Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

oclude and OCLMan

tools to profile and predict the dynamic behavior of standalone OpenCL kernels based on compiling and machine learning techniques

---

Διπλωματικη Εργασια

του

ΝΙΑΡΧΟΥ ΣΩΤΗΡΙΟΥ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής

Αθήνα, Ιούλιος 2020

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

oclude and OCLMan

tools to profile and predict the dynamic behavior of standalone OpenCL kernels based on compiling and machine learning techniques

---

Διπλωματικη Εργασια

του

ΝΙΑΡΧΟΥ ΣΩΤΗΡΙΟΥ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29η Ιουλίου 2020.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Νικόλαος Παπασπύρου
Καθηγητής	Επίκουρος Καθηγητής	Καθηγητής

Αθήνα, Ιούλιος 2020

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Πτυχιακής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης. Αναλαμβάνω την προσωπική και ατομική ευθύνη ότι σε περίπτωση αποτυχίας στην υλοποίηση των ανωτέρω δηλωθέντων στοιχείων, είμαι υπόλογος έναντι λογοκλοπής, γεγονός που σημαίνει αποτυχία στην Πτυχιακή μου Εργασία και κατά συνέπεια αποτυχία απόκτησης του Τίτλου Σπουδών, πέραν των λοιπών συνεπειών του νόμου περί πνευματικών δικαιωμάτων. Δηλώνω, συνεπώς, ότι αυτή η Πτυχιακή Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

(Υπογραφή)

.....
Σωτήριος Νιάρχος

29 Ιουλίου 2020

Περίληψη

Τα τελευταία χρόνια, ο ετερογενής υπολογισμός (heterogeneous computing), δηλαδή η εκτέλεση κώδικα σε μονάδες επεξεργασίας που αντιστοιχούν σε μια ποικιλία διαφορετικών αρχιτεκτονικών (CPUs, GPUs, FPGAs κ.λπ.), αρχίζει να παίζει έναν ολοένα και σημαντικότερο ρόλο στους τομείς του μαζικού και παράλληλου υπολογισμού, καθώς και στο υπολογιστικό νέφος. Ένα από τα κύρια ερωτήματα που θέτει αυτή η προσέγγιση είναι ποια στρατηγική πρέπει να ακολουθηθεί για να κατανέμονται οι υπολογισμοί στις διάφορες διαθέσιμες μονάδες επεξεργασίας προκειμένου να μεγιστοποιηθεί η απόδοση.

Στην προσπάθειά μας να διερευνήσουμε πιθανές απαντήσεις σε αυτό το ερώτημα, παρουσιάζουμε το oclude, ένα εργαλείο που χρησιμοποιεί τεχνικές από τον χώρο των μεταγλωττιστών και επεξεργάζεται τον πηγαίο κώδικα (source code instrumentation) προγραμμάτων γραμμένων σε OpenCL, μία γλώσσα για προγραμματισμό σε παράλληλα και/ή ετερογενή περιβάλλοντα. Χρησιμοποιούμε το oclude για να αναλύσουμε έναν αριθμό πυρήνων OpenCL (OpenCL kernels), δηλαδή να αποκτήσουμε τον αριθμό των LLVM εντολών ανά τύπο που εκτελέστηκαν μέσω μιας δυναμικής διαδικασίας μικροανάλυσης (microprofiling), και ως εκ τούτου να αποκτήσουμε ένα σύνολο χαρακτηριστικών ανεξάρτητων από τη γλώσσα προγραμματισμού και το υλικό. Στη συνέχεια χτίζουμε μοντέλα παλινδρόμησης (regression models) για την πρόβλεψη των χρόνων εκτέλεσης σε μια μονάδα CPU, που βασίζονται αποκλειστικά στο μέγεθος των δεδομένων εισόδου.

Υλοποιήθηκαν δύο στάδια παλινδρόμησης. Το πρώτο στάδιο προβλέπει μετρήσεις εντολών συναρτήσει του μεγέθους των δεδομένων εισόδου και είναι διαφορετικό για κάθε πυρήνα (kernel-specific), ενώ το δεύτερο στάδιο χρησιμοποιεί το πρώτο για να προβλέψει τους χρόνους εκτέλεσης σε μια δεδομένη μονάδα επεξεργασίας, χωρίς η πρόβλεψη αυτή να είναι συγκεκριμένη για τον πυρήνα (kernel-agnostic). Αυτά τα δύο στάδια έχουν υλοποιηθεί ξεχωριστά έχοντας ως στόχο την ανεξαρτησία τους, έτσι ώστε μια διαφορετική προσέγγιση να μπορεί εύκολα να χτιστεί με αυτά, π.χ. ανάλυση υπολογιστικών πυρήνων γραμμένων σε γλώσσες πέραν της OpenCL.

Τέλος, δείχνουμε ότι το μοντέλο μας έχει πολύ καλύτερη απόδοση από ό,τι μοντέλα παλινδρόμησης που βασίζονται μόνο σε στατικά χαρακτηριστικά, το οποίο και αποτελεί τη βασική προσέγγιση που χρησιμοποιείται στη σχετική βιβλιογραφία.

Λέξεις Κλειδιά

ετερογενής υπολογισμός, παράλληλος υπολογισμός, υπολογιστικός πυρήνας, OpenCL, LLVM, δυναμική ανάλυση, μικροανάλυση, μηχανική μάθηση, παλινδρόμηση

Abstract

In recent years, heterogeneous computing, i.e. code execution on processing units that correspond to a variety of different architectures (CPUs, GPUs, FPGAs etc.), starts to play an increasingly important role in the fields of massive, parallel and cloud computing. One of the main questions that this approach raises is which strategy should be followed in order to distribute computation kernels across the different available processing units in order to maximize efficiency.

Towards our effort to investigate possible answers to this question, we present oclude, a tool that uses compiler techniques and source code instrumentation to profile applications written in OpenCL, a parallel and heterogeneous computing framework. We use oclude to profile a number of OpenCL kernels, i.e. to acquire the number of LLVM instructions per type that were executed through a dynamic microprofiling process, and hence to obtain a language and hardware independent feature set. We then build regression models to predict execution times on a CPU, based solely on the size of the input data.

Two regression stages have been implemented. The first predicts instruction counts given the size of the input data and is kernel-specific, while the second utilizes the first in order to predict execution times on a given processing unit without being kernel-specific. These two stages have been implemented separately in order to decouple the language and hardware independent model from the rest of the process, so that a different approach can be easily built on top of them, e.g. profiling a non-OpenCL codebase.

Finally, we show that our model performs significantly better than regression models based on static features only, which is the default approach used in related literature.

Keywords

heterogeneous computing, parallel computing, computational kernel, OpenCL, LLVM, dynamic profiling, microprofiling, machine learning, regression

στο Γιώργο
για την τεχνική, επιστημονική και ψυχολογική του βοήθεια

στην Ελένη
για όλα τα παραπάνω και για άλλα τόσα, και ακόμα πιο πολλά

στην οικογένειά μου
Νικήτα, Βασιλική και Παναγιώτη
για όλα, μα όλα τα υπόλοιπα

Ευχαριστίες

Θα ήθελα καταρχάς να ευχαριστήσω τον καθηγητή κ. Νεκτάριο Κοζύρη για την επίβλεψη αυτής της διπλωματικής εργασίας και για την ευκαιρία που μου έδωσε να την εκπονήσω στο Εργαστήριο Συστημάτων Υπολογιστών.

Επίσης, ευχαριστώ ιδιαίτερα την Δρ. Κατερίνα Δόκα και τον Κωνσταντίνο Μπιτσάκο για την πολύτιμη καθοδήγησή τους και την υποδειγματική συνεργασία που είχαμε. Χωρίς την υποστήριξή τους, η εργασία αυτή θα ήταν πολύ διαφορετική, και σίγουρα όχι τόσο σημαντική, πλήρης και ολοκληρωμένη στα μάτια του γράφοντα όσο είναι τώρα.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς μου και τον αδελφό μου για την πνευματική καθοδήγηση και την ηθική συμπαράσταση που μου προσέφεραν και μου προσφέρουν όλα αυτά τα χρόνια, και που είμαι σίγουρος πως θα συνεχίσουν να μου προσφέρουν για πολλά ακόμα.

Αθήνα, Ιούλιος 2020

Σωτήριος Νιάρχος

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Πρόλογος

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων (CSLab) της σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου στην Αθήνα, κατά το ακαδημαϊκό έτος 2019-2020.

Πίνακας εξωφύλλου: Fernand Léger, *La Ville*, 1919

Κεφάλαιο 1

Εισαγωγή

Σε αυτό το εισαγωγικό κεφάλαιο, θα παρουσιάσουμε το ευρύτερο ερευνητικό πλαίσιο στο οποίο εντάσσεται η παρούσα διπλωματική εργασία, αυτό του ετερογενούς υπολογισμού (heterogeneous computing). Στη συνέχεια, θα ορίσουμε το συγκεκριμένο πρόβλημα από αυτό το πεδίο με το οποίο θα καταπιαστούμε, θα παρουσιάσουμε εποπτικά τη δική μας προσέγγιση της λύσης του και, τέλος, θα παρουσιάσουμε την οργάνωση του παρόντος τόμου σε μέρη και κεφάλαια.

1.1 Μία σύντομη ιστορία της υπολογιστικής

Από τους άβακες των αρχαίων Σουμερίων έως και τον σημερινό ψηφιακό υπολογιστή, η ανθρωπότητα επιθυμούσε ανέκαθεν να βρίσκει και να επανεφευρίσκει νέους τρόπους για να *υπολογίζει* ταχύτερα, αποδοτικότερα, με μεγαλύτερη ακρίβεια και αξιοπιστία. Πληθώρα *υπολογιστικών μονάδων* (processing units) έχουν πλέον σχεδιαστεί με στόχο, επί της ουσίας, απλούς ή και ειδικότερους αριθμητικούς υπολογισμούς, όπως οι επεξεργαστές (CPUs), οι κάρτες γραφικών (GPUs), και άλλα παρόμοια συστήματα.

Πλέον, όμως, και κυρίως χάρη στην έλευση του παγκόσμιου ιστού και του διαδικτύου, βρισκόμαστε στην ανατολή μίας νέας εποχής, όπου υπολογιστικές μονάδες που παραδοσιακά θεωρούνταν είτε απλές είτε ειδικού σκοπού θα χρειαστεί να συνεργαστούν, ώστε να επιλύσουν δυσκολότερα ή μεγαλύτερα προβλήματα.

Καλωσήρθατε στην εποχή της ετερογενούς πληροφορικής!

1.2 Ζούμε σε έναν ετερογενή κόσμο

Η ανάγκη για μια κάποια αλλαγή άρχισε να διαφαίνεται στα μέσα της δεκαετίας του 2000, όταν ο περιβόητος Νόμος του Μουρ (Moore's Law) άρχισε να χάνει τη καθολική του ισχύ, λόγω φυσικών περιορισμών στο μέγεθος των transistors, αλλά και λόγω των υπέρογκων ποσών ενέργειας που άρχισαν να απαιτούνται ανά μονάδα επιφανείας των υπολογιστικών μονάδων.

Με τα παραπάνω δεδομένα, η επιστημονική και τεχνολογική κοινότητα άρχισε να στρέφεται προς την κατεύθυνση της *σύνθεσης* των ήδη υπαρχόντων αρχιτεκτονικών σε συστήματα στα οποία θα συνεργάζονται με στόχο την από κοινού επίλυση πιο σύνθετων, πολύπλοκων,

δύσκολων ή απλά μεγαλύτερων προβλημάτων. Τα συστήματα αυτά ονομάζονται **ετερογενή** (heterogeneous systems).

1.3 Εκμεταλλευόμενοι τη ποικιλομορφία

Παρά τις δυνατότητες που φαίνεται να προσφέρουν τα ετερογενή συστήματα, δεν είναι ακόμα τετριμμένες οι απαντήσεις στα δύο παρακάτω ερωτήματα:

1. Δεδομένου ενός (υπο)προβλήματος, ποια είναι η βέλτιστη αρχιτεκτονική για να το εκτελέσει;
2. Δεδομένης μίας αρχιτεκτονικής υποδομής, ποιο πρόβλημα/ποιο είδος προβλημάτων είναι κατάλληλο για αυτήν;

1.4 Το αντικείμενο της διπλωματικής εργασίας

Η δική μας εργασία θα εστιάσει στην απάντηση των παραπάνω ερωτημάτων μέσω της **πρόβλεψης του χρόνου εκτέλεσης μίας εφαρμογής**. Αυτό θα επιχειρήσουμε να το επιτύχουμε μέσω της **εξαγωγής δυναμικών χαρακτηριστικών** από την εφαρμογή ενδιαφέροντος, την πρόβλεψη των τιμών των δυναμικών αυτών χαρακτηριστικών για μεγαλύτερες τιμές εισόδου και την χρησιμοποίηση των τελευταίων για την τελική πρόβλεψη του χρόνου εκτέλεσης, δεδομένης μίας αρχιτεκτονικής και ενός μεγέθους εισόδου.

Μία εποπτική απεικόνιση του συνολικού συστήματος που σχεδιάσαμε και υλοποιήσαμε φαίνεται στην Εικόνα 2.2. Κάθε ένα από τα υποσυστήματα που απαρτίζουν το παραπάνω συνολικό σύστημα θα αναλυθούν ξεχωριστά, αφού πρώτα παρουσιάσουμε το απαραίτητο θεωρητικό πλαίσιο.

1.5 Σχετική έρευνα

Για μία αναλυτική παρουσίαση της σχετικής έρευνας αναφορικά τόσο με την εξαγωγή στατικών και/ή δυναμικών χαρακτηριστικών αλλά και με την πρόβλεψη του χρόνου εκτέλεσης εφαρμογών, παραπέμπουμε τον αναγνώστη στην αντίστοιχη Ενότητα 2.5.

1.6 Η οργάνωση του παρόντος τόμου

Αυτός ο τόμος αποτελείται από 8 Μέρη. Τα Μέρη 1 έως 4 και 5 έως 8 είναι ίδια, με τη διαφορά πως τα πρώτα είναι στα Ελληνικά, ενώ τα δεύτερα στα Αγγλικά. Επιπλέον, το αγγλικό κείμενο είναι το πρωτότυπο και το πλήρες. Το ελληνικό είναι πολύ σύντομο και περιληπτικό.

Για μία αναλυτική παρουσίαση των Μερών και των Κεφαλαίων, παραπέμπουμε τον αναγνώστη στην αντίστοιχη Ενότητα 2.6.

Chapter 2

Introduction

Welcome my son, welcome to the machine
Where have you been?
It's alright, we know where you've been
You've been in the pipeline filling in time
Provided with toys and 'Scouting for Boys'
You brought a guitar to punish your ma
And you didn't like school
And you know you're nobody's fool

Pink Floyd, Welcome To The Machine

In this introductory chapter, we will start by presenting the broader research context of which this dissertation is a part, that of heterogeneous computing. After that, we will pinpoint the problem that the present work attempts to deal with, and the basis on which we started to design our approach. Lastly, we will present the organization of this volume, clarifying what each part and chapter is about.

2.1 A brief history of computing

Over the years, we have witnessed numerous paradigm shifts in the field of computer science and, more specifically, in the way people perceive and actually do computations. From the ancient abaci to the modern supercomputers, the whole history of computing is permeated by the fundamental need to *compute* fast, reliably, precisely and, nowadays, *massively*.

As mentioned in [2], "the earliest known tool for use in computation is the Sumerian abacus, and it was thought to have been invented in Babylon c. 2700–2300 BC ... This was the first known computer and most advanced system of calculation known to date - preceding Greek methods by 2,000 years". This is it; our journey as a species of counting, computing, measuring the world and the things that surround us begins in Mesopotamia, more than 4 and a half millennia ago.

To say that we have come a lot way since then would be an understatement at best. It did not take long for the whole world to embark on this crucial journey: Ancient Chinese, Indian and Greek philosophers, Muslim astronomers; whichever the field, new compu-

tation methods were always needed and were moving in a perpetual cycle of invention, improvement, optimization, and reinvention. Machines were created (like the Antikythera mechanism, probably the first analog computer [3]), thought experiments were designed (like the "Sand Reckoner", where Archimedes "set out to determine an upper bound for the number of grains of sand that fit into the universe" [4]), algorithms were formulated (like the addendum to Babbage's Analytical Machine [5] by Ada Lovelace who, in poetic style, detailed the first computer program, a formal method to calculate Bernoulli Numbers). Pioneers were in a constant search for better ways to live their lives and to understand the world that surrounds them, ultimately by finding ways to measure the immeasurable, quantify the abstract, compute the incomputable.

Until digital computers came. Somewhere in the middle of the 20^{th} century, the eternal journey for better understanding of the world was reshaped in an irreversible way; speed, formalism, precision, massiveness. Even though not a century has passed, the technological breakthroughs related to the evolution of computing and computer science are too many to list. But, in the context of this dissertation, we are interested in focusing on the hardware, i.e. the *processing units* and the way they evolved. Starting from punched cards, then moving on to electromechanical contraptions and vacuum tubes, making an important stop to witness the laying of the foundations of the digital computer by the works of George Boole and Claude Shannon, and finally arriving to transistors, CPUs, GPUs, FPGAs and more [6]; devices designed to satisfy in the best possible way a universal desire, a universal need.

But we have proven more than a few times in the past that satisfaction is quite an elusive notion, especially when it comes to scientific and technological achievements. A simple question summarizes this elusiveness, in the most profound and, at the same time, dark way. Why not more?

Welcome to the age of heterogeneous computation.

2.2 It is a heterogeneous world

As discussed in [7], "the computer industry has been inspired and motivated by the observation made by Gordon Moore (A.K.A "Moore's law") that the density of transistors on die was doubling every 18 months. This observation created the anticipation that the performance a certain application achieves on one generation of processors will be doubled within two years when the next generation of processors will be announced". Unfortunately, this was not the full picture. First of all, physics had a say in this assumption. During the mid 2000s, it started to became apparent that "Moore's law" was not actually a law. The fact that the transistors kept getting smaller and smaller led to tremendous increases in power consumption and density. But the computer industry was not ready (and most probably will never be) to admit that it had been defeated by something as ordinary as the laws of physics and, therefore, a new approach had to emerge, if we wanted to keep getting harder, better, faster and stronger.

As defined in [8], "heterogeneous computing refers to systems that use more than one kind of processor or cores. These systems gain performance or energy efficiency not just by

adding the same type of processors, but by adding dissimilar coprocessors, usually incorporating specialized processing capabilities to handle particular tasks". A heterogeneous system can combine all sorts of different computational units, in order to achieve its goals, whether these goals have to do with efficiency, speed, increased parallelism, or (as discussed above) low energy consumption. More specifically, regarding the latter, "thermal and power become first class citizens with any design of future architecture. These trends encourage the community to start looking at heterogeneous solutions: systems which are assembled from different subsystems, each of them optimized to achieve different optimization points or to address different workloads. For example, many systems combine "traditional" CPU architecture with special purpose FPGAs or Graphics Processors (GPUs). Such an integration can be done at different levels; e.g., at the system level, at the board level and recently at the core level." [7].

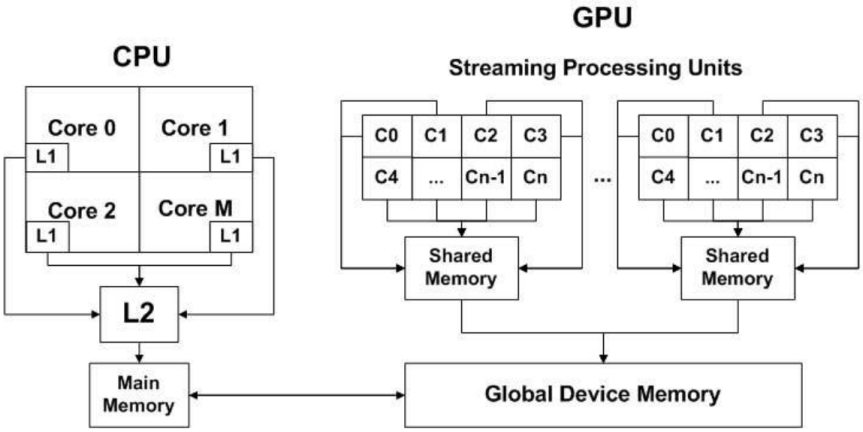

Figure 2.1: *A simple example of a heterogeneous system referenced in [1], consisting of CPUs and GPUs*

2.3 Utilizing diversity

Heterogeneity seems like a natural next step with great potential. As referenced in [7], "heterogeneous computer systems also add richness by allowing the programmer to select the best architecture to execute the task at hand or to choose the right task to make optimal use of a given architecture". That sounds definitely like what we want to accomplish with heterogeneous systems, but important questions unavoidably emerge.

- *How* to select the best architecture for a given task?
- *How* to select the right task for a given architecture?

Furthermore, developing or retargeting applications for such computational systems is far from trivial. Alas, the hardware seems to be more mature than the software. More

specifically, "supporting . . . large (at scale) heterogeneity demands for an adequate software environment able to maximize productivity and to extract maximum performance from the underlying hardware. Such challenge is addressed when one looks at single platforms . . . however, moving at scale, effectively exploiting heterogeneity remains a challenge" [9].

One of the main software frameworks that aim to fulfill this need is OpenCL, an "open standard for parallel programming of heterogeneous systems" [10]. We will delve deeper into the design and implementation of this framework later. For now, let us bare in mind that "OpenCL has been developed specifically to ease the programming burden when writing applications for heterogeneous systems" and that it "also addresses the current trend to increase the number of cores on a given architecture", while its "standard abstractions and interfaces allow the programmer to seamlessly "stitch" together an application within which execution can occur on a rich set of heterogeneous devices from one or many manufacturers" [7].

2.4 The subject of the dissertation

In this work, we are going to present our attempt to path a way towards an alternative answer to the questions of the previous section.

This thesis is part of a larger research effort to create a toolkit and a methodology to profile applications designed for heterogeneous architectures and automatically decide when and/or when to run them, in order to maximize efficiency, throughput and energy conservation. This work attempts to implement the first stages, the backbone of this endeavor. More specifically, the goals of this thesis is to create and present tools that:

1. extract static and, above all, *dynamic* features from the applications of interest
2. interpret and analyze these features in order to build models on them to predict *the execution time* of other (unknown) applications.

In the context of this work, we will focus on OpenCL applications (namely OpenCL *kernels*), given that it is one of the main software frameworks used for developing applications for heterogeneous architectures, as discussed above. However, most of our work can be easily reformed in order to be used with *any* framework or programming language. In fact, given that the collected features meet certain criteria which will be thoroughly discussed, the second part of the pipeline that we will present (i.e. the interpretation and analysis through appropriate machine learning models) is *language-independent*.

To meet our first goal, we created and are proud to present `oclude`, a static and dynamic profiling tool for OpenCL kernels, written in Python 3 and C++. `oclude` is designed to extract static and/or dynamic features from OpenCL kernels, as well as to measure the execution time of a kernel. In the context of our work, the dynamic features of a kernel have been defined as ***the number of bitcode instructions that were executed on the device*** by the kernel. Henceforth, we will interchangeably use the terms ***dynamic features***, ***instruction counts***, ***instcounts*** or ***dynamic instruction counts*** to refer to them.

To meet our second goal, we build regression models that try to determine two key relationships. The first one is the relationship between the size of the input of a kernel and the instruction counts, and the second one is the relationship between the instruction counts and the execution time. For reasons that will become apparent when we discuss `oclude`, we will henceforth refer to the size of the input of a kernel as ***gsize*** *(global size)*. By combining the estimations for these two relationships, we can draw conclusions regarding a third relationship, that between the gsize and the execution time. This grand estimator, the one which combines the simpler models and seemingly estimates the $gsize \longmapsto t_{exec}$ relationship directly, is called `OCLMan`.

It should be noted that our main hope is to present a rigid tool for kernel profiling, static and/or dynamic, and a sound methodology to interpret and analyze these results in order to gain useful insights - that last part is what `OCLMan` is about. However, while our focus on the first part (i.e. `oclude`) is both theoretical and technical (with a slight tendency towards the latter, to be honest), the second part (i.e. `OCLMan`) constitutes mainly a proof of concept. This means that there is a lot more work to be done on the machine learning part of this work and the optimization and fine-tuning of the models that were used. Such an effort is considered to be out of the scope of this dissertation and is included in the "future work" section of the last part.

