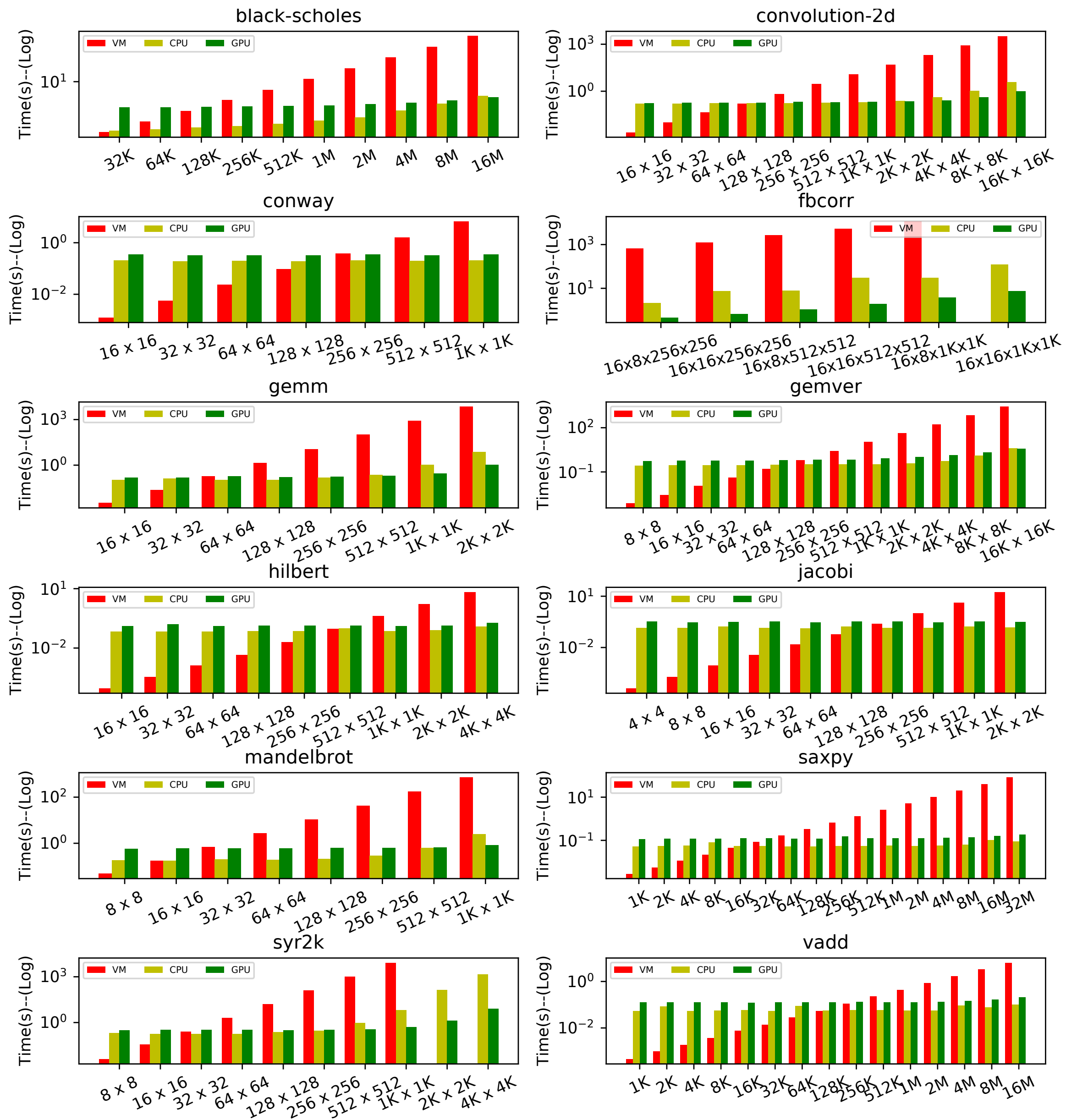



Figure 5.1: Total execution time of ALPyNA generated code on the CPU and GPU vis-a-vis the CPython VM on Platform *T1* (Lower is better). Times are scaled logarithmically. Total execution time includes analysis and code generation, compilation, and execution time. Interpreter execution times for the largest two domains of syr2k ($2k \times 2k$ and $4k \times 4k$) timed out after 3 hours. Results summarised in Table 5.3.

## 5.4 Performance Comparison

The effectiveness of ALPyNA's runtime parallelisation and code generation is evaluated by measuring execution times and relative speedups. Figure 5.1 plots CPython and ALPyNA CPU and GPU runtimes over a large number of iteration domains. The sizes shown on the x–axis on all benchmarks (apart from *fbcorr*) is the size of the largest vector on which the computation is performed. For example, naive matrix multiplication (*gemm*) of two $64 \times 64$ matrices is shown in Figures 5.1, 5.2, 5.3, 5.4 and 5.5 as $64 \times 64$. For *fbcorr*, the iteration domain size is depicted as (number of images)x(number of filters)x(2-D image size). For example, the smallest iteration size ($16 \times 8 \times 256 \times 256$) was performed on $16$ images with $8$ filters and each image is ($256 \times 256$) pixels in size. The domain sizes on the x–axis are depicted on a logarithmic scale. To account for the orders of magnitude difference in execution times between the CPython interpreter and the CPU and GPU kernels that ALPyNA generates, the time axis is also logarithmic. All relative execution time graphs and tables shown in this chapter pertain to platform *T1*. Graphs and tables corresponding to platform *T2* are provided in Appendix A.

Numba caches JIT compiled kernels to reduce compilation times. To compare the effectiveness that adoption of ALPyNA might have in the real world, the performance of the CPython interpreter is compared with the total time taken to complete execution of the loop nest on a cold run. This includes the overhead of *analysis* and *compilation*, as well as actual *execution* of CPU and GPU kernels. Execution times of the GPU are inclusive of data transfer time.

### 5.4.1 Comparison of ALPyNA CPU Variant with CPython Interpreter Execution

Table 5.2 reports the minimum, maximum, and arithmetic mean speedup achieved by the CPU variant of each benchmark over the corresponding execution by the CPython interpreter on platform *T1*. The CPython interpreter executes faster than the JIT compiled CPU variant for the smallest domain sizes. At smaller dimension sizes, the worst relative performance of the CPU variant occurs for the *jacobi* (0.0003x) and *hilbert* (0.001x) at their smallest iteration sizes – $(16 \times 16)$, $(4 \times 4)$ and $(8 \times 8)$ respectively. The largest maximum speedup is achieved for *syr2k* (1199.95x) at $(1K \times 1K)$, *saxpy* (934.84x) at (32M) and *gemm* (925.27x) at domain size $(2K \times 2K)$. The interpreter variant of *syr2k* timed out for the larger execution sizes ($2k \times 2k$ and $4k \times 4k$).

Similarly, columns 2–4 of Table A.1 report minimum, maximum and arithmetic mean speedups by the CPU on platform *T2*. The *hilbert* (0.001x), *jacobi* (0.0002x) and *gemver* (0.002x) benchmarks are the slowest at their respective smallest domain sizes The maximum speedup

| Benchmark | JIT CPU vs CPython | | | Benchmarks | JIT CPU vs CPython | | |
|---|---|---|---|---|---|---|---|
| | min | max | mean | | min | max | mean |
| black-scholes | 0.89 (32K) | 47.58 (16M) | 17.57 | hilbert | 0.001 (16 x 16) | 53.25 (4K x 4K) | 8.95 |
| convolution-2d | 0.015 (16 x 16) | 888.31 (16K x 16K) | 222.05 | jacobi | 0.0003 (4 x 4) | 116.94 (2K x 2K) | 15.23 |
| conway | 0.006 (16 x 16) | 31.85 (1K x 1K) | 6.07 | mandelbrot | 0.26 (8 x 8) | 285.90 (512 x 512) | 98.48 |
| fbcorr | 159.91 (16x16x256x256) | 371.44 (16x8x1Kx1K) | 272.06 | saxpy | 0.053 (1K) | 934.84 (32M) | 125.03 |
| gemm | 0.029 (16 x 16) | 925.27 (2K x 2K) | 282.43 | syr2k | 0.019 (8 x 8) | 1199.95 (1K x 1K) | 346.17 |
| gemver | 0.003 (8 x 8) | 645.83 (16K x 16K) | 137.25 | vadd | 0.008 (1K) | 63.32 (16M) | 10.33 |

Table 5.2: Speedup of JIT compiled CPU loop nest variant relative to CPython interpreter execution on platform *T1*. Iteration sizes corresponding to each data points are shown in brackets.

on platform *T2* is observed for *gemver* (3859.98) at a size $(16K \times 16K)$ and for *syr2k* (1290.19x) at a size $(1K \times 1K)$.

For all benchmarks, ALPyNA's analysis and compilation time dominate execution time for smaller domain sizes. At a large enough domain size threshold, the JIT compiled CPU variant becomes faster than the interpreter. Figures 5.1 and A.1 show the increasing speedups on platforms *T1* and *T2* for all benchmarks as domain sizes increase.

### 5.4.2 Comparison of ALPyNA GPU Variant with CPython Interpreter Execution

Columns 2–4 of Table 5.3 report minimum, maximum, and arithmetic mean speedup achieved by the GPU variant of each benchmark over the corresponding execution by the CPython interpreter on platform *T1*. For the smallest domain sizes, the CPython interpreter is faster than the GPU variant. The lowest speed-ups are achieved for the *hilbert* (0.0001x) and jacobi (0.0007x) benchmarks at the tiny domain sizes of $(4 \times 4)$ and $(16 \times 16)$ respectively. The largest maximum speedup is achieved for *syr2k* (16435x) at $(1K \times 1K)$.

Similarly, columns 2–4 of Table A.2 report minimum, maximum and arithmetic mean speedups by the GPU on platform *T2*. The *hilbert* (0.001x), *jacobi* (0.002x) and *gemver* (0.002x) benchmarks are the slowest at their respective smallest domain sizes $(16 \times 16$, $4 \times 4$ and $8 \times 8)$. The maximum speedup on platform *T2* is observed for *syr2k* (13895x) at a size $(1K \times 1K)$.

The results for the two largest iteration domains for *syr2k* have not been included due to our experimental set-up timing out. The diverging execution times of the interpreter and the

| Benchmark | Relative speedup | | | | | |
|---|---|---|---|---|---|---|
| | ALPyNA GPU vs CPython | | | ALPyNA GPU vs CPU JIT | | |
| | min | max | mean | min | max | mean |
| black-scholes | 0.19 (32K) | 51.85 (16M) | 12.33 | 0.22 (32K) | 3.87 (16M) | 0.46 |
| convolution-2d | 0.013 (16x16) | 3439.31 (16Kx16K) | 592.42 | 0.82 (256x256) | 3.87 (16Kx16K) | 1.40 |
| conway | 0.003 (16x16) | 19.29 (1Kx1K) | 3.67 | 0.58 (16x16) | 0.61 (64x64) | 0.6 |
| fbcorr | 1427.84 (16x8x256x256) | 3082.90 (16x8x1Kx1K) | 2283.3 | 4.64 (16x8x256x256) | 16.49 (16x16x1Kx1K) | 10.49 |
| gemm | 0.021 (16x16) | 6621.08 (2Kx2K) | 1274.87 | 0.58 (64x64) | 7.15 (2Kx2K) | 1.96 |
| gemver | 0.001 (8x8) | 702.22 (16Kx16K) | 100.66 | 0.37 (2Kx2K) | 1.08 (16Kx16K) | 0.52 |
| hilbert | 0.0007 (16x16) | 34.5 (4Kx4K) | 5.62 | 0.43 (32x32) | 0.72 (512x512) | 0.56 |
| jacobi | 0.0001 (4x4) | 53.94 (2Kx4K) | 7.11 | 0.42 (4x4) | 0.52 (16x16) | 0.46 |
| mandelbrot | 0.08 (8x8) | 870.61 (1Kx1K) | 153.73 | 0.29 (16x16) | 3.09 (1Kx1K) | 0.76 |
| saxpy | 0.02 (1K) | 435.6 (32M) | 62.15 | 0.35 (256K) | 0.67 (16M) | 0.46 |
| syr2k | 0.012 (8x8) | 16435.68 (1Kx1K) | 2471.3 | 0.53 (32x32) | 179.55 (4Kx4K) | 30.11 |
| vadd | 0.003 (1K) | 30.37 (16M) | 5.04 | 0.41 (32K) | 0.7 (64K) | 0.49 |

Table 5.3: Speedup of ALPyNA GPU kernels relative to CPython interpreter execution and CPU JIT compiled loop nest on Platform *T1*. The iteration size for each data point is shown in brackets.

GPU as shown in Figures 5.1 and A.1 suggest that GPU speedups will increase exponentially until performance levels off when the GPU is heavily saturated with execution threads.

Figures 5.1 and A.1 show that for all the tested benchmarks, above a threshold iteration domain size, GPU execution becomes profitable. At the smallest domain sizes, the GPU performance lags behind CPython interpreter performance. Below the crossover thresholds, analysis and compilation time dominates the GPU execution time (Section 5.5).

All benchmarks executed on the CPython interpreter on platforms *T1* and *T2* were observed to run at their maximum CPU frequency of 3.0 GHz and 3.9 GHz respectively throughout the duration of execution. This is due to the CPython interpreter running in single-threaded mode needing only one CPU core to be fully utilised and leaving the others idle. Thus no throttling of the CPU due to potential thermal issues was observed.

### 5.4.3 Comparison of ALPyNA GPU Variant with JIT compiled CPU Variant

Figures 5.1 and A.1 show a clear and increasing relative speedups for ALPyNA's GPU variants compared to the CPython interpreter across all benchmarks for platforms *T1* and *T2*. However, the same trend cannot be clearly established for the speed-up of the GPU variant relative to the CPU variant. Figures 5.2 and A.3 show execution time (i.e. without analysis and compilation time) for CPU and GPU code generated by ALPyNA for platforms *T1* and *T2*.

The results in columns 5–7 of Table 5.3 report the minimum, maximum and arithmetic mean of total execution speed-ups over the JIT compiled CPU variant. These results are classified into groups, based on relative execution speedups observed and amount of computation to be executed. This occurs either due to a large number of statements within a loop nest, data transfer overhead, or the repeated execution of parallel kernels within a loop nest by an outer loop executing sequentially.

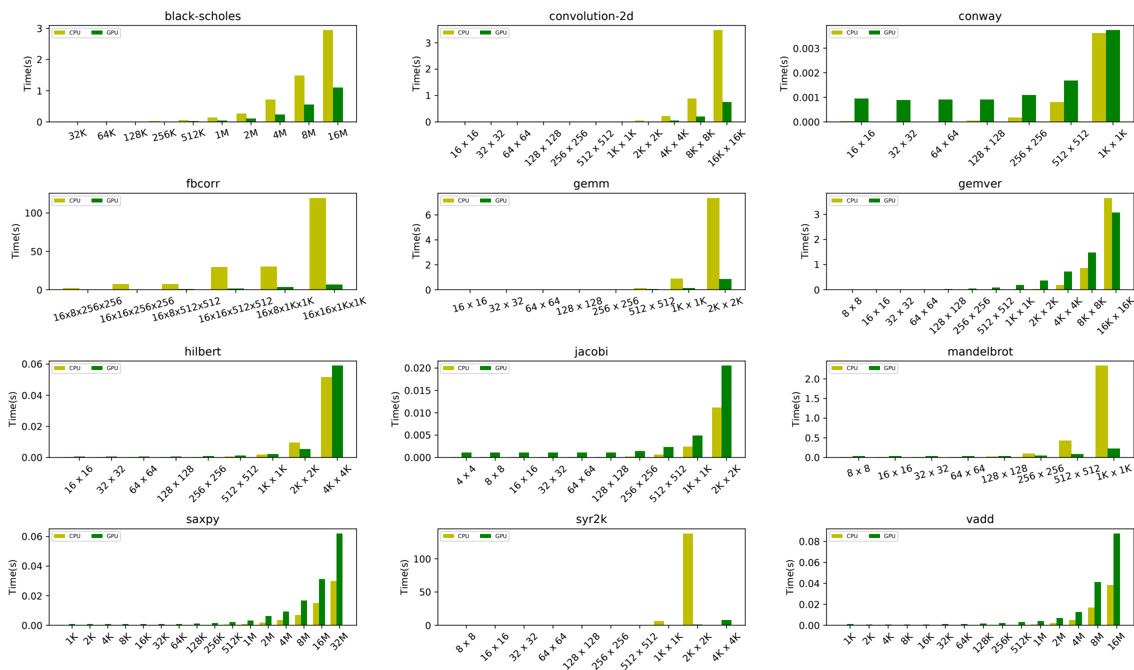



Figure 5.2: Execution time of ALPyNA generated code on the CPU vs the GPU on Platform *T1* (Lower is better). Times are scaled logarithmically. CPU execution time for the largest domain of syr2k ($4k \times 4k$) is omitted to better represent execution times of smaller domain sizes.

**Light parallel workloads** are loop nests with small amounts of work to be done within the body of the parallelisable loop. Benchmarks such as *saxpy, conway, hilbert, jacobi,*

*gemver* and *vadd* fall into this category. The analysis, compilation and data transfer overhead incurred by ALPyNA to execute on the GPU is consistently greater than compiling and executing the benchmark on the CPU. The relative speedup observed on both platforms *T1* and *T2* remains constant for increasing iteration sizes as shown in Figure 5.1 and A.1 respectively. The maximum speed-up of total execution time for these benchmarks ranged from 0.52x – 1.08x on *T1* and 0.83x – 1.1x on *T2*.

**Heavy parallel workloads** are workloads where the per-loop work is substantial. The performance boost that can be obtained from these kernels is substantial despite the data communication overhead of the GPU. Kernels such as *black-scholes, conv-2d, gemm, mandelbrot, fbcorr* and *syr2k* show significant speedup at much lower iteration domain sizes. *fbcorr* is classified as a heavy parallel workload, due to the consistent speedups obtained over the JIT-compiled CPU variant. This is due to the kernel being able to run four out of its seven loops in parallel and execute these faster than the CPU for a range of data sizes. For these benchmarks, speedups will continue to increase with iteration domain size until GPU memory is exhausted. The maximum speed-up of total execution time for these benchmarks ranged from 3.09x – 179.55x on *T1* and 1.12x – 50.68x on *T2*.

The CPU variants of all benchmarks were compiled by Numba to execute in single threaded mode. The CPU variants were able to run at their maximum CPU frequency of 3.0 GHz and 3.9 GHz respectively throughout the duration of execution without any throttling of the CPU. The JIT compiled GPU variants are also observed to execute on *T1* and *T2* at their rated clock frequencies.

### 5.4.4 Comparison of ALPyNA GPU code with hand-written GPU code

Figure 5.3 shows the performance of ALPyNA's generated GPU code compared to hand-written GPU benchmarks on platform *T1*. The hand-written kernels are compiled using Numba. The comparison is performed for four benchmarks namely, *conway, gemm, mandelbrot* and *vadd*. According to the classification in Section 5.4.3, *conway* and *vadd* are light parallel workloads while *gemm* and *mandelbrot* are heavy parallel workloads. Amongst these benchmarks, the maximum slowdown of ALPyNA's generated GPU code relative to the hand-written code across all input domain sizes is 1.9x (*conway*) for light parallel workloads and 54.09x (*mandelbrot*) for heavy workloads. These slowdowns point to the possibility of additional performance gains through further optimisation.

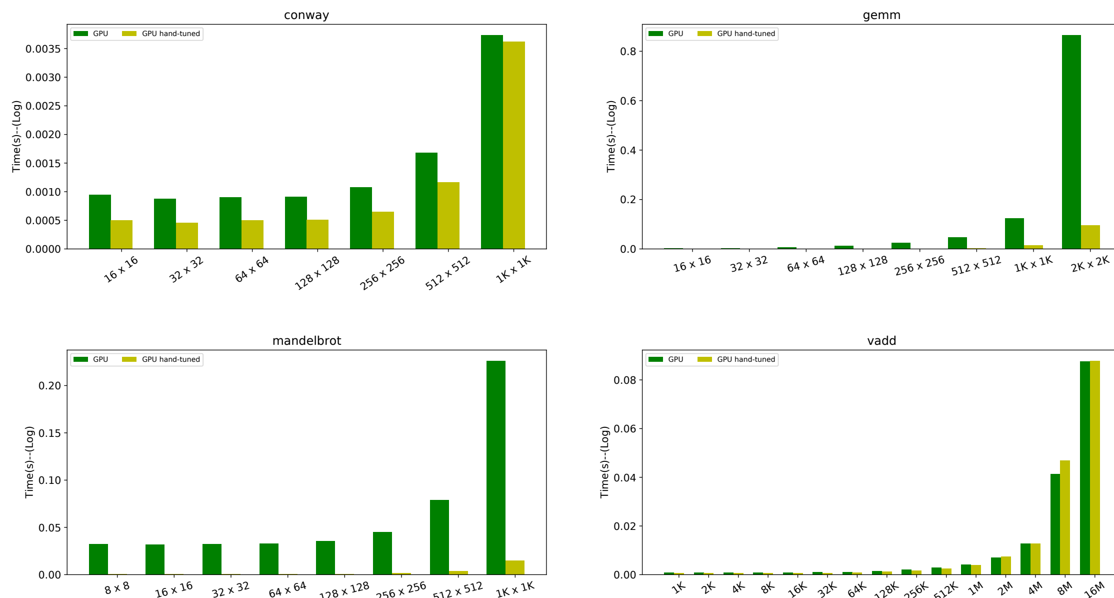



Figure 5.3: Comparison of ALPyNA generated GPU code with hand written GPU kernels on Platform *T1*. Lower is better.

### 5.4.5 Comparison of ALPyNA GPU code with other solutions

Section 3.3.1 discusses prior work in the area of automatic loop parallelisation in Python targeting accelerators such as GPUs and FPGAs. This includes Megaguards [115] and Three-Fingered-Jack (TFJ) [127, 128].