In summary, the ultimate goal of our work is to create a novel tool to extract critical dynamic information from OpenCL kernels and then, in a language-independent way, propose a methodology to determine the following function f:

$$t_{exec} = f_{kernel,device}(gsize) \tag{2.1}$$

proving that the dynamic features that our tool extracted in the first phase hold valuable information. A more accurate depiction of our approach on determining f would be the following:

$$t_{exec} = f_{kernel,device}(gsize) = g \circ h \tag{2.2}$$

where

$$instcounts = h_{kernel,device}(gsize) \tag{2.3}$$

$$t_{exec} = g_{device}(instcounts) \tag{2.4}$$

Finally, we should make one last note before moving on, regarding the way `oclude` provides `OCLMan` with the data it needs. As will be seen in the next section, the dominant method used for achieving the above goals by the research community revolves around the extraction and analysis of ***static source code features***. This approach has a major advantage but also a significant disadvantage. By using static features, one can extract them almost instantly for any kernel, hence being able to train inline models on the fly with negligible time overhead. On the other hand, a lot of information regarding the dynamic behavior of the application of interest is lost. A simple example is a loop that operates on all the elements of an input vector. By extracting and analyzing the static features

of this application, it is almost impossible to take this behavior into consideration during the analysis phase. On the other hand, if the profiling phase consists of ***running the application with multiple, small and random inputs***, then one can have the best of both worlds. We call this approach ***microprofiling*** and it constitutes the connecting link between the first and second part of our pipeline.

2.4.1 A complete overview of our work

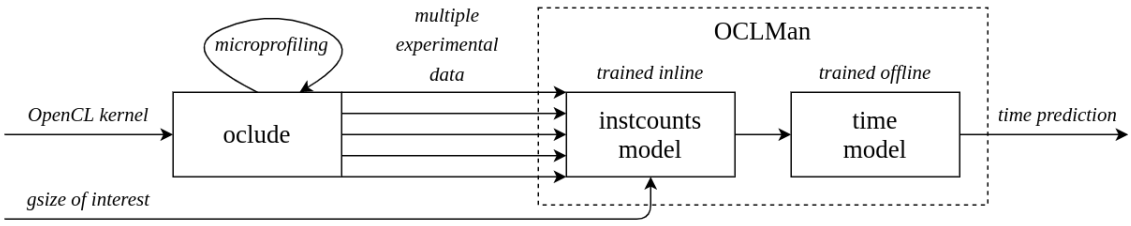

Figure 2.2: *An abstract overview of the complete pipeline*

The above illustration presents an overview of the whole pipeline of our work. Firstly, we are utilizing `oclude` to implement the microprofiling phase, by using it to run the kernel of interest multiple times for different, small, random inputs. Afterwards, we provide the data we collected (i.e. instcounts and execution times for different gsizes) to `OCLMan`, which is responsible for producing a time execution prediction for a given gsize.

`OCLMan` is comprised of *two* regression models, one for the h function and another for the g function, as shown in equations (2.3) and (2.4), respectively. This design derives from 2 motives:

1. The first model (henceforth the *instcounts model*) is ***kernel-specific but hardware-agnostic***, while the second (henceforth the *time model*) is ***kernel-agnostic and hardware-specific***. This decoupling can lead to easier migrations of our work to other software frameworks and/or processing units of different architectures.

2. Minimization of the time needed to make a prediction for a new kernel. This minimization is crucial in order to be comparable to the results of the related literature which, as mentioned before, relies mainly on static features. As will be clearly explained in the `OCLMan` part, the time model (i.e. the kernel-agnostic one) will be *pretrained* for a given processing unit when a new kernel arrives, at which point a kernel-specific instcounts model is trained on the fly, based on the results of the microprofiling conducted with the help of `oclude`. That is, the time model is trained *offline* while the instcounts model is trained *inline*, during the prediction process.

2.5 Related work

Literature related to our research subject is rich and diverse. Previous work has been done with many different approaches but, most of the time, using solely static features (or at least relying extensively on them) instead of dynamic ones.

2.5.1 Regarding feature extraction

Starting with ***feature extraction from applications*** (not necessarily OpenCL), the `Mantis` tool [11] was one of our earliest inspirations. It is a dynamic feature extractor for Java applications and its overall architecture is quite similar to the one of the `oclude - OCLMan` pipeline that we will present here. One of the most interesting parts of the paper is the following section from the "Architecture" section: "*[...] The feature instrumentor analyzes the code of the program and automatically adds instrumentation code that extracts program features. Then the profiler runs this instrumented program with sample input data to collect performance metrics and feature values. [...] Then, the model generator runs machine learning algorithms to generate a prediction model, i.e., select a subset of key features that are relevant to the performance metrics and create a function of the selected features to predict the performance metrics with high accuracy. To use the model, we need a way to compute feature values. The feature evaluator generator uses program slicing to automatically extract small code snippets (which we call feature evaluators) that compute feature values from the instrumented program*". Everything up until the *feature evaluators* is used in our work. However, instead of feature evaluators, we designed `OCLBoi`. The approach of `OCLBoi` is designed to tackle the same problem as the one for which feature evaluators are designed, but in a quite different way, and constitutes the core idea upon which we built the rest of our pipeline. Furthermore, we believe that it represents the most novel part of our work. One more crucial difference between `Mantis` and our approach is that we have selected a richest set of features by passing through the *intermediate representation (IR)* of the source code, while `Manits` follows a different approach by using loop counts, branch counts,and variable values in different versions as features.

The rest of the research on the field of program feature extraction is not as "a radical departure from traditional approaches" (from the `Mantis` paper abstract) as `Mantis` is. Efforts have been made to model execution time, as well as dynamic features like loop counts, as *random variables* [12], but this approach is purely theoretical and does not seem to be suitable for an actual implementation. Beyond that, the dominant approach is that of *static feature extraction*.

Wen et al. [13] present a tool that profiles OpenCL applications by using a set of static features like the number of instructions and the number of load/store operations, and another set which they call *dynamic features*. In this set, they include the input size, the local and global OpenCL NDRange, as well as the output size of the kernel. However, these features are not considered to be dynamic in our work; they are features that are known *before kernel execution*, and therefore we can not consider them as being dynamic. In fact, the approach described by Wen et al. is followed in our work when we design a baseline estimator to compare our pipeline to.

2.5.2 Regarding execution time prediction

Feature extraction is but a means to our goal, which is no other than ***execution time prediction*** or, as frequently mentioned in related literature, ***worst case execution time prediction***. It is interesting to start by noting that Huang et al. [14] deem the following

two approaches as obsolete and outdated when it comes to our problem:

1. building *analytical models* based on (static) raw source code features, due to the complexity of the process the over-simplistic assumptions that are needed
2. the ***"program-as-a-black-box" approach***: dealing with the program of interest as if it was a function, were the (*input*, *output*) pairs are sufficient to estimate it

Despite that, all related literature seems to agree on the fact that code profiling, in a smaller or a larger extent, is an idea to the right direction [12, 14, 15, 16]. ***"Partial execution"*** is a nice term used for this microprofiling process [16] that we will heavily rely on. Yang et al. [16] give one of the most accurate descriptions of this dynamic microprofiling or partial execution process: *"We enable very short "testdrives" of applications on multiple candidate platforms to quickly derive the execution time of much longer runs. The timing results of these testdrives can be stored in a database for reuse in future predictions. This approach facilitates cross-platform performance estimation as an affordable utility"*. They continue by pointing out that "*the objective of partial execution is not to obtain numerical results from scientific codes but to quickly and cheaply capture their rudimentary execution behavior. Only requiring high-level knowledge about the application's control flow and as few as two extra lines of code inserted, partial execution [...] is affordable, scalable and portable. The metrics obtained during partial execution can then be utilized to predict the performance of an application run across different platforms*".

2.5.3 Regarding source code analysis and instrumentation

We can see that the notion of source code instrumentation is present in the excerpt above ("*Only requiring [...] as few as two extra lines of code inserted*" [16]). Source code instrumentation is also used in the `Mantis` tool that we saw earlier [11].

Moreover, techniques like usage of control flow graphs of the input source code [15, 17], as well as more advanced subjects from the field of compilers like value analysis, loop bound analysis and path analysis [17] seem to be proposed and used.

The common basis of all these approaches is the central role that the source code *abstract syntax tree* (AST) and its parsing plays on the analysis and/or instrumentation of the input source code.

2.6 The organization of the volume

This volume is comprised of 8 parts. Parts 1-4 and 5-8 are the same except for the language used; the first 4 parts are written in Greek, while the last 4 parts in English.

Parts 1/5 present the necessary theoretical background needed to understand this work. Each section presents a different topic. The sections of this part are unrelated to each other and can be separately studied. We start by introducing the reader to the most important characteristics and aspects of OpenCL. We then move on to some selected subjects from compiler theory, namely *intermediate representation (IR) code* and *basic*

blocks, a concept of the uttermost importance for `oclude`, as well as a humble peak to the world of *virtual machines (VMs)*, in order to better comprehend the way `oclude` is designed. We finish our theoretical part with a reference to *regression analysis* and, more specifically, to the regression models that we will be using in our work.

Parts 2/6 constitute a thorough presentation of the `oclude` OpenCL kernel profiling tool. We approach `oclude` in a top-down manner; we start by presenting its usage and then delve into its internals.

Parts 3/7 present the `OCLMan` methodology and the corresponding experiments we conducted. We start by our several experiments on the data we gathered from the OpenCL kernels of the Rodinia Benchmark Suite [18]. We then move on to the building blocks of `OCLMan`, the *instcounts and time models*. Lastly, we present the final experiment we conducted to create, train, test and evaluate a simple implementation of the `OCLMan` methodology.

Parts 4/8 list some concluding remarks and propose future work on the field of heterogeneous computing in general and in the tools and methodology presented in our work in particular.

Μέρος Ι

Θεωρητικό Υπόβαθρο

Κεφάλαιο 3

Μία εισαγωγή στην OpenCL

Ξεκινάμε την παρουσίαση του απαραίτητου θεωρητικού υπόβαθρου με την OpenCL, στην οποία στηριζόμαστε εκτενώς στην παρούσα εργασία.

3.1 Οι προδιαγραφές της OpenCL

Η OpenCL είναι ένα σύνολο ευρέως διαδεδομένων προδιαγραφών (specification) για την υλοποίηση παράλληλων και/ή ετερογενών εφαρμογών, καθώς και για τα χαρακτηριστικά που πρέπει να έχουν οι συσκευές στις οποίες οι εφαρμογές αυτές θα τρέχουν. Είναι σημαντικό να θυμόμαστε ότι η OpenCL δεν είναι παρά ένα σύνολο προδιαγραφών και *όχι μία συγκεκριμένη υλοποίηση*.

3.2 Οι πλατφόρμες και οι συσκευές της OpenCL

Μία **OpenCL πλατφόρμα** είναι μία συγκεκριμένη υλοποίηση των προδιαγραφών της OpenCL, π.χ. η πλατφόρμα OpenCL της Intel. Μία **OpenCL συσκευή** είναι μία πραγματική υπολογιστική μονάδα (π.χ. ένας επεξεργαστής ή μία κάρτα γραφικών) που αποτελεί μέρος μίας OpenCL πλατφόρμας, όντας έτσι ικανή να εκτελέσει OpenCL εφαρμογές, τις οποίες αργότερα θα ονομάσουμε *πυρήνες*.

3.3 Το μοντέλο εκτέλεσης της OpenCL

Στην Εικόνα 13.1 βλέπουμε μία απεικόνιση του μοντέλου εκτέλεσης της OpenCL, όπου στα αριστερά φαίνεται ο hostcode ο οποίος τρέχει στον επεξεργαστή και είναι υπεύθυνος για την ενορχήστρωση των εργασιών των πραγματικών υπολογιστικών προγραμμάτων, των πυρήνων (kernels), οι οποίοι και εκτελούνται σε OpenCL συσκευές, όπως φαίνεται στα αριστερά της εικόνας.

Όπως φαίνεται στην Εικόνα 13.2, τα στιγμιότυπα των πυρήνων που εκτελούνται (work-items) οργανώνονται σε ομάδες (work-groups), για λόγους συγχρονισμού.

3.4 Το μοντέλο μνήμης της OpenCL

Στην Εικόνα 13.3 φαίνεται η ιεραρχία μνήμης όπως ορίζεται από την OpenCL. Ο host έχει τη δική του ανεξάρτητη μνήμη. Έπειτα, υπάρχει η καθολική μνήμη (global memory) της συσκευής, η οποία είναι μεν η πιο αργή, αλλά έχουν πρόσβαση σε αυτήν όλα τα work-items. Έπειτα, ακολουθεί η τοπική μνήμη (local memory), η οποία είναι πιο γρήγορη και είναι προσβάσιμη από όλα τα work-items ενός συγκεκριμένου work-group. Τέλος, υπάρχει και η ιδιωτική μνήμη (private memory) του κάθε work-item ξεχωριστά.

Κεφάλαιο 4

Στοιχεία θεωρίας μεταγλωττιστών

Σε αυτό το δεύτερο κεφάλαιο του θεωρητικού υπόβαθρου θα αναφερθούμε σε ό,τι χρειαζόμαστε από τη θεωρία των μεταγλωττιστών.

4.1 Ενδιάμεσος κώδικας

Ο ενδιάμεσος κώδικας είναι το τελευταίο στάδιο του front-end μέρους ενός μεταγλωττιστή, δηλαδή του μέρους που είναι ανεξάρτητο της εκάστοτε μηχανής, όπως φαίνεται στην Εικόνα 14.1.

Συνήθως, η ενδιάμεση αναπαράσταση είναι πιο απλή και εύκολη στην επεξεργασία, στην εξαγωγή χαρακτηριστικών, καθώς και στην εισαγωγή κώδικα για τη λήψη μετρήσεων (code instrumentation).

4.2 Η έννοια των basic blocks

Ένα basic block ορίζεται ως μία μεγιστική ακολουθία εντολών ενδιάμεσου κώδικα με τις εξής ιδιότητες:

1. Η ροή ελέγχου μπορεί να εισέλθει στο basic block μόνο μέσω της πρώτης εντολής του μπλοκ. Με άλλα λόγια, δεν υπάρχουν άλματα προς το μέσον του μπλοκ.

2. Η ροή ελέγχου θα εξέλθει από το μπλοκ χωρίς να σταματήσει ή να πάρει κάποια διακλάδωση, εκτός ίσως από την τελευταία εντολή του μπλοκ.

Η ιδιότητα πως, άπαξ και η ροή ελέχγου μπει σε ένα basic block, τότε όλες οι εντολές αυτού θα εκτελεστούν, θα μας φανεί πολύ χρήσιμη στη συνέχεια.

4.3 Virtual Machines (VMs)

Σε αντίθεση με τους μεταγλωττιστές, οι διερμηνείς ή virtual machines λαμβάνουν τον πηγαίο κώδικα μίας εφαρμογής και είσοδο από το χρήστη, ερμηνεύουν κατά την εκτέλεση τις εντολές από τον πηγαίο κώδικα και παράγουν κάποιο αποτέλεσμα. Δεν υπάρχει, με άλλα λόγια, η φάση της παραγωγής κάποιου εκτελέσιμου αρχείου.

Στη λογική των διερμηνέων βασίζεται η επεξεργασία του κώδικα του πυρήνα που μας έχει δοθεί από τον χρήστη κατά την φάση του instrumentation στο πρώτο εργαλείο που θα παρουσιάσουμε, το `oclude`.

Κεφάλαιο 5

Μοντέλα παλινδρόμησης

Η πρόβλεψη του χρόνου εκτέλεσης μίας εφαρμογής, καθώς και η πρόβλεψη των δυναμικών χαρακτηριστικών της για μεγαλύτερες εισόδους είναι προβλήματα που προσφέρονται για επίλυση μέσω της τεχνικής της παλινδρόμησης. Σε αυτό το κεφάλαιο, θα παρουσιάσουμε τα μοντέλα παλινδρόμησης (regression models) που χρησιμοποιήσαμε στη δουλειά μας.

5.1 Γραμμική παλινδρόμηση

Η γραμμική παλινδρόμηση προσπαθεί να εκτιμήσει τη παρακάτω γραμμική συνάρτηση από τα δεδομένα εκπαίδευσης:

$$y = b_0 + b_1 x_1 + b_2 x_2 + ... + b_n x_n$$

Στη περίπτωσή μας όπου έχουμε να κάνουμε με πολλές μετρήσεις εντολών, η εξίσωση αυτή παίρνει την κάτωθι μορφή:

$$\vec{y} = \vec{b}x \Leftrightarrow \begin{bmatrix} counter_{add} \\ counter_{sub} \\ counter_{mul} \\ \vdots \end{bmatrix} = \begin{bmatrix} coef_{add} \\ coef_{sub} \\ coef_{mul} \\ \vdots \end{bmatrix} gsize$$

5.2 Παλινδρόμηση τύπου Elastic Net

Η παλινδρόμηση τύπου Elastic Net είναι ίδια με τη γραμμική, με τη διαφορά ότι στη τιμή προς ελαχιστοποίηση, μαζί με τα ελάχιστα τετράγωνα προστίθενται και οι δύο πρώτες νόρμες, με στόχο την κανονικοποίηση της εκτίμησης της συνάρτησης.

5.3 Πολυωνυμική παρεμβολή

Η πολυωνυμική παρεμβολή χρησιμοποιεί τις δυνάμεις και τα γινόμενα των χαρακτηριστικών του κάθε δείγματος (μέχρι κάποιο προκαθορισμένο βαθμό), με στόχο να εκτιμήσει τις

παραπάνω εξισώσεις συναρτήσει και αυτών των δυνάμεων και γινομένων, οδηγούμενη, έτσι, από την εκτίμηση μίας (υπερ)ευθείας στην εκτίμηση ενός πολυωνύμου.

5.4 Γιατί να περιοριστούμε;

Θα παραμείνουμε στα γραμμικά και τα πολυωνυμικά μοντέλα που παρουσιάσαμε. Ποιο πολύπλοκα και ενδιαφέροντα μοντέλα παλινδρόμησης, όπως διάφορα δενδρικά μοντέλα (π.χ. XGBoost) δεν είναι κατάλληλα για το δικό μας πρόβλημα, διότι δεν μπορούν να γενικεύσουν έξω από το πεδίο τιμών των ανεξάρτητων τιμών στο οποίο εκπαιδεύτηκαν (extrapolation).

Μέρος II

oclude - ο αναλυτής

Κεφάλαιο 6

Χρησιμοποιώντας το oclude

Είμαστε πλέον έτοιμοι να παρουσιάσουμε το `oclude`, το πρώτο εργαλείο μας. Το `oclude` είναι σχεδιασμένο ώστε να μπορεί να:

- μετρά τον χρόνο εκτέλεσης μίας εφαρμογής,
- μετρά το πλήθος των LLVM εντολών που αντιστοιχούν στην εκτέλεση μίας εφαρμογής.

Για να το επιτύχει αυτό, χρησιμοποιεί διάφορα εργαλεία και τεχνικές, μεταξύ των οποίων:

1. Instrumentation του πηγαίου κώδικα μίας OpenCL εφαρμογής, και συγκεκριμένα ενός OpenCL πυρήνα,
2. Δημιουργία τυχαίων ορισμάτων για την εκτέλεση των πυρήνων,
3. Ένας γενικευμένος hostcode ικανός να εκτελέσει οποιονδήποτε OpenCL πυρήνα.

6.1 Χρήση ως εργαλείου της γραμμής εντολών

Το `oclude` μπορεί να χρησιμοποιηθεί από την γραμμή εντολών ενός Unix-like λειτουργικού συστήματος. Μπορεί να εκτελέσει τις δύο παρακάτω εντολές, device και command, τις οποίες θα εξηγήσουμε αμέσως μετά:

```
$ oclude device <flags>

$ oclude [kernel] <flags>
```

6.1.1 Η εντολή "device"

Χρησιμοποιείται για την ανάλυση των χαρακτηριστικών και του τρόπου λειτουργίας μίας συγκεκριμένης OpenCL συσκευής. Παραδείγματα χρήσης:

```
$ oclude device -p 0 -d 0
[hostcode] Collecting profiling info for the following device:
[hostcode] Platform:    Intel(R) OpenCL HD Graphics
[hostcode] Device:      Intel(R) Gen9 HD Graphics NEO
```

```
[hostcode] Version:     OpenCL 2.1 NEO
[hostcode] Please wait, this may take a while...
Profiling info for selected OpenCL device:
        profiling overhead (time) - 0.011303499341011047
  profiling overhead (percentage) - 17.80%
                  command latency - 0.06351426243782043
  host-to-device transfer latency - 0.011074915528297424
  device-to-host transfer latency - 0.011512413620948792
device-to-device transfer latency - 0.06323426961898804
host-device bandwidth bandwidth @ 64 bytes - 0.005645181903735443 GB/s
host-device bandwidth bandwidth @ 256 bytes - 0.022125035974706695 GB/s
host-device bandwidth bandwidth @ 1024 bytes - 0.08657326467722175 GB/s
... a lot of bandwidth measurements follow ...
```

6.1.2 Η εντολή "kernel"

Χρησιμοποιείται για την εκτέλεση ενός OpenCL πυρήνα. Παράδειγμα χρήσης:

```
$ oclude -f tests/rodinia_kernels/dwt2d/com_dwt.cl -k c_CopySrcToComponents -g 1024 -l 128
[oclude] INFO: Input file tests/rodinia_kernels/dwt2d/com_dwt.cl is not cached
[oclude] Running kernel 'c_CopySrcToComponents' from file <path omitted to fit>/com_dwt.cl
[hostcode] Using the following device:
[hostcode] Platform:    Intel(R) OpenCL HD Graphics
[hostcode] Device:      Intel(R) Gen9 HD Graphics NEO
[hostcode] Version:     OpenCL 2.1 NEO
[hostcode] Kernel name: c_CopySrcToComponents
[hostcode] Kernel arg 1: d_r (int*, global)
[hostcode] Kernel arg 2: d_g (int*, global)
[hostcode] Kernel arg 3: d_b (int*, global)
[hostcode] Kernel arg 4: cl_d_src (uchar*, global)
[hostcode] Kernel arg 5: pixels (int, private)
[hostcode] About to execute kernel with Global NDRange = 1024 and Local NDRange = 128
[hostcode] Number of executions (a.k.a. samples) to perform: 1
[hostcode] Kernel run completed successfully
```

6.1.3 Η λειτουργία "instcounts"

Στα πλαίσια της εντολής "kernel", η λειτουργία "instcounts" χρησιμοποιείται για τη μέτρηση των LLVM εντολών που αντιστοιχούν στην εκτέλεση ενός OpenCL πυρήνα. Παράδειγμα χρήσης:

```
$ oclude -f <path omitted to fit>/com_dwt.cl -k c_CopySrcToComponents -g 1024 -l 128 -i
[oclude] INFO: Input file tests/rodinia_kernels/dwt2d/com_dwt.cl is cached
[oclude] INFO: Using cached instrumented file
[oclude] Running kernel 'c_CopySrcToComponents' from file <path omitted to fit>/com_dwt.cl
[hostcode] Using the following device:
[hostcode] Platform:        Intel(R) OpenCL HD Graphics
[hostcode] Device:      Intel(R) Gen9 HD Graphics NEO
```

```
[hostcode] Version:     OpenCL 2.1 NEO
[hostcode] Kernel name: c_CopySrcToComponents
[hostcode] Kernel arg 1: d_r (int*, global)
[hostcode] Kernel arg 2: d_g (int*, global)
[hostcode] Kernel arg 3: d_b (int*, global)
[hostcode] Kernel arg 4: cl_d_src (uchar*, global)
[hostcode] Kernel arg 5: pixels (int, private)
[hostcode] About to execute kernel with Global NDRange = 1024 and Local NDRange = 128
[hostcode] Number of executions (a.k.a. samples) to perform: 1
[hostcode] Collecting instruction counts...
[hostcode] Kernel run completed successfully
Instructions executed for kernel 'c_CopySrcToComponents':
           26920 - load private
           20776 - alloca
           14336 - store private
           12288 - add
           11631 - getelementptr
           11264 - mul
            8855 - store callee
            7245 - load callee
            4096 - call
            3072 - load global
            3072 - load local
            3072 - store local
            3072 - zext
            2415 - sub
            1829 - br
            1024 - ret
            1024 - icmp
```

6.1.4 Η λειτουργία "timeit"

Στα πλαίσια της εντολής "kernel", η λειτουργία "timeit" χρησιμοποιείται για τη μέτρηση του χρόνου εκτέλεσης ενός OpenCL πυρήνα. Παράδειγμα χρήσης:

```
$ oclude -f <path omitted to fit>/com_dwt.cl -k c_CopySrcToComponents -g 1024 -l 128 -t
[oclude] INFO: Input file tests/rodinia_kernels/dwt2d/com_dwt.cl is not cached
[oclude] Running kernel 'c_CopySrcToComponents' from file <path omitted to fit>/com_dwt.cl
[hostcode] Using the following device:
[hostcode] Platform:        Intel(R) OpenCL HD Graphics
[hostcode] Device:        Intel(R) Gen9 HD Graphics NEO
[hostcode] Version:        OpenCL 2.1 NEO
[hostcode] Kernel name: c_CopySrcToComponents
[hostcode] Kernel arg 1: d_r (int*, global)
[hostcode] Kernel arg 2: d_g (int*, global)
[hostcode] Kernel arg 3: d_b (int*, global)
[hostcode] Kernel arg 4: cl_d_src (uchar*, global)
[hostcode] Kernel arg 5: pixels (int, private)
[hostcode] About to execute kernel with Global NDRange = 1024 and Local NDRange = 128
[hostcode] Number of executions (a.k.a. samples) to perform: 1
[hostcode] Collecting time profiling info...
[hostcode] Kernel run completed successfully
```

```
Time measurement info regarding the execution for kernel 'c_CopySrcToComponents'
(in milliseconds):
hostcode - 1.9354820251464844
  device - 0.013415999999999999
transfer - 1.9220660251464843
```

6.2 Χρήση ως πακέτου Python

Το `oclude` διαθέτει και Python 3 API. Για την πλήρη ανάλυση και παρουσίαση της χρήσης του `oclude` μέσω της Python, παραπέμπουμε τον αναγνώστη στο wiki του `oclude` [19].

Κεφάλαιο 7

Υλοποιώντας το oclude

Το `oclude`, όπως είπαμε και στο προηγούμενο κεφάλαιο, είναι το πρώτο από τα εργαλεία που αναπτύξαμε στα πλαίσια της παρούσας διπλωματικής εργασίας. Στόχος του είναι η εκτέλεση μεμονωμένων OpenCL πυρήνων (δηλαδή χωρίς hostcode), η μέτρηση του χρόνου εκτέλεσής τους, καθώς και η μέτρηση των LLVM εντολών που αντιστοιχούν στην εκτέλεση αυτή.

Δεν θα μπορούσαμε να προχωρήσουμε εάν δεν παρουσιάζαμε συνολικά και εποπτικά την αρχιτεκτονική του `oclude` και τις συνιστώσες που το αποτελούν.

7.1 Μία εποπτεία της αρχιτεκτονικής

Όπως φαίνεται στο UML διάγραμμα της Εικόνας 17.1, το `oclude` αποτελείται από τις εξής υπομονάδες (components):

1. Την υπομονάδα του instrumentation
2. Την υπομονάδα του instrumentation-parser
3. Την υπομονάδα του hostcode
4. Την υπομονάδα της κρυφής μνήμης (cache)

7.2 Μία εποπτεία του τρόπου λειτουργίας

Στο UML διάγραμμα της Εικόνας 17.2, βλέπουμε τη συνολική ροή της λειτουργίας του `oclude` κατά το γενικό σενάριο χρησιμοποίησής του από κάποιον χρήστη.

7.3 Οι υπομονάδες του oclude

7.3.1 Η υπομονάδα του instrumentation

Η υπομονάδα του instrumentation είναι υπεύθυνη για:

- την διάσχιση του AST του κώδικα του OpenCL πυρήνα εισόδου, και

- το instrumentation του AST του κώδικα του OpenCL πυρήνα εισόδου ανά basic block με τα στατικά χαρακτηριστικά που συνέλεξε η υπομονάδα του instrumentation-parser.

Για περισσότερες λεπτομέρειες του τρόπου λειτουργίας και υλοποίησης αυτής της υπομονάδας, παραπέμπουμε τον ενδιαφερόμενο αναγνώστη στην αντίστοιχη Ενότητα 17.3.1.

7.3.2 Η υπομονάδα του instrumentation-parser

Η υπομονάδα του instrumentation-parser είναι υπεύθυνη για την εξαγωγή των στατικών χαρακτηριστικών από τον κώδικα του OpenCL πυρήνα εισόδου. Για να το επιτύχει αυτό, λαμβάνει το LLVM bitcode του κώδικα από την υπομονάδα του instrumentation και το διασχίζει με τη βοήθεια του LLVM C++ API, οπότε και μετράει τις εντολές του κώδικα ανά basic block.

7.3.3 Η υπομονάδα του hostcode

Η υπομονάδα του hostcode είναι υπεύθυνη για την εκτέλεση του OpenCL πυρήνα εισόδου στην OpenCL συσκευή του συστήματος που επιλέχθηκε από τον χρήστη. Για να το επιτύχει αυτό, χρησιμοποιεί το PyOpenCL API [20], καθώς και το εργαλείο `rvg` [21] για την τυχαία αρχικοποίηση των ορισμάτων του πυρήνα.

Για περισσότερες λεπτομέρειες του τρόπου λειτουργίας και υλοποίησης αυτής της υπομονάδας, παραπέμπουμε τον ενδιαφερόμενο αναγνώστη στην αντίστοιχη Ενότητα 17.3.3.

7.3.4 Η υπομονάδα της κρυφής μνήμης (cache)

Η υπομονάδα της κρυφής μνήμης (cache) είναι υπεύθυνη για την αποθήκευση των OpenCL πυρήνων που έχουν υποβληθεί σε instrumentation ώστε να μην χρειάζεται η διαδικασία να επαναλαμβάνεται κάθε φορά.

Το βασικό κίνητρο πίσω από την υλοποίηση αυτής της μονάδας ήταν η μείωση του απαιτούμενου χρόνου για εφαρμογές όπου το oclude χρειάζεται να παίρνει πολλές μετρήσεις από τον ίδιο OpenCL πυρήνα - για παράδειγμα, σε κάποια εφαρμογή μηχανικής μάθησης (machine learning applications).

Κεφάλαιο 8

Σχετικά με την αναγκαιότητα ύπαρξης του oclude

Πριν αφήσουμε πίσω μας το `oclude`, θα το συγκρίνουμε με άλλα δύο αντίστοιχα εργαλεία τα οποία χρησιμοποιήσαμε εκτενώς τις πρώτες μέρες της έρευνάς μας. Τα εργαλεία αυτά είναι το `Oclgrind` και το `cldrive`.

8.1 Άλλοι OpenCL αναλυτές

8.1.1 Το Oclgrind

Το `Oclgrind` είναι ένας προσομοιωτής OpenCL. Τούτο σημαίνει πως ο OpenCL πυρήνας εισόδου δεν εκτελείται στην πραγματικότητα, αλλά η εκτέλεσή του προσομοιώνεται από ένα υβριδικό OpenCL/LLVM virtual machine.

8.1.2 Το cldrive

Το `cldrive` είναι ένα εργαλείο για την εκτέλεση μεμονωμένων OpenCL πυρήνων, όπως και το `oclude`. Η κύρια διαφορά μεταξύ τους είναι η δυσκολία χρήσης του πρώτου, καθώς και το ότι δεν μετρά τις LLVM εντολές που αντιστοιχούν στην εκτέλεση του πυρήνα.

8.2 Σύγκριση των εργαλείων

Παραπέμπουμε τον ενδιαφερόμενο αναγνώστη στον Πίνακα 18.1, όπου παραθέτουμε διάφορα χαρακτηριστικά των τριών εργαλείων και τα συγκρίνουμε μεταξύ τους μέσω αυτών. Σε αυτόν τον Πίνακα διαφαίνεται η υπεροχή και η αναγκαιότητα ύπαρξης του `oclude`.

Μέρος III

OCLMan - το μαντείο

Κεφάλαιο 9

Συλλέγοντας και αναλύοντας δεδομένα

Αφού έχουμε παρουσιάσει το `oclude`, είμαστε έτοιμοι να πάρουμε μετρήσεις από κάποιους OpenCL πυρήνες. Στα πλαίσια της παρούσας εργασίας, θα χρησιμοποιήσουμε το Rodinia Benchmark Suite [18], το οποίο περιλαμβάνει, μεταξύ άλλων, μία πληθώρα OpenCL πυρήνων κατάλληλων για λήψη μετρήσεων.

9.1 Η πειραματική διαδικασία

1. Λήψη μετρήσεων από κάθε OpenCL πυρήνα για πολλά διαφορετικά μεγέθη εισόδου (εφεξής **gsizes**), ανάλογα με τη συμπεριφορά και την ευκολία εκτέλεσης του καθενός.

2. Λήψη 100 διαφορετικών μετρήσεων για κάθε gsize με στόχο τον υπολογισμό του μέσου όρου τους ώστε να εξαλείψουμε τον θόρυβο που υπεισέρχεται στις μετρήσεις μας κατά το δυνατό. Για περισσότερες λεπτομέρειες, παραπέμπουμε τον ενδιαφερόμενο αναγνώστη στο Παράρτημα A.

9.2 Γενικά στατιστικά από την ανάλυση των πυρήνων

Αρχικά, χωρίσαμε τους πυρήνες σε 'ανεπίδεκτους ανάλυσης', 'σχετικά γρήγορους' και 'σχετικά αργούς', όπως φαίνεται στην Εικόνα 19.1.

Στον Πίνακα 19.1 βλέπουμε το πλήθος των μετρήσεων (με 100 δείγματα ανά μέτρηση) που κάναμε ανά πυρήνα του Rodinia.

9.3 Διερευνητική ανάλυση των δεδομένων (EDA)

9.3.1 Αναφορικά με όλους τους πυρήνες

Διαπιστώσαμε πως τα διάφορα είδη LLVM εντολών είναι ισχυρά αλληλοσχετιζόμενα, όπως φαίνεται στον Πίνακα 19.2.

9.3.2 Αναφορικά με τους 'σχετικά γρήγορους' πυρήνες

Στην Εικόνα 19.4 βλέπουμε κάποια γραφήματα από μερικούς ενδεικτικούς πυρήνες της κατηγορίας των 'σχετικά γρήγορων', όπου και διακρίνεται μία σχετικά γραμμική σχέση μεταξύ του gsize και του πλήθους των LLVM εντολών.

9.3.3 Αναφορικά με τους 'σχετικά αργούς' πυρήνες

Στην Εικόνα 19.6 βλέπουμε κάποια γραφήματα από μερικούς ενδεικτικούς πυρήνες της κατηγορίας των 'σχετικά αργών', όπου και διακρίνεται μία σχετικά πολυωνυμική -ίσως δεύτερης τάξης- σχέση μεταξύ του gsize και του πλήθους των LLVM εντολών.

9.4 Συγκριτικά συμπεράσματα

Στη πίτα της Εικόνας 19.7, βλέπουμε πως σχεδόν 2 στους 3 πυρήνες είναι 'σχετικά γρήγοροι', ενώ 1 στους 3 είναι 'σχετικά αργοί'. Αυτό, σε συνδυασμό με τη γραμμική και τη πολυωνυμική συμπεριφορά τους αντίστοιχα, είναι σημαντικά δεδομένα για τον σχεδιασμό των δύο επόμενων εργαλείων, του `OCLBoi` και του `OCLMan`, που ακολουθεί.

Κεφάλαιο 10

OCLBoi: Το εξειδικευμένο ανά πυρήνα μοντέλο

Έχοντας στο μυαλό μας τα αποτελέσματα από την ανάλυση των πειραματικών δεδομένων του προηγούμενου κεφαλαίου, είμαστε πλέον σε θέση να σχεδιάσουμε και να υλοποιήσουμε το `OCLBoi`, ένα μοντέλο ειδικό και διαφορετικό για κάθε OpenCL πυρήνα, το οποίο θα δέχεται ως είσοδο ένα gsize και θα προβλέπει τις LLVM εντολές που αντιστοιχούν στην εκτέλεση του πυρήνα, δηλαδή τα δυναμικά του χαρακτηριστικά.

10.1 Τα μοντέλα παλινδρόμησης που χρησιμοποιήσαμε

Για να επιτύχουμε τα παραπάνω, θα χρησιμοποιήσουμε τα τέσσερα μοντέλα παλινδρόμησης που αναφέραμε στο Θεωρητικό Μέρος:

1. Έναν γραμμικό παλινδρομητή
2. Έναν παλινδρομητή τύπου Elastic Net
3. Έναν πολυωνυμικό παλινδρομητή $2^{ου}$ βαθμού, βασισμένο σε γραμμικό παλινδρομητή
4. Έναν πολυωνυμικό παλινδρομητή $2^{ου}$ βαθμού, βασισμένο σε παλινδρομητή τύπου Elastic Net

10.2 Επιλέγοντας μοντέλο παλινδρόμησης για κάθε πυρήνα

Στην Εικόνα 20.1 βλέπουμε τη μέση τιμή του R^2 που είχαν τα διάφορα μοντέλα παλινδρόμησης στους πυρήνες από τους οποίους 'επιλέχθηκαν' (δηλαδή που είχαν το μεγαλύτερο σκορ).

Στην Εικόνα 20.2, βλέπουμε ότι το πιο δημοφιλές μοντέλο είναι ο **πολυωνυμικός παλινδρομητής με βάση τον γραμμικό** (12 πυρήνες). Ακολουθεί ο **γραμμικός παλινδρομητής** (με 10 πυρήνες), και έπειτα ο παλινδρομητής τύπου **Elastic Net** (9 πυρήνες) και **ο πολυωνυμικός παλινδρομητής με βάση το Elastic Net** (8 πυρήνες).

Τέλος, στην Εικόνα 20.3 βλέπουμε ότι ***τα γραμμικά μοντέλα*** (γραμμική παλινδρόμηση και παλινδρόμηση τύπου Elastic Net) ***επιλέχθηκαν από την πλειοψηφία των 'σχετικά γρήγορων' πυρήνων σε ποσοστό 60%***, ενώ τα ***πολυωνυμικά μοντέλα*** (πολυωνυμική παλινδρόμηση με βάση απλή γραμμική παλινδρόμηση και πολυωνυμική

παλινδρόμηση με βάση το Elastic Net) ***επιλέχθηκαν από την πλειονότητα των 'σχετικά γρήγορων' πυρήνων σε ποσοστό 71,4%***.

Κεφάλαιο 11

OCLMan: Ο άρχοντας των μοντέλων

Είμαστε πλέον έτοιμοι να παρουσιάσουμε το OCLMan, τη μεθοδολογία/μοντέλο που θα χρησιμοποιήσουμε για να ελέγξουμε ότι πράγματι η προσέγγισή μας εξάγει χρήσιμη και σημαντική πληροφορία από τους OpenCL πυρήνες η οποία δεν θα ήταν διαθέσιμη εάν χρησιμοποιούσαμε τα στατικά μοντέλα προς τα οποία κλίνει η σχετική έρευνα.

11.1 Μία εποπτεία του OCLMan

Όπως φαίνεται και στην Εικόνα 2.2, ο OCLMan αποτελείται από δύο μοντέλα που συντίθενται, το μοντέλο των μετρήσεων εντολών (instcounts model) και το χρονικό μοντέλο (time model).

11.2 Από τις πειραματικές μετρήσεις στο πλήθος εντολών

Το πρώτο μοντέλο, το μοντέλο των μετρήσεων εντολών, είναι το γνωστό μας OCLBoi από το προηγούμενο κεφάλαιο.

11.3 Από τις πειραματικές μετρήσεις στο χρόνο εκτέλεσης

Για το δεύτερο μοντέλο, το χρονικό, κάνουμε την απλοϊκή υπόθεση πως ο χρόνος εκτέλεσης είναι γραμμική συνάρτηση των LLVM εντολών που εκτελέστηκαν. Συνεπώς, το μοντέλο αυτό θα είναι ένας απλός γραμμικός παλινδρομητής με είσοδο το διάνυσμα των LLVM εντολών που αντιστοιχούν σε μία εκτέλεση και έξοδο τον χρόνο αυτής της εκτέλεσης.

11.4 OCLMan, όλοι για έναν!

Είμαστε έτοιμοι να συνθέσουμε, να εκπαιδεύσουμε, να τεστάρουμε και να χρησιμοποιήσουμε τον OCLMan.

11.4.1 Εκπαίδευση και έλεγχος

Η διαδικασία της εκπαίδευσης (training) και ελέγχου (testing) του OCLMan φαίνεται στην Εικόνα 21.1. Καθότι ο χειρισμός των δεδομένων εκπαίδευσης και ελέγχου δεν ήταν μία τετριμμένη διαδικασία, παραπέμπουμε τον αναγνώστη στην αντίστοιχη Ενότητα 21.4.1.

11.4.2 Αξιολόγηση

Για να αξιολογήσουμε τον `OCLMan`, σχεδιάζουμε ένα απλό στατικό πολυωνυμικό μοντέλο με τον ίδιο στόχο, την πρόβλεψη χρόνου εκτέλεσης με είσοδο κάποιο gsize. Για περισσότερες λεπτομέρειες σχετικά με τις σχεδιαστικές επιλογές που αφορούν σε αυτό το μοντέλο βάσης (εφεξής `OCLBase`), παραπέμπουμε τον ενδιαφερόμενο αναγνώστη στην αντίστοιχη Ενότητα 21.4.2.

Στον Πίνακα 21.1 φαίνεται η σύγκριση των τιμών του R^2 και του μέσου τετραγωνικού σφάλματος μεταξύ του `OCLMan` και του `OCLBase` σε 5 τυχαίους OpenCL πυρήνες, όπου και διαφαίνεται η σαφής υπεροχή του δικού μας μοντέλου. Αντίστοιχα, αυτή η υπεροχή είναι σαφής και στα γραφήματα της Εικόνας 21.2.

Μέρος IV

Τελικές παρατηρήσεις

Κεφάλαιο 12

Προτάσεις για μελλοντική έρευνα

Θα ολοκληρώσουμε αυτό το όμορφο ταξίδι με μερικές προτάσεις για μελλοντική έρευνα στον τομέα της παρούσας διπλωματικής εργασίας, αλλά και πιο ειδικά σχετικά με τα εργαλεία που σχεδιάσαμε, υλοποιήσαμε και παρουσιάσαμε.

1. Όπως είδαμε, ο σχεδιασμός, η κατασκευή και η συντήρηση του `oclude` ως instrumentor πηγαίου κώδικα είναι ένα εξαιρετικά δύσκολο έργο. Θα ήταν πολύ χρήσιμο εάν το `oclude` ξαναγραφόταν ως instrumentor ενδιάμεσου κώδικα. Εάν γινόταν κάτι τέτοιο, η διαδικασία του instrumentation θα ήταν τετριμμένη. Ο λόγος που το `oclude` σχεδιάστηκε κατ' αυτόν τον τρόπο είναι επειδή δεν μπορούσε να βρεθεί καμία μέθοδος κατά τη διάρκεια της έρευνάς μας για την εκτέλεση OpenCL εφαρμογών σε μορφή ενδιάμεσου κώδικα (και συγκεκριμένα σε LLVM IR).

2. Μία από τις πιο απλουστευτικές υποθέσεις μας, κυρίως όσον αφορά τους επεξεργαστές ως OpenCL συσκευές, είναι ότι ο χρόνος εκτέλεσης είναι μια γραμμική συνάρτηση των LLVM εντολών και, με βάση αυτήν την υπόθεση, χρησιμοποιήσαμε έναν γραμμικό παλινδρομητή ως το χρονικό μας μοντέλο στον `OCLMan` και το `OCLBase`. Αυτή η υπόθεση πρέπει να ελεγχθεί, με περισσότερες τεχνικές παλινδρόμησης για το χρονικό μοντέλο.

3. Ο `OCLMan` πρέπει να ωριμάσει και να εξελιχθεί από μεθοδολογία σε εργαλειοθήκη. Για να επιτευχθεί κάτι τέτοιο, η αρχιτεκτονική του και πολλές από τις παραμέτρους του πρέπει να αναπροσαρμοστούν και να ρυθμιστούν πιο ντελικάτα. Το καλύτερο δυνατό σενάριο είναι ο `OCLMan` να μετατραπεί σε ένα αυτόνομο λογισμικό που αναλύει OpenCL συσκευές και απομονωμένους πυρήνες.

4. Απαιτούνται περισσότερες συσκευές και περισσότεροι πυρήνες για να δοκιμάσουμε περαιτέρω την αρχιτεκτονική μας, τα εργαλεία μας, τις ιδέες και τις υποθέσεις μας.

Part V

Theoretical Background

Chapter 13

An Introduction to OpenCL

As explained in the Introduction, we will be using OpenCL as the software framework of choice throughout our work. As a result of this decision, the design and implementation of `oclude`, our profiler and feature extractor, could not be unrelated to the design and implementation of OpenCL. Therefore, the most important concepts of OpenCL in relation to our work are presented in this Chapter.

13.1 The OpenCL specification: an overview

As mentioned in [22], "The OpenCL standard offers a common API for program execution on systems composed of different types of computational devices such as multicore CPUs, GPUs, or other accelerators." Therefore, OpenCL is nothing more than a mere **specification**, i.e. guidelines to:

- the users, describing a way to design, create and run applications on parallel/heterogeneous systems
- the hardware vendors, defining the protocols that processing units (CPUs, GPUs, etc) must follow in order to facilitate the above

OpenCL has no standard implementation; it differs from vendor to vendor, but the API and the observable behavior (should) stay the same.

13.2 OpenCL platform and devices

An **OpenCL platform** is a specific OpenCL implementation, e.g. the Intel OpenCL platform. An **OpenCL device** is an actual processing unit (e.g. a CPU, a GPU) that is part of an OpenCL platform, hence is able to execute OpenCL applications (kernels).

13.3 The OpenCL execution model

Observe Figure 13.1. On the right, we can see the host component of the OpenCL execution environment. This is commonly called hostcode and it is the part of an OpenCL application that runs on the main processing unit of a system, most commonly a CPU,

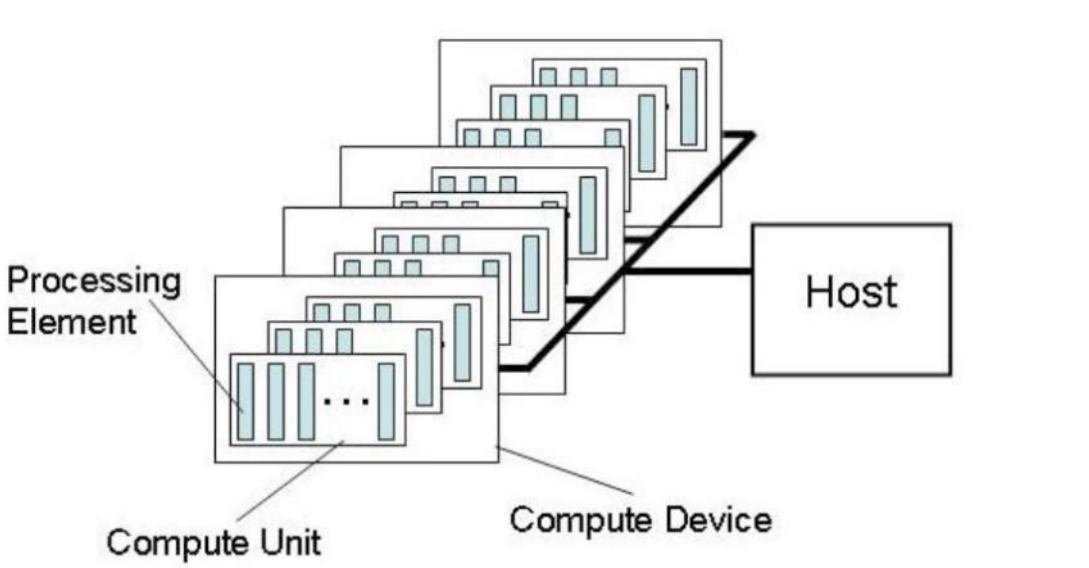

Figure 13.1: *A graphic overview of the OpenCL execution model ©Khronos Group, 2012*

which does not need to be an OpenCL device, and orchestrates the execution of the actual task(s) on the OpenCL devices that the programmer or user specifies. The competences and capabilities of the hostcode include, but are not limited to:

1. Reading input data and transforming them into appropriate memory objects, e.g. OpenCL buffers and images
2. Instructing the target OpenCL device to JIT (just-in-time) compile the kernel(s) that will execute the needed tasks
3. Dispatching the kernel and its properly formatted data to the appropriate OpenCL device
4. Deciding the size of the problem, the number of kernel instances (i.e. the *work-items*) needed to tackle it and their division into small groups where inner synchronization is allowed (i.e. the *work-groups*)
5. Profiling the device and kernel runtime behavior directly (e.g. execution time measurement) or indirectly (e.g. hostcode synchronization barriers)
6. Collecting the arguments of the kernel(s) after their execution is completed

As mentioned several times, what actually does beneficial computational work are the kernels, C-like functions that typically operate on a single element or a small group of elements of a 1, 2 or 3-dimensional buffer[1]. Their dispatch on OpenCL devices and computational units is shown on the left side of Figure 13.1.