Megaguards is built on top of ZipPy which is a Python interpreter built on top of the JVM, while ALPyNA uses the standard CPython interpreter along with the Numba JIT compiler. In the presence of cross-loop iterations, Megaguards will JIT compile the loop nest for the CPU. However ALPyNA can generate parallel code for inner loops by sequentially executing outer loops that carry dependences. Benchmarks used to evaluate the performance of ALPyNA (Section 5.1) which contain cross loop iterations include *conv-2d, gemm, gemver, mandelbrot,syr2k* and *fbcorr*. While TFJ vectorises code for FPGAs and for CPUs with SIMD extensions, it does not generate GPU kernels. It also does not generate parallel kernels for loop nest bodies with control flow divergence. Benchmarks with control flow divergence used for the evaluation of ALPyNA include *mandelbrot* and *black-scholes*. Due to these reasons, these systems cannot be directly compared and hence a performance comparison has not been performed.

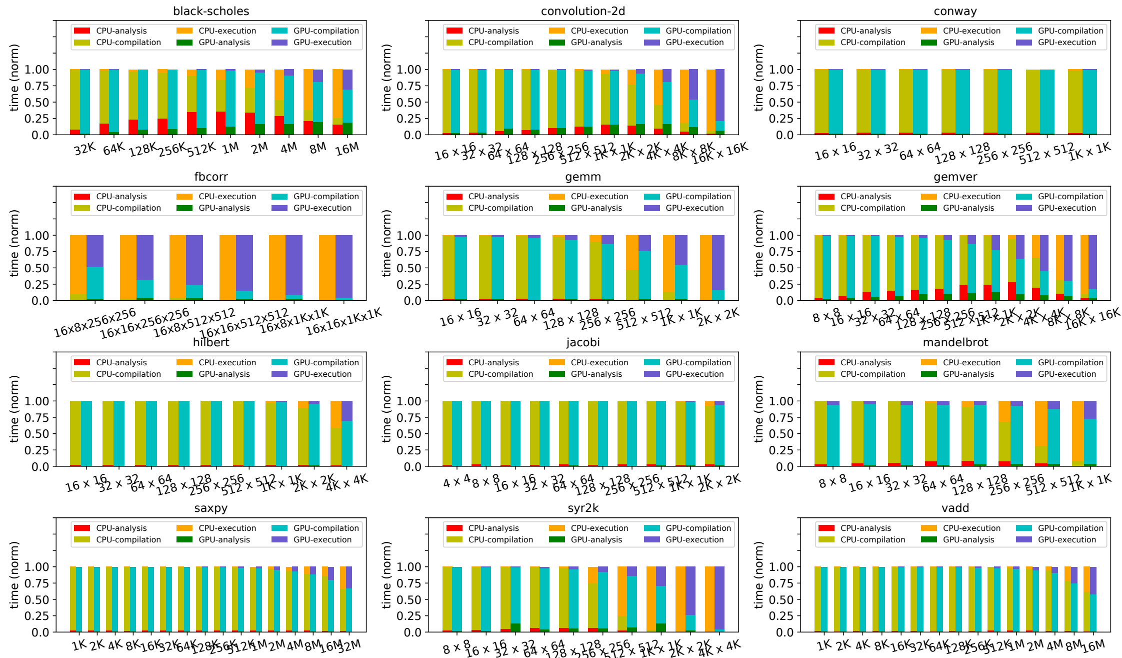



Figure 5.4: Proportion of time spent by ALPyNA for analysis, compilation and execution on JIT compiled CPU and GPU code on Platform *T1*.

## 5.5 Analysis and Compilation Overhead

Section 5.4 establishes that for small domain sizes, interpreter execution is faster than either CPU or GPU variants of a loop nest. However the threshold domain sizes at which it is advantageous to execute JIT compiled code on either the CPU or GPU is small.

Figures 5.4 and A.2 show the relative time spent by ALPyNA for analysis, compilation and for execution of the benchmarks on platforms *T1* and *T2* respectively. Table 5.4 and A.3 present the time overhead incurred by ALPyNA for analysis and compilation on platform *T1* and *T2* respectively. The runtime analysis and code generation component includes ALPyNA's runtime dependence analysis, cost modelling and code-generation for the selected device for JIT compilation. The mean values for analysis and code generation range from 1ms (*vadd,saxpy*) – 280ms (*black-scholes*) for CPU code generation on *T1*. The corresponding mean values for the GPU on *T1* ranged from 1ms (*vadd*) – 304ms (*black-scholes*). On platform *T2*, the mean values for analysis and code generation took between 1ms (*hilbert*) and 509ms (*black-scholes*) for the CPU and, between 1ms (*hilbert*) and 543ms (*black-scholes*) for the GPU.

Compilation of generated code by Numba takes significantly longer than ALPyNA's analysis and code generation phase. Compilation took a maximum of 384ms and 1.84sec for the *black-scholes* benchmark on the CPU and GPU respectively. On *T2*, compilation of code

took upto 321ms and 992ms for the CPU and GPU. For each benchmark, these values are consistent across domain sizes. The variability of the compilation times for both CPU and GPU was +/- 10%.

| Benchmark | ALPyNA analysis + runtime code generation (sec) | | | | | | Compilation (sec) | |
|---|---|---|---|---|---|---|---|---|
| | CPU | | | GPU | | | CPU | GPU |
| | min | max | mean | min | max | mean | | |
| black-scholes | 0.034 | 0.616 | 0.280 | 0.046 | 0.682 | 0.304 | 0.384 | 1.847 |
| convolution-2d | 0.004 | 0.079 | 0.028 | 0.005 | 0.063 | 0.029 | 0.153 | 0.167 |
| conway | 0.006 | 0.007 | 0.006 | 0.006 | 0.007 | 0.007 | 0.194 | 0.330 |
| fbcorr | 0.008 | 0.101 | 0.056 | 0.012 | 0.118 | 0.061 | 0.222 | 0.208 |
| gemm | 0.002 | 0.005 | 0.004 | 0.003 | 0.006 | 0.004 | 0.114 | 0.154 |
| gemver | 0.010 | 0.157 | 0.074 | 0.013 | 0.164 | 0.081 | 0.252 | 0.525 |
| hilbert | 0.001 | 0.002 | 0.002 | 0.002 | 0.002 | 0.002 | 0.072 | 0.134 |
| jacobi | 0.004 | 0.004 | 0.004 | 0.004 | 0.005 | 0.004 | 0.143 | 0.312 |
| mandelbrot | 0.006 | 0.028 | 0.017 | 0.007 | 0.029 | 0.018 | 0.173 | 0.556 |
| saxpy | 0.001 | 0.002 | 0.001 | 0.001 | 0.002 | 0.002 | 0.057 | 0.122 |
| syr2k | 0.005 | 0.069 | 0.022 | 0.006 | 0.068 | 0.027 | 0.191 | 0.298 |
| vadd | 0.001 | 0.001 | 0.001 | 0.001 | 0.002 | 0.001 | 0.061 | 0.122 |

Table 5.4: Analysis and code generation time taken by ALPyNA for CPU and GPU on Platform *T1*.

Figure 5.5 shows the increase in total analysis and compilation time for each benchmark for various domain sizes. While compilation time remains constant across all domain sizes, analysis and code generation time increases as the domain sizes increase. The increase in analysis time occurs due to the increased convergence time required for the thread hierarchy calculation (described in Section 4.4.1 – GPU Thread Hierarchy) for larger domain sizes. This calculation is performed before the code is generated so that the ALPyNA Cost Model (Chapter 6) can inform the selection of an optimal target device for compilation.

### 5.5.1 Static Analysis overhead

ALPyNA's warm-up phase for each benchmark is measured to demonstrate the low overhead of static analysis. Figure 5.6 shows an overhead of around 0.1s for all benchmarks on platform *T2*. The *black-scholes* analysis takes the longest amount of time; there are a total of 66 subscript pairs (49 cross and 17 output variable subscript pairs) to analyse amongst 13 lines of code in the target loop nest. Conditional code within the loop body creates compiler generated predicate variables due to *if-conversion* which increases the time to do static analysis.

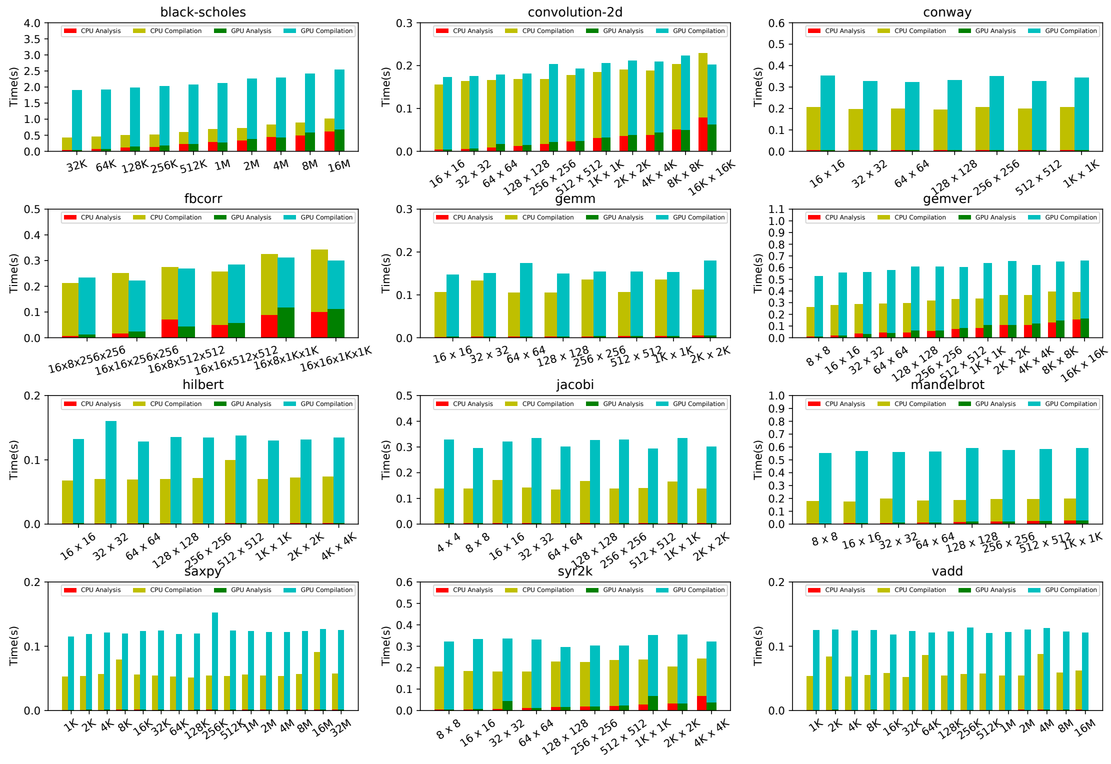



Figure 5.5: ALPyNA runtime analysis and compilation overhead (Platform *T1*).

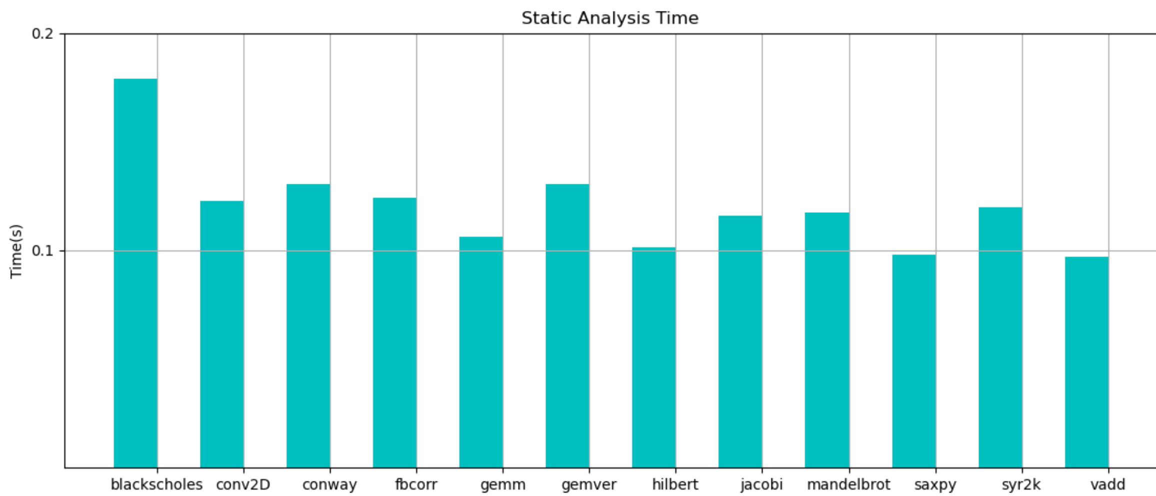



Figure 5.6: ALPyNA static analysis times for each benchmark on Platform *T2*.

## 5.6 Summary

This chapter evaluates the performance of ALPyNA's dynamic dependence analysis and loop parallelisation for CPUs and GPUs. The results show that dynamically generating CPU and GPU variants of data parallel Python loop nests is not only viable but also profitable.

ALPyNA's runtime capabilities are evaluated using 12 loop intensive benchmarks. These benchmarks vary in complexity from single-loop single-statement loop nests that are embarrassingly parallel to more complex benchmarks containing multiple statements, loop carried dependences, imperfect loop nests, control flow divergence and calls to pure intrinsic functions.

The benchmarks are measured on two CPU – GPU systems (Section 5.2). One platform (*T1*) is a server grade machine and the other (*T2*) is a typical desktop setup. Between these two systems, over 240 data points were collected for the 12 benchmarks. Section 5.4 shows that once the overheads of analysis and compilation are overcome, large speedups are achieved for the JIT compiled CPU (max 1290x) and GPU variants (max 16435x) relative to the interpreter. The gradient of the relative performance also consistently increases with increasing domain ranges.

Section 5.5 measures the overheads of runtime analysis, cost modelling, code generation and compilation. The compilation times were slightly higher for the GPU (0.069 – 1.847 seconds) compared to the CPU (0.045 – 0.384 seconds). Compilation of generated code is observed to be constant for each benchmark. Runtime dependence analysis, cost modelling and code generation takes between (0.001 – 0.5 seconds) on the CPU and (0.001 – 0.54 seconds) for the GPU. Although analysis time increased for larger domain sizes, compilation time dominated analysis and code generation time.

Analysis of the relative performance of the JIT-compiled GPU kernels *vis-à-vis* the JIT-compiled CPU version reveals two kinds of workloads (*light* and *heavy*) characterised by the amount of total parallel computation in the loop nest. The evaluation reveals a relative speedup ranging from 0.22x for *light* workloads (i.e. a slowdown) up to 179.55x for *heavy* workloads. Within the distribution of iteration domains for each benchmark, light workloads are consistently faster on the CPU. For heavy workloads, the CPU is generally faster at lower iteration domain thresholds while the GPU executes faster at higher iteration domains.

The performance evaluation of ALPyNA's runtime code generation and execution shows that execution by the interpreter is fastest for small iteration domain sizes while the JIT compiled version is faster for large input sizes when execution time dominates the analysis and compilation overhead. The GPU executes loop nests with *heavy* parallel workloads faster than the CPU.

However we cannot naïvely always expect a speedup from a GPU JIT compiler, even when all the loop instances can be executed in parallel. This suggests the need for a *dynamic cost model* to guide the selection of the optimal target device for the execution of a loop nest in a heterogeneous environment. Chapter 6 will introduce and evaluate such a cost model.

# Chapter 6

# The ALPyNA Cost Model

ALPyNA auto-parallelises loop nests for GPU execution in Python, a dynamic language. Auto-parallelising compilers are common for static languages. They often use cost models to determine when GPU execution is likely to be faster and whether this will be be enough to outweigh offload overheads. Cost models for JIT compilation and parallelisation of dynamic languages [10, 98, 110] concentrate on CPU acceleration. However, few cost models currently exist for JIT compilation of dynamic languages in CPU–GPU environments.

ALPyNA can deliver significant speedups if the analysis and compilation overheads are amortised over the whole iteration domain range. However, relative performance between the CPU and the GPU reveals the need for a more nuanced approach to selecting the optimal target device to execute a loop nest computation (Section 5.4.3).

This chapter presents an analytical cost model for auto-parallelising loop nests in a dynamic language such as Python on heterogeneous architectures. Predicting execution time in a language like Python is relatively complicated since aspects like the element types, size of the iteration space, and amenability to parallelisation can only be determined at runtime. To achieve optimised parallel code, ALPyNA stages dependence analysis and code generation between compile time and runtime. To reflect ALPyNA's architecture, the cost model is also staged.

Section 6.1 presents the motivations behind the cost model for ALPyNA and the factors that inform its development. Section 6.2 systematically develops the ALPyNA Cost Model (ACM) which guides the selection of a target device for JIT compilation. Section 6.3 evaluates the accuracy of ACM using two measures – *misprediction penalty* and *misprediction range*. Finally Section 6.4 summarises the chapter and its main contributions.

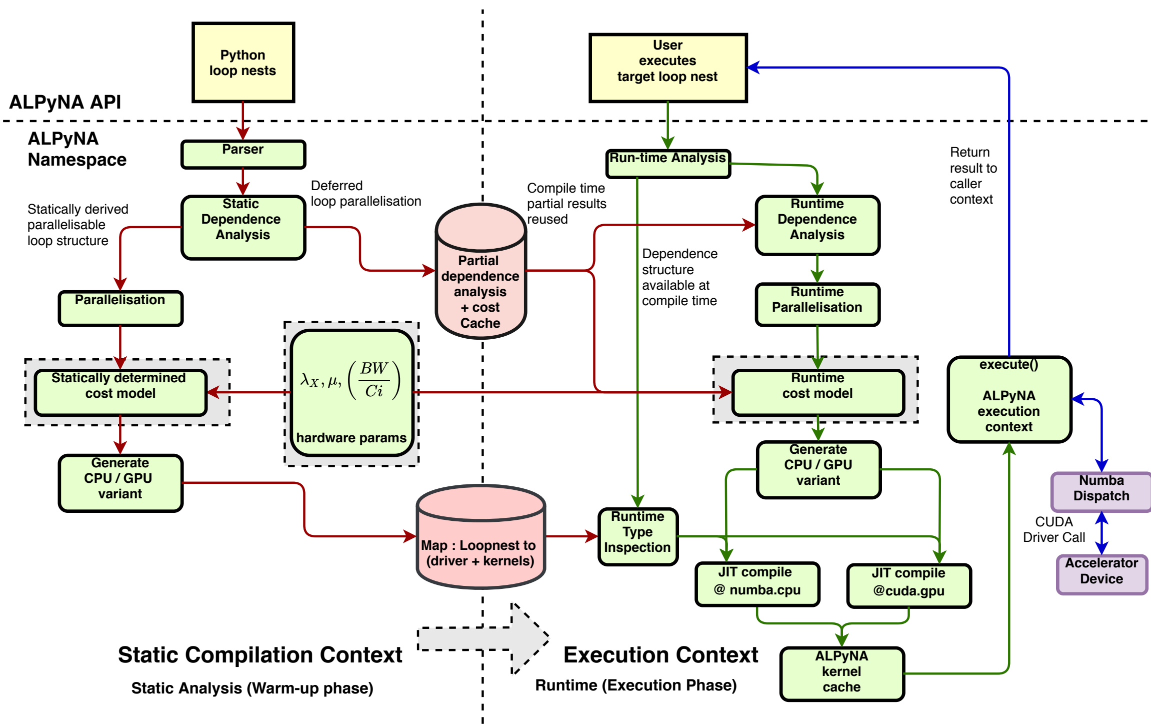



Figure 6.1: ALPyNA system architecture (introduced in Figure 4.4), generating kernels specialised to runtime dependences and domain sizes. Install time profiled constants and hardware parameters inform the selection of the optimal device for compilation.

## 6.1 Motivation

The performance evaluation of ALPyNA (Chapter 5) demonstrated the potential for significant reduction in runtimes of large and moderately complex loop nests (Section 5.4.3). However, this is not the case for all loop nest instances. The GPU variants of loop nests with *light* workloads never catch up and execute faster than the CPU counterpart across the entire range of iteration domains. For such loop nests, it is always better to JIT compile for the CPU rather than the GPU. With loop nests executing *heavy* computational parallel workloads, the CPU variant executes faster than the GPU up to a threshold iteration domain size. Beyond this threshold, a GPU is the faster device. The overall work performed is represented by the overall iteration domain size. The iteration domain space at which this threshold occurs varies between the 12 benchmarks (Figure 5.1).

Analytical performance predictors that rely on extensive profiling [63, 73] accurately generate optimised code for specific execution profiles of an application. Machine learning techniques [9, 19] for performance prediction of sequential loop nests on parallel hardware is dependent on the diversity of data used for training. Ideally, any cost prediction for a particular execution instance would be generated directly from the source code.

Listing 6.1: Example abstract loop nest structure for cost modelling.

```
𝓕1:for itr_𝓕1 in range(𝓛(𝓕1)):
       𝓢1
𝓕2:    for itr_𝓕2 in range(𝓛(𝓕2)):
           ...
𝓕N:        for itr_𝓕N in range(𝓛(𝓕N)):
               𝓢j
       𝓢M
```

One of the primary goals of a cost model in a JIT compilation environment is to be light weight, so as to reduce perceived execution latency. ALPyNA's dynamic loop dependence analysis optimises and generates code for target devices based on the loop dependence structures and iteration domains that emerge at runtime. Hence, any cost model must be staged to combine static and dynamic information while also being lightweight. It must account for the execution paradigm of a target device, the physical hardware specification, and execution speed relative to the host device. It would also be advantageous for any cost model to be easily retargetable to new hardware with a different parallel execution model. ALPyNA's cost model parameters inform the selection of target devices for JIT compilation (as shown in Figure 6.1). This includes the generation of untyped skeletal kernels and drivers during the static phase as well as during dynamic runtime kernel generation.