To fully grasp the execution model of OpenCL, imagine that we want to apply a naive denoising filter on the image shown in Figure 13.2. Let us assume that this filter is a simple

[1] In OpenCL, 1-dimensional arrays are called *Buffers*, while 2 and 3-dimensional arrays are called *Images*.

average of the arithmetic value of each pixel and that of its eight neighbors. To achieve that goal, we write a simple OpenCL kernel that accepts a 2D image as its argument, finds out its location on the 1024×1024 grid through the `get_{local,global}_id` intrinsic OpenCL functions, and computes this average for a single pixel. We then write a hostcode that spawns 1024×1024 instances of this kernel (i.e. work-items) by dispatching them on the specified OpenCL device for execution, and then collects the result, through a possible second image argument were all of the work-items write their average pixel. Note that the work-items can be divided into work-groups (and *will* be, in a nearly optimal way, if the programmer does not specify anything). These work-groups offer some shinny features like **inner synchronization** (which is impossible outside of a work-group) and **local memory**, an address space which is accessible by all the members of a group and only by them and, per the OpenCL specification, is closer and/or faster than the global memory of the OpenCL device.

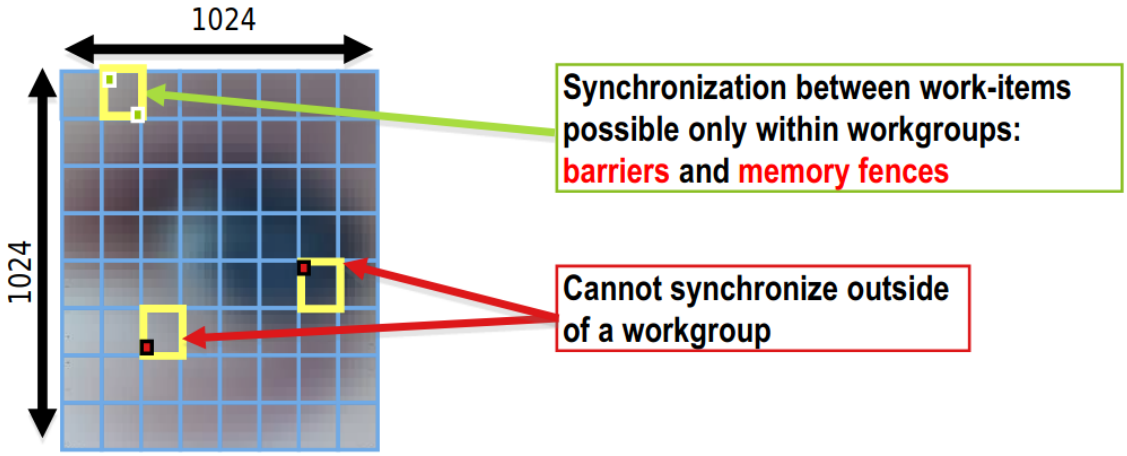

Figure 13.2: *OpenCL work-items and work-groups ©Khronos Group, 2012*

13.4 The OpenCL memory model

But where do the work-items write their results for the hostcode to retrieve them? Apart from its own memory, the hostcode can access only the global and constant memory of the OpenCL device. These two address spaces are typically the largest and the slowest in an OpenCL device, but are visible from every work-item. Immediately after comes the local memory, which is faster and/or closer to the work-items and it is different and isolated for each work-group. Lastly, the fastest address space is the private memory of each work-item. A graphical representation of the above can be found in Figure 13.3.

We can not emphasize enough the importance of knowing which address space is accessed by a work-item for our dynamic profiling process. By differentiating between the loads and stores on each address space we essentially mark the relationship of each one with our kernel in a different and distinct way, mirroring what actually happens on the hardware; a thousand global memory accesses are totally different from a thousand local or private memory access in terms of how they impact execution time or energy consump-

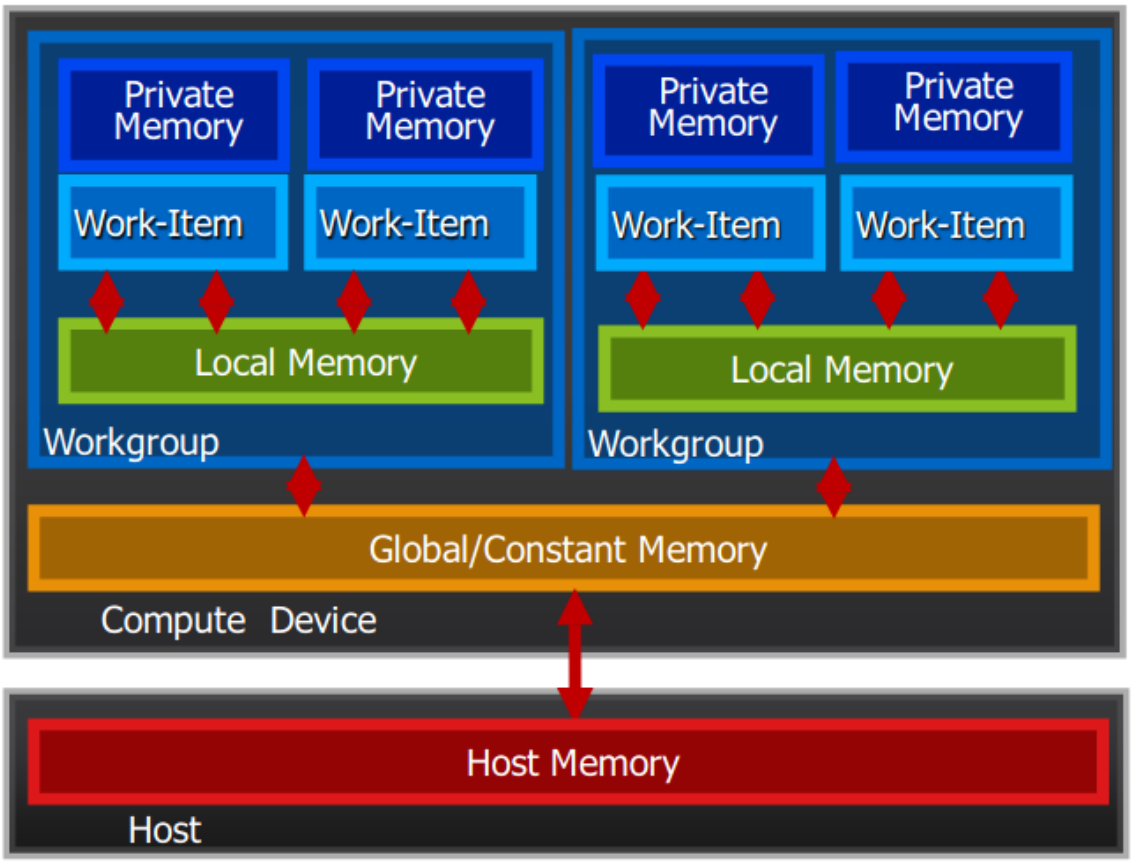

Figure 13.3: *A graphic overview of the OpenCL memory model ©Khronos Group, 2012*

tion, for example. As the reader will find out later, `oclude` does make this differentiation by treating loads and stores on each one of the four OpenCL address spaces as **different instructions**.

13.5 OpenCL APIs

As mentioned before, an OpenCL kernel is a C-like function. To be more precise, an OpenCL kernel is a function written in OpenCL C, a language which is essentially identical to C99, plus some keywords (mainly to directly manage the address spaces that a kernel uses) and some intrinsic functions, like `get_{local,global}_id` that were mentioned before.

But what is the hostcode of an OpenCL application written in? Unlike OpenCL kernels, there is no dedicated language for that component of an OpenCL application. To build a hostcode, an API is needed which is essentially nothing more than including a library in C or C++.

However, using the official C or C++ APIs was too low-level and complicated for our purposes. Therefore, we used PyOpenCL [20], the standard Python API to build the hostcode component of OpenCL applications. PyOpenCL is merely a wrapper API around the official C OpenCL API.

Chapter 14

Elements of Compiler Theory

In Chapter 17, where the architecture of `oclude` is presented, the reader will realize that the technical pillar on top of which `oclude` is built, is the *intermediate representation (IR) code* of the input OpeCL kernel, the *basic blocks* of this intermediate representation and, lastly, a simplified *virtual machine* that parses the abstract syntax trees (ASTs) of the original source code and the instrumented one. Therefore, we deem it necessary to present what we need from the world of compilers now.

14.1 Intermediate representation (IR) code

The *intermediate representation code* or *intermediate code* is the last compilation phase of the *front-end* of a compiler, i.e. the *machine-independent* part of it, as shown in Figure 14.1.

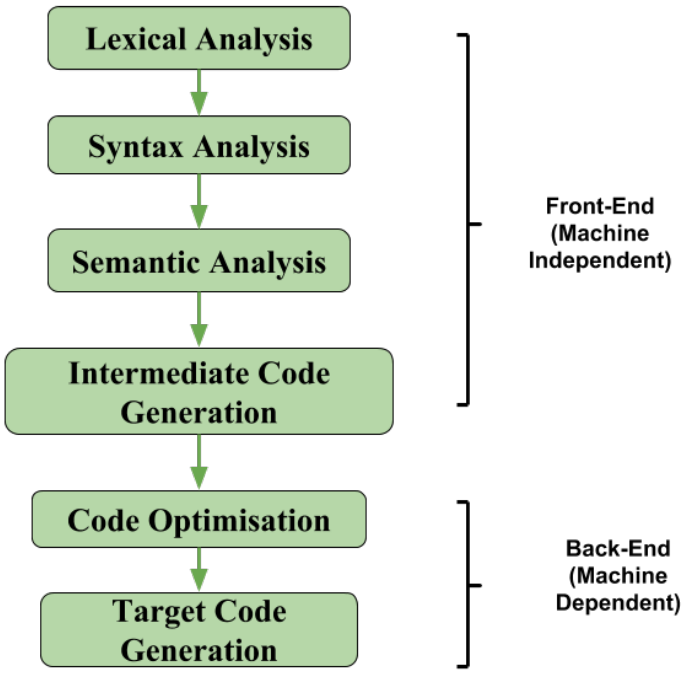

Figure 14.1: *A simplified overview of the compilation process*

It should be noted that intermediate code representations tend to look like assembly code, in terms of structure.

The reason why the compilation process passes through the phase of intermediate code generation is twofold:

1. The source code is, most of the time, a high-level representation of the task the programmer wants to describe and tackle. By transcribing the source code to IR code, we simplify it in terms of form and, hence, make it easier to optimize it and/or produce machine code (e.g. assembly).

2. A uniform and simple intermediate and, most importantly, machine-independent representation of the source code is very useful in order to facilitate the retargeting of an application to different end systems, like different processor architectures or operating systems.

Regarding our work, we are interested in the first bullet; this simplified representation of an application is easier to parse, extract features from and, ultimately, *instrument*.

14.1.1 The LLVM project

The intermediate representation of programs is not universally standardized. LLVM [23] is an umbrella project composed of many tools to facilitate any compiling toolchain. In these tools, an intermediate language, the **LLVM bitcode**, and a C-family compiler, **clang**, are included. Actually, clang is the front-end tool that compiles the source code of applications to LLVM bitcode.[1]

`Clang` supports many source programming languages of the wider C family, including OpenCL. In our work, we use clang to compile OpenCL kernels to LLVM bitcode and then process it using **the LLVM C++ wrapper API**, a C++ library to parse, process and produce LLVM bitcode modules which is nothing more than a wrapper of the LLVM C API. The reason why the C++ API was used instead of the C API is that the latter is too low-level, introducing flexibility as well as complexity that were not needed in our case.

14.2 The concept of basic blocks

The source code and the intermediate representation of a task differ in many things, one of which is the concept of basic blocks. Basic blocks is one of the most important notions of any intermediate representation and aims at simplifying the optimization process and various types of analysis, like control flow and data flow analysis, as well as the generation of machine code.

An accurate and concise definition of the basic block can be found in [24], which is one of the dominant references for compiler theory and design. In [24], basic blocks are defined as "maximal sequences of consecutive IR code instructions with the properties that:

[1] In the official site of the LLVM project, it is mentioned that LLVM is not an acronym and does not mean anything. However, scattered references from the vastness of the past of the internet world imply that LLVM did use to be an acronym and stood for "Low Level Virtual Machine".

1. The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.

2. Control will leave the block without halting or branching, except possibly at the last instruction in the block."

In general, the intermediate representation of a program or an application (i.e. an IR module) is nothing more than a graph of basic blocks called **flow graph**, i.e. a directed graph with the basic blocks of the modules as its nodes and the jumps and branches between the basic blocks as its edges.

Simplifying the above definition to the extent that it almost ceases being one, and if we consider the execution system to be isolated and, therefore, uninterrupted, if the execution flow enters a basic block, it is a given that *all the instructions in it will be executed*. If the reader wishes to keep a single thing from this chapter it should be this certainty, which will later prove to be extremely helpful for the instrumentation component of `oclude` and the algorithm it implements.

14.3 Virtual Machines (VMs)

A last subject that we need to discuss before we depart from the land of compilers are virtual machines (or interpreters), close cousins of compilers.

Let us firstly consider a simple definition of a compiler. As mentioned in [24], "a compiler is a program that can read a program in one language, the *source language*, and translate it into an equivalent program in another language, the *target language*. [...] If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs". On the other hand, a virtual machine or "an interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to **directly execute the operations specified in the source program** on inputs supplied by the user". The difference between a compiler and a virtual machine is shown through their usage in Figure 14.2. Typical examples of virtual machines (or VMs) are the Java Virtual Machine (or JVM) and the Python interpreter.

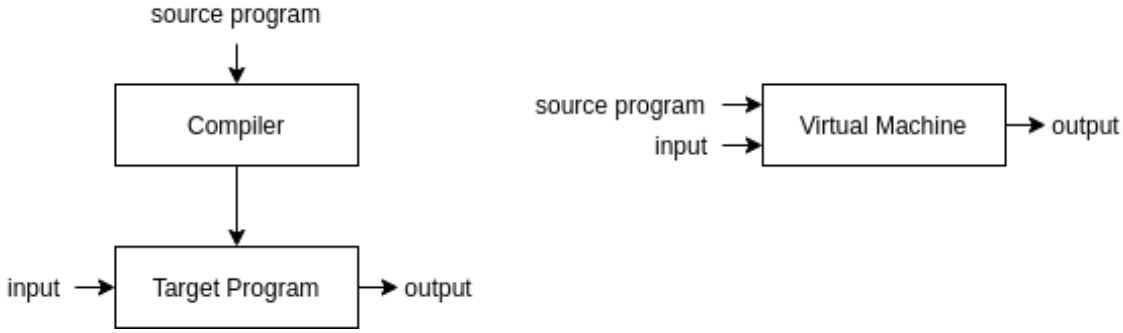

Figure 14.2: *Using a compiler (left) versus using a virtual machine (right)*

The reason why we had to mention virtual machines is that the instrumentation component of `oclude` is basically an OpenCL virtual machine; after having collected the instrumentation data from the source code, the instrumentation component of `oclude` closely resembles a virtual machine were the source code is parsed and each expression is transcribed, if necessary, in order to rewrite the source code in terms of basic blocks, as accurate as such a process can be when it comes to mixing high-level source code with the concept of basic blocks.

Chapter 15

Regression models

Regression is a statistical method, lately used extensively in the field of machine learning, that attempts to estimate the nature of the relationship between one or more *independent variables* and one or more *dependent variables*.

There are several methods with which regression can be used, the suitability of which relies heavily on the problem. In our work, there are many variables that relate to each other. To name them, we need to know what is the relationship, quantitatively and qualitatively, between:

- the size of the input of an OpenCL kernel and the LLVM instructions that were executed,
- the LLVM instructions that were executed and the execution time of a kernel and, finally,
- the size of the input of an OpenCL kernel and its execution time.

All of the above are very suitable problems for a regression algorithm. However, before choosing which algorithm to use, it is essential to investigate these relationships in a quantitative way first, and then choose the appropriate model. Does the relationship seem (or expected to be) linear? Super-linear? Is it polynomial? Something else? These questions need to be answered before proceeding to model selection. For the time being, however, we will present to the reader the four models that we used throughout our work, and ask them to keep them in mind. When the time comes, we will show and explain exactly how and why each one was selected.

15.1 Linear regression

The ***linear regression model*** fits an equation of the following form to the input data:

$$y = b_0 + b_1 x_1 + b_2 x_2 + ... + b_n x_n \tag{15.1}$$

where y is the independent variable, x_i are the dependent variables or features, b_i are the coefficients the model tries to estimate and n is the number of features. The goal of

the b_i estimation algorithm is to minimize the *mean squared error* between the calculated line and the actual data.

In our case, one of our regressions is **multioutput**, i.e. we have a vector $\vec{y}$ instead of a scalar y as our independent variable. More specifically, a linear model will be presented later that predicts executed instructions. Therefore, $\vec{y}$ will hold these counts (aka `instcounts`), and the length of $\vec{y}$ will be equal to the number of the instructions of the LLVM bitcode instruction set which, at the time of the writing, equals about 70. Moreover, this model will have *a single independent variable* as input, the `gsize` parameter.

Therefore, in our case, the equation we are trying to fit looks more like that:

$$\vec{y} = \vec{b}x \Leftrightarrow \begin{bmatrix} counter_{add} \\ counter_{sub} \\ counter_{mul} \\ \vdots \end{bmatrix} = \begin{bmatrix} coef_{add} \\ coef_{sub} \\ coef_{mul} \\ \vdots \end{bmatrix} gsize \tag{15.2}$$

and what we try to estimate is $\vec{b}$, the vector of the coefficients.

A last important note on linear regression is that it is ***strongly biased***, given that it assumes that the dependent variable is a linear function of the dependent variables.

15.2 Elastic Net regression

Elastic Net is a linear regression algorithm that uses *regularization* techniques in order to solve an ill-posed problem or to prevent overfitting (it should be noted that, through are work, there will be times when overfitting will be *needed*). Elastic Net attempts to minimize the same objective function as that of the simple linear regression (mean squared error), plus the $l1$ and $l2$ norms (namely Manhattan and Euclidean, respectively) of the coefficients vector:

$$error_{elastic\ net} = error_{linear\ regression} + \lambda_1 \|\vec{b}\| + \lambda_2 \sqrt{\|\vec{b}\|^2} \tag{15.3}$$

λ_1 and λ_2 are weights that regulate which norm to take more into account, and are the most important hyperparameters of the Elastic Net model. It should be noted that the actual equation is more complex and contains some more hyperparameters, but the above form is sufficient for the supervisory purpose of this Chapter.

By taking both these norms into consideration, Elastic Net attempts to shrink (or even eliminate) the contribution of certain features to the dependent variable and does a better job when dealing with correlated features, which is a plus in our case.

Further analysis regarding the algorithmic and mathematical properties of Elastic Net is considered to be outside the scope of this dissertation.

15.3 Polynomial regression

Polynomial regression tries to fit a polynomial to the training data instead of a straight line. It does so in 2 steps:

1. Computes the ***polynomial features*** of the training set, i.e. all the products and the powers of the features of the training set samples, up to a certain order. For example, if one of the samples is the following:

   $$[a, b, c]$$

   then the polynomial version of its features is the following (assuming a max order of 2):

   $$[\underbrace{1}_{\text{product of order 0}}, \overbrace{a, b, c}^{\text{products of order 1}}, \underbrace{a^2, b^2, c^2, ab, bc, ac}_{\text{products of order 2}}]$$

2. Uses a linear regression model, like the ones discussed above.

This way, the polynomial regression strategy assumes a linear relationship between the features (as before), ***their powers and their mixed products -up to a certain order-*** and the independent variable. This is why a polynomial regressor is defined by two parameters:

1. the maximum degree of the polynomial it will attempt to fit
2. the linear regression algorithm that will be used on the polynomial features

Therefore, we will be using two polynomial regressors in our work: one based on simple linear regression and another based on the Elastic Net model.

15.4 Why limit ourselves?

Linear and polynomial regression are very powerful and can be successfully used in the vast majority of regression problems, but there are far more sophisticated regression algorithms used, mainly in the machine learning community. The most prominent among these models are the ***tree-based regression algorithms***. However, such models (e.g. gradient boosting or even `XGBoost`) can not be used in our case. The reason is that no tree regressor is good at *extrapolating*, i.e. predicting values outside of the feature domain on which they were trained. This is crucial for our application, given that we need our regressors to predict `instcounts` for values of `gsize` that they have never seen, and possibly orders of magnitude larger.

Part VI

oclude - the profiler

Chapter 16

Using oclude

In the previous chapters, we covered everything we needed to present our work. The time has come; in this chapter, we are pleased to start our presentation of `oclude`, which we will approach in a top-to-bottom fashion.

`oclude` [19] is an open-source tool designed to run and test arbitrary standalone OpenCL kernels, without the need to write hostcode or specify its arguments. In that sense, we call `oclude` a ***profiler of standalone OpenCL kernels***. Besides simply running the OpenCL kernel, oclude can also:

- measure its execution time,
- count the instructions executed through an accurate and meaningful mapping from the OpenCL kernel code to the LLVM instruction set, created by using the clang toolkit,
- profile a specified OpenCL device.

Some of the most important characteristics of `oclude` are the following (each one will be thoroughly discussed in the appropriate chapter/section of this part):

1. **Source code instrumentation** of the input OpenCL kernel file via **a custom virtual machine (VM) for a subset of the OpenCL C kernel language**
2. **Inline creation of random inputs for arbitrary argument data types**, utilizing a seperate package for random values generation, also developed by the writer. These data types include primitive OpenCL data types, OpenCL vector data types, custom user data types (structs), and nested/encapsulated combinations of the above.
3. **An OpenCL hostcode wrapper** for arbitrary kernels based on the `PyOpenCL` API.

We will start by presenting the way `oclude` is designed to be used as a command line tool and as a Python package. Afterwards, in the next chapter, we will delve deeply into its inner proceedings and present all of its parts, how they work, and how they are interconnected. By presenting `oclude` that way, we hope that the reader will get familiar with *what* `oclude` is designed to do before getting to know *how* it does it.

16.1 As a command line tool

`oclude` can be used as a command line tool on Unix-based Operating Systems. Detailed instructions on its prerequisites and how to install it are included in its GitHub repository [19].

Starting from the highest level possible, `oclude` supports 2 commands, the ***device*** command:

```
$ oclude device <flags>
```

and the ***kernel*** command:

```
$ oclude [kernel] <flags>
```

Note that the `kernel` keyword can be ommited and, in that case, oclude defaults to the `kernel` family of its functionalities.

Given that our approach is a top-to-bottom one, we will start by presenting these two commands.

16.1.1 The "device" command

The `device` command is designed to profile general characteristics of a specific OpenCL device. This information can be useful in the case of an heterogeneous system where we want to gather information regarding all available OpenCL devices and take decisions (either manual or automatic) based on this intel. This mode has not been used in this work.

As an example, in case we wanted to profile the first device of the first OpenCL platform in our system, we would do the following in a terminal:

```
$ oclude device -p 0 -d 0
```

and get an output that would look like this:

```
[hostcode] Collecting profiling info for the following device:
[hostcode] Platform:    Intel(R) OpenCL HD Graphics
[hostcode] Device:      Intel(R) Gen9 HD Graphics NEO
[hostcode] Version:     OpenCL 2.1 NEO
[hostcode] Please wait, this may take a while...
Profiling info for selected OpenCL device:
        profiling overhead (time) - 0.011303499341011047
  profiling overhead (percentage) - 17.80%
                  command latency - 0.06351426243782043
  host-to-device transfer latency - 0.011074915528297424
  device-to-host transfer latency - 0.011512413620948792
device-to-device transfer latency - 0.06323426961898804
host-device bandwidth bandwidth @ 64 bytes - 0.005645181903735443 GB/s
```

```
host-device bandwidth bandwidth @ 256 bytes - 0.022125035974706695 GB/s
host-device bandwidth bandwidth @ 1024 bytes - 0.08657326467722175 GB/s
... a lot of bandwidth measurements follow ...
```

Obviously, the flags `-p` and `-d` correspond to the OpenCL platform and device of choice, respectively.

16.1.2 The "kernel" command

The `kernel` command is the main job `oclude` is designed to do - to profile OpenCL kernels. It supports *two different modes of operation*, apart from simply executing the kernel:

1. count **the LLVM instructions that were executed**, code-named ***instcounts***, and/or
2. measure **the execution time**, code-named ***timeit***.

The `kernel` command can be used with either, one, or none of these modes on.

An example of the `kernel` command in the `oclude` CLI could be the following (note that the `kernel` keyword is omitted as it is implied when absent and that, besides running the kernel, nothing else really happens):

```
$ oclude -f tests/rodinia_kernels/dwt2d/com_dwt.cl -k c_CopySrcToComponents -g 1024 -l 128
[oclude] INFO: Input file tests/rodinia_kernels/dwt2d/com_dwt.cl is not cached
[oclude] Running kernel 'c_CopySrcToComponents' from file <path omitted to fit>/com_dwt.cl
[hostcode] Using the following device:
[hostcode] Platform:    Intel(R) OpenCL HD Graphics
[hostcode] Device:      Intel(R) Gen9 HD Graphics NEO
[hostcode] Version:     OpenCL 2.1 NEO
[hostcode] Kernel name: c_CopySrcToComponents
[hostcode] Kernel arg 1: d_r (int*, global)
[hostcode] Kernel arg 2: d_g (int*, global)
[hostcode] Kernel arg 3: d_b (int*, global)
[hostcode] Kernel arg 4: cl_d_src (uchar*, global)
[hostcode] Kernel arg 5: pixels (int, private)
[hostcode] About to execute kernel with Global NDRange = 1024 and Local NDRange = 128
[hostcode] Number of executions (a.k.a. samples) to perform: 1
[hostcode] Kernel run completed successfully
```

Observe the following from the usage above:

- Firstly, an OpenCL kernel file (*.cl) is specified with the `-file/-f` flag.
- A kernel from inside this file is chosen with `-kernel/-k`. This is optional; if it is not used, `oclude` will inform the user of the kernels present in the input file and they will be able to choose which one to run interactively.
- The global and local OpenCL NDRanges are specified with the `-gsize/-g` and `-lsize/-l` flags, respectively. Only one dimension is supported for now, therefore these flags accept only a single positive integer.

Nothing interesting happened though... That is why the kernel command has two modes of operation.

16.1.3 The "instcounts" mode of operation

For the `instcounts` mode of operation of the `kernel` command, the `-inst-counts/ -i` flag is used to instrument the kernel and count the LLVM instructions that correspond to the instructions that were actually ran by the kernel (more on that on the next chapter):

```
$ oclude -f <path omitted to fit>/com_dwt.cl -k c_CopySrcToComponents -g 1024 -l 128 -i
[oclude] INFO: Input file tests/rodinia_kernels/dwt2d/com_dwt.cl is cached
[oclude] INFO: Using cached instrumented file
[oclude] Running kernel 'c_CopySrcToComponents' from file <path omitted to fit>/com_dwt.cl
[hostcode] Using the following device:
[hostcode] Platform:        Intel(R) OpenCL HD Graphics
[hostcode] Device:      Intel(R) Gen9 HD Graphics NEO
[hostcode] Version:     OpenCL 2.1 NEO
[hostcode] Kernel name: c_CopySrcToComponents
[hostcode] Kernel arg 1: d_r (int*, global)
[hostcode] Kernel arg 2: d_g (int*, global)
[hostcode] Kernel arg 3: d_b (int*, global)
[hostcode] Kernel arg 4: cl_d_src (uchar*, global)
[hostcode] Kernel arg 5: pixels (int, private)
[hostcode] About to execute kernel with Global NDRange = 1024 and Local NDRange = 128
[hostcode] Number of executions (a.k.a. samples) to perform: 1
[hostcode] Collecting instruction counts...
[hostcode] Kernel run completed successfully
Instructions executed for kernel 'c_CopySrcToComponents':
           26920 - load private
           20776 - alloca
           14336 - store private
           12288 - add
           11631 - getelementptr
           11264 - mul
            8855 - store callee
            7245 - load callee
            4096 - call
            3072 - load global
            3072 - load local
            3072 - store local
            3072 - zext
            2415 - sub
            1829 - br
            1024 - ret
            1024 - icmp
```

It should be mentioned that the output of this mode was designed to resemble that of `Oclgrind` [25], a tool that helped a lot during the first stages of our research as our dynamic features extractor. When we started to design `oclude`, it was much easier to keep the consumers of the output of `Oclgrind` that we had already implemented. Note that,

had it not being an OpenCL runtime emulator instead of actually running the input kernel on an OpenCL device, we would not have needed to develop `oclude` to begin with.[1]

16.1.4 The "timeit" mode of operation

For the `timeit` mode of operation of the `kernel` command, the `-time-it/-t` is used to measure the execution time of the specified kernel:

```
$ oclude -f <path omitted to fit>/com_dwt.cl -k c_CopySrcToComponents -g 1024 -l 128 -t
[oclude] INFO: Input file tests/rodinia_kernels/dwt2d/com_dwt.cl is not cached
[oclude] Running kernel 'c_CopySrcToComponents' from file <path omitted to fit>/com_dwt.cl
[hostcode] Using the following device:
[hostcode] Platform:        Intel(R) OpenCL HD Graphics
[hostcode] Device:        Intel(R) Gen9 HD Graphics NEO
[hostcode] Version:        OpenCL 2.1 NEO
[hostcode] Kernel name: c_CopySrcToComponents
[hostcode] Kernel arg 1: d_r (int*, global)
[hostcode] Kernel arg 2: d_g (int*, global)
[hostcode] Kernel arg 3: d_b (int*, global)
[hostcode] Kernel arg 4: cl_d_src (uchar*, global)
[hostcode] Kernel arg 5: pixels (int, private)
[hostcode] About to execute kernel with Global NDRange = 1024 and Local NDRange = 128
[hostcode] Number of executions (a.k.a. samples) to perform: 1
[hostcode] Collecting time profiling info...
[hostcode] Kernel run completed successfully
Time measurement info regarding the execution for kernel 'c_CopySrcToComponents'
(in milliseconds):
hostcode - 1.9354820251464844
  device - 0.013415999999999999
transfer - 1.9220660251464843
```

The two modes of the `kernel` command can be combined to measure the execution time of the instrumented OpenCL code.

16.1.5 The "samples" flag

One of the most important flags of the "kernel" command is the `-samples/-s` flag. This flag asks `oclude` to run the specified kernel *more than once*.

When using either of the `device` or `kernel` commands (or both), the results regarding the instruction counts and the different times that were measured are averaged across all runs. Keep in mind that this behavior occurs in the command line interface of `oclude` only. If used as a Python package, things are slightly different. Let us see exactly how different.

[1] This is not the whole truth; Indeed, Oclgrind is an emulator and, as mentioned in the "Usage" section in [25], "Since it is interpreting an abstract intermediate representation and bounds-checking each memory access, `Oclgrind` will run quite slowly (typically a couple of orders of magnitude slower than a regular CPU implementation). Therefore, it is recommended to run your application with a small problem if possible". But maybe this would not pose a problem in our microprofiling methodology (we will never know for sure, we migrated to the `oclude` solution very early in our research). What would certainly be a problem is the fact that `Oclgrind` **does not support vector types or custom user types**, as mentioned in its GitHub wiki. This is another reason why `oclude` had to be built.

16.2 As a Python package

By exporting a Python 3 package API, `oclude` can be used inside a Python script or project directly (e.g. in a Jupyter Notebook) to perform kernel profilings and experiments.

To achieve that, `oclude` offers three different entry points in the form of Python functions. For a more detailed description of their input arguments or the results they return, the reader is kindly requested to consult the GitHub wiki of the project [19]. These functions are the following:

1. The `oclude.profile_opencl_device` function exports the **device** command.
2. The `oclude.profile_opencl_kernel` function exports the **kernel** command.
3. The `oclude.get_opencl_kernel_static_instcounts` function exports a functionality which we have not mentioned before.

All three functions return Python dictionaries as results of their execution (again, for a thorough description of the functions, please visit the GitHub page of `oclude` [19]).

The first two functions return dictionaries that include as much entries as the number of runs defined through the `samples` argument, which corresponds to the `-samples/-s` flag of the command line interface of `oclude`. That means that, in contrast to what happens in the command line interface, the ***Python package interface does not average results***. Therefore, if extensive and not aggregated results are needed, the Python package interface is the way to go. That is why we used this interface to get the data for the experiments that are presented in the `OCLMan` part of this volume.

The last function returns a dictionary containing ***static features of the OpenCL kernel file source code, i.e. instruction counts regarding the intermediate representation (IR) of the source code alone***. This functionality is offered through the Python package interface only and could be useful for analyzing data and creating models based solely on static features. This could help in at least two different ways:

- Use such a model to tackle the problems mentioned in the Introduction, as per the standard approach on the subject.
- Use such a model as a baseline to assess one built on dynamic features.

In case the reader did not understand it immediately, we are going to use this `oclude` functionality for the second reason. More on that on the last part.

Chapter 17

Building oclude

As you may already know or have realized by now, profiling standalone OpenCL kernels is not a trivial task at all. There are many questions that have to be answered before moving into designing and implementing such a tool:

- *How complicated kernels should the tool be able to handle?*
- *How much freedom can we grant to the user regarding the kernel arguments? e.g. as we have seen, OpenCL allows a user to define their own custom data types, even nested ones. Are we able and/or do we want to handle that?*
- *Where should we intercept and instrument the execution process in order to be able to count the executed instructions? On dispatch to the device through the queue? On the device itself during the execution? Maybe before even starting to execute the hostcode? Or a combination of all/some of these?*
- *How to enrich the profiling process with useful data regarding the kernel execution(s)? What information is safe to dismiss?*
- *and much, much more...*

By presenting the entire `oclude` architecture, starting with an overview and then part by part, we hope to achieve a dual goal; on the one hand, to help the reader delve deeper into the gears of `oclude` and really understand the way it is structured and implemented and, on the other hand, to convey the choices we made and consider worth-mentioning while designing such a complicated utility, as well as the insights we gained in the virtues and needs of the world of heterogeneous computing and its software.

17.1 An architectural overview

Let us start with an overview of `oclude` the building blocks of `oclude` and how they are interconnected.

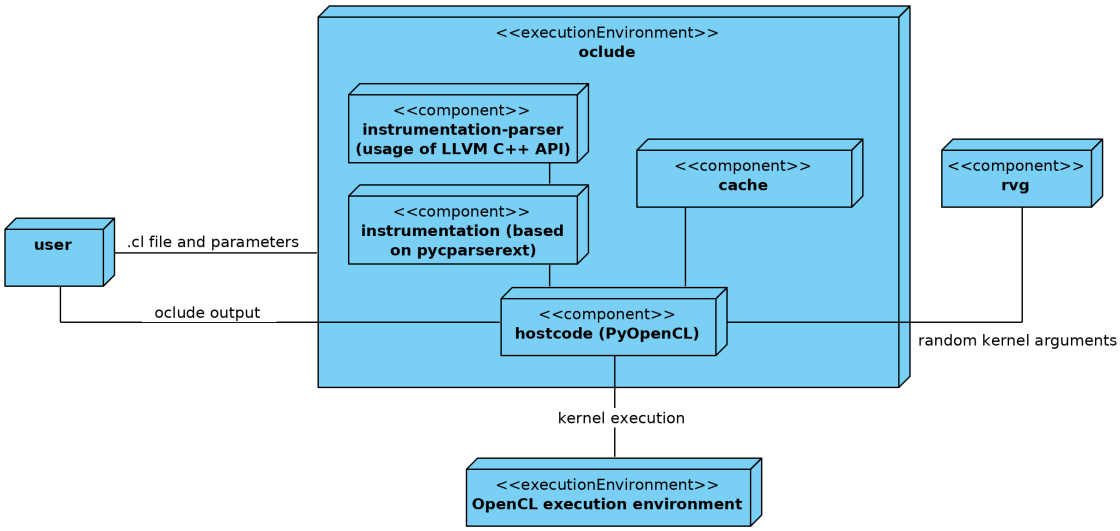

Figure 17.1: *A UML deployment diagram of oclude*

In the deployment diagram above, all the major components of `oclude` are shown. We can see that the user provides an OpenCL kernel source code file to `oclude`, as well as values for `oclude` parameters, e.g. some of the flags that we previously mentioned. After that, four components work together in order to prepare the kernel for dispatch to the OpenCL execution environment, which is a device on the system (and not a part of `oclude`).

Before the dispatch to the OpenCL device, some steps need to be made. Firstly, ***the instrumentation component*** is responsible for injecting instrumentation code in the provided source code, in order for the counting of the executed instructions to take place. To extract the information needed to instrument the kernel, the instrumentation component uses ***the instrumentation-parser component***, which parses the LLVM bitcode of the input file, counts the (static) instruction counts and organizes them into basic blocks. It should be noted that we need to go only this far if the user has requested a static profiling of the input kernel. After that, the instrumented source code is dispatched to the OpenCL execution environment by ***the hostcode component***, which is responsible for the execution of the given kernel. Last but not least, the execution results are collected, formatted and returned to the user.

The observant reader should have noticed that we completely omitted ***the cache component*** that is depicted in the UML diagram above from our architectural overview. No need to worry; the cache component is quite autonomous and is not crucial for understanding the way oclude works. Therefore, we will save it for last.

17.2 A functional overview

Before moving on into breaking down each one of the components discussed above, a functional overview of all the scenarios for which `oclude` was designed would be quite helpful in order to fully grasp the way `oclude` is meant to be used and operate.

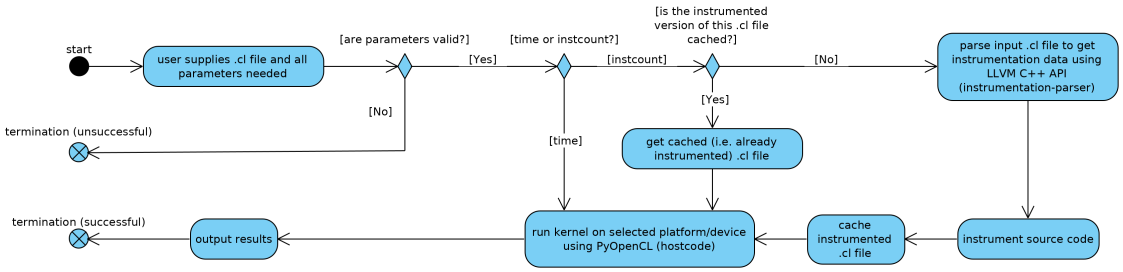

Figure 17.2: *A UML activity diagram of oclude*

It is reminded that we are not going to talk about the cache functionality yet, although it is depicted for the sake of completeness.

After the user supplies the OpenCL kernel source code file, `oclude` checks whether the provided information is valid. If it is, `oclude` checks whether an instruction counting or a time measurement was requested. In the case of the latter, then `oclude` needs only to run the provided kernel with random arguments, and report the hostcode, device and transfer times. In the case that an instruction counting was asked, then a query is sent to the cache. We will ignore this for now and assume that the cache does not exist, hence following the "No" arrow. Then, the instrumentation phase begins, in which the source file is compiled to LLVM IR bitcode that is parsed, in order to extract all the information needed to instrument the original source code, and send that over for execution.

Note that the full path will be followed whether the user asked for an instruction counting or for both measurements (instruction counting and time measurement).

We feel the need to turn once again to the perceptive reader, and inform them that the static feature extraction scenario was omitted from the above UML diagram on purpose, for simplicity.

17.3 The components of oclude

We are now ready to dissect `oclude` and find out how each of the components is designed, how it is implemented, what it does, and how `oclude` glues them all together.

17.3.1 The instrumentation component

The instrumentation component is the biggest and most complex component of `oclude`, and it is what differentiates `oclude` from a simple standalone OpenCL kernel driver.

The instrumentation component:

1. Receives the OpenCL kernel and several parameters from the user.

2. If the user did not request an instruction count, e.g. they requested a time measurement only, using the `-time-it/-t` flag, then this component is *transparent*; it passes everything to the hostcode component.

3. If the user requested an instruction count, using the `-instcounts/-i` flag, then it proceeds.

4. Checks whether the instrumented version of the input source file is cached (see 17.3.4). If it is, then the instrumentation component passes the cached instrumented source code to the hostcode component. If not, it *compiles the input source code to LLVM bitcode* using the `clang` compiler infrastructure.

5. It passes the LLVM bitcode to the instrumentation-parser component (see 17.3.2) to extract the static instruction counts per basic block.

6. It uses this information to insert the following instrumentation to the source code that the user provided:

   (a) **Two hidden counters as kernel arguments**. These counters will serve as the instruction counters, therefore the length of these counter buffers is equal to the number of the instructions of the LLVM instruction set, which is about 70 at the time of the writing (August 6, 2020). More specifically, The first buffer is **a local buffer**, which means that each work-group has a different one and is responsible for its initialization. Each work-item of a specific work-group accesses the local counter only, in order to record instruction counts. After the end of the execution of the kernel, the *leader* of each group (i.e. the work-item with local id equal to 0) is responsible to add the counts of the group to the overall counts, by writing to the second buffer, **the global one**, which is the buffer that is ultimately read by the hostcode and contains the `instcounts` that are returned to the user. All of the write accesses to the counter buffers described above are obviously *atomic*. The reason for the existence of two buffers instead of just the global one, is to speed up the process by limiting the accesses to the global memory and distribute the load of book-keeping across the work-groups. As a note, helper functions use the local counter buffer only.

   (b) **An initialization preamble**, where the leader of each work-group initializes the local buffer to 0.

   (c) **An aggregating epilogue**, where the leader of each work-group adds its local counter buffer to the local counter buffer, atomically.

   (d) **Increments of the appropriate locations of the local counter buffer at the start of each "basic block"**. We use quotes around the term basic block because, as we said in Chapter 14, we can not talk about basic blocks in the source code; basic block are a feature of the intermediate representation languages. Therefore, we have designed a ***mapping effort between the LLVM bitcode (IR) basic blocks and the source code***. This process was

the toughest thing we had to do in `oclude`, technically speaking. As a simple example, imagine the following conditional:

```
if (a && b || c) { ... }
```

This simple conditional is translated, in fact, into **three basic blocks**; one for the resolution of each variable of the conditional (omitting the basic blocks of the taken/non-taken paths). Therefore, it should have become clear by now that **a virtual machine for (a subset of) OpenCL** had to be implemented, in order to deal with each possible expression and instrument it in the proper way, taking into consideration the way that each expression is mapped into basic blocks in the LLVM intermediate representation. For obvious reasons, we will not delve deeper into this virtual machine we created. The interested reader is kindly referred to the `oclude` repository [19].

7. After the instrumentation process is completed, the instrumented kernel is handed over to the hostcode component.

17.3.2 The instrumentation-parser component

The instrumentation-parser component is, in fact, a subcomponent of the instrumentation component, which is responsible for parsing the LLVM IR bitcode of the input source file, and report back (static) instruction counts, organized in basic blocks. Moreover, it should be noted that it is the only part of the current version of `oclude` (as of August 6, 2020) that is written in C++.

Before moving on, it is interesting to note that an effort was made to rewrite this component in Python in order to make `oclude` a pure Python project. All attempts lead to disappointing and fruitless results, though, because `llvmlite`, part of the `Numba` project [26] and, apparently, the only maintained (up until the moment of the writing, August 6, 2020) Python API for LLVM, has decided to completely decouple the parsing of a source file and the processing (even visiting) of the resulting AST. Therefore, we had no alternative but to use the LLVM C++ wrapper API.

17.3.3 The hostcode component

The hostcode component is the functional heart of `oclude`; it is the interconnection between `oclude` and the actual OpenCL runtime environment of the system.

It is interesting to note that, like the instrumentation-parser component, it used to be implemented in C++ during the early development stages of `oclude`. Although, at a point it became clear that the complexity of the needs of `oclude` going hand by hand with the low-level-ness of the C and C++ OpenCL APIs, was an explosive combination. In the light of this realization, we migrated the whole component to PyOpenCL [20] which, at the time of the writing (August 6, 2020), is the core of the `oclude` hostcode component.

The hostcode component is responsible for:

1. receiving the (possibly instrumented) OpenCL source code and the name of the kernel of interest,

2. randomly initializing the arguments of the kernel using the external `rvg` component [21],
3. running the kernel on the specified device as many times as the user requested,
4. getting back the results (instruction counts and/or time measurements),
5. calculating their average if necessary, and
6. returning the final results to the user.

As a final note, the latest `PyOpenCL` version that was available at the time of building `oclude` was version `2019.1.2`. This version did not support user defined structs with array members as kernel arguments (or any nested types other than primitive OpenCL types inside of structs, for that matter). Therefore, a contribution had to be made to add this feature, given that there were OpenCL kernels in the Rodinia Benchmark Suite that had such structs. The `PyOpenCL` maintainer accepted this contribution and is now a part of version `2020.1` that was used in `oclude`. The current version (August 6, 2020) is `2020.2` and still includes this feature.

17.3.4 The cache component

The cache component is an addition to the architecture of `oclude` designed to facilitate experiments were repeated profiling of the same kernels is necessary. One such experiment is our work.

The cache is merely a directory were the instrumented version of a kernel is stored, when it is handled by `oclude` for the first time. The unique identification of this (instrumented) file is considered to be the *hash value* of the *original* version of the file. That way, when this kernel (or, more precisely, the source code file where this kernel is defined) is seen again by `oclude`, there is no need to re-instrument it. Note that the instrumentation process is the most time consuming part of the `oclude` pipeline. Therefore, when a user wants to conduct tens or hundreds of different experiments on the same kernel for different values of `gsize` and/or different inputs (like we needed), a huge amount of time is saved.

Under the light of the above, the reader is kindly encouraged to revisit the UML Deployment 17.1 and Activity 17.2 Diagrams and observe the role of the cache component as depicted there.

Chapter 18

On the necessity of the existence of `oclude`

As mentioned before, the problem of profiling OpenCL applications is very popular and studied. Therefore, it is reasonable and expected that there are already tools to do this job. In this Chapter, we present two such tools, `Oclgrind` and `cldrive`, both of which were used during the early stages of our research (i.e. before building `oclude`).

18.1 Other OpenCL kernel profiling tools

18.1.1 Oclgrind

In the abstract of the `Oclgrind` paper [25], it is described as "*a platform designed to enable the creation of developer tools for analysis and debugging of OpenCL programs. `Oclgrind` simulates how OpenCL kernels execute with respect to the OpenCL standard, adhering to the execution and memory models that it defines.*" Furthermore, `Oclgrind` reports the total LLVM instructions that were (simulated to have been) executed by the specified OpenCL kernel. In fact, the format of the output of `oclude` was designed to ressemble that of `Oclgrind`.

Even though `Oclgrind` is an extremelly user-friendly and helpful tool, seemingly doing exactly what we want to do in our work, it remains a *simulator*. This means that `Oclgrind` is *device-agnostic*, something that goes against the goals of our work. But even if we could find a way to take the OpenCL device of interest into consideration in another way, one more problem remains: `Oclgrind`, being a simulator, is quite slow for higher values of `gsize`. `oclude`, on the other hand, was designed as an actual driver for arbitrary kernels, as well as `gsize` values.

18.1.2 cldrive

`cldrive` "textitis a tool for running arbitrary OpenCL kernels to record their runtimes and outputs. It reads OpenCL kernels from an input file, and for each, generates random inputs (parameterized by a given size), runs the kernel and records its execution time and outputs." [27].

`cldrive` would be the ideal profiler for our worker if it counted LLVM instructions, which it does not. But, even if we ignore that, using it was a very cumbersome experience. One of the side goals of `oclude` was to be as user-friendly as possible.

18.2 A cross-comparison

These two tools, `Oclgrind` [25] and `cldrive` [27], are the best standalone OpenCL kernel drivers that we found. Unfortunatelly, they lack some characteristics that were crucial for our work. Therefore, `oclude` turned out to be a necessity.

For the shake of brevity, these characteristics are presented in the cross-reference Table 18.1, were it becomes crystal clear why `oclude` had to be created. Note that each and every one of the features listed in the leftmost column was considered as nothing less than an absolute necessity for `oclude`.

	`Oclgrind`	`cldrive`	`oclude`
installation process	cumbersome (source) straightforward (binary)	cumbersome (uses bazel)	straightforward (uses make)
usage process	straightforward (command line utility)	cumbersome (bazel run)	straightforward (Python 3 script)
execution environment	simulator	underlying OpenCL env.	underlying OpenCL env.
hardware	agnostic	dependent	dependent
speed	very slow (simulator)	fast (runs on device)	fast (runs on device)
caching of results	✗	✗	✓
static instruction counts	✗	✗	✓
dynamic instruction counts	✓	✗	✓
execution time measurement	not applicable	✓	✓
OpenCL vectors as kernel arguments	✗	✓	✓
custom user types as kernel arguments	✗	✓	✓

Table 18.1: *A cross-comparison between* `oclude` *and other tools*

Part VII

OCLMan - the predictor

Chapter 19

Gathering and analyzing data

After having presented `oclude` in the previous chapter, we are ready to use it in order to profile some OpenCL kernels and draw some conclusions which will guide us through the process of designing prediction models later on.