## 6.2 ALPyNA Cost Model

Parallelisation during static compilation usually makes conservative assumptions when a dependence cannot be accurately resolved. Such limitations may be caused by unknown loop limits and other variables within the loop nest (such as loop invariants). These assumptions prevent opportunities to maximise parallel execution of a loop nest (Section 4.2).

Execution on a GPU is worthwhile if the interpreted execution time exceeds the time taken to compile kernels, transfer kernels and data between the host and the accelerator, and executing the computation on the device. The current cost model accounts for transfer time and execution time but omits compilation time as discussed in Section 7.3.2. The ACM determines which device to execute a loop nest on, in a heterogeneous CPU–GPU compute environment. It does so by using staging and a family of lightweight models to compare the predicted relative runtimes of loop nest instances on each execution device.

Consider an imperfect loop nest as shown in Listing 6.1 comprising a set of $N$ Python `for` loop headers $\mathcal{P} = \{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_N\}$. The execution domain (i.e. number of iterations) of indexed `for` loop $\mathcal{F}_i$ is denoted as $\mathcal{L}(\mathcal{F}_i)$. The loop nest contains $M$ distinct Python statements to be executed; each statement is represented as an indexed value $\mathcal{S}_i$, with $1 \leq i \leq M$.

Statements are restricted to assignments to variables or arrays with non side-effecting functions as this is a requirement for ALPyNA to perform dependence analysis (Section 7.3.1). Note that there is no particular correspondence between the integer index of a `for` loop $\mathcal{F}_i$ and that of a statement $\mathcal{S}_i$. Some loop bodies only contain nested loop bodies; others may contain multiple statements.

We relate the control flow structure of `for` loops and statements in a loop nest, using the graph theoretic notion of dominance [81]. We designate the set of loop headers enclosing an arbitrary statement $s$ as $\mathcal{D}(s)$. In other words, $\mathcal{D}(s)$ is the set of loop headers that *dominate* $s$. We designate the set of statements enclosed within an loop header $f$ as $\mathcal{E}(f)$. In other words, $\mathcal{E}(f)$ is the set of statements *dominated by* $f$. $\mathcal{D}$ and $\mathcal{E}$ are duals in the dominance relation, i.e.

$$f \in \mathcal{D}(s) \iff s \in \mathcal{E}(f) \tag{6.1}$$

The ACM assumes that loops conform to this style, with $N$ loop headers and $M$ statements inside a single loop nest. Thus the outermost loop header $\mathcal{F}_1$ in the original loop structure always dominates all other loop headers and statements. The form is not restrictive as multiple top level loops can be modelled by introducing a top level loop to enclose them. ALPyNA uses *if-conversion* (Section 4.3.1) to transform statements in loop bodies with control flow divergence to predicated statements during AST parsing.

### 6.2.1 Modelling Interpreter Execution

$I_{\text{int}}$ is a function that maps any individual loop nest statement $S_j$ to an abstract cost, effectively a predicted execution time in the CPython interpreter. Such values could be profiled ahead-of-time. However, a novel feature of ACM is that *all costs are expressed relative to* $I_{\text{int}}$*, so profiling never actually takes place*. $C_{\text{int}}$ is a function that predicts the total cost of interpreting all instances of statement $S_j$ in the loop nest as the product of $I_{\text{int}}(\mathcal{S}_j)$ and all of the iteration domain sizes, Equation 6.2. This requires loop limits to have been resolved to numerical constants, which may require runtime introspection. $T_{\text{int}}$ is a function that predicts the total execution cost of the entire loop nest with top-level `for` loop header $f$ as the sum of the total execution costs of all statements in the loop nest, Equation 6.3.

$$C_{\text{int}}(s) = I_{\text{int}}(s) \prod_{f \in \mathcal{D}(s)} \mathcal{L}(f) \tag{6.2}$$

$$T_{\text{int}}(f) = \sum_{s \in \mathcal{E}(f)} C_{\text{int}}(s) \tag{6.3}$$

### 6.2.2 Modelling JIT Compiled CPU Execution

When ALPyNA JIT compiles a loop nest targeting the CPU, the cost model assumes an execution cost very similar to that of the interpreter. $I_{\text{cpu}}$ maps a loop nest statement to an abstract cost. One-time profiling at installation time is performed to express $I_{\text{cpu}}$ in terms of $I_{\text{int}}$ for each statement (Section 6.2.3 – Calibrating ACM). $C_{cpu}$ is a function that predicts the total cost of executing all instances of a statement in the loop nest as a product of the individual statement cost and the loop domain limits, Equation 6.4. $T_{cpu}$ is a function that predicts the total execution cost of the entire loop nest, Equation 6.5.

$$C_{\text{cpu}}(s) = I_{\text{cpu}}(s) \prod_{f \in \mathcal{D}(s)} \mathcal{L}(f) \tag{6.4}$$

$$T_{\text{cpu}}(f) = \sum_{s \in \mathcal{E}(f)} C_{\text{cpu}}(s) \tag{6.5}$$

Relating $I_{\text{cpu}}$ and $I_{\text{int}}$ assumes that the JIT compiler only compiles the loop into sequential binary instructions, and does not vectorise the loop body for Single Instruction Multiple Data (SIMD) execution units on the CPU. During the course of evaluation (Section 6.3), compilation by Numba was verified to not automatically parallelise loop nest execution to multiple cores on the CPU.

### 6.2.3 Modelling GPU Execution

Dependence theory states that if a dependence in a loop nest is *carried* by a loop and this loop is executed sequentially, all other loops nested within this loop may be executed in parallel. ALPyNA uses dependence analysis to parallelise such inner loops. To accommodate imperfect loop nests and the possibility that each statement in a loop nest may have different loop carried dependences, a light-weight kernel is created for each statement in the body of the loop nest (Section 4.4.1).

**GPU Execution Cost Model** Dependence analysis determines which loops to execute sequentially to maintain dependences between each statement. In theory, all other loops can be executed in parallel. Loops that *must* be executed sequentially are transformed into a GPU kernel call within the interpreter and called sequentially maintaining dependence relationships. Every statement which can be executed in parallel is executed within a GPU kernel. However, current GPGPU programming semantics restrict the number of dimensions along which threads can be scheduled to three (Section 2.3.2). This means parallel threads can be assigned for a triple nested `for` loop at best using CUDA thread semantics alone. Nested loops with a greater depth are supported by sequentially executing such loops

within a parallel GPU kernel. A *block-cyclic scheduling* approach (Bacon *et al* [17]) is used for the decomposition of loop domains along axes and their structuring into a two-tier thread hierarchy.

To model the cost of executing a loop nest that has been parallelised, the set of `for` loops enclosing each statement $s$ is split into distinct partitions:

1. $\mathcal{D}_{seq}(s)$ – the set of outer loop headers enclosing statement $s$ which *must* be executed sequentially, to satisfy the constraints imposed by a loop carried dependence. ALPyNA executes sequential loops in the interpreter on the host and calls parallel GPU kernels within the the scope of these sequential loops.

2. $\mathcal{D}_{par}(s)$ – the set of all loop headers enclosing statement $s$ that may be executed in parallel. This may be either because there are no loop-carried dependences or all loops carrying dependences are executed sequentially and accounted for in the set $\mathcal{D}_{seq}(s)$. The GPU kernels are guaranteed to execute in the order of being invoked by the driver code executing in the interpreter on the host. In general, the number of loops in $\mathcal{D}_{par}(s)$ may be greater than the number of parallel axes on the GPU (i.e. three). To model this ALPyNA further partitions the set $\mathcal{D}_{par}(s)$ into :

   (a) $\mathcal{D}_{gpu}(s)$ – the set of loops that ALPyNA has mapped to hardware axes. ALPyNA transforms each instance of execution along these iteration domains into a GPU kernel execution instance. On an NVIDIA GP104 (Pascal microarchitecture) GPU for example, each thread-block can only be allocated a maximum of 1024 threads. ALPyNA calculates a thread hierarchy from the loop domain sizes at run time and splits it into a tuple of grid sizes and block sizes. The loop domains scheduled along the logical *x,y,z* hardware axes are intended to maximise parallel work while meeting the *threads per block* constraint imposed by the GPU.

   (b) $\mathcal{D}_{\overline{gpu}}(s)$ – the set of all remaining loops that cannot be mapped onto a GPU parallel axis, these are executed sequentially within each GPU kernel. Although these loops are executed sequentially within a parallel kernel, the order of statement execution does not matter as loop carried dependences have been maintained by the sequentially executed outer loops in $\mathcal{D}_{seq}(s)$ Therefore these loops can be executed safely without synchronisation barriers inserted into the loop body within these loops.

To model the parallel execution cost of a statement $s$ on a GPU, the number of executions are measured as the product of the loop limits of the enclosing `for` loops, as for the interpreter and the CPU. However unlike the interpreter and CPU, which are modelled as executing each statement instance sequentially, the cost model uses a time-step factor to represent any speed

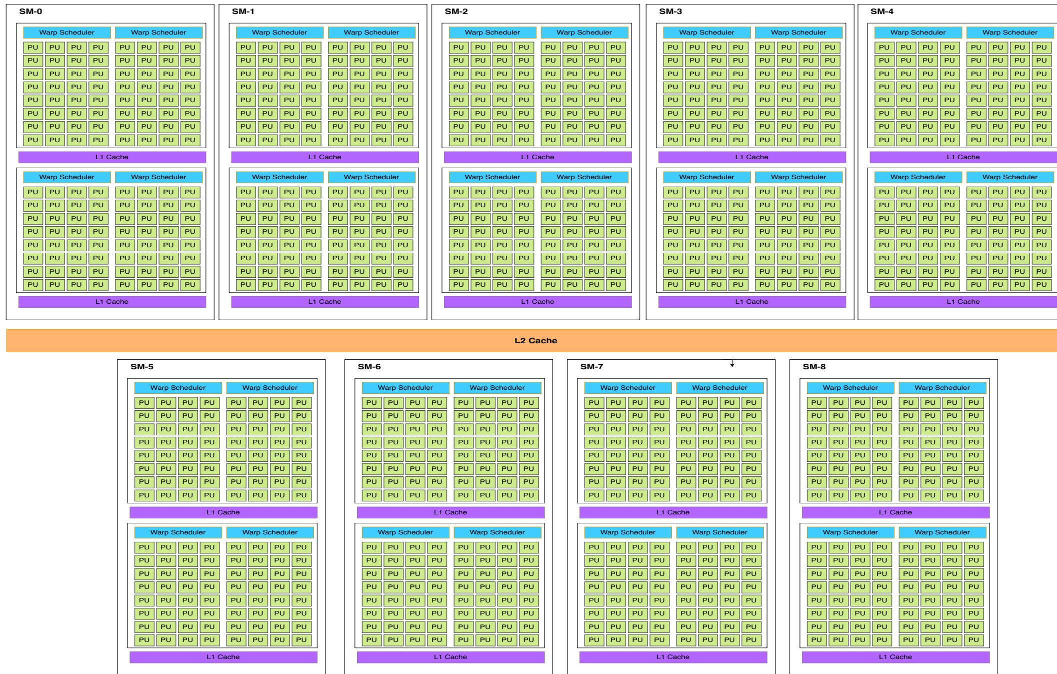



Figure 6.2: Simplified block diagram of a GTX-1060, (platform *T2*) that uses the GP104 microarchitecture.

up from parallel execution. The net effect of this factor is to divide sequential execution time by the parallel speedup of the computation being executed. At runtime, ALPyNA introspects and resolves loop domain limits as part of its dependence analysis. During code generation, total domain size is padded and split into a thread hierarchy of *grids* and *thread-blocks*. These threads are split along a 3-D plane (at-most) of thread axes. To estimate the parallel execution cost, a function $\{\mathcal{G}(\mathcal{L}(f)),\ f \in \mathcal{D}_{gpu}(s)\}$ is defined to be the number of grids along a particular dimension in the 3-D thread plane. $\mathcal{G}$ maximises the threads-per-block to calculate the number of grids within a thread hierarchy while maintaining the maximum threads-per-block constraint of each device by loop-interchange and sorting for the largest loops within the set $\mathcal{D}_{par}(s)$ (Section 4.4.1).

While the number of threads allocated to execute on an Streaming Multiprocessor (SM) can be greater than the number of CUDA cores in the SM, the maximum number of threads executing in parallel at any one time in an SM is the product of the number of warp schedulers per SM (denoted $v$) and the warp size (denoted $w$). The *block-cyclic* distribution of threads in parallelised GPU code means that any parallelisation speed-up factor should take into account precisely how the execution is mapped onto SMs in a GPU. Figure 6.2 shows a simplified diagram of an NVIDIA GTX-1060 GPU. This device has 9 SMs (denoted $u$); each SM uses the GP104 NVIDIA Pascal microarchitecture. Each SM has 128 CUDA cores

on them and 4 *warp* schedulers. For any statement $s$ that is executed in parallel, the amount of work done by each GPU kernel invocation is denoted as $\lambda_{exec}(s)$ (Equation 6.6).

$$\begin{aligned} \lambda_{exec}(s) &= \left\lceil \frac{g}{u} \right\rceil \times \frac{1}{g.v.w} \times \prod_{f \in \mathcal{D}_{gpu}(s)} \mathcal{L}(f) \times \prod_{f \in \mathcal{D}_{\overline{gpu}}(s)} \mathcal{L}(f) \\ g &= \prod_{f \in \mathcal{D}_{gpu}(s)} \mathcal{G}(\mathcal{L}(f)) \end{aligned} \tag{6.6}$$

All the threads in a *thread-block* are allocated to a single SM. Intuitively, the term $g.v.w$ provides overall parallel speed-up if a theoretical GPU built from the same microarchitecture had an infinite number of SMs and each thread-block can be allocated to simultaneously execute on all SMs. If the number of thread-blocks exceeds the available SMs, these thread-blocks are allocated in batches to an SM and sequentially executed. The slowdown relative to the theoretical maximum is captured by the slowdown factor $\left\lceil \frac{g}{u} \right\rceil$.

#### Modelling GPU Starvation

ALPyNA maintains loop-carried dependences on code transformed for the GPU by scheduling outer loops that carry dependences to execute in the CPython interpreter. The GPU executes JIT compiled binary code much faster than the interpreter. If the interpreter executes each kernel invocation faster than the GPU can execute the kernel, each kernel is queued for execution on the GPU and overall execution time is bound by kernel execution. Otherwise, the GPU will finish each kernel before the interpreter can schedule the next one, and the GPU is *starved* of work.

There are two cases to consider. If all the loops that dominate a statement $s$ are parallelisable, i.e. $\mathcal{D}_{seq}(s) = \emptyset$, the code is transformed into a single invocation of a kernel that executes all loop iterations of $\mathcal{D}(s)$ on the GPU. Hence there is a single kernel invocation latency cost $Cki_{vm}$. Otherwise $\mathcal{D}_{seq}(s) \neq \emptyset$ and the execution cost is the greater of the kernel invocation or the GPU execution cost. Here $I_{\text{gpu}}(s)$ is the cost of executing a single instance of the compiled kernel that represents statement $s$. Equation 6.7 models the cost of executing a loop nest statement while taking into consideration whether the relatively slow execution of the interpreter is acting as a bottleneck while scheduling kernels.

$$C_{\text{gpu}}(s) = \begin{cases} \max\left(Cki_{vm},\ \lambda_{exec}(s).I_{\text{gpu}}(s)\right) \prod_{f \in \mathcal{D}_{seq}(s)} \mathcal{L}(f) & \text{if } D_{seq}(s) \neq \emptyset \\ \\ Cki_{vm} + \lambda_{exec}(s).I_{\text{gpu}}(s) & \text{if } D_{seq}(s) = \emptyset \end{cases} \tag{6.7}$$

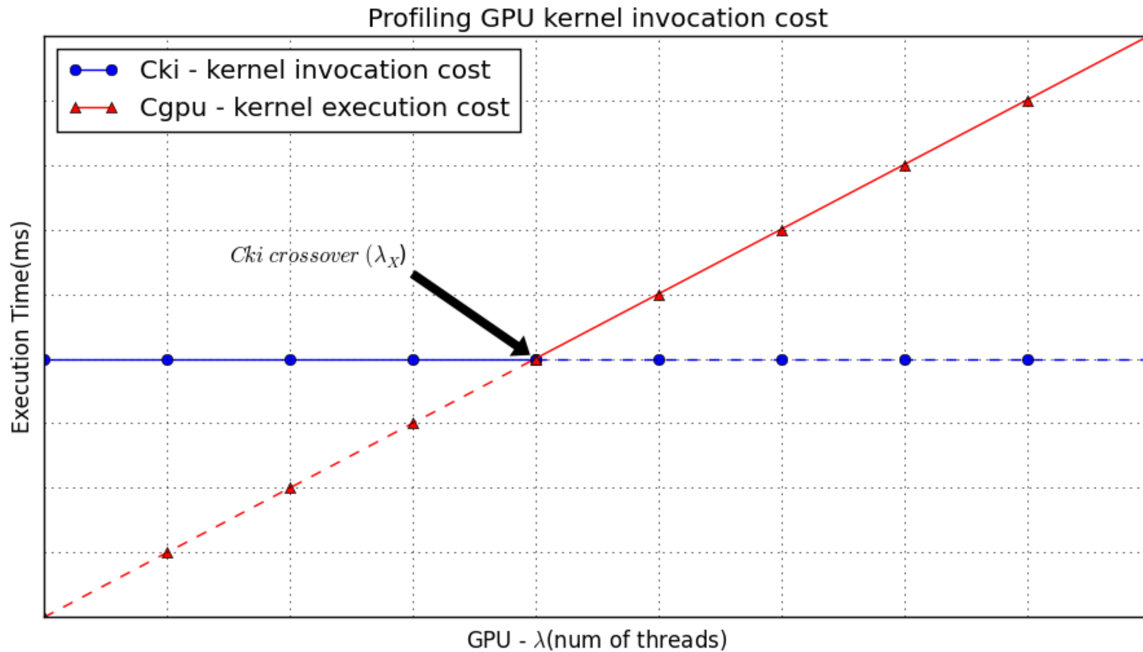



Figure 6.3: ALPyNA profiles a very simple kernel to discover the minimum work rate required to keep the GPU busy

Figure 6.3 depicts how for smaller amounts of parallel work, the GPU kernel completes early and the interpreter loop execution time dominates. However, once there is enough work in each kernel invocation to keep the GPU busy, GPU execution time dominates. This threshold varies depending on the relative performance of the GPU and the interpreter and is specific to each CPU–GPU combination.

To ascertain the GPU invocation cost threshold for each hardware setup, a very simple loop statement (as shown in Listing 6.2) is profiled once at installation time. We use a two dimensional loop where only the inner loop can be parallelised. We profile this loop nest in ALPyNA using varying domain sizes to arrive at the GPU throttling threshold. The profiling starts at a parallel domain size $\{\mathcal{L}(f) = w | f \in \mathcal{D}_{par}(s), w \leftarrow warpsize\}$ and the domain size is increased exponentially until the profiler detects execution time has gone beyond its inflexion point. The profiler then interpolates the number of threads at the inflection point and calculates $\lambda_{exec}$ (Equation 6.7) for the domain size at the inflection point.

Listing 6.2: Simple kernel used to profile value of $Cki_{vm}$

```
def _static_profile_kernel(arg_a, i_dom):
    for i in range(i_dom):
        for j in range(arg_a.shape[0]):
            arg_a[j] = arg_a[j] + 1
    return arg_a
```

This profiling and calculation is done to estimate the amount of GPU parallel work (in terms of domain sizes) required to overcome the kernel invocation cost $Cki_{vm}$, and is designated $\lambda_X$. At the kernel invocation cost threshold, the following relationship is assumed :

$$Cki_{vm} \equiv (\lambda_X \times I_{\text{gpu}}(s)) \tag{6.8}$$

Profiling a simple statement $s$ to determine the cross-over point $\lambda_X$, provides a maximal number of threads beyond which GPU execution time will dominate the loop iterations executing in the interpreter to schedule kernels representing statement $s$. For a kernel representing more complex statements, this assumption leads to the ACM selecting a higher, (conservative), threshold of parallel work to offload to the GPU. Substituting $Cki_{vm}$ into Equation 6.7 the estimated cost of executing each kernel is shown in Equation 6.9.

$$C_{\text{gpu}}(s) = \begin{cases} \max\left(\lambda_x,\ \lambda_{exec}(s)\right).I_{\text{gpu}}(s) \times \prod_{f \in \mathcal{D}_{seq}(s)} \mathcal{L}(f) & \text{if } D_{seq}(s) \neq \emptyset \\ \\ (\lambda_x + \lambda_{exec}(s)).I_{\text{gpu}}(s) & \text{if } D_{seq}(s) = \emptyset \end{cases} \tag{6.9}$$

The full cost of executing the loop nest with outermost `for` loop $f$ on the GPU is the summation in Equation 6.10 where $C_{\text{xfer}}(GPU)$ is the data transfer cost outlined next.

$$T_{\text{gpu}}(f) = C_{\text{xfer}}(f) + \sum_{s \in \mathcal{D}(f)} C_{\text{gpu}}(s) \tag{6.10}$$

### Modelling GPU Transfer Time

Executing on accelerators like GPUs incurs overhead for transferring data between the host CPU and the accelerator. Loop nests with light parallel work are sensitive to the overheads of data transfer.

Following common practice ALPyNA normalises data transfer time against the estimated cost of executing a very simple statement in the CPython interpreter, $I_{\text{int}}(s)$. Bandwidth profiling of the PCIe bus on which the GPU resides is performed once at install time, along with the measurements of GPU starvation factor $\lambda_X$ (Equation 6.8), and the JIT speed-up factor $\mu$ (Section 6.2.3, Equation 6.12).