To achieve the above, we are going to use the OpenCL kernels from the Rodinia Benchmark Suite [18]. Rodinia includes roughly about 60 OpenCL kernels which implement algorithms originating from various fields, like computer vision and medical imaging.

We will start by presenting the experimental procedure that we will follow. We will then investigate the general behavior of the Rodinia OpenCL kernels and categorize them for later reference. Lastly, after profiling the kernels, we will conduct an explaratory data analysis (EDA) on our results, in order to draw some conclusions and gain much needed insights for the design process that will follow in the next Part of this volume.

19.1 The experimental procedure

Each kernel from the Rodinia benchmark suite was treated separately, even those that were in the same file.

From these, some could not be profiled with `oclude` due to frequent segmentation faults, which most probably were caused by the numerical values of the input buffers. At this point, it should be reminded that `oclude` initializes the kernel arguments in a random way, taking into consideration the argument type only (e.g. it will never initialize an unsigned type with a value below zero). It is also reminded that an external system, the `rvg` package, is responsible for this randomization process [21]. An OpenCL kernel, though, may rely on the input data meeting certain requirements, either regarding the numerical values of the input or even the buffer sizes. For example, a kernel may take for granted that its first argument has twice the size of the second. It is obvious that, in such a case, segmentation faults are to be expected and `oclude`, in its current development status, can not do anything about them.

From the kernels that *could* be profiled (i.e. their execution did not lead to -frequent-segfaults), a number of profiling samples was taken. More precisely:

- First of all, the number of samples (flag `-samples/-s`) was kept constant and equal to 100 for all experiments described in this section (see Appendix A).

- The only variable of the profiling experiments was the global NDRange (flag `-gsize/-g`), what we have mentioned before as ***gsize***. The local NDRange was not defined, which means that it was automatically and (nearly) optimally selected by the OpenCL runtime.

- All kernels were profiled for gsizes starting from 100 and upwards, with a step of 50. The profiling process was terminated in case the 100 samples took longer than 3 seconds to run (flag `-timeout/-x`) or the size reached a value of 100000. Only the former happened.

- Some kernels were too slow to get 100 samples in less than 3 seconds for a gsize greater than 100, or we managed to get only a single digit number of 100-samples profilings. For these kernels (labeled "relatively slow"), the range from 10 to 100 was used regarding the gsize, with a step of 5. All the other kernels are labeled "relatively fast".

- The values of gsize and the timeout flags were manually fine-tuned when needed, in order to be certain that at least 20 100-samples profilings were successfully acquired from each kernel.

- Each profiling corresponds to a dictionary, the return value of the `profile_opencl_kernel` function of `oclude`. Every such profiling was stored as a JSON file for subsequent processing.

19.2 Kernel profiling general statistics

The first step of the experimental process was to gain an overview of the Rodinia kernels general behavior. A pie chart showing the behavior of the OpenCL kernels during our experiments follows.

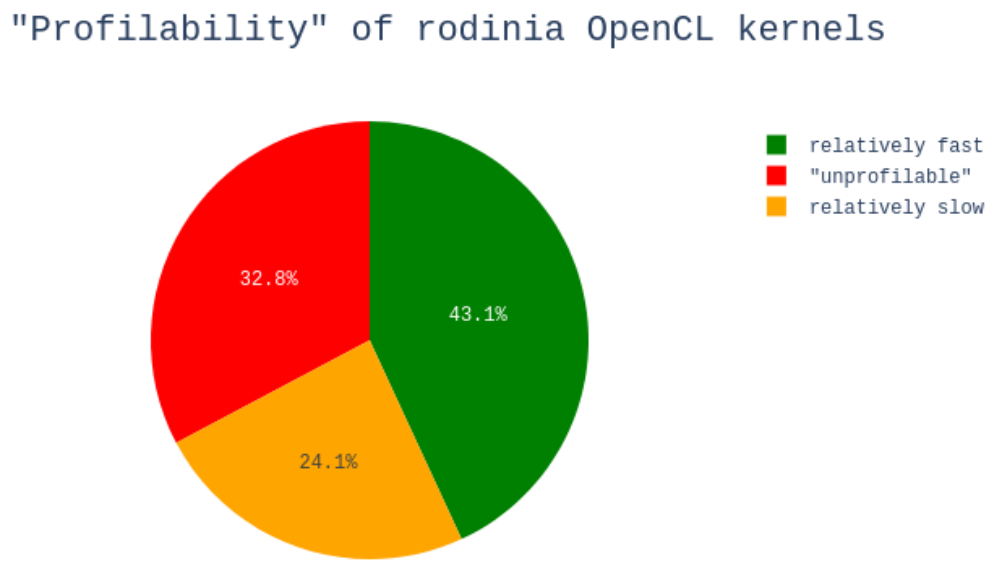

Figure 19.1: *"Profilability" of all OpenCL kernels from the Rodinia Benchmark Suite*

We can see that the vast majority (**43.1%**) of the kernels are "relatively fast", meaning that we managed to perform tens or even hundreds of 100-samples profiling experiments on them. They represent the majority of the Rodinia Benchmark Suite and, as we will see later, the execution time and instructions tend to have a relatively linear relation to gsize in these kernels.

More specifically, the following table shows how many 100-sample profilings were performed on each kernel. Note that the data below refer to the "profilable" (i.e. "relatively fast" and "relatively slow") kernels, not the "unprofilable" ones.

rodinia benchmark	benchmark file	kernel name	# profilings	smallest gsize	largest gsize
b+tree	kernel_gpu_opencl.cl	findK	39	10	200
b+tree	kernel_gpu_opencl_2.cl	findRangeK	21	10	110
backprop	backprop_kernel.cl	bpnn_adjust_weights_ocl	382	100	19150
backprop	backprop_kernel.cl	bpnn_layerforward_ocl	50	100	2550
bfs	Kernels.cl	BFS_1	178	10	895
bfs	Kernels.cl	BFS_2	894	100	44750
dwt2d	com_dwt.cl	c_CopySrcToComponent	550	100	27550
dwt2d	com_dwt.cl	c_CopySrcToComponents	573	100	28700
gaussian	gaussianElim_kernels.cl	Fan1	903	100	45200
gaussian	gaussianElim_kernels.cl	Fan2	623	100	31200
heartwall	kernel_gpu_opencl.cl	kernel_gpu_opencl	33	10	170
hotspot	hotspot_kernel.cl	hotspot	44	10	225
hotspot3D	hotspotKernel.cl	hotspotOpt1	49	10	250
hybridsort	bucketsort_kernels.cl	bucketprefixoffset	123	10	620
hybridsort	bucketsort_kernels.cl	bucketsort	77	100	3900
hybridsort	histogram1024.cl	histogram1024Kernel	20	5	100
hybridsort	mergesort.cl	mergeSortFirst	288	100	6350
kmeans	kmeans.cl	kmeans_kernel_c	22	10	115
kmeans	kmeans.cl	kmeans_swap	133	10	670
lud	lud_kernel.cl	lud_internal	33	10	170
myocyte	kernel_gpu_opencl.cl	kernel_gpu_opencl	893	100	44700
nn	nearestNeighbor_kernel.cl	NearestNeighbor	723	100	36200
particlefilter	particle_double.cl	find_index_kernel	173	100	8700
particlefilter	particle_double.cl	likelihood_kernel	44	10	225
particlefilter	particle_double.cl	normalize_weights_kernel	183	100	9200
particlefilter	particle_double.cl	sum_kernel	1373	100	68700
particlefilter	particle_naive.cl	particle_kernel	194	100	9750
particlefilter	particle_single.cl	find_index_kernel	173	100	8700
particlefilter	particle_single.cl	likelihood_kernel	49	10	250
particlefilter	particle_single.cl	normalize_weights_kernel	183	100	9200
particlefilter	particle_single.cl	sum_kernel	1083	100	54200
pathfinder	kernels.cl	dynproc_kernel	42	10	215
srad	kernel_gpu_opencl.cl	compress_kernel	803	100	40200
srad	kernel_gpu_opencl.cl	extract_kernel	833	100	41700
srad	kernel_gpu_opencl.cl	prepare_kernel	773	100	38700
srad	kernel_gpu_opencl.cl	reduce_kernel	53	100	2700
srad	kernel_gpu_opencl.cl	srad2_kernel	453	100	22700
srad	kernel_gpu_opencl.cl	srad_kernel	413	100	20700
streamcluster	Kernels.cl	memset_kernel	1150	100	57550

Table 19.1: *A quantitive overview of the microprofiling process results*

We can observe that another way of differentiating between "relatively fast" and "relatively slow" kernels is that the former are the ones the largest `gsize` of which is greater that 1000, while the latter are the ones the smallest `gsize` of which is smaller than 100.

19.3 An explanatory data analysis (EDA)

We are now ready to investigate the relationship between `gsize` and the instructions executed from an OpenCL kernel (i.e. `instcounts`). To achieve this, we are going to use the data we acquired with the help of `oclude`.

It is reminded that each of the profilings of the kernels refers to a single `gsize` value and consists of 100 samples, taken with different random initialization of the kernel input data (i.e. the value(s) of its arguments). In that way, we are able to:

1. inspect a single execution of a specific kernel for a specific `gsize`, or

2. inspect the 100-samples profiling as a whole (e.g. by taking the average instructions that were executed, as we have mentioned in Chapter 16), in order to smooth out the effects of randomness regarding the arguments initialization (see Appendix A) and/or to study how "predictable" or "well-behaved" the kernel is, in terms of the variance of the instructions executed for the same `gsize`.

We will focus on the second approach.

19.3.1 Regarding all kernels

In this section, we will work on all kernels and start gaining some insight on what is going on. For this section, we will work with the *average* executed instructions for each profiling and not for each one of the 100 measurements separately.

As a start, we are interested into how the `gsize` parameter and `instcounts` are correlated. This is a very valuable insight, because it will let us know:

1. whether the executed instructions (`instcounts`, our dependent variables) are correlated with `gsize` (our independent variable)

2. whether the executed instructions are correlated with each other

It is easy to do so for a single kernel (i.e. by calculating the respective correlation matrix), but it is not obvious how to gain an insight regarding *the whole benchmark suite*. To be more specific, it is easy to answer the following question:

"How is instruction X correlated with instruction Y in kernel K?"

But it is not obvious how to answer the (much more interesting) question:

"How is instruction X correlated with instruction Y ***across all kernels****?"*

To answer the latter, we will plot the ***average correlation matrix*** across all kernels. More specifically:

1. We will compute the correlation matrices for every kernel

2. We will take the average of these matrices

That way, we will learn how each instruction is correlated with every other instruction across all kernels *in average*.

Figure 19.2: *The average correlation matrix of LLVM instructions executed across all kernels*

From the aggregating heatmap above, it is clear that:

1. the executed instructions (our dependent variables) are **strongly correlated** with `gsize` (our independent variable)
2. `instcounts` (our dependent variables) are **strongly correlated** with each other

These are great news indeed, because we are led to the following conclusions:

1. The strong correlation between `instcounts` and `gsize` means that we can expect to be able to predict executed instructions given the `gsize` fairly accurately.
2. The strong correlation between `instcounts` means that there are groups of "similarly escalating" instruction counts (e.g. groups that are alltogether scaling linearly, squarely etc).

Combining the above, we can draw even more interesting conclusions regarding the groups of "similarly escalating" instruction counts. For example, if such a group correlates

strongly with `gsize`, we can safely assume that the instructions of this group scale linearly, given that `gsize` scales linearly.

For now, we will study the correlation between `instcounts` and `gsize` only, because we want to focus on our first main goal, which is to predict `instcounts` given a `gsize`.

A list of the correlation of `instcounts` and `gsize` follows, in descending strength, along with the number of kernels in which they have appeared.

Table 19.2: *Correlation of each type of LLVM instruction executed per kernel with* `gsize`

Table 19.3: *sorted by number of kernels* Table 19.4: *sorted by correlation with* `gsize`

instruction	# kernels	corr. with gsize	instruction	# kernels	corr. with gsize
alloca	39	0.997600	udiv	2	1.000000
trunc	39	0.993089	shl	1	1.000000
store private	39	0.986098	sdiv	6	0.999330
call	39	0.982756	lshr	4	0.998961
ret	39	0.972261	alloca	39	0.997600
load private	39	0.968990	insertelement	1	0.994463
getelementptr	39	0.874323	extractelement	1	0.994463
icmp	38	0.977983	trunc	39	0.993089
br	38	0.973891	fptoui	1	0.992097
sext	38	0.905347	srem	9	0.990848
load global	38	0.860842	phi	4	0.989927
add	36	0.925258	store private	39	0.986098
store global	33	0.915318	call	39	0.982756
mul	26	0.978386	load local	12	0.982649
sub	16	0.913548	or	1	0.978429
fmul	15	0.815356	mul	26	0.978386
store local	13	0.879171	icmp	38	0.977983
fdiv	13	0.832970	br	38	0.973891
load local	12	0.982649	ashr	1	0.972627
fadd	12	0.822982	ret	39	0.972261
fsub	10	0.842560	load private	39	0.968990
fcmp	10	0.842339	zext	9	0.961617
srem	9	0.990848	add	36	0.925258
zext	9	0.961617	fptosi	4	0.922889
store callee	9	0.869554	store global	33	0.915318
load callee	9	0.868066	sub	16	0.913548
sitofp	8	0.813739	sext	38	0.905347
sdiv	6	0.999330	store local	13	0.879171
lshr	4	0.998961	getelementptr	39	0.874323
phi	4	0.989927	store callee	9	0.869554
fptosi	4	0.922889	load callee	9	0.868066
fptrunc	4	0.599275	load global	38	0.860842
fpext	4	0.598968	and	2	0.851791
udiv	2	1.000000	fsub	10	0.842560
and	2	0.851791	fcmp	10	0.842339
shl	1	1.000000	fdiv	13	0.832970
insertelement	1	0.994463	fadd	12	0.822982
extractelement	1	0.994463	fmul	15	0.815356
fptoui	1	0.992097	sitofp	8	0.813739
or	1	0.978429	fptrunc	4	0.599275
ashr	1	0.972627	fpext	4	0.598968
fneg	1	0.115662	fneg	1	0.115662

We observe that all the instructions that appear in multiple kernels (i.e. they are of greater significance to us) are strongly correlated with `gsize`. We obviously do not need to worry about instructions like `fptrunc`, `fpext` and `fneg` which have quite weaker correlations with `gsize` (especially the later), given that they appear in 4, 4 and 1 kernel respectively, and thus are insignificant for our analysis.

19.3.2 Regarding the "relatively fast" kernels

We will continue our analysis with the "relatively fast" kernels. Specifically, we want to show that their fastness is a result of ***a linear relationship between `gsize` (independent variable) and `instcounts` (dependent variables)***.

Let us visit the average correlation matrix again but, this time, regarding the "relatively fast" kernels only.

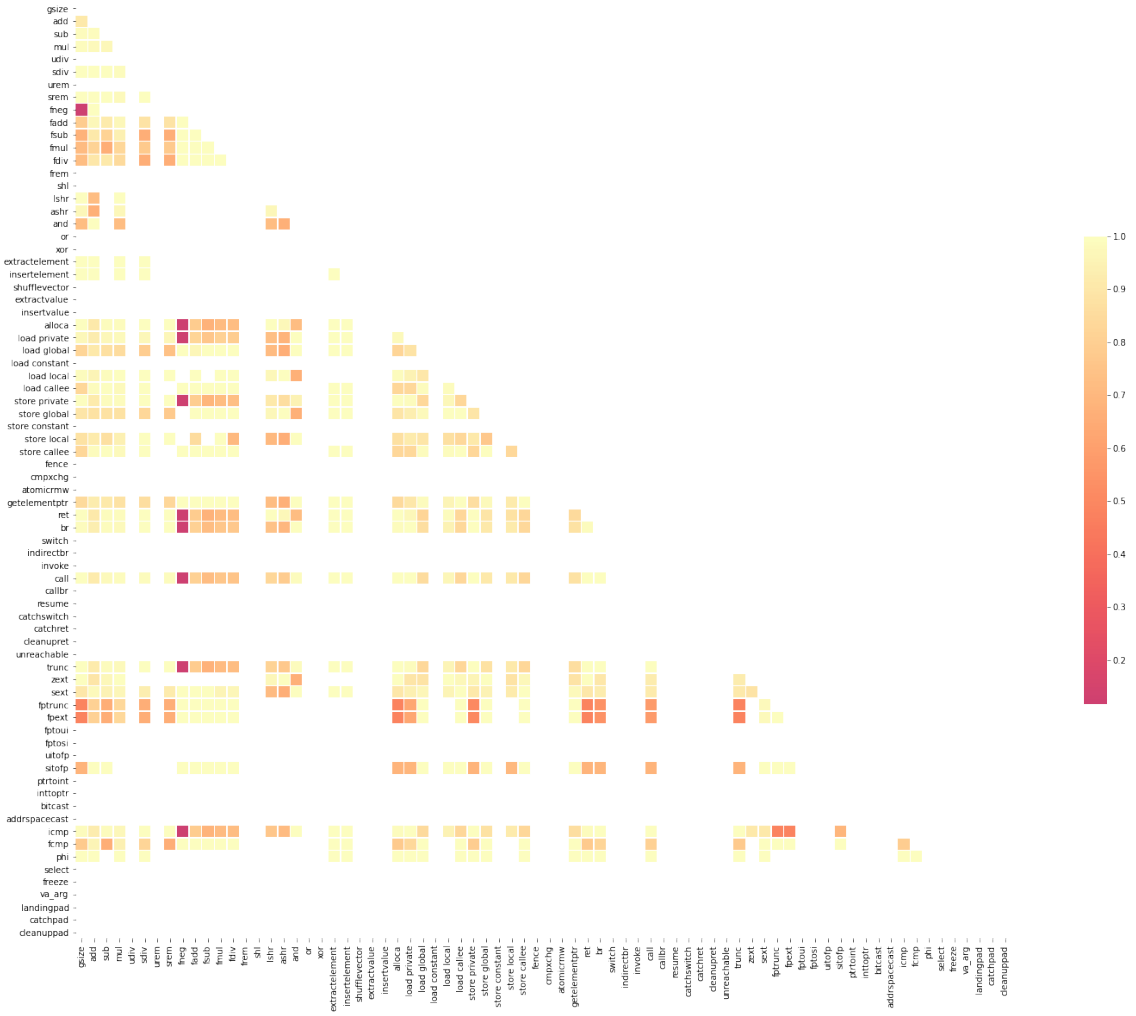

Figure 19.3: *The average correlation matrix of LLVM instructions executed across the "relatively fast" kernels only*

Obviously excluding the `fneg` instruction, as well as the `fptrunc` and `fpext` instructions (which tend not to scale at all and rather stay constant in the kernels in which they exist), every other correlation between `gsize` and `instcounts` and between `instcounts` themselves is generally strong.

To get a better glimpse at what is actually going on, we believe that it would be helpful to plot some `instcounts` against `gsize`. For the sake of brevity, we will plot only the following instructions:

1. `store global`: It correlates extremely strongly with every other instruction and is

very important in our analysis, given that accesses to the global OpenCL memory address space are one of the most expensive operations in terms of execution time.

2. `br`: It is one of the possible instructions we can use -along `call` and `ret`- to study branching, another expensive operation. Obviously, `call` and `ret` correlate extremely strongly, given that whenever the former happened, the later must follow. We can see from the heatmap 19.3 that `call`, `ret` and `br` form one of the many strongly correlated groups of instruction counts. Furthermore, it can be shown that `call` and `ret` are ***absolutely linear functions of gsize*** (the corresponding diagrams are omitted for brevity). Therefore, `br` is "more interesting" than `call` and `ret` at this point in our analysis.

Having all of the above in mind, we will plot `store global` and `br` for three indicative "relatively fast" kernels. The Figure 19.4 is quite big and is therefore locate at the end of this Section.

The first observation we can make is that `store global` and `ret` are ***near-linear or linear functions of*** `gsize`. This is an extremely important observation regarding the construction of regression models later; it may mean that we need to focus on linear models only.

With a cooler approach, we have to record the following observations (some of which refer to kernels that we omitted from Figure 19.4 for brevity):

1. In some kernels, ‘store global‘ and/or ‘br‘ have **zero counts**.

   (a) Regarding `store global`, this may seem odd; what is the utility of an OpenCL kernel if it does not write data somewhere where the hostcode can retrieve them? It turns out, however, that some of them, e.g. the `dwt2d` kernels, are actually helper functions for other kernels, while others like the `myocyte` kernel use other helper functions to write to their global buffers. The store operations associated with the later behavior are categorized as `store callee`, because `oclude` can not infer the OpenCL address space of a memory operation when it occurs outside of a kernel function. Every kernel that exhibits this behavior belongs to one of the two categories mentioned above.

   (b) Regarding `br`, only the `streamcluster` kernel does not record any, and this is because it actually does not include any branching. This is easily deduced by looking at its source code:

```
__kernel void memset_kernel(__global char * mem_d, short val, int number_bytes){
    const int thread_id = get_global_id(0);
    mem_d[thread_id] = val;
}
```

2. There are relations that are ***absolutely linear***. In fact, all relations seem to be linear, ***with more or less noise*** (see Appendix A).

3. The `srad` benchmark exhibits some ***worrying behavior*** regarding the relation between the executed instructions we plotted and `gsize`. More specifically, it also

follows a linear-like trend by tends to spike a lot. Could this behavior be more generalized than it seems here? How can such relations be regressed and predicted?

We will keep in mind all of the above observations when we move to the design and selection of our regression models, and we will see if similar conclusions are drawn from the "relatively slow" kernels.

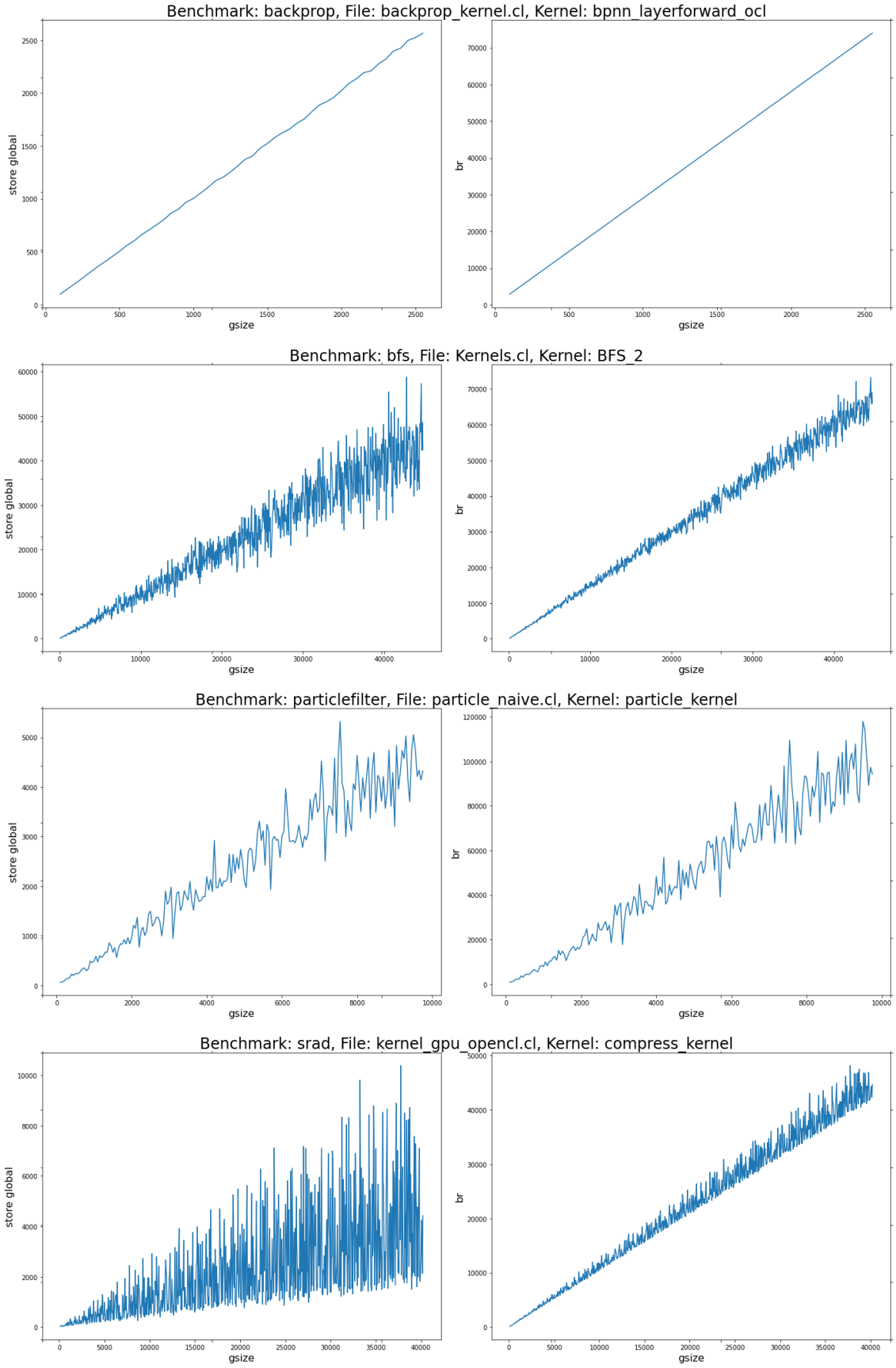

Figure 19.4: *`store global` and `br` instructions against `gsize` for some of the "relatively fast" kernels*

19.3.3 Regarding the "relatively slow" kernels

We will continue our analysis with the "relatively slow" kernels. Specifically, we want to show that their slowness is a result of ***a super-linear relationship between `gsize` (independent variable) and `instcounts` (dependent variables)***.