Assuming that the set $\mathcal{P}_d$ is the set of all vectors that are transferred from the host to the GPU and $\mathcal{P}_h$ is the set of all vectors transferred back to the host after execution, Equation 6.11

shows the normalised cost of data transfer.

$$C_{\text{xfer}}(GPU) = \sum_{m \in \mathcal{P}_d} \frac{| < m > | \times size(T(< m >))}{BW/I_{\text{int}}(s)} + \sum_{n \in \mathcal{P}_h} \frac{| < n > | \times size(T(< n >))}{BW/I_{\text{int}}(s)}$$
$$where < m > \in P_d, < n > \in P_h$$
$$| < m > | \mapsto \text{number of elements in vector } < m >$$
$$T(< m >) \mapsto \text{data type of each element of vector } < m >$$
$$size(type) \mapsto \text{number of bytes in memory representation of type} \quad (6.11)$$

While the transfer model is fairly standard, a novel feature is that it is staged. That is ALPyNA identifies the set of vectors to be transferred to/from the GPU, and resolves their types and sizes at runtime. These vectors may include dynamically generated vectors of type - marshalled vectors (Section 4.4.2 – GPU Scalar Marshalling). These are combined with the ahead-of-time bandwidth measurements to estimate the transfer overhead, $C_{\text{xfer}}(GPU)$.

### Calibrating ACM

ACM normalises the cost of executing a statement on the CPU or GPU relative to the interpreter. Previous GPU cost models such as Leung *et al* [82] approximate the relative speed up by profiling the execution of a single instruction in the interpreter to the speed up achieved executing the same instruction on the GPU. Such cost estimations do not take into consideration differing speedups of the same instruction on the GPU due to scheduling constraints of different (*grid, thread-block*) thread hierarchies. ACM is more sophisticated in this respect.

Like many cost models ACM takes parameters that characterise the specific execution platform, e.g. CPython and the Numba JIT compiler on a specific CPU, and CUDA on a specific GPU. Specifically the key cost equations, 6.2, 6.4 and 6.7 take parameters representing the predicted runtime $I_{\text{platform}}(s)$ of executing a statement $s$ on a given $\text{platform}$, e.g. $I_{cpu}(s)$ in Equation 6.4. The platform costs for a statement $s$ are computed *relative* to the predicted interpreter cost $I_{\text{int}}(s)$.

Calibration is required to determine the value of the model parameters for each execution platform, and this is achieved as follows. The cost of executing a JIT compiled statement relative to the interpretation cost is profiled once at install time. A very simple kernel similar to the one used to profile $Cki_{vm}$ (Section 6.2.3 – Modelling GPU Starvation) is used to relate the runtimes of JIT compiled, and CPython interpreted code. The array size in the loop nest is chosen to ensure that there is only one cache miss (on the first iteration). The relative performance factor obtained is designated as $\mu$ in Equation 6.12. This provides a close approximation of the relative runtimes of compiled and interpreted code without caching, and allows us to separately account for caches when comparing CPU and GPU runtimes.

$$\mu = \frac{I_{\text{int}}(s)}{I_{cpu}(s)} \tag{6.12}$$

The cost of GPU execution clearly depends on the relative clock frequencies of the CPU and GPU, $f_{cpu}$ and $f_{gpu}$. Moreover, Armih et al [12] and Belikov et al [24] both report the size of the last level cache as a significant factor while comparing relative performance of data intensive programs on heterogeneous platforms. The last level cache (L2) on the GPU $LC_{\text{gpu}}$ is shared by all SMs. Each SM has its own set of L1 caches. For example, in the NVIDIA Pascal (GP104) microarchitecture, each SM has two L1 caches [133] as shown in figure 6.2. This cache sharing is represented by the factor $\sigma = num_SM \times L1_caches_per_SM$ in Equation 6.13 that computes relative CPU/GPU performance. A cache sharing factor is not introduced for the CPU core because Numba JIT compiles a loop nest for a single core. The ACM assumes that the CPU core that executes the code has exclusive use of the L3 cache ($LC_{cpu}$).

$$\frac{I_{cpu}(s)}{I_{\text{gpu}}(s)} = \psi \approx \frac{f_{gpu} \times (LC_{\text{gpu}}/\sigma)}{f_{cpu} \times LC_{cpu}} \tag{6.13}$$

Equation 6.14 shows how the cost of GPU execution relative to the interpreter is directly computed as the product of CPU/GPU cost $\psi$ and the interpreter/CPU ratio $\mu$ (Equation 6.12).

$$\frac{I_{\text{int}}(s)}{I_{\text{gpu}}(s)} \approx \mu \times \psi \tag{6.14}$$

Substituting Equations 6.12, 6.13 and 6.14 into Equations 6.2, 6.4 and 6.7 provides ALPyNA with an overall cost for each execution device that is normalised with respect to the cost of executing the loop nest on the interpreter. The platform with the minimal execution cost is selected *i.e.* $min(T_{\text{int}}, T_{cpu}, T_{\text{gpu}})$. These relationships provide an analytical cost model parameterised on :

1. dynamically determined dependence relationships in a loop nest

2. the thread hierarchy calculated from this dependence structure and the domains of all parallelisable loops dominating a statement

3. hardware characteristics of the CPU and GPU

4. whether the relative speed of the interpreter causes starvation effects on the GPU.

## 6.3 Evaluation

The evaluation in Section 5.4.3 shows that for *light* parallel workloads, the CPU is the optimal target device for JIT compilation. For *heavy* parallel workloads the CPU variant is the optimal choice up to an iteration domain threshold, and beyond this the GPU is the optimal target device. This section is an evaluation of the effectiveness of ACM to guide selection of a target device for JIT compilation of loop nests.

### 6.3.1 Experimental Setup

**Benchmarks** ACM is evaluated using twelve loop-intensive benchmarks of varying complexities and features. Section 5.1 describes the loop level characteristics of these benchmarks that affect the performance of JIT compiled code. The benchmarks (summarised in Table 5.1) represent a variety of characteristics that test ACM's capability to predict the optimal target device for compilation.

**Hardware Platforms** ACM is evaluated on two machines *M1* and *M2*. While *M1* is a server grade machine, *M2* is a typical desktop machine. The hardware specifications and the software stack used by ALPyNA for this evaluation has been previously specified in Section 5.2. A third machine configuration (*M3*) is created by limiting the frequency of machine *M2* to 800 MHz, while maintaining all other parameters the same. This evaluation follows the same convention in Chapter 5 and refers to the combination of hardware specifications and software stacks on machines *M1*, *M2* and *M3* as platforms *T1*, *T2* and *T3* respectively.

### 6.3.2 Methodology

ALPyNA has been developed with a profiling mode that disables the cost model guided JIT compilation. Instead it allows a developer to override the cost model's choice of target device with a specific device (one of *interpreter, CPU or GPU*). This mode is used to disable the cost-model guided selection and compilation for a target device. However, the profiling mode will still evaluate a relative cost and predict the performant device to execute a given loop nest instance.

Before using ACM on a target CPU–GPU platform the values of $\lambda_X, \mu$ (Eqs. 6.12 and 6.8) and relative GPU bandwidth for data transfer (Section 6.2.3) must be profiled. A 10% safety margin is used over the predicted inflection point ($\lambda_X$) at which the interpreter is able to keep the GPU busy. The calculation of $\frac{BW}{I_{\text{gpu}}}$, i.e. data transfer speeds in execution time units $(I_{\text{int}})$, is done while profiling for the value $\mu$ to ensure consistency.

The Numba compiler compiles and executes single threaded code on the CPU. During experimentation no other cores execute computationally intensive code and hence the core can reach maximum frequencies without being throttled.

To profile a wide range of domain sizes for each benchmark the iteration domain sizes in each loop are increased exponentially. Runtime on the ACM predicted device is compared with runtime on the optimal device (that would be identified by an oracle predictor) at each domain size. Reported runtimes are the arithmetic mean of 5 executions. We interpolate the iteration domain size at which ACM predicts GPU execution becomes profitable, to calculate the intervals of mispredicted domain sizes.

### 6.3.3 Comparative Baselines

For each domain size, execution time on the optimal device predicted by ACM device is compared with two baselines: (i) execution time on the optimal device identified by an 'oracle' predictor and (ii) execution time on the device selected by a two-class Support Vector Machine (SVM). An SVM [150] is a supervised machine learning model used for classification and regression analysis. The classification algorithm generally works for linearly separable data. The 'oracle' training data shows that the predictions are linearly separable based on input sizes for each individual benchmark. Because of its intuitive simplicity and the availability of high-performance SVM library implementations [33], it is a popular machine learning (ML) algorithm in program optimisation contexts. For instance, Table 1 in [15] shows that SVM is the second most deployed machine learning approach after decision trees, for automated ML-based compiler optimisations. SVMs are found to be particularly effective for identifying hot code regions for selective optimisation [90].

The SVM is trained for each benchmark using the labelled experimental data of all other benchmarks – i.e. Leave-One-Out Cross Validation (LOOCV). Training is performed separately for each of the three evaluation platforms. The optimal performant device at each iteration domain size is identified for each benchmark using the oracle predictor. The SVM's feature vector comprises structural components of each loop nest (cf. Table 5.1), iteration domain size, raw execution times on target devices and the optimal device chosen by the oracle predictor. All feature values are scaled with min-max normalization to the range $[0, 1]$.

### 6.3.4 Cost Model Performance

**GPU Speedups.** Table 6.1 shows the range of speedups obtained for each benchmark being executed on the GPU. The measured computation time on the GPU includes data transfer time and the time for execution. It shows that most benchmarks benefit from exploiting the GPU beyond iteration domain sizes that provide enough parallel computation to amortise the

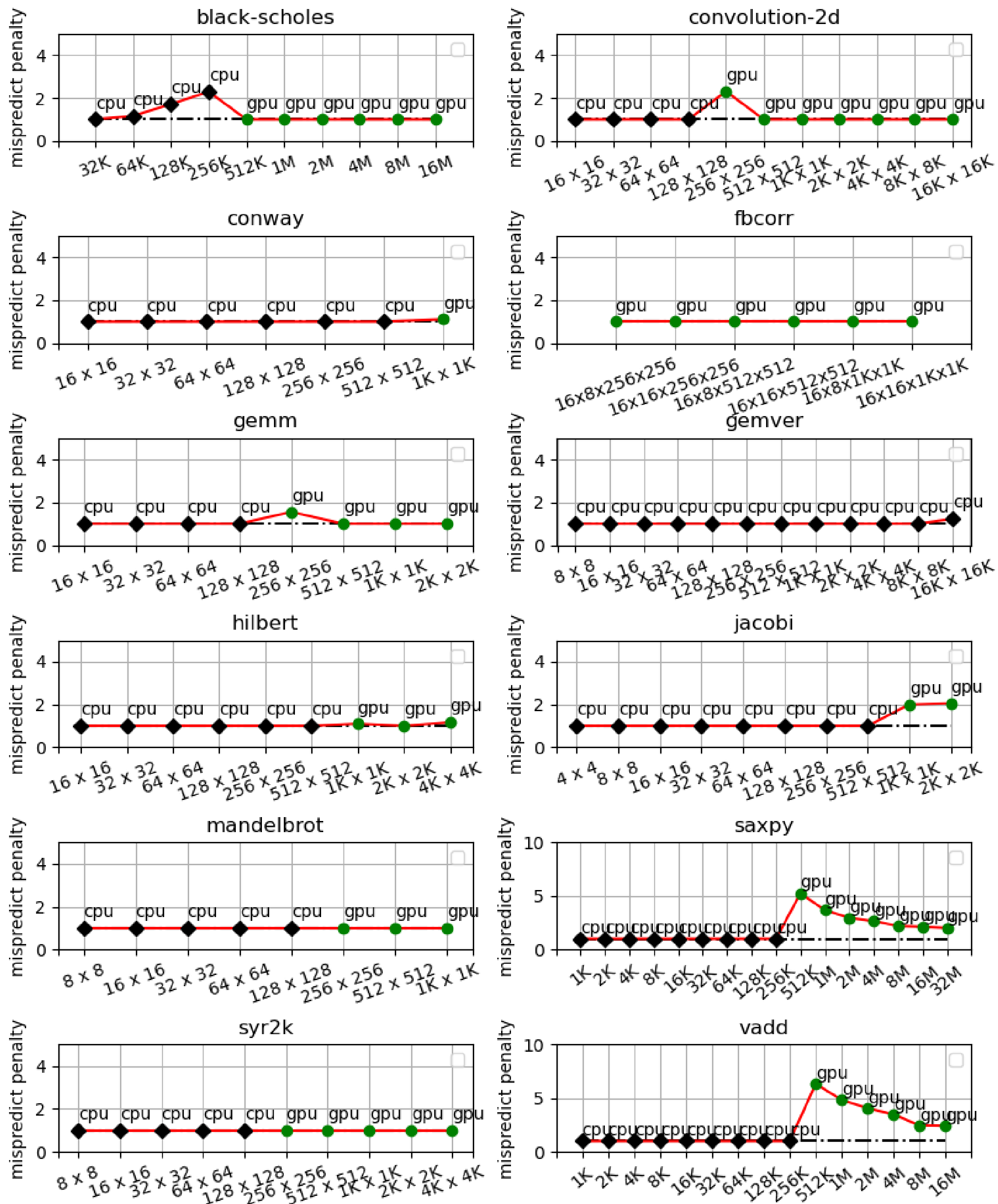



Figure 6.4: ALPyNA Cost Model misprediction penalties for 12 loop-intensive benchmarks with varying domain sizes on platform *T1*. Misprediction slowdown is the ratio of predicted device runtime and faster device runtime, so optimal is always 1.0. Each experiment is annotated with the predicted optimal target device.

| Benchmark | GPU speedup | | |
|---|---|---|---|
| | *T1*: CPU/GPU 2960/1405MHz | *T2*: CPU/GPU 3900/1500MHz | *T3*: CPU/GPU 800/1500 MHz |
| black-scholes | 1.17 – 3.83 | 1.13 – 2.61 | 1.60 – 8.21 |
| conv-2d | 1.31 – 4.61 | 1.06 – 2.05 | 1.45 – 6.92 |
| conway | na | na | 1.36 |
| fbcorr | 8.16 – 17.55 | 3.07 – 8.06 | 9.87 – 51.38 |
| gemm | 2.46 – 8.68 | 2.19 – 2.71 | 2.23 – 11.69 |
| gemver | 1.23 | 1.12 | na |
| hilbert | 1.55 | na | 1.25 – 2.51 |
| jacobi | na | na | 1.15 |
| mandelbrot | 1.80 – 9.52 | 1.59 – 4.48 | 1.95 – 12.39 |
| saxpy | na | na | na |
| syr2k | 2.87 – 192.45 | 2.65 – 53.03 | 2.59 – 134.69 |
| vadd | na | na | na |

Table 6.1: ALPyNA GPU speedups. Some benchmarks are always faster on the CPU.

data transfer and overcome the slower per-core execution speed of the GPU. As discussed in Section 5.4.3, GPU execution of some benchmarks is not faster than the CPU at any domain sizes.

**Misprediction Penalty** is reported in Figure 6.4 for each of the 12 benchmarks on target platform *T1*. Figures B.1 and B.3 in Appendix B show the misprediction penalties for platforms *T2* and *T3* respectively. At each domain size these figures show the platform device (CPU or GPU) selected and a misprediction penalty. The misprediction penalty is the slowdown incurred when the ACM predicts differently compared to the oracular predictor. In Figure 6.4, these are shown as peaks in the graph at each benchmark size that is mispredicted compared to the optimal value of 1.0. The relevant domain sizes are plotted on a logarithmic x-axis.

For all benchmarks and platforms Table 6.2 shows the geometric mean penalties over all domain sizes and the maximum penalties. ACM provides entirely accurate predictions for four benchmarks on *T2* and for three on *T1* and *T3*. The mean penalties for different benchmarks vary from 1.0 (optimal) to 1.68, 1.50, and 2.70 for *T1*, *T2*, and *T3* respectively. ACM has the same or lower misprediction penalty than the SVM predictor in 6 benchmarks on *T1* and *T2*, and 10 benchmarks on *T3*. Table 6.2 also shows the mean penalty and mean maximum penalty on each platform. The geometric mean penalty across all experiments is 1.14 for ACM and 1.46 for SVM predictor.

ACM delivers similar mean misprediction penalties for eight benchmarks on all three platforms (*T1*, *T2* and *T3*). Between *T1* and *T2* similar misprediction penalties are observed for nine benchmarks. The mean misprediction penalty for *black-scholes* is lower for platform *T1* (1.16) than for platforms *T2* (1.5) and *T3* (2.7). Benchmarks *saxpy* and *vadd* have lower mean misprediction penalties on *T3* (*saxpy*:1.07, *vadd*:1.10) than on *T1* (*saxpy*:1.57, *vadd*:1.68) or *T2* (*saxpy*:1.35, *vadd*:1.30).

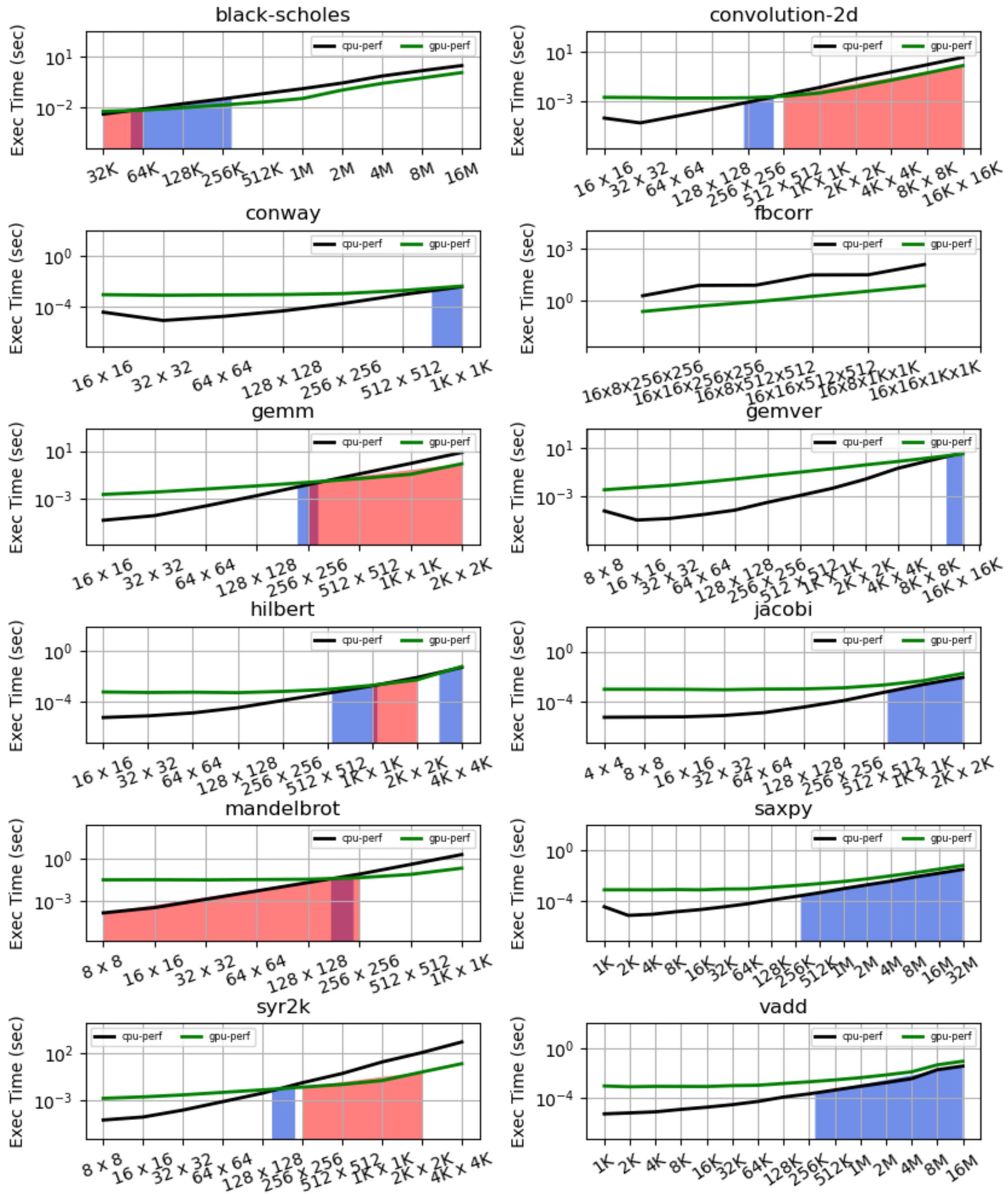



Figure 6.5: ALPyNA Cost Model misprediction ranges (shaded blue) for 12 loop-intensive benchmarks with varying domain sizes on platform *T1*. ACM's domain crossover point is interpolated from the measured values. The misprediction range of the SVM predictor is shaded red. The intersection of mispredicted domain ranges is shaded purple.

| Benchmark | T1: CPU/GPU 2960/1405MHz | | | | T2: CPU/GPU 3900/1500MH | | | | T3: CPU/GPU 800/1500 MHz | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ACM | | SVM | | ACM | | SVM | | ACM | | SVM | |
| | mean | max | mean | max | mean | max | mean | max | mean | max | mean | max |
| black-scholes | 1.16 | 2.29 | 1.04 | 1.45 | 1.50 | 2.60 | 1.03 | 1.34 | 2.70 | 8.21 | 3.61 | 8.21 |
| conv-2d | 1.07 | 2.30 | 1.69 | 4.56 | 1.00 | 1.00 | 1.27 | 2.05 | 1.07 | 2.12 | 2.33 | 6.91 |
| conway | 1.01 | 1.10 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.04 | 1.36 | 1.04 | 1.36 |
| fbcorr | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 20.44 | 51.38 |
| gemm | 1.05 | 1.54 | 1.90 | 8.68 | 1.00 | 1.00 | 1.40 | 2.70 | 1.07 | 1.82 | 1.96 | 11.69 |
| gemver | 1.01 | 1.23 | 1.00 | 1.00 | 1.01 | 1.12 | 1.00 | 1.00 | 1.00 | 1.00 | 1.01 | 1.16 |
| hilbert | 1.02 | 1.16 | 1.05 | 1.54 | 1.02 | 1.27 | 1.00 | 1.03 | 1.11 | 2.08 | 1.23 | 2.50 |
| jacobi | 1.15 | 2.05 | 1.00 | 1.00 | 1.09 | 2.53 | 1.00 | 1.00 | 1.01 | 1.14 | 1.01 | 1.15 |
| mandelbrot | 1.00 | 1.00 | 7.12 | 237.12 | 1.05 | 1.58 | 7.80 | 296.69 | 1.00 | 1.00 | 7.53 | 260.25 |
| saxpy | 1.57 | 5.23 | 1.00 | 1.00 | 1.35 | 3.64 | 1.00 | 1.00 | 1.07 | 1.57 | 1.00 | 1.00 |
| syr2k | 1.00 | 1.00 | 2.27 | 90.33 | 1.10 | 2.64 | 1.95 | 26.38 | 1.10 | 2.59 | 2.21 | 87.83 |
| vadd | 1.68 | 6.33 | 1.00 | 1.00 | 1.30 | 5.07 | 1.00 | 1.00 | 1.10 | 2.24 | 1.00 | 1.00 |
| Geo.Mean | 1.17 | 1.75 | 1.34 | 3.34 | 1.13 | 1.73 | 1.27 | 2.50 | 1.14 | 2.16 | 1.81 | 6.25 |

Table 6.2: Misprediction penalties for all benchmarks on all platforms: showing Geometric Mean Penalty across all input sizes, and Maximum Penalty.

**Misprediction Ranges.** Another metric to investigate the prediction performance of ACM is the range of domain sizes over which ACM correctly predicts the performant device in a platform. Figure 6.5 shows CPU and GPU benchmark runtimes of the 12 benchmarks on *T1* with ACM's misprediction intervals shaded in blue. The corresponding range mispredicted by the SVM is shown in Figure 6.5 by misprediction intervals shaded red. Where both ACM and the SVM model mispredict, the overlap between both ranges are shaded purple. For benchmarks which ACM correctly predicts across all domain sizes, there are no regions shaded blue. Likewise, benchmarks for which the SVM predicts correctly across all domain sizes do not have any region shaded red. Figures B.2 and B.4 in Appendix B show the misprediction ranges for platforms *T2* and *T3*.

| Benchmarks | ACM Misprediction Range | | | SVM Misprediction Range | | |
|---|---|---|---|---|---|---|
| | T1 | T2 | T3 | T1 | T2 | T3 |
| black-scholes | 0.30 | 0.60 | 0.70 | 0.10 | 0.10 | 0.90 |
| conv-2d | 0.09 | 0.00 | 0.09 | 0.45 | 0.45 | 0.54 |
| conway | 0.14 | 0.00 | 0.14 | 0.00 | 0.00 | 0.14 |
| fbcorr | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 1.0 |
| gemm | 0.13 | 0.00 | 0.13 | 0.38 | 0.38 | 0.38 |
| gemver | 0.08 | 0.08 | 0.00 | 0.00 | 0.00 | 0.08 |
| hilbert | 0.22 | 0.11 | 0.22 | 0.11 | 0.11 | 0.33 |
| jacobi | 0.20 | 0.10 | 0.10 | 0.00 | 0.00 | 0.10 |
| mandelbrot | 0.00 | 0.13 | 0.00 | 0.63 | 0.63 | 0.63 |
| saxpy | 0.43 | 0.25 | 0.18 | 0.00 | 0.00 | 0.00 |
| syr2k | 0.00 | 0.10 | 0.10 | 0.30 | 0.30 | 0.30 |
| vadd | 0.40 | 0.20 | 0.13 | 0.00 | 0.00 | 0.00 |
| mean | 0.16 | 0.13 | 0.15 | 0.16 | 0.16 | 0.37 |

Table 6.3: Ratio of mispredicted/correct ranges for all benchmarks on platforms *T1*, *T2* and *T3* using ACM and an SVM predictor

If ACM selects GPU execution over the CPU, the threshold domain size predicted by ACM is interpolated from the results of the cost model predictions. The oracular threshold domain size (if it exists) is interpolated from the actual execution results. The misprediction range is calculated as the domain sizes between these two points. ACM delivers similar misprediction ranges for eight benchmarks on platforms *T1*, *T2* and *T3*. Table 6.3 shows the ratio of mispredicted domain ranges over which the benchmarks are executed. ACM has the same or smaller misprediction intervals than the SVM predictor for 5,7 and 10 of the benchmarks on *T1*,*T2* and *T3* respectively. The overall mean mispredicted domain range ratio is 0.146 for ACM and 0.23 for the SVM predictor.

The misprediction ranges are generally small for benchmarks with sharply diverging CPU and GPU runtime curves such as *conv-2d*, *gemm*, *mandelbrot* and *syr2k*. The exceptions are *black-scholes* on *T2* (0.6) and *T3* (0.7), where ACM mispredicts that CPU execution will be faster over the initial range before correctly selecting the GPU at medium to large domain sizes. For benchmarks where the CPU always executes faster than the GPU, *gemver* and *conway* have a small range of mispredicted domains but mispredicts at large domain sizes for *saxpy* and *vadd*.

## 6.4 Summary

This chapter has presented the ALPyNA Cost Model, a lightweight cost model to select the most performant target device in a CPU – GPU heterogeneous environment. ACM is staged, combining compile-time analysis with runtime introspection in the CPython interpreter to parameterise the static model. The model is designed to be lightweight as it is evaluated at runtime. The ACM models for interpreted and JIT-compiled CPU execution are simple and relatively standard. In contrast the GPU model is both elaborate and novel as it accounts for key device costs, including the time to transfer data to and from the device, block structured execution and starvation effects arising due to the relatively slower execution speed of the interpreter. All of the platform models are parametric in key characteristics of the devices in a heterogeneous platform, like warp size, number of SMs, operating frequency and cache size.

An important scientific contribution of ACM is to show that a comparatively simple staged analytical model can effectively determine at runtime whether to offload a loop nest in a dynamic language. ACM is evaluated on three heterogeneous platforms *T1*, *T2* and *T3* across 360 experiments with 12 loop-intensive Python benchmark programs (Section 6.3). The results show small misprediction intervals and a mean slowdown of just 14.6%, relative to the optimal (oracular) offload strategy. This is a lower overall mean misprediction penalty compared to the predictions of a trained SVM model (45.0%). The overall domain range

misprediction across all experiments is 14.6% which is lower than the overall mispredicted range of the SVM (23.0%).

Comparatively simple cost models for the parallel platforms suffice because the system does not attempt to accurately predict *absolute* loop runtimes, rather it compares the *relative* runtimes on the GPU and CPU with interpretation or JIT compilation.

# Chapter 7

# Conclusion

Parallel programming of accelerator devices such as GPUs enable programmers to significantly reduce execution time. However, directly programming such devices e.g. in CUDA or OpenCL requires low-level expertise. The research in this thesis aims to aid non-expert developers programming in an interpreted dynamic language (Python) to automatically exploit accelerator parallelism.

This thesis presents the development of ALPyNA, a staged loop dependence analysis framework and code generator for Python. An extensive evaluation of the execution profile and performance gains from ALPyNA led to the identification of the requirement for a lightweight cost model to be used in a JIT environment. The developed cost model is integrated into ALPyNA to direct the automatic runtime exploitation of the most performant target device in a heterogeneous execution environment.

This chapter reviews the challenges and opportunities identified by this thesis. Section 7.1 reviews the problem addressed by this thesis. Section 7.2 summarises the main contributions of this thesis. Section 7.3 presents a critical analysis of these contributions and presents solutions that could potentially overcome or resolve some of these constraints. Finally, Section 7.4 shows future directions that this work could potentially take.

## 7.1 The Problem

Accelerators such as GPUs provide large computational speedups of data parallel programs. Modern GPUs speed up data parallel computation by executing a large number of threads in parallel. Threads are scheduled on a large number of simple CUDA cores arranged in blocks called SMs. These threads are scheduled for execution in *block-cyclic* manner. To exploit the performance potential, GPUs have to be programmed using low-level programming C-like languages such as CUDA and OpenCL. These programming languages require developers

to have a deep understanding of the hardware characteristics of each GPU and the GPGPU parallel programming model.

Dynamic languages such as Python are increasingly being used in industry and academia for numerical computation. Language bindings to CUDA and OpenCL interfaces enable programmers to write GPU kernels using the semantics of the host language. However, such methods still expose programmers to the hardware complexity of a GPU. Parallel programming interfaces to reduce programming complexity of GPUs include providing APIs and libraries with specific operations.

Loop parallelisation has been researched and incorporated into parallelising compilers for static languages such as FORTRAN and C/C++. In dynamic languages, loop parallelisation has mainly focussed on runtime JIT compilation for CPUs. Trace based JIT compilation for dynamic languages target the CPU because it relies on optimising and executing binary instructions of the CPU that have already been executed. Many efforts to automatically exploit parallelisation on GPUs have concentrated on algorithmic skeletons as discussed in Chapter 3. However, little work has been done to automatically parallelise loops in dynamic languages such as Python and to exploit the performance benefits of accelerators in a heterogeneous environment.

## 7.2 Contributions

This thesis addresses the hypotheses outlined in Section 1.1 as follows :

### 7.2.1 Staged Automatic Loop Parallelisation

Hypothesis H1. *Staged static and dynamic dependence analysis on array-centric loop nests, in a general purpose dynamic scripting language, will yield higher performance parallel code than static dependence analysis alone.*

Chapter 4 presented ALPyNA, a novel loop parallelisation framework to parallelise linear loop nests written in plain Python. ALPyNA stages dependence analysis to maximise the opportunity to parallelise loop nests. The static phase parses the AST of loop nests and converts them into closures. Each closure is associated with in-memory data structures holding dependence relationships between various *load–store* memory accesses. If these relationships cannot be resolved statically, resolution of these dependence relationships is deferred to runtime and the whole loop nest is marked for runtime analysis. If all dependence relationships can be statically inferred, then ALPyNA generates and caches untyped skeletal kernels.

At runtime, loop nests marked for runtime analysis are analysed after augmenting dependence relationships that were not resolved during static compilation. ALPyNA introspects the CPython runtime environment to resolve dependence relationships and maximise the potential for parallelisation. Once the dependence relationships are determined, CPU or GPU kernels are generated.

Chapter 4 also presented the techniques used for generation of GPU kernel and driver code. GPU kernel and driver generation is customised at runtime to maximise the parallelism obtained from all parallel loops. Loops carrying dependences are executed sequentially in the appropriate order to preserve dependence constraints.

Runtime introspection is used to determine types for the data structures referenced within kernels. The runtime discovered types are patched into the kernels at runtime to enable JIT compilation. ALPyNA utilises the Numba compiler for JIT compilation of CPU code and GPU kernels.

### 7.2.2 Systematic Analysis of ALPyNA

Hypothesis H2. *Code can be automatically synthesised to target heterogeneous architectures, with minimal user intervention. When such code is executed on a resource-aware adaptive Virtual Machine (VM), it will*

- *almost never degrade performance despite the overheads of dynamically re-targeting code*
- *in many instances significantly reduce execution time.*

A systematic analysis of the performance of ALPyNA is presented in Chapter 5. ALPyNA's capabilities are exercised using 12 benchmarks of varying complexities. These vary from easily parallelisable singly nested loops to nested loops with loop carried dependences and control flow divergence within the body of the loop. Over 240 experiments were performed across two heterogeneous platforms *T1* and *T2*. The execution times were measured on a range of iteration domain sizes to compare the relative performance between the interpreter, CPU and GPU.

**Interpreter vs CPU**: For each benchmark, performance of the JIT compiled CPU variant was faster except at lower iteration domain sizes. At lower domain sizes ALPyNA's analysis time and code generation, and Numba's compilation time dominated overall execution time. Measured execution speed-ups ranged from 0.001x – 1199.9x (*T1*) and 0.002x – 1290.19x (*T2*). CPU speedups continued to increase across all domain ranges once the overhead of analysis and compilation were overcome.

**Interpreter vs GPU**: The performance of GPU code generated by ALPyNA showed large speedups for larger iteration domain sizes when the interpreter execution time was greater than dependence analysis and compilation time. GPU speedups were between 0.0001x (*T1*) and 0.001x (*T2*) at the lower end to a maximum of 16435x (*T1*) and 13895x (*T2*).

**GPU vs CPU** A comparison between the GPU and CPU variant showed that the CPU is the performant device for smaller iteration domain sizes for all benchmarks. The GPU executes *heavy* parallel loop nests faster than the CPU at larger iteration domain sizes. Execution is found to always be faster on the CPU for *light* parallel workloads. The relative execution speed of the GPU variant was between 0.22x – 179.55x (*T1*) and 0.33x – 50.68x (*T2*).

### 7.2.3 Lightweight analytical cost model for ALPyNA

Hypothesis H3. *A staged static and dynamic analytical cost-model can accurately determine the quicker device that will execute a given instance of an array-centric loop nest written in a dynamic scripting language in a heterogeneous CPU–GPU environment. Such a cost model need only be parameterised on the hardware characteristics of the CPU and the GPU, and requires only installation time profiling of the relevant compute devices using a simple pre-determined kernel.*

Chapter 6 presented the design, implementation and evaluation of ACM, the first analytical cost model for loop nest parallelisation supporting automatic runtime exploitation of GPUs in a dynamic language. ACM is staged and lightweight, and is used to select between compute devices to effectively parallelise moderately complex loop nests using ALPyNA.

For each instance of a loop nest, ACM dynamically predicts the relative runtimes on alternative devices so that ALPyNA can select the fastest. The models for the CPython interpreter, and for JIT compiled CPU execution are relatively standard, but the GPU model is both novel and elaborate. It accounts for key costs like data transfer time and starvation effects. All of the platform models are parametric in key characteristics of the platforms, like cache size and sharing, and warp size on the GPU.

A systematic evaluation of ACM on three heterogeneous platforms is also performed using 12 standard loop-intensive Python benchmarks. Each benchmark covers a wide range of domain sizes. The cost model proves to be effective, with a mean misprediction range of 14.66% and a mean misprediction penalty of just 14.6% slowdown, relative to optimal, across all benchmarks. The ACM is better on all three platforms compared to a trained SVM predictor model when considering overall misprediction penalty. ACM's overall misprediction range is also better than or equal to the predictions of a trained SVM machine learning model on all three platforms.

Machine Learning models require the upfront effort of training using representative data sets.

This would require extensive profiling of each benchmark on a wide range of domain sizes to accurately predict the most performant device at runtime. Accurate predictions are also dependent on a similar loop dependence structure being identified by the model. ALPyNA uses a one time simple profiling phase to determine cost factors for each accelerator, to use in ACM. ALPyNA's runtime dependence analysis and cost model customised to the hardware characteristics of the CPU – GPU system provides better performant device prediction while being sufficiently lightweight.

## 7.3 Limitations

While ALPyNA can parallelise moderately complex loop nests written in plain Python code, there are some restrictions that limit the scope of analysis. These limitations are engineering design decisions carefully made to minimise implementation effort. These constraints do not detract from the overall findings of this thesis.

### 7.3.1 ALPyNA

**Subscript indexing**: As ALPyNA uses Numba for GPU compilation and data transfer between host and GPU memory, currently only Numpy arrays and basic scalar types are allowed. Each variable subscript must be a linear function of loop iterator values. ALPyNA only supports basic indexing; array slices are not allowed. All dereferencing of a Numpy vector read should evaluate to a value which has that vector's '*dtype*' and every write should access the location of a scalar within a Numpy vector. This precludes Numpy array broadcasting semantics.

**Side effects**: ALPyNA requires that any loop nest analysed is side-effect free. Function calls within a loop nest body are checked against a known list of side-effect free, read-only functions. JIT compilation is performed on loop nests with homogeneous vectors of types that can be JIT compiled for both CPU and GPU. These are obtained at runtime through introspection. Within the body of the loop nest, operators on these vectors should not be overridden. This can be checked by checking whether the status of the method mapped to the operator is *read-only*. Restricting developers to these operations enables ALPyNA to ensure that parallel execution instances of GPU kernels do not break dependence constraints. ALPyNA does not currently support runtime detection of side-effecting functions through the use of Python packages such as `purefunc` [2].

**Deoptimisation and exceptions**: For GPU JIT compilation with Numba, all vectors within a loop nest are required to be homogeneous Numpy vectors. Enabling standard Python lists for JIT compilation would require transforming data to Numpy arrays. A vector detected

as non-homogeneous during this transformation would require deoptimisation i.e. execution would have to be continued in the interpreter. However this is currently not implemented. ALPyNA does not speculatively JIT compile code for partial paths within the loop nest. Analysis and compilation is instead performed on each loop nest in its entirety after runtime dependence analysis. Exception handling within JIT compiled code is not currently handled automatically. All exceptions that are raised within a loop nest are reported back to the user (Section 4.1.1).

**Inter-procedural analysis**: ALPyNA considers each loop nest within a function as a single unit for dependence analysis. Currently only intra-procedural dependence analysis is performed on the loop nest. Function calls within a loop nest allow for a richer representation of programs. Although ALPyNA recognises calls to pure intrinsic functions that are supported by Numba, these calls are not subject to inter-procedural analysis and their validity must be guaranteed by the user.

### 7.3.2 ALPyNA Cost Model

The ALPyNA Cost Model (ACM) is sufficient to support research hypothesis H3., but also has some limitations. ACM currently considers the time taken for execution and data transfer to select the performant device for JIT compilation. Compilation costs are currently not taken into consideration. The implication of this limitation is that ACM never selects the interpreter as the fastest device even for small iteration domains. However, JIT compilation time for kernels can be a significant factor in overall execution time (Section 5.5). For frequently executed kernels, it helps that Numba persists generated code in a compilation cache. Fetching compiled kernels from a persistent cache will improve overall execution times. Approaches to account for runtime compilation and fetching cached code (e.g. models like [88]) are expected to be relatively straightforward to implement in ALPyNA. This work is left for future work to improve ACM.

## 7.4 Future Work

**Multi-device support**: An immediate avenue for future work is to evaluate the performance of code generated by ALPyNA on different kinds of devices in a heterogeneous environment. Single board computers such as the NVIDIA Jetson [54] integrate the CPU and GPU into a single system-on-chip architecture. Such systems can then exploit the shared RAM to reduce the data transfer time. These could include devices such as parallelisation on multi-core CPUs or other devices such as FPGA devices. ALPyNA's Hardware Abstraction Layer

(HAL) layer is modular and extensible, enabling easy addition of code generators for each individual device.

The parameters required by the cost model are abstracted within this HAL. This allows ACM's cost model for each individual device to incorporate as many parameters as required to accurately model device performance. ACM normalises the cost of execution of each statement to the interpreter execution time. A relative cost calculation can be performed across all devices This enables the prediction of a performant device for each loop nest.

The addition of a OpenCL based code generator would expand the range of target devices that ALPyNA can parallelise and JIT compile code for. Support for a OpenCL compiler has been very recently added to Numba to enable support for AMD GPUs.

**Support for other linear co-routines**: Loop nest domains are currently expressed only using the `range` function. Other well understood non-mutable linear iterators such as `enumerate` and `list` iterators are planned to be added for analysis. These may be implemented either by transforming the loop nest during AST parsing to use the `range` function or by building a map of (*start, end , stride*) values for each known co-routine for the purpose of dependence analysis.

**Support for comprehensions**: Comprehensions are Python constructs that are generated from other iterable sequences. They are often generated using nested loop structures. Currently, comprehension generators for GPUs are not supported. However, this is not a design limitation and can be added with relative ease.

**Optimisation improvements**: ALPyNA generates one GPU kernel per statement to maximise opportunities for parallelisation (Section 6.2.3). When correctness can be ensured, loop fusion may provide execution performance improvement of perfect loop nests. Loop fusion reduces the overhead of launching GPU kernels from the interpreter. This will favour GPU execution at smaller iteration domain sizes.

**Loop analysis and GPU compilation for other dynamic languages**: ALPyNA primarily focusses its implementation on Python. However, the fundamental principles used in this thesis can be transferred to other interpreted dynamic languages such as JavaScript and Ruby. An interesting research avenue would be to see if a common IR or interface can be developed to support other dynamic languages. This would be similar to the direction taken by MLIR [80] for static languages. A corollary of this research could also look into how language level features may affect the cost modelling to guide JIT compilation for target devices.

While parallelisation in general is a well-studied topic, the work in this thesis shows that runtime parallelisation and JIT compilation of loops for dynamic languages is a feasible and potentially rewarding area for exploration.

# Appendix A

# Evaluation of ALPyNA on platform *T2*

Chapter 5 evaluated the performance of ALPyNA on two platforms *T1* and *T2*. The runtime performance and overheads for platform *T1* are displayed within the chapter. This appendix provides the corresponding results for *T2*.

Figure A.1 plots the total execution time of the CPython and ALPyNA runtimes over a large number of domain sizes for the 12 benchmarks as described in Section 5.1. Table A.1 extracts the minimum, maximum and mean relative speed-ups of the CPU relative to the interpreter. Table A.2 shows the minimum, maximum and mean speed-ups of ALPyNA's generated GPU code relative to the CPython interpreter and JIT compiled CPU code. Figure A.2 shows the proportion of time spent by ALPyNA for analysis, compilation and execution across all iteration domain sizes. Figure A.3 plots the execution time of JIT compiled CPU and GPU code generated by ALPyNA over a large number of domain sizes for the 12 benchmarks. for analysis, compilation and execution across all iteration domain sizes. Figure A.4 displays a graph showing the overheads incurred by ALPyNA for loop nests and Table A.3 summarises these results.