Let us visit the average correlation matrix again but, this time, regarding the "relatively slow" kernels only.

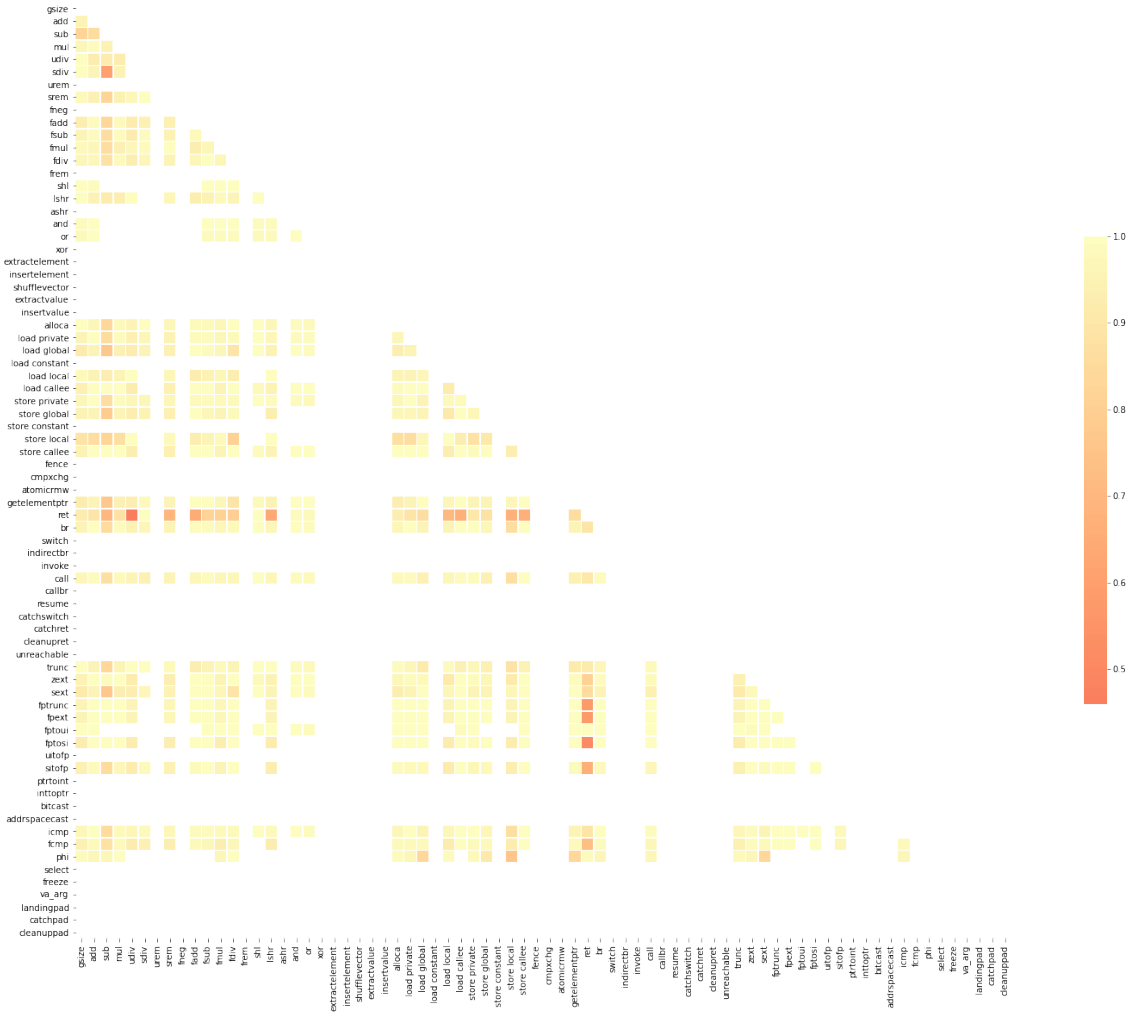

Figure 19.5: *The average correlation matrix of LLVM instructions executed across the "relatively slow" kernels only*

We see once again that the correlation between `gsize` and `instcounts`, as well as between `instcounts` themselves are really strong. Therefore, the answer regarding the slowness of these kernels will probably not be found here. Or will it?

Let us take a closer look the way we did before, i.e. by presenting four indicative kernels from the "relatively slow" group in Figure 19.6 at the end of the Section.

Things are way more interesting than what we expected before starting our experiments. First of all, it should be noted that we now have ***way fewer profilings compared to what we had from the "relatively fast" kernels***, due to their slowness. This is the main reason for the graphs not being so smooth now. But, even with these few samples,

we can make some really important observations:

1. ***Almost all the "relatively slow" kernels seem to exhibit a super-linear behavior***. This is not very visible because of the small number of profilings, but it is clearly observable in the `bfs` and `hybridsort` benchmarks above, amongst others not shown here.

2. There is ***one kernel, the one from the `lud` benchmark, that exhibits linear behavior***. Observe, though, that we get a really big count for the `br` instruction - around 100000 - with `gsize` being as small as 130. This explains why `lud`, despite exhibiting a linear behavior, belongs to the "relatively slow" group.

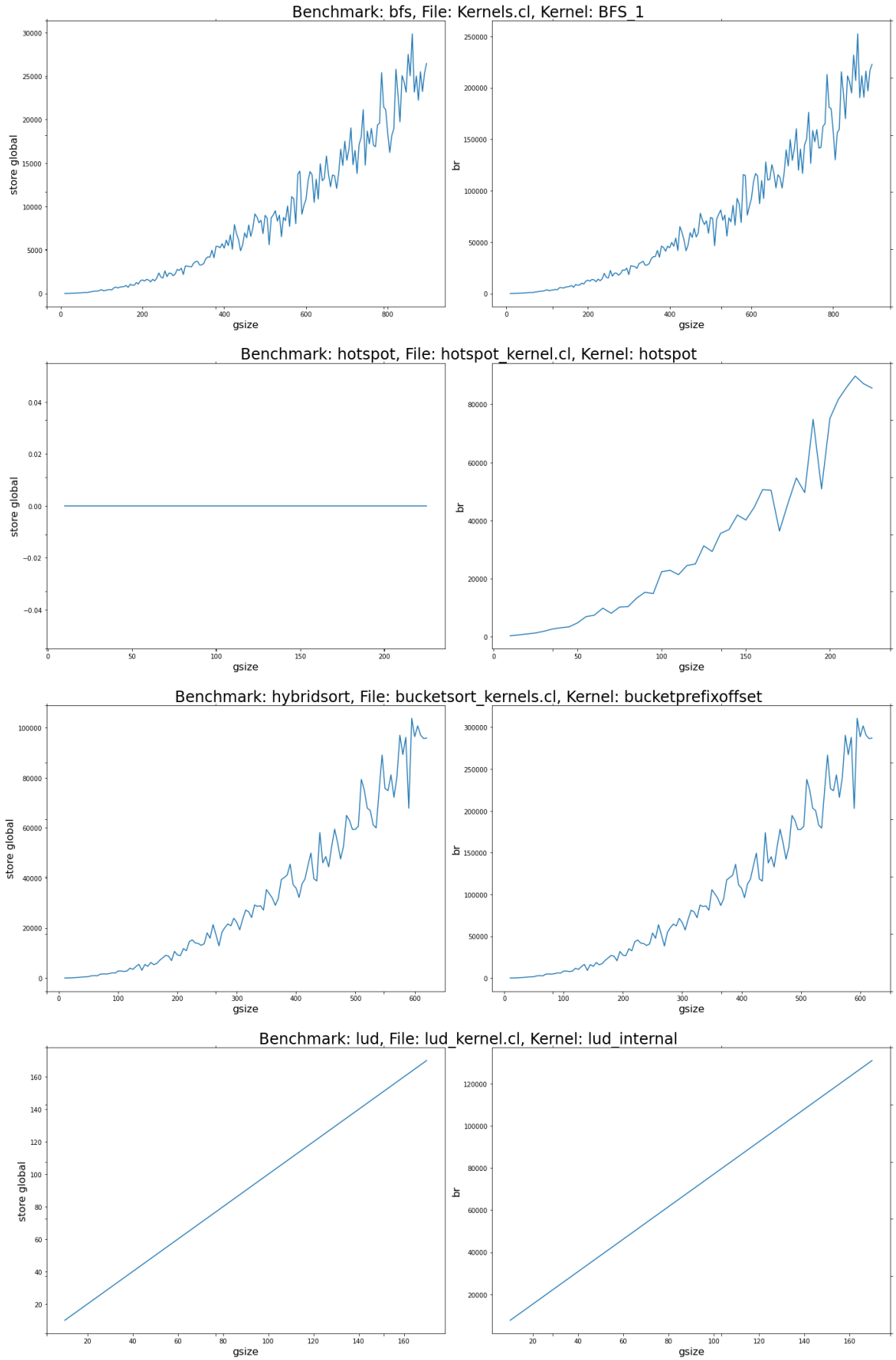

Figure 19.6: *`store global` and `br` instructions against `gsize` for some of the "relatively slow" kernels*

19.4 Some comparative conclusions

We will wrap up this Chapter by presenting one last pie chart (let the reader know that the writer is aesthetically attracted by this kind of charts for some reason):

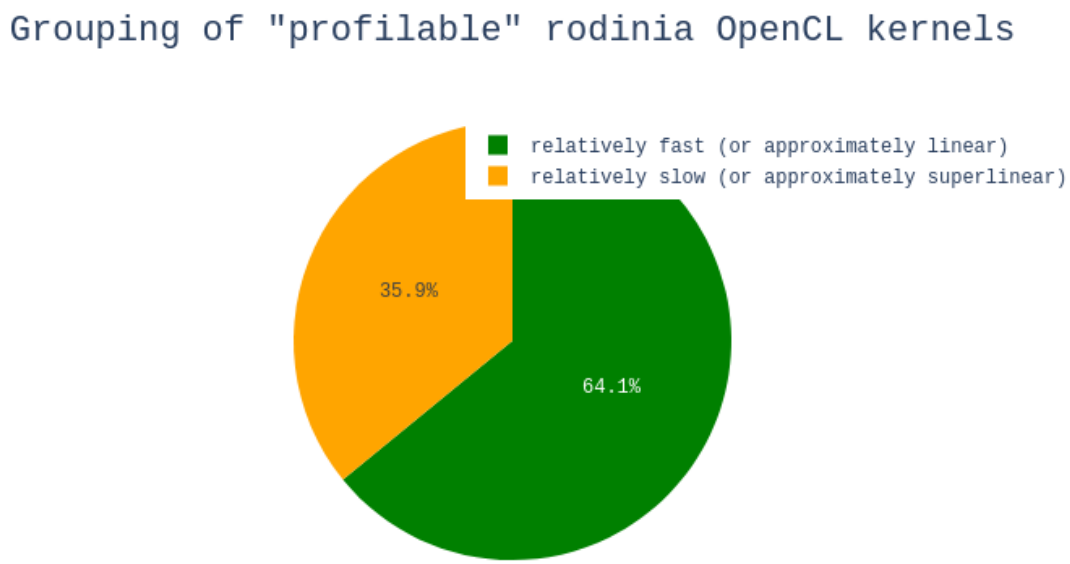

Figure 19.7: *"Profilable" Rodinia kernels by category ("relatively fast/slow")*

We concluded that:

- ***64.1% of the "profilable" kernels exhibit linear behavior***. This explains their fastness and means that we expect linear regression models to be more accurate than other, more complicated models.
- ***35.9% of the "profilable" kernels exhibit super-linear behavior***. This explains their slowness and means that more complicated regression models than the linear ones will be needed. But, given that most kernels of this kind ***most probably seem to exhibit a quadratic behavior***, a polynomial model of the appropriate degree may suffice. We must always keep in mind that the profilings are very few and thus very noisy, though, so any conclusion regarding the "relatively slow" kernels should be considered unsafe for now.

Last but (certainly) not least, a secondary but extremely important conclusion is that it may be the case that ***a single regression model will not be accurate enough for the whole benchmark suite***. There will most probably be a single linear regression model that will be very accurate for the "relatively fast" kernels alone, but it is quite uncertain whether:

1. a global one

2. one for the "relatively slow" kernels alone

will (or can) be found. But, in cases like this, one should remember this cheerful little quote by writer Seth Godin:

> *"Surprise comes from defying expectations."*

This writer (not Seth Godin) loves surprises. What about the reader?

Chapter 20

OCLBoi: The kernel-specific model

We are now ready to create machine learning models based on various regression techniques which will attempt to predict the total number of instructions per type that would be executed by an OpenCL kernel, given a specific global NDRange (i.e. a `gsize`).

In other words, we are ready to design a strategy to build ***a different regression model per kernel*** in order to fit the function $h_{kernel,device}$ as best as possible, where $h_{kernel,device}$ is the function defined in Equation (2.3). We will call this family of regressors ***OCLBois*** - `OCLboi` in singular, meaning "OpenCL but one in-particular", given that each `OCLBoi` will correspond to one kernel only.

As discussed in the Introduction, the $h_{kernel,device}$ function is kernel-specific, because instruction counts are related to the `gsize` input parameter in different ways depending on what each kernel is designed to do (i.e., depending on its *source code*).

Given our prior knowledge from the EDA we conducted (see Chapter 19), we assume we are not going to need regressors of order greater than that of the polynomial with order 2. More specifically, we assume that:

- The $h_{kernel,device}$ function of the "relatively fast" kernels can be satisfactorily approached with a linear regression model.
- The $f_{kernel,device}$ function of the "relatively slow" kernels can be satisfactorily approached with a polynomial regression model of order no greater than 2.

These 2 hypotheses derive from the graphs we saw in Chapter 19.

Therefore, we are going to use four different regression models, two linear and two polynomial, where the latter will be built on the former respectively.

20.1 The regression models used

We are going to use four different regression models in our experiments:

1. A linear regressor
2. An Elastic Net regressor
3. A polynomial regressor of degree 2, based on a linear regressor

4. A polynomial regressor of degree 2, based on an Elastic Net regressor

For more information on the characteristics of each regression model and the differences between them, we refer the reader to Chapter 15.

20.2 Selecting a regression model per kernel

In our experiments, we determined the best model for each kernel via *a multi-model gridsearch* and *a* 5*-fold cross validation* process. More specifically, `OCLBoi` fits the function (2.3) that maps `gsize` to `instcounts` for a specific kernel of interest. To do that, we split all of our samples into a train and a test set, we train our models on the former and test them with the latter. The model selection for each kernel is done based on ***the highest*** R^2 ***score on the test set across all models***. To learn (or remember) everything you need to know about R^2 in the context of this work, we kindly propose to the reader to (re)visit Chapter 15.

It should be noted that, contrary to Chapter 19, we ***did not use the average of the 100 samples of each profiling experiment***; we rather used each sample separately, as a different and independent train/test sample. The same holds for Chapter 21 that follows.

All of the above is a pretty straight-forward (one could even say *textbook*) case of the designing, training, testing and selecting a regression model for a given problem (the only non-textbook part is the multi-model gridsearch). Therefore, we will not procrastinate any longer and we will jump write into our results from the experimental process described above.

Firstly, we will plot the mean R^2 scores for each one of the four models that we used, regardless of whether or not they were they had the best score for none, one or more kernels.

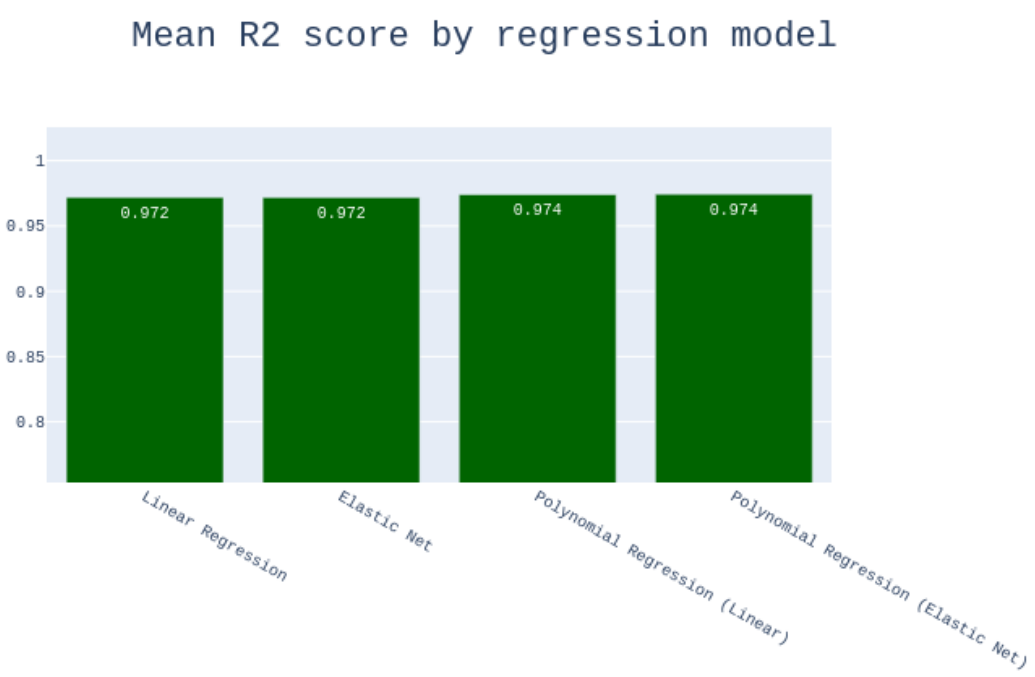

Figure 20.1: *The mean R^2 score by regression model used for* `OCLBois`

The results confirm that our assumptions regarding the sufficiency of the linear models and the polynomial models of order no more than 2 were correct; the mean R^2 score is very satisfactory for all four models, which means that the predictions of the models we chose explain the variance of the data to a large extent.

Let us now investigate which models are the more "popular" ones, i.e. they have the greatest R^2 score for more kernels than the other models.

In Figure 20.2, we see that the most popular model is the **polynomial regressor based on the linear regressor** (12 kernels). **Linear regression** (10 kernels) comes next, followed by **Elastic Net** (9 kernels) and **polynomial regression based on Elastic Net** (8 kernels).

Lastly, it should be quite interesting to study which regression models were preferred by the kernels *based on whether they were categorized as "relatively fast" or "relatively slow"* in the EDA Chapter (Chapter 19).

As expected, in Figure 20.3 we can see that the ***linear models*** (linear regression and Elastic Net) ***were selected by the majority of the "relatively fast" kernels at a rate of 60%***, while the ***polynomial models*** (polynomial regression based on simple linear regression and polynomial regression based on Elastic Net) ***were selected by the majority of the "relatively fast" kernels at a rate of 71.4%***.

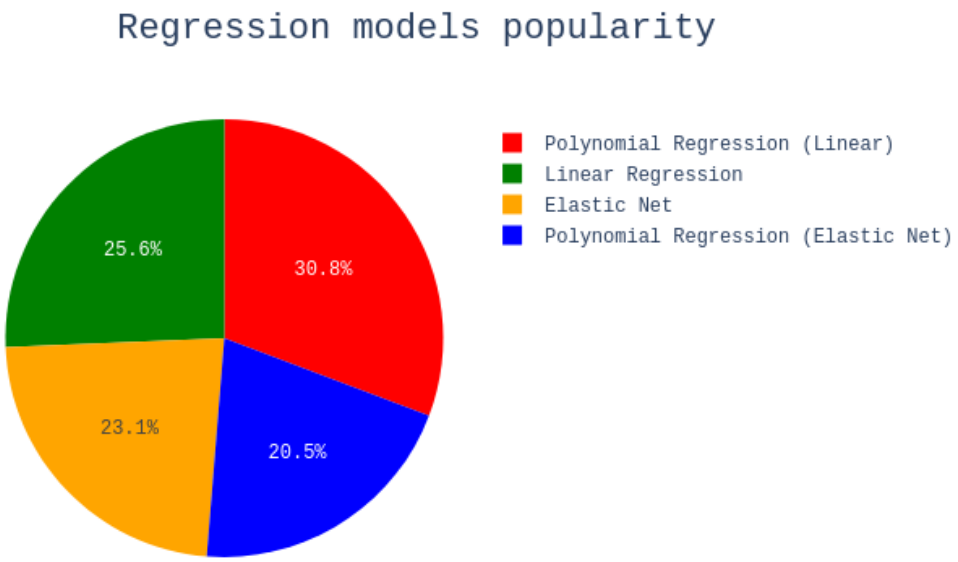

Figure 20.2: *Popularity of the four regression models amongst all kernels*

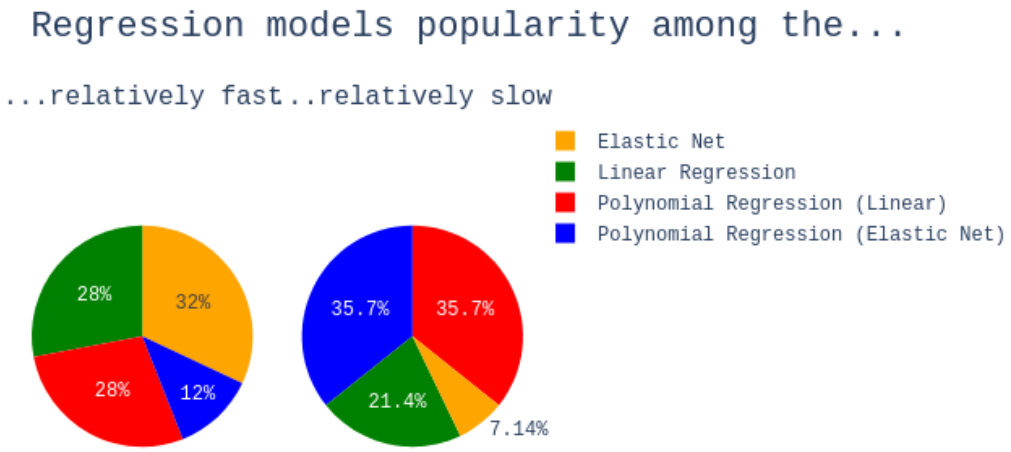

Figure 20.3: *Popularity of the four regression models grouped by category ("relatively fast/slow")*

Chapter 21

OCLMan: One model to rule them all

The time has come. Raise the flags with shouts of gladness, for `OCLMan` is about to arrive.

`OCLMan` stands for *"OpenCL Maybe? Approximately? Nope!"*. Obviously, the bar is set quite high when such an acronym is chosen. However, it is crucial to remind the reader of the fact that ***OCLMan is more of a proposed methodology than a fully implemented and fine-tuned model***. That has been said more than enough times up until now, so let us mark this time as the last (at least until Chapter 22).

Having said the above, the goal of this Chapter is to present the methodology we designed in order to predict execution times given a kernel and a `gsize` of interest.

21.1 An overview of OCLMan

The overall model consists of two parts that it ultimately synthesizes, following the abstract mathematical formalization of Equation (2.2). In fact, what `OCLMan` is trying to do is fit this f function, but *not directly*. This is merely the behavior we expect (and want) to be seen from the outside. But what `OCLMan` *really* does is fit the h and g functions from Equations (2.3) and (2.4), respectively.