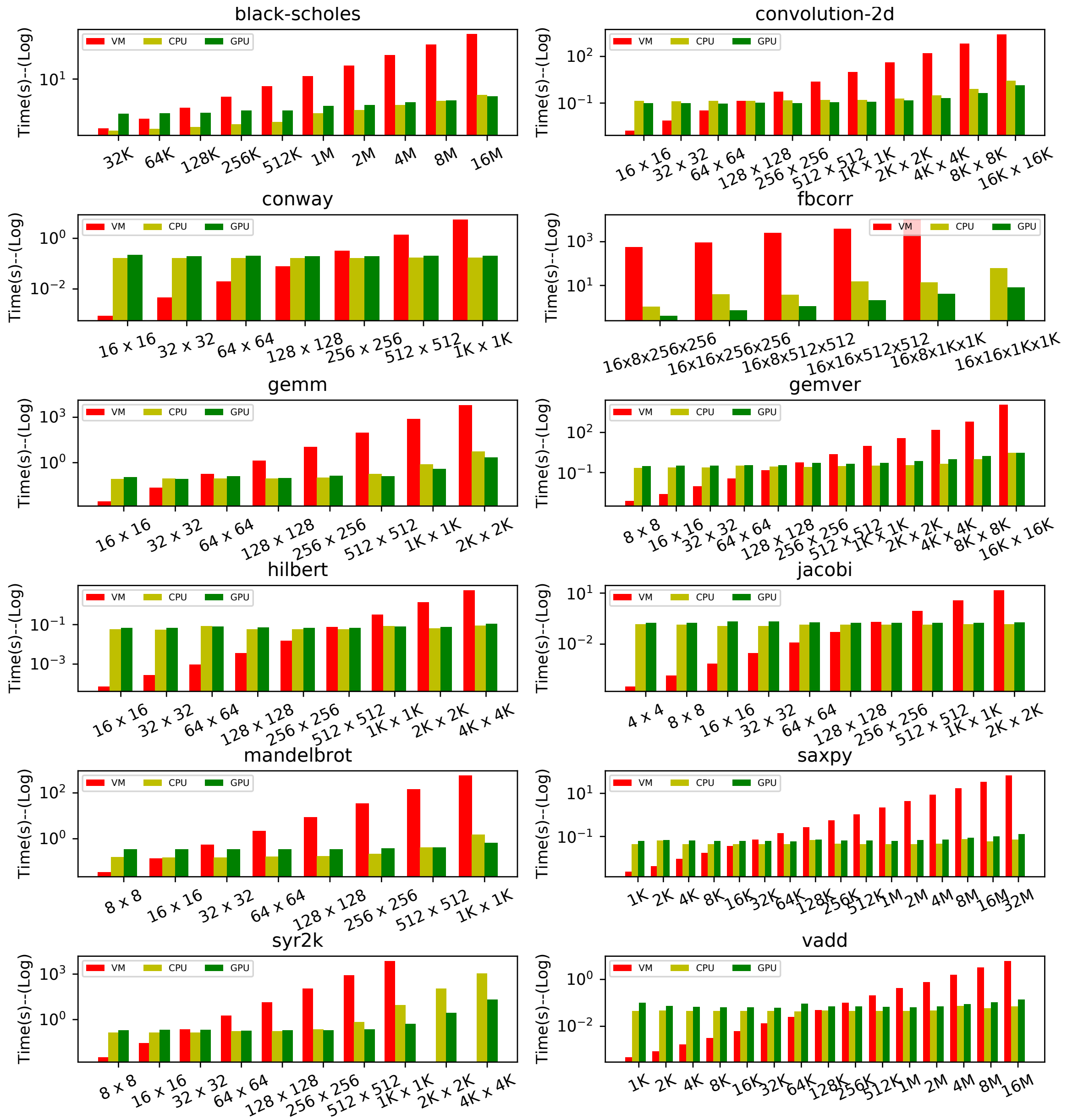



Figure A.1: Total execution time of ALPyNA generated code on the CPU and GPU vis-a-vis the CPython VM on Platform *T2* (Lower is better). Times are scaled logarithmically. Total execution time includes analysis and code generation, compilation, and execution time. Interpreter execution for the largest two domains of syr2k ($2k \times 2k$ and $4k \times 4k$) timed out after 3 hours. Results are summarised in Table A.2.

| Benchmarks | JIT CPU vs CPython | | |
|---|---|---|---|
| | min | max | mean |
| black-scholes | 1.14 (32K) | 53.04 (16M) | 17.27 |
| convolution-2d | 0.012 (16 x 16) | 973.37 (16K x 16K) | 234.93 |
| conway | 0.005 (16 x 16) | 32.02 (1K x 1K) | 6.05 |
| fbcorr | 229.31 (16x16x256x256) | 722.62 (16x8x1Kx1K) | 540.05 |
| gemm | 0.032 (16 x 16) | 1143.62 (2K x 2K) | 353.83 |
| gemver | 0.003 (8 x 8) | 3859.98 (16K x 16K) | 412.59 |
| hilbert | 0.001 (16 x 16) | 61.43 (4K x 4K) | 9.71 |
| jacobi | 0.0002 (4 x 4) | 96.56 (2K x 2K) | 12.96 |
| mandelbrot | 0.22 (8 x 8) | 364.85 (1K x 1K) | 116.47 |
| saxpy | 0.05 (1K) | 931.89 (32M) | 132.70 |
| syr2k | 0.02 (8 x 8) | 1290.19 (512 x 512) | 335.33 |
| vadd | 0.010 (1K) | 86.62 (16M) | 13.15 |

Table A.1: Performance of JIT compiled CPU loop nest variant relative to CPython interpreter execution on platform *T2*. Iteration sizes corresponding to each data points are shown in brackets.

| Benchmarks | Relative speedup | | | | | |
|---|---|---|---|---|---|---|
| | ALPyNA GPU vs CPython | | | ALPyNA GPU vs CPU JIT | | |
| | min | max | mean | min | max | mean |
| black-scholes | 0.38 (32K) | 59.55 (16M) | 15.02 | 0.33 (32K) | 1.12 (16M) | 0.62 |
| convolution-2d | 0.018 (16x16) | 1897.53 (16Kx16K) | 411.4 | 1.32 (2Kx2K) | 1.94 (16Kx16K) | 1.50 |
| conway | 0.004 (16x16) | 27.08 (1Kx1K) | 5.13 | 0.76 (16x16) | 0.86 (512x512) | 0.83 |
| fbcorr | 1247.82 (16x16x256x256) | 2416.78 (16x8x1Kx1K) | 1820.78 | 2.65 (16x8x256x256) | 7.61 (16x16x1Kx1K) | 4.97 |
| gemm | 0.025 (16x16) | 2885.78 (2Kx2K) | 732.73 | 0.71 (64x64) | 2.52 (2Kx2K) | 1.28 |
| gemver | 0.002 (8x8) | 3795.96 (8Kx8K) | 366.67 | 0.46 (4Kx4K) | 0.98 (16Kx16K) | 0.67 |
| hilbert | 0.001 (16x16) | 49.2 (4Kx4K) | 8.0 | 0.8 (128x128) | 1.10 (1Kx1K) | 0.87 |
| jacobi | 0.002 (4x4) | 77.12 (2Kx4K) | 10.33 | 0.53 (16x16) | 0.83 (4Kx4K) | 0.74 |
| mandelbrot | 0.1 (8x8) | 869.02 (1Kx1K) | 167.78 | 0.43 (32x32) | 2.38 (1Kx1K) | 0.78 |
| saxpy | 0.037 (1K) | 507.9 (32M) | 80.54 | 0.54 (32M) | 0.98 (128K) | 0.72 |
| syr2k | 0.016 (8x8) | 13894.955 (1Kx1K) | 2287.63 | 0.64 (32x32) | 50.68 (4Kx4K) | 11.48 |
| vadd | 0.016 (1K) | 44.62 (16M) | 7.77 | 0.44 (1K) | 0.86 (4M) | 0.64 |

Table A.2: Performance of ALPyNA GPU kernels relative to CPython interpreter execution and CPU JIT compiled loop nest on platform *T2*. Iteration sizes corresponding to each data points are shown in brackets.

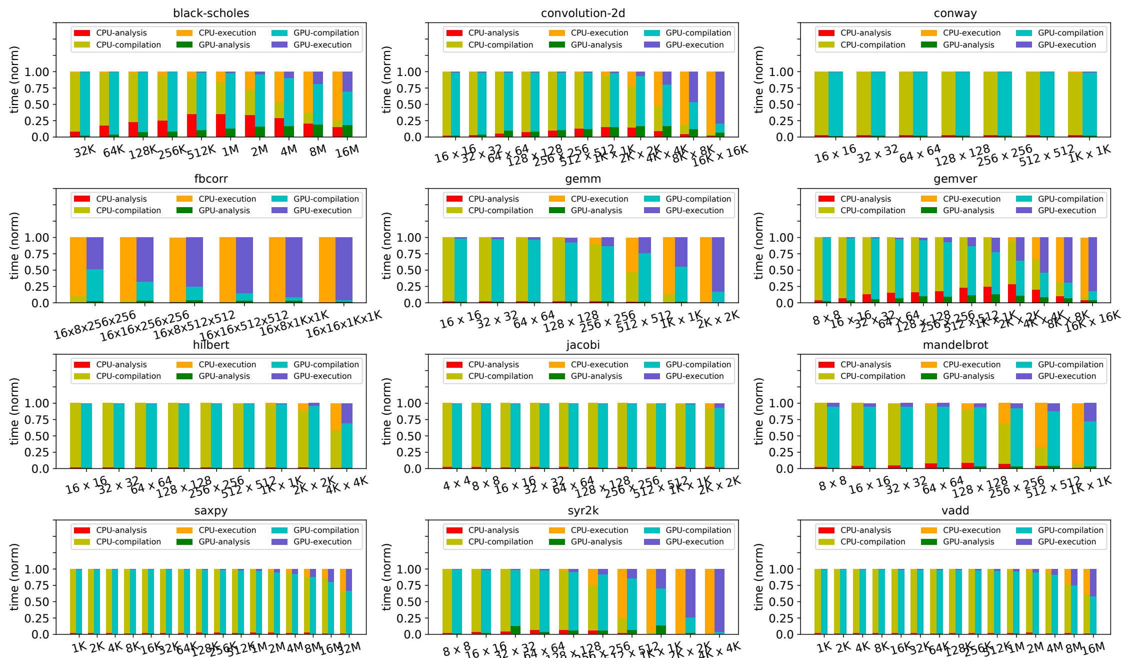



Figure A.2: Proportion of time spent by ALPyNA for analysis, compilation and execution on JIT compiled CPU and GPU code on Platform *T2*.

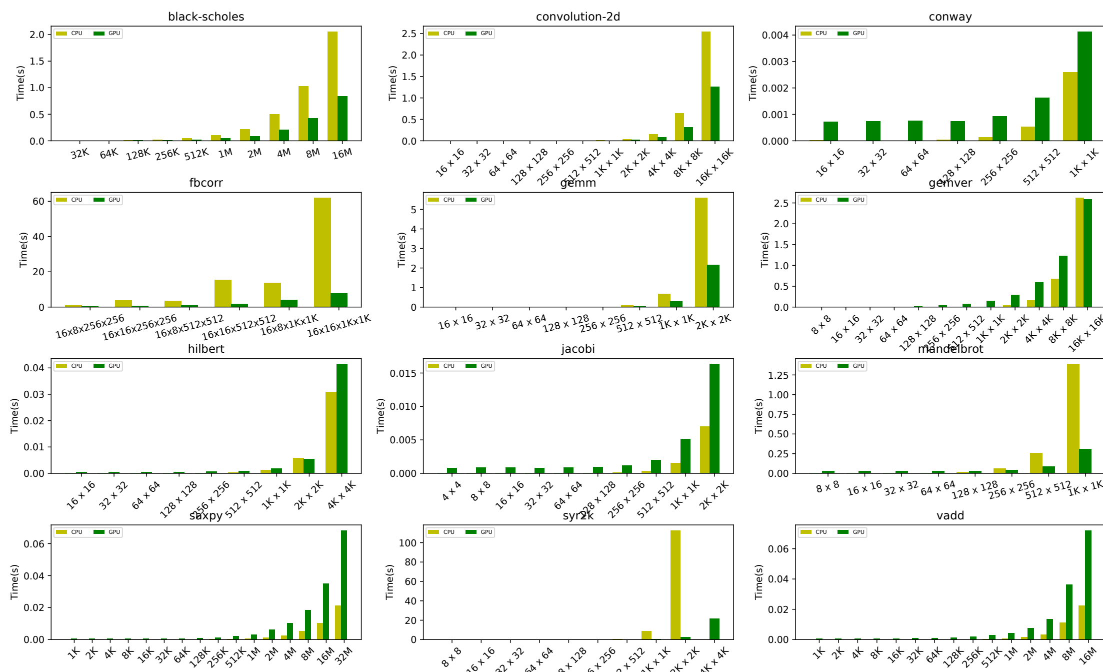



Figure A.3: Execution time of ALPyNA generated code on the CPU vs the GPU on Platform *T2* (Lower is better). Times are scaled logarithmically. CPU execution time for the largest domain of syr2k ($4k \times 4k$) is omitted to better represent execution times of smaller domain sizes.

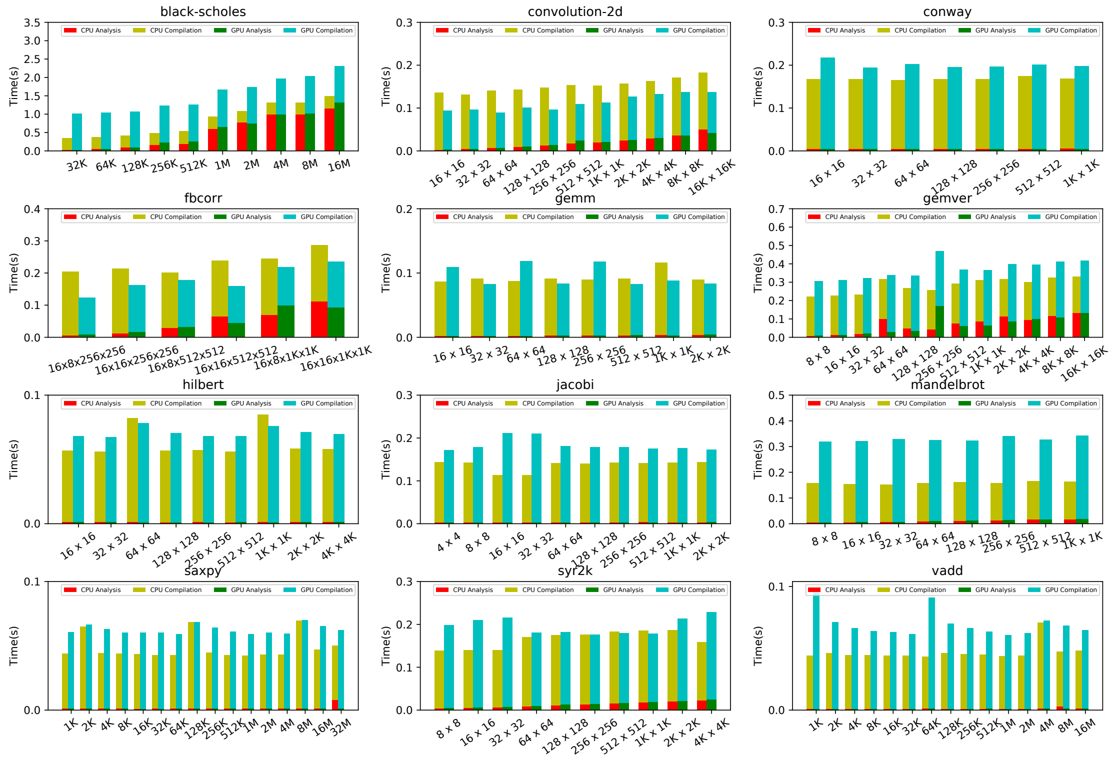



Figure A.4: ALPyNA runtime analysis and compilation overhead (Platform T2).

| Benchmark | ALPyNA analysis + code generation (sec) | | | | | | Compilation (sec) | |
|---|---|---|---|---|---|---|---|---|
| | CPU | | | GPU | | | CPU | GPU |
| | min | max | mean | min | max | mean | | |
| black-scholes | 0.024 | 1.152 | 0.509 | 0.031 | 1.328 | 0.543 | 0.321 | 0.992 |
| convolution-2d | 0.003 | 0.050 | 0.019 | 0.003 | 0.042 | 0.020 | 0.133 | 0.092 |
| conway | 0.004 | 0.006 | 0.005 | 0.004 | 0.005 | 0.005 | 0.164 | 0.196 |
| fbcorr | 0.006 | 0.111 | 0.049 | 0.009 | 0.100 | 0.049 | 0.183 | 0.130 |
| gemm | 0.002 | 0.004 | 0.003 | 0.002 | 0.004 | 0.003 | 0.090 | 0.093 |
| gemver | 0.008 | 0.132 | 0.071 | 0.009 | 0.172 | 0.070 | 0.212 | 0.300 |
| hilbert | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.062 | 0.069 |
| jacobi | 0.003 | 0.004 | 0.003 | 0.003 | 0.004 | 0.003 | 0.133 | 0.180 |
| mandelbrot | 0.005 | 0.018 | 0.011 | 0.005 | 0.019 | 0.012 | 0.149 | 0.317 |
| saxpy | 0.001 | 0.008 | 0.002 | 0.001 | 0.001 | 0.001 | 0.047 | 0.061 |
| syr2k | 0.005 | 0.023 | 0.013 | 0.005 | 0.026 | 0.014 | 0.153 | 0.182 |
| vadd | 0.001 | 0.003 | 0.001 | 0.001 | 0.001 | 0.001 | 0.045 | 0.069 |

Table A.3: Analysis and code generation time taken by ALPyNA on platform *T2* for CPU and GPU. All times are in seconds.

# Appendix B

# Evaluation of ACM on platform *T2* and *T3*

Chapter 6 evaluated the performance of the cost model used to predict the performant target device for JIT compilation. The evaluation was performed on three platforms *T1*, *T2* and *T3*. The graphs for misprediction penalites and mispredicted ranges for platform *T1* is shown within the chapter. This appendix provides the corresponding graphs for platforms *T2* and *T3*.

Figures B.1 and B.3 plot the predicted performant device for each benchmark over increasing domain ranges for platforms *T2* and *T3* respectively. An optimal prediction is the value 1.0. The misprediction penalty is the slowdown in execution due to a wrong prediction. Figure B.2 and B.4 show the domain range over which the ACM predicts the wrong device for platforms *T2* and *T3* respectively. This is done by plotting the actual execution times for the CPU and GPU and interpolating the cross-over domain threshold of ACM. These domain ranges are shaded in blue. Alongside this, the misprediction range of a trained SVM model is also plotted in Figures B.1 and B.3 (shaded red). The intersecting domain ranges are shaded purple.

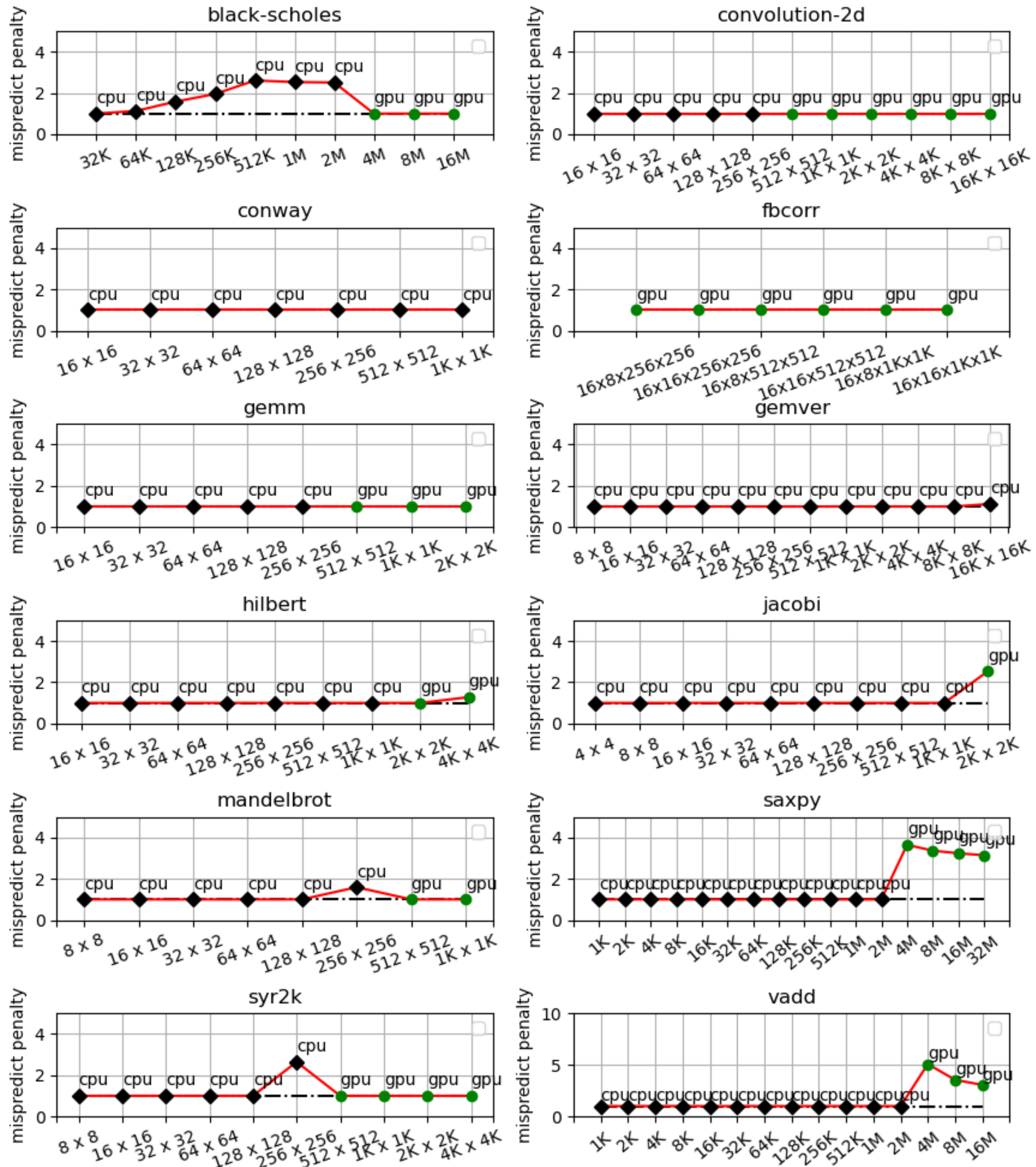



Figure B.1: ALPyNA Cost Model misprediction penalties for 12 loop-intensive benchmarks with varying domain sizes on platform *T2*. Misprediction slowdown is the ratio of predicted device runtime and faster device runtime, so optimal is always 1.0.

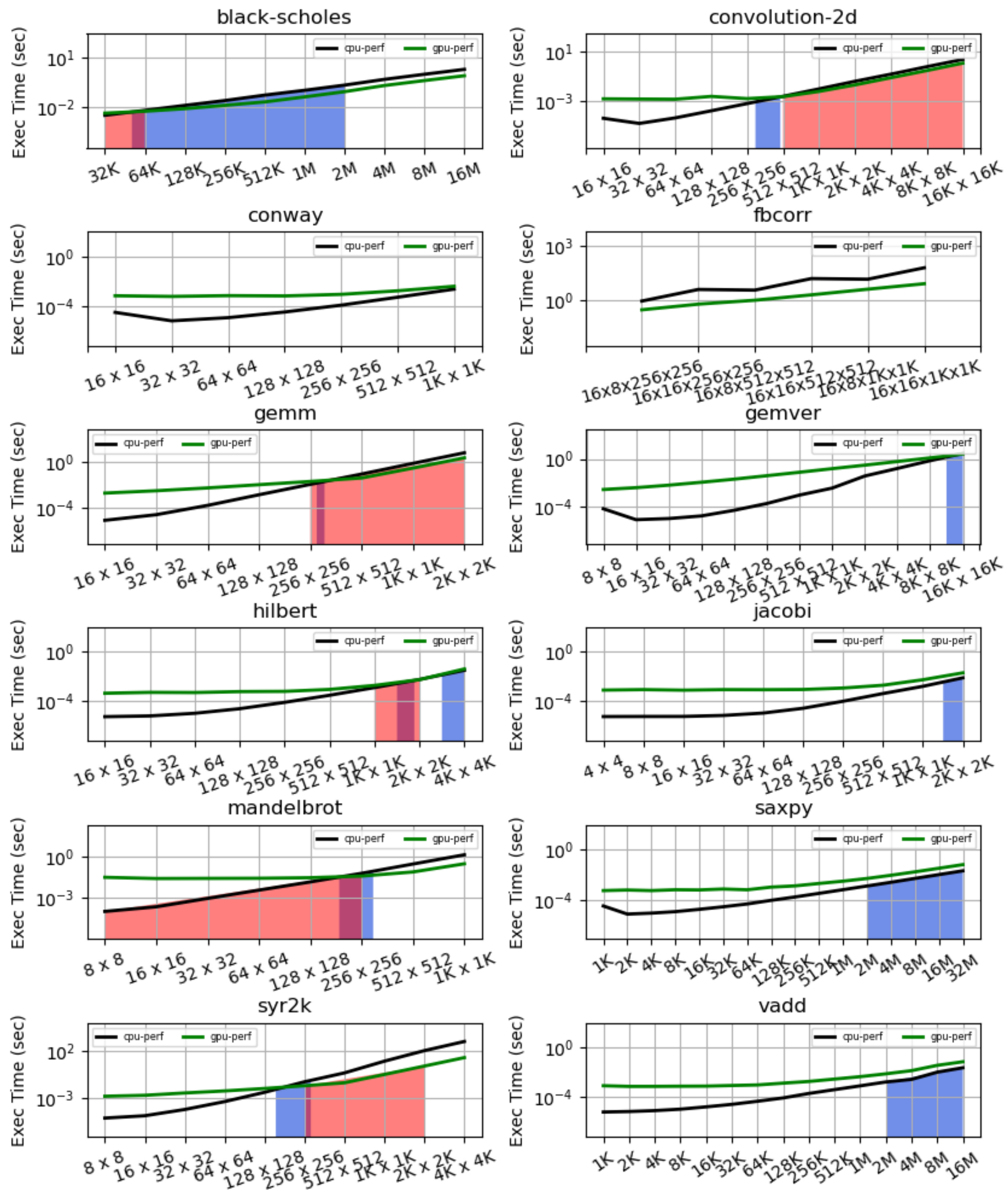



Figure B.2: ALPyNA Cost Model misprediction ranges (shaded blue) for 12 loop-intensive benchmarks with varying domain sizes on platform *T2*. ACM's domain crossover point is interpolated from the measured values. Misprediction range of SVM model is also shown (shaded red). The domain ranges at which both models mispredict the performant device is shaded purple.

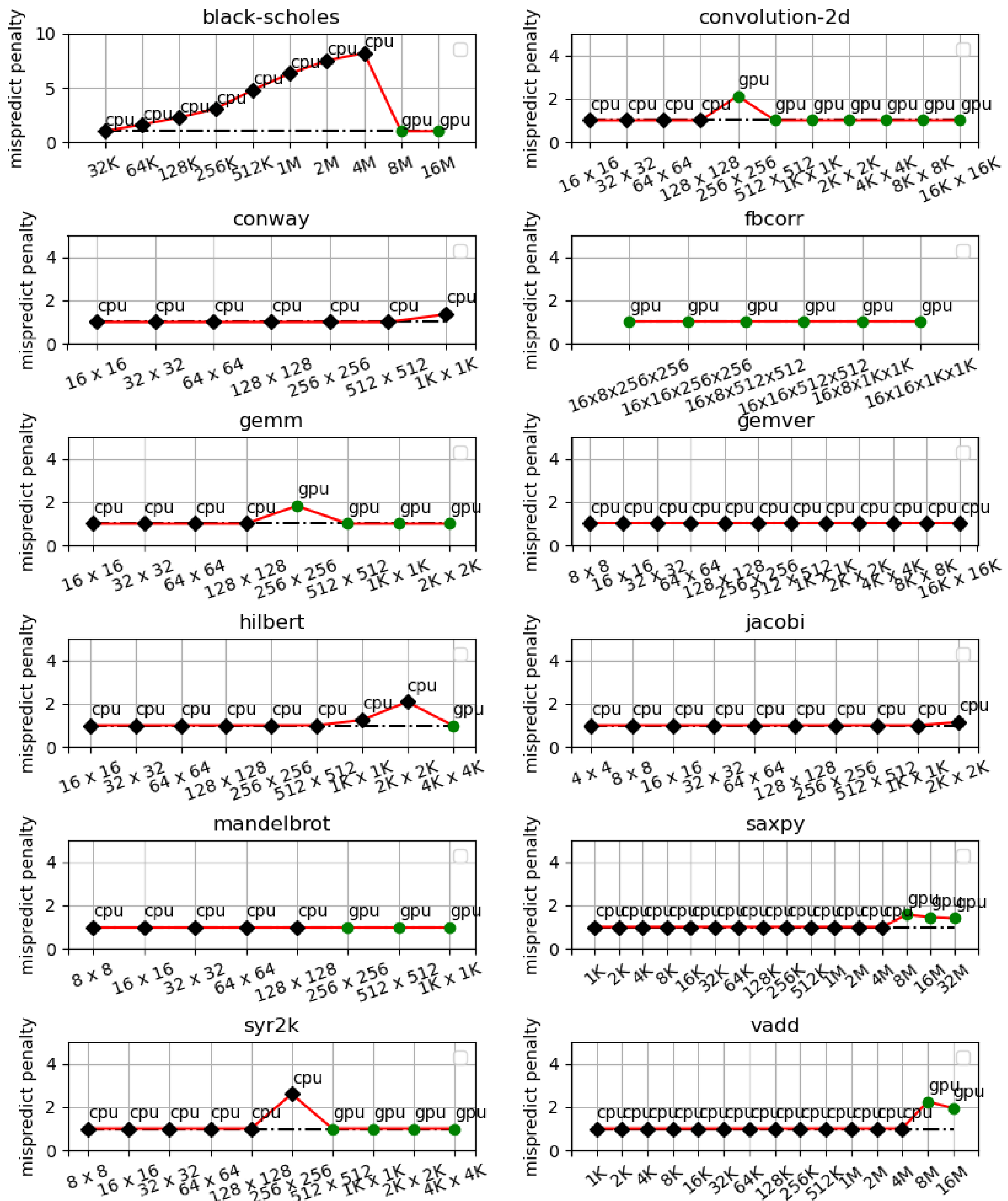



Figure B.3: ALPyNA Cost Model misprediction penalties for 12 loop-intensive benchmarks with varying domain sizes on platform *T3*. Misprediction slowdown is the ratio of predicted device runtime and faster device runtime, so optimal is always 1.0.

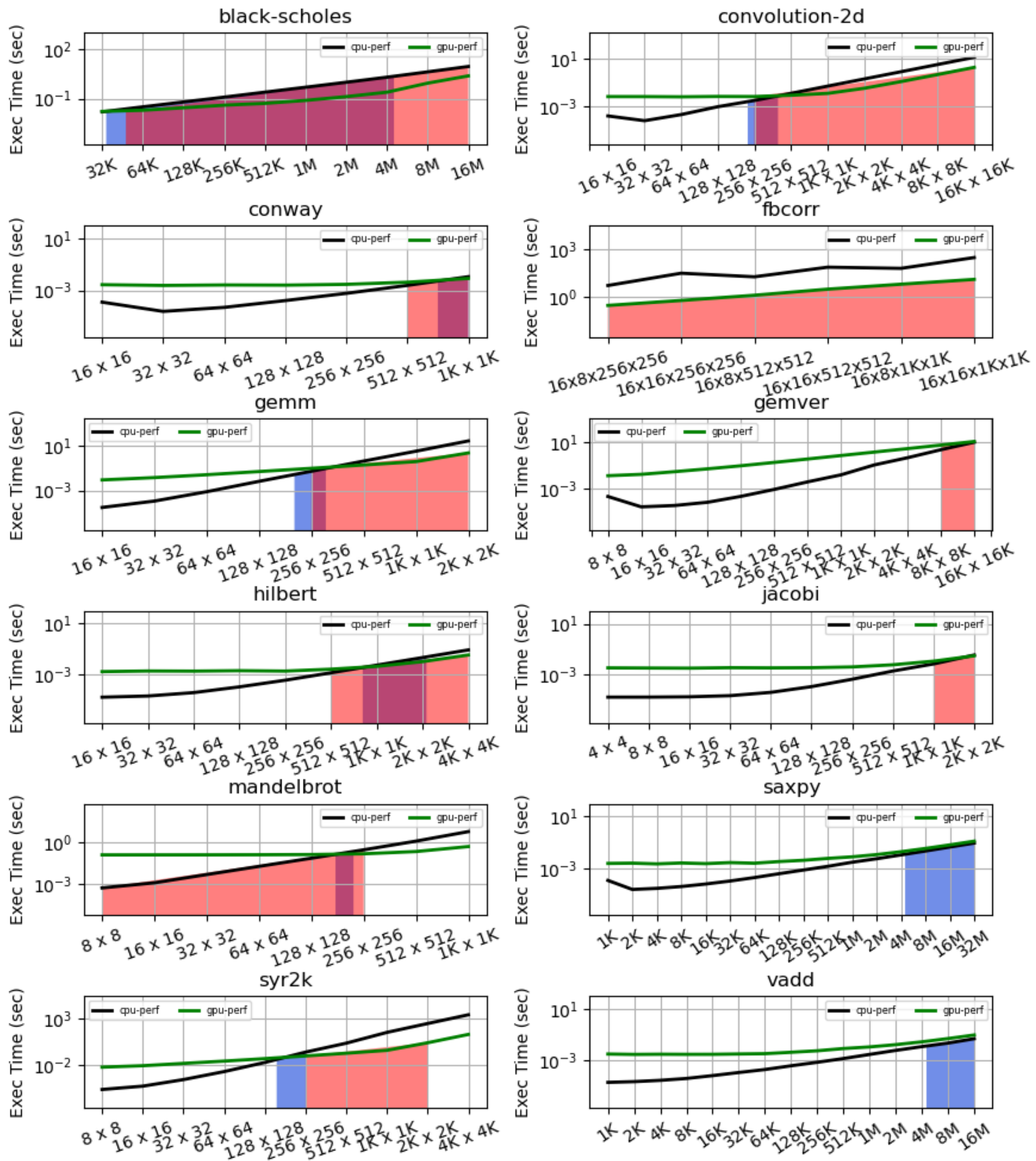



Figure B.4: ALPyNA Cost Model misprediction ranges (shaded orange) for 12 loop-intensive benchmarks with varying domain sizes on platform *T3*. ACM's domain crossover point is interpolated from the measured values. Misprediction range of SVM model is also shown (shaded red). The domain ranges at which both models mispredict the performant device is shaded purple.

# Bibliography

[1] An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw. 28*, 2 (June 2002), 135–151.

[2] ADFINIS SYGROUP AG. Python runtime pure function. https://pypi.org/project/pure-func, 2020. Accessed: 2020-11-10.

[3] ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. Conversion of control dependence to data dependence. In *Proc. POPL* (1983), pp. 177–189.

[4] ALPERN, B., ATTANASIO, C. R., BARTON, J. J., BURKE, M. G., CHENG, P., CHOI, J. ., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M. F., NGO, T., RUSSELL, J. R., SARKAR, V., SERRANO, M. J., SHEPHERD, J. C., SMITH, S. E., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. The Jalapeño virtual machine. *IBM Systems Journal 39*, 1 (2000), 211–238.

[5] AMARÍS, M., DE CAMARGO, R. Y., DYAB, M., GOLDMAN, A., AND TRYSTRAM, D. A comparison of GPU execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)* (2016), pp. 326–333.

[6] AMD. HSA Platform System Architecture Specification. http://www.hsafoundation.com/?ddownload=5702, 2018. Accessed: 2020-03-25.

[7] AMD. AMD Radeon RX570 technical specifications. https://www.amd.com/en/products/graphics/radeon-rx-570, 2020. Accessed: 2020-02-27.

[8] AMD. AMD ROCm Documentation. https://rocm-documentation.readthedocs.io/en/latest/index.html, 2020. Accessed: 2020-04-04.

[9] ARDALANI, N., LESTOURGEON, C., SANKARALINGAM, K., AND ZHU, X. Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In *Proceedings of the 48th International Symposium on Microarchitecture*

(New York, NY, USA, 2015), MICRO-48, Association for Computing Machinery, p. 725–737.

[10] ARDÖ, H., BOLZ, C. F., AND FIJAŁKOWSKI, M. Loop-aware optimizations in PyPy's tracing JIT. In *Proceedings of the 8th Symposium on Dynamic Languages* (New York, NY, USA, 2012), DLS '12, Association for Computing Machinery, p. 63–72.

[11] ARM DEVELOPER. Introducing NEON for Armv8-A. `https://developer.arm.com/architectures/instruction-sets/simd-isas`, 2020. Accessed: 2020-02-25.

[12] ARMIH, K., MICHAELSON, G., AND TRINDER, P. Cache size in a cost model for heterogeneous skeletons. In *Proceedings of the Fifth International Workshop on High-Level Parallel Programming and Applications* (New York, NY, USA, 2011), HLPP '11, Association for Computing Machinery, p. 3–10.

[13] ARMIH, K. A., ET AL. *Toward optimised skeletons for heterogeneous parallel architecture with performance cost model*. PhD thesis, Heriot-Watt University, 2013.

[14] ASCHER, D. Dynamic languages: ready for the next challenges, by design. *white paper, ActiveState* (2004).

[15] ASHOURI, A. H., KILLIAN, W., CAVAZOS, J., PALERMO, G., AND SILVANO, C. A survey on compiler autotuning using machine learning. *ACM Comput. Surv. 51*, 5 (Sept. 2018).

[16] AYCOCK, J. A brief history of Just-in-Time. *ACM Comput. Surv. 35*, 2 (June 2003), 97–113.

[17] BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. Compiler transformations for high-performance computing. *ACM Comput. Surv. 26*, 4 (Dec. 1994), 345–420.

[18] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2000), PLDI '00, Association for Computing Machinery, pp. 1–12.

[19] BALDINI, I., FINK, S. J., AND ALTMAN, E. Predicting GPU performance from CPU runs using machine learning. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing* (2014), pp. 254–261.

[20] BANERJEE, U. *Dependence analysis for supercomputing*, vol. 60. Springer Science & Business Media, 2013.

[21] BASTOUL, C. Code generation in the polyhedral model is easier than you think. In *IEEE International Conference on Parallel Architecture and Compilation Techniques* (Juan-les-Pins, France, September 2004), pp. 7–16.

[22] BEAZLEY, D. Understanding the Python GIL. In *Proc. PyCON* (2010). `https://www.dabeaz.com/python/UnderstandingGIL.pdf`.

[23] BEHNEL, S., BRADSHAW, R., CITRO, C., DALCIN, L., SELJEBOTN, D. S., AND SMITH, K. Cython: The best of both worlds. *Computing in Science Engineering 13*, 2 (2011), 31–39.

[24] BELIKOV, E., LOIDL, H.-W., MICHAELSON, G., AND TRINDER, P. Architecture-aware cost modelling for parallel performance portability. In *Software Engineering 2012. Workshopband* (Bonn, 2012), S. Jähnichen, B. Rumpe, and H. Schlingloff, Eds., Gesellschaft für Informatik e.V., pp. 105–120.

[25] BISSYANDÉ, T. F., THUNG, F., LO, D., JIANG, L., AND RÉVEILLÈRE, L. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th Annual Computer Software and Applications Conference* (2013), pp. 303–312.

[26] BLACK, FISCHER. The pricing of commodity contracts. *Journal of financial economics 3*, 1-2 (1976), 167–179.

[27] BOLZ, C. F., CUNI, A., FIJAŁKOWSKI, M., AND RIGO, A. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (New York, NY, USA, 2009), ICOOOLPS '09, Association for Computing Machinery, p. 18–25.

[28] BROWN, N. ePython: An implementation of python for the many-core Epiphany Co-processor. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)* (Nov 2016), pp. 59–66.

[29] CAAMAÑO, M., MANUEL, J., SELVA, M., CLAUSS, P., BALOIAN, A., AND WOLFF, W. Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience 29*, 15 (2017), e4192.

[30] CASS, S., AND BULUSU, P. IEEE Spectrum top programming languages survey. `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019`, 2020. Accessed: 2020-02-12.

[31] CATANZARO, B., GARLAND, M., AND KEUTZER, K. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2011), PPoPP '11, Association for Computing Machinery, p. 47–56.

[32] CATANZARO, B., KAMIL, S., LEE, Y., ASANOVIC, K., DEMMEL, J., KEUTZER, K., SHALF, J., YELICK, K., AND FOX, A. SEJITS: Getting productivity and performance with selective embedded JIT specialization. *Programming Models for Emerging Architectures 1*, 1 (2009), 1–9.

[33] CHANG, C.-C., AND LIN, C.-J. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol. 2*, 3 (May 2011).

[34] CHIKIN, A., AMARAL, J. N., ALI, K., AND TIOTTO, E. Toward an analytical performance model to select between GPU and CPU execution. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (May 2019), pp. 353–362.

[35] CHIKIN, A., LLOYD, T., AMARAL, J., AND TIOTTO, E. Compiler for restructuring code using interation-point algebraic difference analysis. *Patent Reference: P201706298US01, Filled on March 12*, 2018 (2018), 79.

[36] CLARKSON, J., FUMERO, J., PAPADIMITRIOU, M., ZAKKAK, F. S., XEKALAKI, M., KOTSELIDIS, C., AND LUJÁN, M. Exploiting high-performance heterogeneous hardware for Java programs using Graal. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes* (New York, NY, USA, 2018), ManLang '18, Association for Computing Machinery.

[37] DAGA, M., AJI, A. M., AND FENG, W. On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing. In *2011 Symposium on Application Accelerators in High-Performance Computing* (July 2011), pp. 141–149.

[38] DENNARD, R. H., GAENSSLEN, F. H., RIDEOUT, V. L., BASSOUS, E., AND LEBLANC, A. R. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits 9*, 5 (1974), 256–268.

[39] DUBACH, C., CHENG, P., RABBAH, R., BACON, D. F., AND FINK, S. J. Compiling a high-level language for GPUs: (via language support for architectures and compilers). *SIGPLAN Not. 47*, 6 (June 2012), 1–12.

[40] DUBOSCQ, G., WÜRTHINGER, T., STADLER, L., WIMMER, C., SIMON, D., AND MÖSSENBÖCK, H. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines*

*and Intermediate Languages* (New York, NY, USA, 2013), VMIL ’13, Association for Computing Machinery, p. 1–10.

[41] EECKHOUT, L. Is Moore’s law slowing down? What’s next? *IEEE Micro 37*, 04 (jul 2017), 4–5.

[42] EKMEKCI, B., MCANANY, C. E., AND MURA, C. An introduction to programming for bioscientists: a Python-based primer. *PLoS Computational Biology 12*, 6 (2016), e1004867.

[43] FANG, J., CHEN, H., AND MAO, J. Understanding data partition for applications on cpu-gpu integrated processors. In *International Conference on Mobile Ad-Hoc and Sensor Networks* (2017), Springer, pp. 426–434.

[44] FLYNN, M. J. Very high-speed computing systems. *Proceedings of the IEEE 54*, 12 (Dec 1966), 1901–1909.

[45] FUMERO, J., AND KOTSELIDIS, C. Using compiler snippets to exploit parallelism on heterogeneous hardware: A Java reduction case study. In *Proc. Virtual Machines and Intermediate Languages* (2018), pp. 16–25.

[46] FUMERO, J., PAPADIMITRIOU, M., ZAKKAK, F. S., XEKALAKI, M., CLARKSON, J., AND KOTSELIDIS, C. Dynamic application reconfiguration on heterogeneous hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2019), VEE 2019, Association for Computing Machinery, p. 165–178.

[47] FUMERO, J., STEUWER, M., STADLER, L., AND DUBACH, C. Just-in-time GPU compilation for interpreted languages with partial evaluation. In *Proc. VEE* (2017), pp. 60–73.