We refer the reader to Figure 2.2, where the two functions that we are talking about correspond to the *instcounts and time models*.

21.2 From experimental measurements to instruction counts

The first model, the *instcounts model*, is already familiar to the reader; it is no other than the `OCLBoi` model of the previous Chapter.

We propose (and experimented with) the following approach: *a different `OCLBoi` should be created and trained, inline and on the fly, based on the input kernel.* That is:

1. ***why*** we designed the microprofiling phase to extract runtime features with small `gsize` values; to make the training of our `OCLBoi`s faster
2. ***how*** we achieve having the second model, the time model, being *pretrained* for a specific OpenCL device, but *kernel-agnostic* nonetheless; everything kernel-related falls on the shoulders of an `OCLBoi`, which will spawn and be trained when needed.

Let us now summarize what the *instcounts model*, or an `OCLBoi`, needs in order to be created, trained and used.

- Regarding the **training** phase:
  - inputs:
    - measurements regarding the kernel `instcounts` as taken by `oclude`
- Regarding the **prediction** phase:
  - inputs:
    - a `gsize` value
  - output:
    - a vector of predicted `instcounts` for the given `gsize` value

21.3 From experimental measurements to execution time

The next step of the pipeline that constitutes `OCLMan` is the *time model*. This model fits a function that maps `instcounts` to execution times.

Two very crucial assumptions have been made regarding this model and the function (2.4) that it fits, which must be noted at this point.

Our first assumption is that this function is considered to be ***kernel-agnostic***. This is, in fact, a hypothesis, although it may not be so obvious. Several factors that may contribute to the execution time have been omitted - e.g. the order of the instructions (a kernel-specific feature) has not been taken into consideration. However, we consider this assumption to be quite balanced between the simplicity it offers and the information we may lose by making it. After all, our experiments will tell us whether we are on the right path or not.

Our second assumption is that ***the execution time of a kernel is a linear combination of its instcounts***. We base this assumption on nothing more than common sense; doesn't it sound rational that the time a kernel needs to execute is a linear function of the different individual jobs (i.e. instructions) that it has to complete? More specifically, we can imagine a function of the following form:

$$t_{exec} = t_{add}count_{add} + t_{sub}count_{sub} + t_{mul}count_{mul} + \dots \tag{21.1}$$

where:

- t_i is the time that a *single* LLVM instruction of the i type needs to execute on the processing unit of interest, and
- $count_i$ is the number of LLVM instructions of the i type that were executed (i.e. our familiar `instcounts`).

It is obvious that these two assumptions lead to simplifications regarding the behavior of the execution environment. However, the simplicity of this model is sufficient to achieve some of the most important objectives of this dissertation, which are to prove that:

1. `oclude`, as a dynamic OpenCL kernel profiler, works as designed, expected and promised.
2. the measurements taken by using `oclude` contain valuable information that deepens our understanding of the behavior of OpenCL kernels and facilitates our effort to better deal with the problems of the heterogeneous computing world; in other words, the dynamic microprofiling process was worth it.
3. `OCLMan`, as a methodology, works. It may be far from finalized and/or optimized, but it yields significant results, which would otherwise be infeasible.

Let us now summarize what the *time model* needs in order to be created, trained and used.

- Regarding the **training** phase:
  - inputs:
    - *(more like a dependency)* a kernel-specific predictor (i.e. an `OCLBoi`) for each kernel of the training benchmark suite
    - measurements regarding **the execution times of all the kernels of the training benchmark suite** as taken by `oclude`
- Regarding the **prediction** phase:
  - inputs:
    - an OpenCL kernel
    - measurements regarding the `instcounts` of the kernel as taken by `oclude`, to create and train a kernel-specific `OCLBoi` on the fly (see above)[1]
    - a `gsize` value
    - *(internal process)* a vector of predicted `instcounts` for the given `gsize` value, from the `OCLBoi` of the given kernel
  - output:
    - the predicted kernel execution time for the given `gsize` value

21.4 OCLMan, assemble!

In order to present the way we built `OCLMan` in order to evaluate our methodology, we need to refer to two things:

1. The way data was used during the training and testing phase, and
2. The baseline model that we used in order to evaluate `OCLMan`.

[1] Note that this prediction phase input implies that these measurements have already been taken at some point in the past. To fully automate the whole process, and hence create a complete predictor which will accept a kernel and a `gsize` value as its sole inputs, the implementation of an automatic measurement subsystem is needed, which should be based on `oclude`.

21.4.1 Training and testing

To train and test `OCLMan` and its two components, the *instcounts model* and the *time model*, we had to be very careful with the handling of our data, mainly due to `OCLBoi`. More specifically, the steps followed to *train* `OCLMan` are shown in the following Figure and are explained soon after.

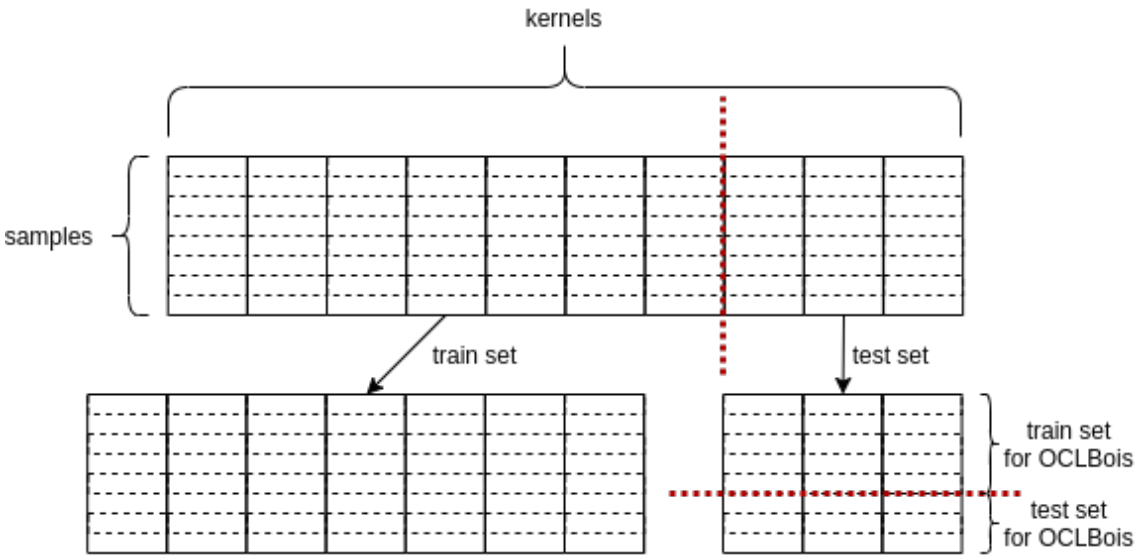

Figure 21.1: *The training and testing phases of the* `OCLMan` *methodology*

1. we organized our data into three different groups, without omitting which sample came from which kernel:

   (a) the *gsizes*

   (b) the *instcounts*

   (c) the *execution times*

   In the *instcounts model* component of `OCLMan` (i.e. `OCLBoi`), (a) will be used as the inputs and (b) as the outputs, while the *time model* component will use (b) as inputs and (c) as outputs.

2. Having completed the above data compartmentalization for each sample, we split ***the kernels*** (and not the samples taken from the kernels) into a train and a test set. That way, we make sure that `OCLMan` and its components will know nothing about the kernels of the testing phase. This is important because, if we had split the samples and not the kernels (i.e. the top dashed red line in Figure 21.1 was horizontal instead of vertical) our experimental results would be misleading. The reason is that there is a chance that `OCLMan` would have figured out patterns and relations amongst the experimental data of *all* kernels, thus indirectly canceling the split into train and test sets and, ultimately, the whole experiment.

3. We train **a simple linear regressor** on all of our train data (i.e. the bottom left rectangle in Figure 21.1), specifically the part of the training data that is the

`instcounts` and execution time measurements. This is our *time model*, now pre-trained and ready for our OpenCL device. It is a simple linear regressor because of the assumptions explained earlier in the Chapter. Note that this linear regression is a *multiple* one; it accepts many inputs (our `instcounts`) and produces a single output (execution time). At this point, `OCLMan` is ***ready to be used***. Note that ***there is no need to test the time model individually at this point***. The testing of `OCLMan` will be performed at the end, including all of its components.

4. `OCLMan` is ready and awaits for your questions. The training phase is complete.

In order to *test* `OCLMan`, we are obviously going to use the test set only (bottom right rectangle in Figure 21.1). But remember; each prediction of `OCLMan` results into the spawning, training *and testing* of a new kernel-specific `OCLBoi`, an *instcounts model*. It is reminded (see Chapter 20) that the testing of an `OCLBoi` is crucial because it ultimately leads to the selection of one of the four regression models available. Therefore, we need *a training and a testing set for `OCLBoi`, every time `OCLMan` is called to predict*. The only way to achieve that without breaking the inviolable machine learning comandments of train-test split (*"thou shalt not allow your model to see or know anything about the test set while training it"*), we need to do what is shown in the bottom right corner of Figure 21.1: *to split the `OCLMan` test set itself into train and test sets for each `OCLBoi`*. That way, when a prediction is needed (e.g. in the testing phase of our experiment), `OCLMan` will be able to rely solely on the data for the kernel of interest.

21.4.2 Evaluating

To evaluate the `OCLMan` methodology and conclude our humble but hopefully memorable journey through the lands of heterogeneity and machine learning, we will compare it with the standard methodology regarding this type of problems; *a static model*.

By static, we mean that this model, hencforth `OCLBase`, will be trained on *static features only*. More specifically, we will train `OCLBase` with three types of measurements:

1. *`gsizes`*,
2. *static `instcounts`*, and
3. *execution times*.

The static `instcounts` for each kernel, i.e. source code LLVM instruction counts, have been extracted by `oclude`, through the corresponding function of the Python package API (see Chapter 16). Therefore, in `OCLBase`, no notion of dynamic microprofiling is present. That way, we will be able to see whether getting into all this trouble regarding the microprofiling process and the design of `OCLMan` was worth it or not.

Now, to be absolutely fair, we must give `OCLBase` the same freedom and constrain it with the same bias as we do with `OCLMan`. That means, that we will make `OCLBase` a ***polynomial regressor of order 2 based on linear regression***. The reason we do this is the relationships between `gsize`, `instcounts` and *execution time*:

$$gsize \longmapsto instcounts \ (at\ most\ polynomial,\ proven) \tag{21.2}$$

$$instcounts \longmapsto t_{execution} \ (linear,\ assumed) \tag{21.3}$$

$$\Downarrow$$

$$gsize \longmapsto t_{execution} \ (at\ most\ polynomial,\ assumed) \tag{21.4}$$

Therefore, if the regressor of `OCLMan` is a simple linear regressor[2] and the `OCLBase` is a polynomial regressor of order 2 based on simple linear regression, both models have, in theory, the same capabilities.

Talk is cheap; we present the following table, were the R^2 *scores* and *root mean square errors* of both models on the five kernels that constituted our test set are shown. We also present the corresponding plots in Figure 21.2.

test set kernel index	`OCLMan` **R2 score**	`OCLBase` **R2 score**	`OCLMan` **RMS error**	`OCLMan` **RMS error**
1	0.902958	0.8998345	1.838034	1.867378
2	-98.151065	$-1.246192 \cdot 10^2$	176.792961	198.995943
3	0.467303	$-1.970361 \cdot 10^3$	19.129077	1163.689462
4	0.467474	$-1.730095 \cdot 10^7$	3.405769	19412.417732
5	0.792912	$-2.005667 \cdot 10^2$	12.833066	400.370287

Table 21.1: R^2 *scores and RMS errors of* `OCLMan` *and* `OCLBase` *on the test set*

We can see from both Table 21.1 and Figure 21.2 that the gods are kind to us. More specifically, we can safely say that:

> ***The OCLMan methodology we propose performs better than a simple static model and, most of the times, by many orders of magnitude.***

It is important to note that this experimented was conducted many times with different splits of the our data into train and test sets, but the results always followed the trend seen in Table 21.1 and Figure 21.2.

More specifically, we observe the following:

1. Most of the times, `OCLman` achieves R^2 scores that are orders of magnitude better than the ones that `OCLBase` achieves and, in fact, lie in the (0. 1] interval, in which the percentage of the variance of the experimental data explained by our predictions is greater than zero.

2. `OCLMan` manages to capture the curvature of the experimental measurements when needed, while `OCLBase` does not (even though we implemented the latter as a polynomial regressor of degree 2).

[2] it is reminded that, regarding `OCLMan`, the polynomial nature of the last relation is manifested (if needed) through the kernel-specific `OCLBoi` predictor that `OCLMan` uses internally.

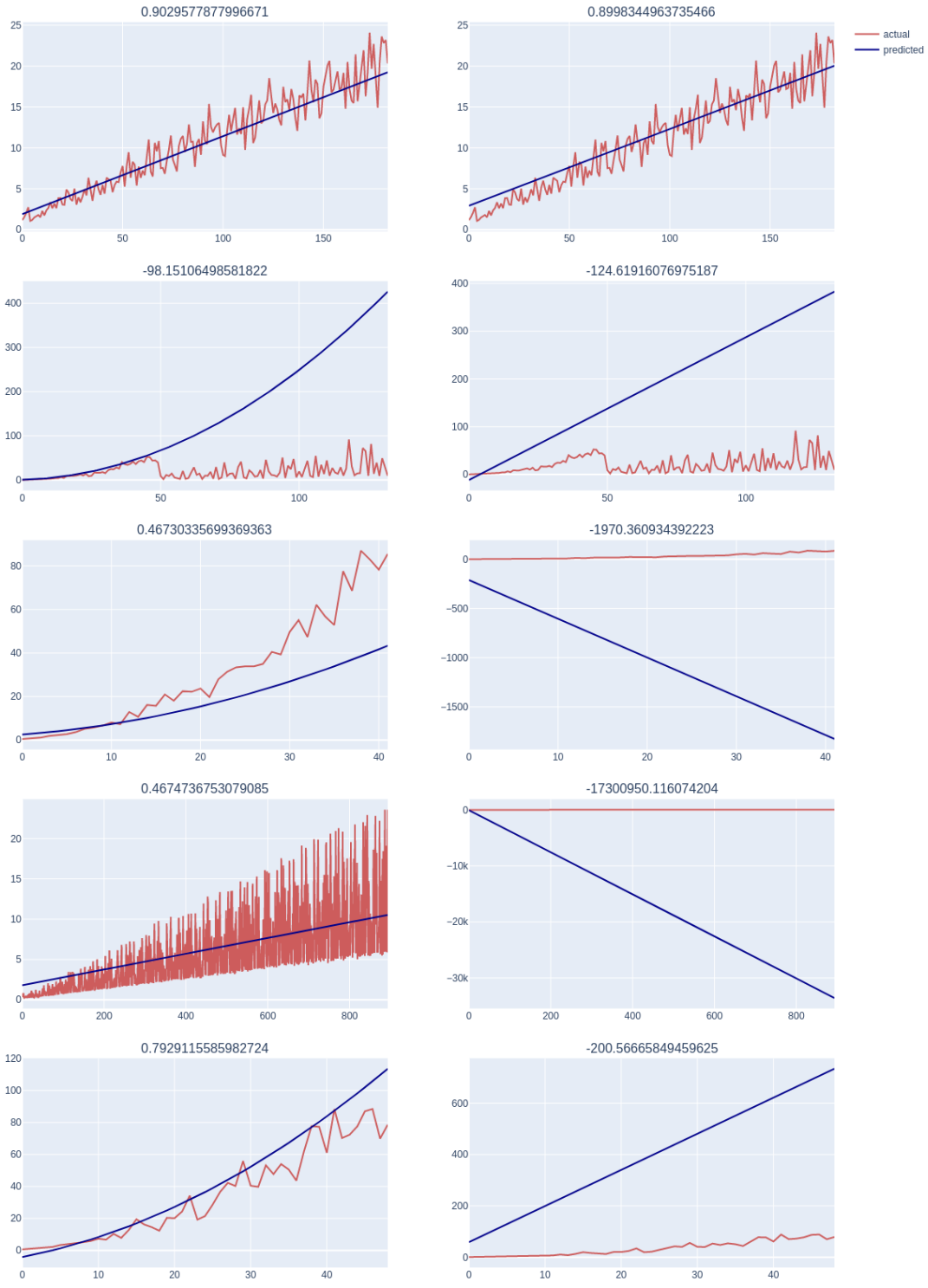

Figure 21.2: *Plots of the predicted execution times on the final test set. Over each diagram, the corresponding R^2 score is written. On the left column,* `OCLMan` *results. On the right column,* `OCLBoi` *results.*

3. There are times where `OCLMan` performs badly, but clearly not as badly as `OCLBase`. Observe the second kernel; it is interesting to note that the reason `OCLMan` performed badly is that it followed the trend of the small `gsize` values, which unfortunatelly was not indicative of the overall behavior of the kernel. This is a very interesting research subject for future work.

Part VIII

Final Remarks

Chapter 22

Suggestions for future work

We are going to wrap up this beautiful journey with some suggestions for future work.

1. As you have seen, designing, building and maintaining `oclude` as a source code instrumentor is an extremely challenging task. It would be extremely helpful if `oclude` could be re-written as an IR instrumentor. If that was achieved, the instrumentation process would be trivial: We would iterate every basic block and add incrementors for the instructions it contained. The reason `oclude` is not designed that way is because no method could be found during our research to run OpenCL applications that are available as (LLVM) IR.

2. One of our most simplistic assumptions, mainly regarding CPUs as OpenCL devices, is that the execution time is a linear function of the instcounts vector and, based on that assumption, we used a linear regressor as our time model in `OCLMan` and `OCLBase`. This assumption needs to be tested, with more regression techniques for the time model.

3. `OCLMan` needs to grow up and evolve from being a methodology to a toolkit. To do so, its architecture and many of its parameters need to be readjusted and fine-grained. The nicest possible scenario is for `OCLMan` to turn into a standalone software that profiles OpenCL devices and kernels out-of-the-box.

4. More devices and more kernels are needed in order to further test our designs, tools, ideas and assumptions.

Παραρτήματα

Appendix A

In defence of the (large) number of samples for the same gsize and why it is essential

In this appendix, we would like to add a small but significant note to wrap up the investigative analysis presented in Chapter 19 regarding the way profiling data was collected using `oclude`. More specifically, we would like to convince the reader of the necessity to use a relatively large value for the `-samples/-s` flag of `oclude`. We will prove our point with an experimental example.

By profiling a kernel for 100 different gsizes, and for different sample sizes, we get the results illustrated in the graphs below.

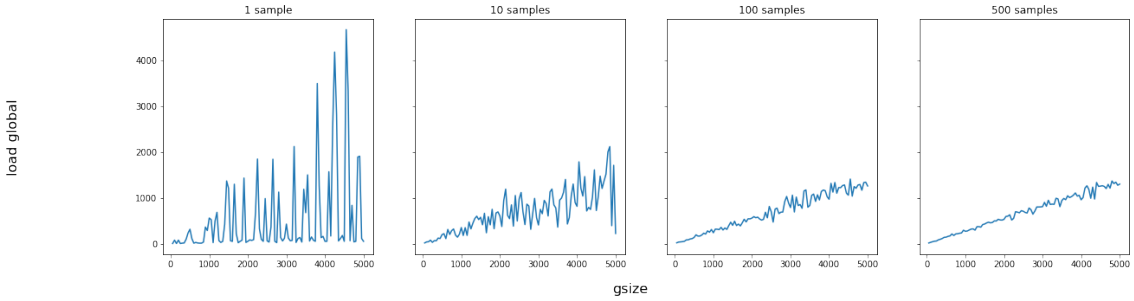

Figure A.1: *Profiling a kernel for increasing sample sizes*

The absolute values are of no importance, the above illustration is presented in the context of a qualitative analysis.

We can clearly see that ***the more samples there are, the less influential the randomness that comes from the initialization of the kernel arguments becomes***.

In the example above, it is obvious that the linear nature of the kernel is not visible at all when we get a single sample. What's more, it seems like we have to deal with greater instruction counts than in reality (i.e. there are a lot of spikes that reach values greater than 2000). But when we start to increase the number of samples (and calculate the average instruction counts), the more visible and obvious it becomes that:

1. the kernel exhibits a linear behavior, and
2. the actual instruction counts are much smaller on average (i.e. the spikes/outliers have been neutralized).

Βιβλιογραφία

[1] Vignesh Ravi, Wenjing Ma, David Chiu και Gagan Agrawal. *Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations*. σελίδες 137–146, 2010.

[2] *History of computation*. https://en.wikipedia.org/wiki/History_of_computing. [Online; accessed 22-July-2020].

[3] *Antikythera mechanism*. https://en.wikipedia.org/wiki/Antikythera_mechanism. [Online; accessed 22-July-2020].

[4] *The Sand Reckoner*. https://en.wikipedia.org/wiki/The_Sand_Reckoner. [Online; accessed 22-July-2020].

[5] *Charles Babbage's Analytical Engine*. https://en.wikipedia.org/wiki/Analytical_Engine. [Online; accessed 22-July-2020].

[6] *The History of Computing Hardware*. https://en.wikipedia.org/wiki/History_of_computing_hardware. [Online; accessed 22-July-2020].

[7] Benedict Gaster, Lee Howes, David Kaeli κ.ά. . *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 1η έκδοση, 2011.

[8] *Heterogeneous Computing*. https://en.wikipedia.org/wiki/Heterogeneous_computing. [Online; accessed 22-July-2020].

[9] Olivier Terzo; Karim Djemame; Alberto Scionti; Clara Pezuela. *Heterogeneous Computing Architectures: Challenges and Vision*. CRC Press, 2020.

[10] *OpenCL Overview - The Khronos Group Inc*. https://www.khronos.org/opencl/. [Online; accessed 22-July-2020].

[11] Byung Gon Chun, Ling Huang, Sangmin Lee κ.ά. . *Mantis: Predicting System Performance through Program Analysis and Modeling*. 2010.

[12] L. David και I. Puaut. *Static determination of probabilistic execution times*. *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, σελίδες 223–230, 2004.

[13] Yuan Wen, Zheng Wang και Michael O'Boyle. *Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms*. σελίδες 1–10, 2014.

[14] Ling Huang, Jinzhu Jia, Bin Yu κ.ά. . *Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression*. *Advances in Neural Information Processing Systems 23*J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor κ.ά. , επιμελητές, σελίδες 883–891. Curran Associates, Inc., 2010.

[15] M. A. Iverson, F. Ozguner και L. Potter. *Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment*. *IEEE Transactions on Computers*, 48(12):1374–1379, 1999.

[16] L. T. Yang, Xiaosong Ma και F. Mueller. *Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution*. *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, σελίδες 40–40, 2005.

[17] Reinhold Heckmann και Christian Ferdinand. *aiT: Worst-Case Execution Time Prediction by Static Program Analysis*. *International Federation for Information Processing Digital Library; Building the Information Society;*, 156, 2004.

[18] S. Che, M. Boyer, J. Meng κ.ά. . *Rodinia: A benchmark suite for heterogeneous computing*. *2009 IEEE International Symposium on Workload Characterization (IISWC)*, σελίδες 44–54, 2009.

[19] Sotirios Niarchos. *oclude: An OpenCL driver to test, run and profile standalone kernels on arbitrary devices*. https://github.com/zehanort/oclude, 2020. GitHub repository.

[20] Andreas Klöckner, Nicolas Pinto, Yunsup Lee κ.ά. . *PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation*. *Parallel Computing*, 38(3):157–174, 2012.

[21] Sotirios Niarchos. *rvg: A random values generator for any data type in Python 3*. https://github.com/zehanort/rvg, 2020. GitHub repository.

[22] J. E. Stone, D. Gohara και G. Shi. *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems*. *Computing in Science Engineering*, 12(3):66–73, 2010.

[23] Chris Lattner και Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation*. *CGO*, σελίδες 75–88, San Jose, CA, USA, 2004.

[24] Ravi Sethi Jeffrey D. Ullman Alfred V. Aho, Monica S. Lam. *Compilers: principles, techniques & tools*. Pearson/Addison Wesley, 2η έκδοση, 2007.

[25] James Price και Simon McIntosh-Smith. *Oclgrind: An Extensible OpenCL Device Simulator*. *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL '15, σελίδες 12:1–12:7, New York, NY, USA, 2015. ACM.

[26] Siu Kwan Lam, Antoine Pitrou και Stanley Seibert. *Numba: A LLVM-Based Python JIT Compiler*. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[27] Chris Cummins. *cldrive: Run arbitrary OpenCL kernels*. https://github.com/ChrisCummins/cldrive, 2020. GitHub repository.