[48] FUMERO, J. J., REMMELG, T., STEUWER, M., AND DUBACH, C. Runtime code generation and data management for heterogeneous computing in Java. In *Proceedings of the Principles and Practices of Programming on The Java Platform* (New York, NY, USA, 2015), PPPJ ’15, Association for Computing Machinery, p. 16–26.

[49] FUMERO, J. J., STEUWER, M., AND DUBACH, C. A composable array function interface for heterogeneous computing in Java. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (New York, NY, USA, 2014), ARRAY’14, Association for Computing Machinery, p. 44–49.

[50] FURSIN, G., KASHNIKOV, Y., MEMON, A. W., CHAMSKI, Z., TEMAM, O., NAMOLARU, M., MENDELSON, B., ZAKS, A., COURTOIS, E., BODIN, F., BARNARD, P., ASHTON, E., BONILLA, E., THOMSON, J., WILLIAMS, C., AND O'BOYLE, M. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming 39* (06 2011), 296–327.

[51] GARDNER, MARTIN. Mathematical games: The fantastic combinations of John Conway's new solitaire game life. *Scientific American 223*, 4 (1970), 120–123.

[52] GOFF, G., KENNEDY, K., AND TSENG, C.-W. Practical dependence testing. *SIGPLAN Not. 26*, 6 (May 1991), 15–29.

[53] GROSSER, T., ZHENG, H., ALOOR, R., SIMBÜRGER, A., GRÖSSLINGER, A., AND POUCHET, L.-N. Polly—polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)* (2011), vol. 2011, p. 1.

[54] HALAWA, H., ABDELHAFEZ, H. A., BOKTOR, A., AND RIPEANU, M. Nvidia jetson platform characterization. In *Euro-Par 2017: Parallel Processing* (Cha, 2017), F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds., Springer International Publishing, pp. 92–10.

[55] HENRIKSEN, T., SERUP, N. G. W., ELSMAN, M., HENGLEIN, F., AND OANCEA, C. E. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proc. PLDI* (2017), pp. 556–571.

[56] HERHUT, S., HUDSON, R. L., SHPEISMAN, T., AND SREERAM, J. River Trail: A path to parallelism in JavaScript. In *Proc. OOPSLA* (2013), pp. 729–744.

[57] HONG, S., AND KIM, H. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News 37*, 3 (June 2009), 152–163.

[58] INTEL. Intel OneAPI. `https://software.intel.com/en-us/oneapi`, 2020. Accessed: 2020-03-27.

[59] ISHIZAKI, K., HAYASHI, A., KOBLENTS, G., AND SARKAR, V. Compiling and optimizing Java 8 programs for GPU execution. In *2015 International Conference on Parallel Architecture and Compilation (PACT)* (Oct 2015), pp. 419–431.

[60] JACOB, D., AND SINGER, J. ALPyNA: Acceleration of Loops in Python for Novel Architectures. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (2019), Association for Computing Machinery, pp. 25–34.

[61] JACOB, D., TRINDER, P., AND SINGER, J. Python programmers have GPUs too: Automatic Python loop parallelization with staged dependence analysis. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (2019), pp. 42–54.

[62] JACOB, D., TRINDER, P., AND SINGER, J. Pricing Python Parallelism: A Dynamic Language Cost Model for Heterogeneous Platforms. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (New York, NY, USA, 2020), DLS 2020, Association for Computing Machinery, p. 29–42.

[63] JEON, D., GARCIA, S., LOUIE, C., AND TAYLOR, M. B. Kismet: Parallel speedup estimates for serial programs. *SIGPLAN Not. 46*, 10 (Oct. 2011), 519–536.

[64] JIBAJA, I., JENSEN, P., HU, N., HAGHIGHAT, M. R., MCCUTCHAN, J., GOHMAN, D., BLACKBURN, S. M., AND MCKINLEY, K. S. Vector parallelism in JavaScript: Language and compiler support for SIMD. In *2015 International Conference on Parallel Architecture and Compilation (PACT)* (2015), pp. 407–418.

[65] JONES, R., HOSKING, A., AND MOSS, E. *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2016.

[66] JORDAN, H., PELLEGRINI, S., THOMAN, P., KOFLER, K., AND FAHRINGER, T. INSPIRE: The insieme parallel intermediate representation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (2013), pp. 7–17.

[67] JUNEAU, J., BAKER, J., WIERZBICKI, F., MUOZ, L. S., NG, V., NG, A., AND BAKER, D. L. *The definitive guide to Jython: Python for the Java platform*. Apress, 2010.

[68] KAELI, D. R., MISTRY, P., SCHAA, D., AND ZHANG, D. P. *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann, 2015.

[69] KAMIL, S., COETZEE, D., AND FOX, A. Bringing parallel performance to Python with domain-specific selective embedded just-in-time specialization. In *Python for Scientific Computing Conference (SciPy)* (2011).

[70] KENNEDY, K., AND ALLEN, J. R. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[71] KENNEDY, K., AND MCKINLEY, K. S. Optimizing for parallelism and data locality. In *Proceedings of the 6th International Conference on Supercomputing* (New York, NY, USA, 1992), Association for Computing Machinery, pp. 323–334.

[72] KHRONOS. Standard portable intermediate representation. https://www.khronos.org/spir, 2020. Accessed: 2020-03-27.

[73] KIM, M., KUMAR, P., KIM, H., AND BRETT, B. Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (2012), pp. 1318–1329.

[74] KLÖCKNER, A. Loo. py: transformation-based code generation for GPUs and CPUs. In *Proc. ARRAY* (2014), pp. 82–87.

[75] KLÖCKNER, A., PINTO, N., LEE, Y., CATANZARO, B., IVANOV, P., AND FASIH, A. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Comput. 38*, 3 (Mar. 2012), 157–174.

[76] KONG, M., AND POUCHET, L.-N. A performance vocabulary for affine loop transformations. *arXiv preprint arXiv:1811.06043* (2018).

[77] KUCHLING, A. Functional programming HOWTO. https://docs.python.org/3/howto/functional.html, 2020. Accessed: 2020-02-18.

[78] LAM, S. K., PITROU, A., AND SEIBERT, S. Numba: A LLVM-based Python JIT compiler. In *Proc. Second Workshop on the LLVM Compiler Infrastructure in HPC* (2015), p. 7.

[79] LATTNER, C., AND ADVE, V. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (March 2004), pp. 75–86.

[80] LATTNER, C., PIENAAR, J., AMINI, M., BONDHUGULA, U., RIDDLE, R., COHEN, A., SHPEISMAN, T., DAVIS, A., VASILACHE, N., AND ZINENKO, O. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[81] LENGAUER, T., AND TARJAN, R. E. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1 (Jan. 1979), 121–141.

[82] LEUNG, A., LHOTÁK, O., AND LASHARI, G. Automatic parallelization for graphics processing units. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java* (New York, NY, USA, 2009), PPPJ '09, Association for Computing Machinery, p. 91–100.

[83] LIAO, C., AND CHAPMAN, B. Invited paper: A compile-time cost model for OpenMP. In *2007 IEEE International Parallel and Distributed Processing Symposium* (2007), pp. 1–8.

[84] LIN, J. W.-B. Why Python is the next wave in earth sciences computing. *Bulletin of the American Meteorological Society 93*, 12 (2012), 1823–1824.

[85] LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2 (2008), 39–55.

[86] LOMONT,CHRIS. Introduction to Intel Advanced Vector Extensions. `https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions`, 2020. Accessed: 2020-02-25.

[87] LOZINSKI, L. The Uber engineering tech stack, part I: The foundation. `https://eng.uber.com/tech-stack-part-one`, 2016. Accessed: 2020-02-12.

[88] LUO, G., CHEN, T., AND YU, H. Toward a progress indicator for program compilation. *Software: Practice and Experience 37*, 9 (2007), 909–933.

[89] LUTZ, T., AND GROVER, V. LambdaJIT: A dynamic compiler for heterogeneous optimizations of STL algorithms. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing* (New York, NY, USA, 2014), FHPC '14, Association for Computing Machinery, pp. 99–108.

[90] MAHAPATRA, A., AND PATRA, P. Support vector machine for frequently executed method prediction. In *2018 International Conference on Information Technology (ICIT)* (Los Alamitos, CA, USA, dec 2018), IEEE Computer Society, pp. 193–198.

[91] MAIER, P., MORTON, J. M., AND TRINDER, P. JIT costing adaptive skeletons for performance portability. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing* (New York, NY, USA, 2016), FHPC 2016, Association for Computing Machinery, pp. 23–30.

[92] MASUHARA, H., AND NISHIGUCHI, Y. A data-parallel extension to Ruby for GPGPU: Toward a framework for implementing domain-specific optimizations. In *Proceedings of the 9th ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution* (New York, NY, USA, 2012), RAM–SE '12, Association for Computing Machinery, pp. 3–6.

[93] MCKINNEY, W. Pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing 14* (2011).

[94] MEIER, R., AND RIGO, A. A way forward in parallelising dynamic languages. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE* (New York, NY, USA, 2014), ICOOOLPS '14, Association for Computing Machinery.

[95] MITTAL, S., AND VETTER, J. S. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv. 47*, 4 (July 2015).

[96] MODZELEWSKI, K., AND WACHTLER, M. Pyston, an open-source Python implementation using JIT techniques. `https://github.com/dropbox/pyston`, 2014.

[97] MOORE, G. E., ET AL. Cramming more components onto integrated circuits, 1965.

[98] MORTON, J. M., MAIER, P., AND TRINDER, P. JIT-based cost analysis for dynamic program transformations. *Electronic Notes in Theoretical Computer Science 330* (2016), 5 – 25. RAC 2016 - Resource Aware Computin.

[99] NUMPY. Numpy - indexing. `https://numpy.org/doc/1.18/reference/arrays.indexing.html`, 2020. Accessed: 2020-05-15.

[100] NVIDIA. NVIDIA's next generation cuda compute architecture. *NVidia, Santa Clara, Calif, USA* (2009).

[101] NVIDIA. NVIDIA GTX1060 technical specifications. `https://www.nvidia.com/en-gb/geforce/products/10series/geforce-gtx-1060`, 2020. Accessed: 2020-02-27.

[102] NVIDIA. NVIDIA Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2020. Accessed: 2020-02-27.

[103] NVIDIA. NVVM IR Specification. `https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html`, 2020. Accessed: 2020-02-18.

[104] PADUA, D. A., AND WOLFE, M. J. Advanced compiler optimizations for supercomputers. *Commun. ACM* (1986).

[105] PALECZNY, M., VICK, C., AND CLICK, C. The Java Hotspot server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium* (2001), vol. 1.

[106] PAUL FEAUTRIER. *The Data Parallel Programming Model*, vol. 1132 of *LNCS*. Springer-Verlag, 1996, ch. Automatic Parallelization in the Polytope Model.

[107] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., ET AL. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research 12*, Oct (2011), 2825–2830.

[108] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND ÉDOUARD DUCHESNAY. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12*, 85 (2011), 2825–2830.

[109] PETERSEN, P. M., AND PADUA, D. A. Static and dynamic evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems 7*, 11 (Nov 1996), 1121–1132.

[110] PLANGGER, R., AND KRALL, A. Vectorization in PyPy's tracing Just-In-Time compiler. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems* (New York, NY, USA, 2016), SCOPES '16, Association for Computing Machinery, p. 67–76.

[111] POP, S., COHEN, A., BASTOUL, C., GIRBAL, S., SILBER, G.-A., AND VASILACHE, N. Graphite: Polyhedral analyses and optimizations for GCC. In *Proceedings of the 2006 GCC Developers Summit* (2006), Citeseer, p. 2006.

[112] POPE-CARTER, F., HARRIS, C., SPARROW, T., AND GAFFNEY, C. Archaeopy: Developing open source software for archaeological geophysics. In *Proc. 43rd Computer Applications and Quantitative Methods in Archaeology* (2015). `https://sites.caa-international.org/caa2015/wp-content/uploads/sites/14/2015/04/Book-of-Abstracts_CAA20151.pdf`.

[113] POUCHET, L.-N., BONDHUGULA, U., ET AL. The polybench benchmarks, 2019. `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench`.

[114] PUGH, W. Uniform techniques for loop optimization. In *Proceedings of the 5th International Conference on Supercomputing* (New York, NY, USA, 1991), ICS '91, Association for Computing Machinery, p. 341–352.

[115] QUNAIBIT, M., BRUNTHALER, S., NA, Y., VOLCKAERT, S., AND FRANZ, M. Accelerating dynamically-typed languages on heterogeneous platforms using guards optimization. In *Proc. ECOOP* (2018), pp. 16:1–16:29.

[116] RAPOPORT, R., MOYLES, B., CISTARO, J., AND BERTRAM, C. Python at Netflix. `https://netflixtechblog.com/python-at-netflix-86b6028b3b3e`, 2016. Accessed: 2020-02-12.

[117] REYES, R., AND LOMÜLLER, V. SYCL: Single-source C++ accelerator programming. In *PARCO* (2015), pp. 673–682.

[118] RICHARDS, A. Update on the SYCL for OpenCL open standard to enable C++ meta programming on top of opencl. In *Proceedings of the 3rd International Workshop on OpenCL* (New York, NY, USA, 2015), IWOCL '15, Association for Computing Machinery.

[119] RIGO, ARMIN AND PEDRONI, SAMUELE. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 2006), OOPSLA '06, Association for Computing Machinery, p. 944–953.

[120] ROBITAILLE, T. P., TOLLERUD, E. J., GREENFIELD, P., DROETTBOOM, M., BRAY, E., ALDCROFT, T., DAVIS, M., GINSBURG, A., PRICE-WHELAN, A. M., KERZENDORF, W. E., ET AL. Astropy: A community Python package for astronomy. *Astronomy & Astrophysics 558* (2013), A33.

[121] ROSSBACH, C. J., YU, Y., CURREY, J., MARTIN, J.-P., AND FETTERLY, D. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, pp. 49–68.

[122] ROSSUM, G. Extending and embedding the python interpreter. Tech. rep., NLD, 1995.

[123] ROSSUM, G. V. CPython: Python Reference Implementation. `https://github.com/python/cpython`, 2020. Accessed: 2020-10-24.

[124] RUBINSTEYN, A., HIELSCHER, E., WEINMAN, N., AND SHASHA, D. Parakeet: A Just-in-Time parallel accelerator for Python. In *Proc. 4th USENIX Conference on Hot Topics in Parallelism* (2012), pp. 14–14.

[125] RULE, A., TABARD, A., AND HOLLAN, J. D. Exploration and explanation in computational notebooks. In *Proc. CHI* (2018), pp. 32:1–32:12.

[126] SAMADI, M., HORMATI, A., LEE, J., AND MAHLKE, S. Paragon: Collaborative speculative loop execution on GPU and CPU. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units* (New York, NY, USA, 2012), GPGPU-5, Association for Computing Machinery, p. 64–73.

[127] SHEFFIELD, D., ANDERSON, M., AND KEUTZER, K. Automatic generation of application-specific accelerators for FPGAs from Python loop nests. In *22nd International Conference on Field Programmable Logic and Applications (FPL)* (Aug 2012), pp. 567–570.

[128] SHEFFIELD, D. B. *Three Fingered Jack: Productively Addressing Platform Diversity*. PhD thesis, UC Berkeley, 2013.

[129] SILVA, H. C. D., PISANI, F., AND BORIN, E. A comparative study of SYCL, OpenCL, and OpenMP. In *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)* (2016), pp. 61–66.

[130] SIM, J., DASGUPTA, A., KIM, H., AND VUDUC, R. A performance analysis framework for identifying potential benefits in GPGPU applications. *SIGPLAN Not. 47*, 8 (Feb. 2012), 11–22.

[131] SMITH, B. C. *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[132] SMITH, J., AND NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[133] SMITH, R. The NVIDIA GeForce GTX 1080 & GTX 1070 Founders Editions Review: Kicking Off the FinFET Generation, 2016. `https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/4`.

[134] SPRINGER, M., WAULIGMANN, P., AND MASUHARA, H. Modular array-based gpu computing in a dynamically-typed language. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (New York, NY, USA, 2017), ARRAY 2017, Association for Computing Machinery, pp. 48–55.

[135] STADLER, L., WELC, A., HUMER, C., AND JORDAN, M. Optimizing R language execution via aggressive speculation. In *Proceedings of the 12th Symposium on Dynamic Languages* (New York, NY, USA, 2016), DLS 2016, Association for Computing Machinery, p. 84–95.

[136] STEUWER, M., FENSCH, C., LINDLEY, S., AND DUBACH, C. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. *SIGPLAN Not. 50*, 9 (Aug. 2015), 205–217.

[137] STEUWER, M., REMMELG, T., AND DUBACH, C. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Feb 2017), pp. 74–85.

[138] SUJEETH, A. K., BROWN, K. J., LEE, H., ROMPF, T., CHAFI, H., ODERSKY, M., AND OLUKOTUN, K. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst. 13*, 4s (Apr. 2014).

[139] SUKUMARAN, J., AND HOLDER, M. T. DendroPy: a Python library for phylogenetic computing. *Bioinformatics 26*, 12 (2010), 1569–1571.

[140] SUKUMARAN-RAJAM, A., AND CLAUSS, P. The polyhedral model of nonlinear loops. *ACM Trans. Archit. Code Optim. 12*, 4 (Dec. 2015).

[141] SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal 30*, 3 (2005), 202–210.

[142] SVENSSON SAND, K., AND ELIASSON, T. A comparison of functional and object-oriented programming paradigms in JavaScript, 2017.

[143] TIM PETERS. PEP 20 – The Zen of Python, 2004. https://www.python.org/dev/peps/pep-0020/.

[144] TIOBE. Tiobe index. http://www.tiobe.com/tiobe-index/python/, 2020. Accessed: 2020-02-12.

[145] TRINDER, P. W., COLE, M. I., HAMMOND, K., LOIDL, H.-W., AND MICHAELSON, G. J. Resource analyses for parallel and distributed coordination. *Concurrency and Computation: Practice and Experience 25*, 3 (2013), 309–348.

[146] UPADRASTA, R., AND COHEN, A. Sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2013), POPL '13, Association for Computing Machinery, p. 483–496.

[147] VAN DER MEER, G. How we use Python at Spotify. https://labs.spotify.com/2013/03/20/how-we-use-python-at-spotify, 2013. Accessed: 2020-02-21.

[148] VAN DER WALT, S., COLBERT, S. C., AND VAROQUAUX, G. The NumPy array: A structure for efficient numerical computation. *Computing in Science Engineering 13*, 2 (March 2011), 22–30.

[149] VANDIERENDONCK, H., AND MENS, T. Techniques and tools for parallelizing software. *IEEE Software 29*, 2 (March 2012), 22–25.

[150] VAPNIK, V. N. An overview of statistical learning theory. *IEEE Transactions on Neural Networks 10*, 5 (1999), 988–999.

[151] VARDI, M. Y. Moore's law and the sand-heap paradox. *Communications of the ACM 57*, 5 (2014), 5–5.

[152] VERDOOLAEGE, S., AND GROSSER, T. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France* (2012), vol. 141.

[153] VIRTANEN, P., GOMMERS, R., OLIPHANT, T. E., HABERLAND, M., REDDY, T., COURNAPEAU, D., BUROVSKI, E., PETERSON, P., WECKESSER, W., BRIGHT, J., ET AL. Scipy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* (2020), 1–12.

[154] WANG, J., RUBIN, N., AND YALAMANCHILI, S. ParallelJS: An execution framework for JavaScript on heterogeneous systems. In *Proceedings of Workshop on General Purpose Processing Using GPUs* (New York, NY, USA, 2014), GPGPU-7, Association for Computing Machinery, p. 72–80.

[155] WANG, Z., GREWE, D., AND O'BOYLE, M. F. P. Automatic and portable mapping of data parallel programs to OpenCL for GPU-based heterogeneous systems. *ACM Trans. Archit. Code Optim. 11*, 4 (Dec. 2014).

[156] WANG, Z., POWELL, D., FRANKE, B., AND O'BOYLE, M. Exploitation of GPUs for the parallelisation of probably parallel legacy code. In *Compiler Construction* (Berlin, Heidelberg, 2014), A. Cohen, Ed., Springer Berlin Heidelberg, pp. 154–173.

[157] WILT, N. *The CUDA handbook: A comprehensive guide to GPU programming*. Pearson Education, 2013.

[158] WOLFE, MICHAEL AND BANERJEE, UTPAL. Data dependence and its application to parallel processing. *International Journal of Parallel Programming 16*, 2 (Apr 1987), 137–178.

[159] WONG, M., RICHARDS, A., ROVATSOU, M., AND REYES, R. Khronos's OpenCL SYCL to support heterogeneous devices for C++, 2016.

[160] WU, G., GREATHOUSE, J. L., LYASHEVSKY, A., JAYASENA, N., AND CHIOU, D. GPGPU performance and power estimation using machine learning. In *2015 IEEE*

*21st International Symposium on High Performance Computer Architecture (HPCA)* (2015), pp. 564–576.

[161] WÜRTHINGER, T., WIMMER, C., WÖUNDEFINED, A., STADLER, L., DUBOSCQ, G., HUMER, C., RICHARDS, G., SIMON, D., AND WOLCZKO, M. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (New York, NY, USA, 2013), Onward! 2013, Association for Computing Machinery, pp. 187–204.

[162] ZAHRAN, M. Heterogeneous Computing: Here to Stay. *Commun. ACM 60*, 3 (Feb. 2017), 42–45.

[163] ZAREI ZEFREH, E., LOTFI, S., MOHAMMAD KHANLI, L., AND KARIMPOUR, J. 3-D data partitioning for 3-level perfectly nested loops on heterogeneous distributed systems. *Concurrency and Computation: Practice and Experience 29*, 5 (2017), e3976. e3976 cpe.3976.

[164] ZHANG, W. *Efficient hosted interpreter for dynamic languages*. PhD thesis, UC Irvine, 2015.