Programming model for heterogeneous computing systems with customizable accelerators

Pervan, Branimir

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Branimir Pervan

PROGRAMMING MODEL FOR HETEROGENEOUS COMPUTING SYSTEMS WITH CUSTOMIZABLE ACCELERATORS

DOCTORAL THESIS

Zagreb, 2022

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Branimir Pervan

PROGRAMMING MODEL FOR HETEROGENEOUS COMPUTING SYSTEMS WITH CUSTOMIZABLE ACCELERATORS

DOCTORAL THESIS

Supervisor: Associate Professor Josip Knezović, PhD

Zagreb, 2022

Sveučilište u Zagrebu
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Branimir Pervan

PROGRAMSKI MODEL ZA RAZNORODNE RAČUNALNE SUSTAVE S PRILAGODLJIVIM UBRZIVAČIMA

DOKTORSKI RAD

Mentor: izv. prof. dr. sc. Josip Knezović

Zagreb, 2022.

Doctoral thesis was made at the University of Zagreb, Faculty of Electrical Engineering and Computing, Department of Control and Computer Engineering

Supervisor: Associate Professor Josip Knezović, PhD

Doctoral thesis contains: 140 pages

Doctoral thesis number: _____

About the Supervisor

Josip Knezović, PhD is an associate professor at the Department of Control and Computer Engineering, Faculty of Electrical Engineering and Computing, University of Zagreb. He obtained his master's degree in 2005 and in 2009 he gained PhD degree with the thesis on the Streaming model of computation for image and video processing.

He is the member of the HPC Architecture and Application Research Center at FER. His research interests include energy efficient, embedded, heterogeneous and parallel computing systems and programming models, parallel programming, and computer architectures and applications of high-performance computing systems. He teaches several courses in undergraduate, graduate, and doctoral studies in computer science. From 2020 he serves as the Head of the Department of Control and Computer Engineering. He is a member of the IEEE, IEEE Computer Society, and ACM.

Personal page: `https://www.fer.unizg.hr/rasip/jknezovic`

O mentoru

Izv. prof. dr. sc. Josip Knezović je izvanredni profesor u Zavodu za automatiku i računalno inženjerstvo Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu. Stupanj magistra znanosti postigao je 2005. g., a 2009. g. obranio je doktorsku disertaciju na temu tokovnog računalnog modela za obradu slikovnih i video podataka.

Član je Centra za istraživanje arhitektura i aplikacija računarstva visokih performanci na FER-u. Njegovi istraživački interesi su energetski učinkoviti ugradbeni, heterogeni i paralelni računalni sustavi, paralelno programiranje i programski modeli, te računalne arhitekture i aplikacije u računarstvu visokih performanci. Nositelj je ili predavač na više predmeta na preddiplomskom, diplomskom i doktorskom studiju računarstva FER-a. Od 2020. g. obnaša dužnost predstojnika Zavoda za automatiku i računalno inženjerstvo. Član je strukovnih udruženja IEEE, IEEE Computer Society i ACM.

Osobna stranica: `http://www.fer.unizg.hr/rasip/jknezovic`

Zahvaljujem prvo Bogu svemogućemu, onome koji bje, koji jest i koji će doći, iz kojega sve izvire i u kojega sve uvire.

Hvala mojoj supruzi Petri, kćeri Marti i sinu Boni. Vaša žrtva i odricanje protkani su ovim stranicama. Bez vaše pomoći ne bih uspio, moj uspjeh je i vaš uspjeh. Hvala i mojoj široj obitelji.

Hvala mom mentoru Josipu za neizostavnu podršku koju mi je pružio tijekom rada na doktorskom studiju, te za sve trenutke u kojima smo se međusobno podučavali.

Special thanks goes to Michel Steuwer at the University of Edinburgh and to his team from Edinburgh, Glasgow, and Münster. I cannot thank you enough for your help, support, and the ability to be a part of the team. Thank you for your expertise and patience.

Hvala svim mojim prijateljima, a ponajprije Barbari i Šili. Posebnu zahvalnost želim iskazati doktoru Turčinoviću, za svaki savjet, kavu, pelin, ručak, raspravu i druženje. Hvala i Emanuelu, Luki, Ivani, Igoru i Ani na podršci.

A ja, Bože sačuvaj da bih se ičim ponosio osim križem Gospodina našega Isusa Krista po kojem je meni svijet raspet i ja svijetu.
(Gal 6, 14)

I'd rather have questions that can't be answered than answers that can't be questioned.
-R. Feynman

Those who would give up essential Liberty, to purchase a little temporary Safety, deserve neither Liberty nor Safety.
-B. Franklin

Abstract

The constant growth of human development in general can be easily perceived at the current point in time. Followed by that development, there is a significant growth in the need for processing power. The amount of aggregated data requiring processing, followed by big data paradigm and brain-inspired computing drive the need to enter the so-called exascale domain, which is additionally confirmed by the existence of multiple scientific projects with the ultimate goal to develop an exascale machine in near future. In reaching the exascale domain, one of the key development points will be heterogeneity, implying systems containing a general-purpose processing core, coupled with at least one non-general purpose accelerator. Accelerators for specific applications usually achieve the best results regarding performance and energy efficiency, but with a cost of being unusable in other domains. Theoretically, there exists a large gap for customizable accelerators which could balance between performance and energy efficiency gains and the lack of customizability in conventional accelerators, by allowing customization of themselves to some extent.

High-performance systems of the future not only have to be heterogeneous but exploit parallelism on every level as well. However, efficiently exploiting heterogeneity and parallelism is inherently hard. Current programming models usually rely on imperative programming paradigms which decompose algorithms and problems with respect to *how* computations are executed, requiring deep knowledge of the underlying hardware and other aspects of the system, that domain scientists usually lack. It is, therefore, necessary to provide adequate models which would efficiently exploit given resources while keeping a relatively simple approach.

This thesis proposes a different approach to programming complex heterogeneous systems, by expressing *what* is being computed, rather than *how* computations are executed. To deliver such a model, the use of RISE language is proposed, which was appropriately extended to deliver the outcomes of this thesis. Additionally, ELEVATE, a domain-specific language for describing optimization strategies was used to demonstrate and enable optimizations targeting the specific parts of the system. As a testing platform, GAP8, a System-on-Chip containing a general-purpose fabric controller loosely coupled with a customizable accelerator containing eight RISC-V cores and a Hardware Convolution Engine was used. The proposed model was evaluated by executing typical parallel benchmarks and comparing the RISE-generated code with hand-tuned code. The results show that it is possible to provide an arguably simpler and more concise programming model while keeping performance and energy efficiency at least on par compared to the conventional programming model for the target platform.

Keywords: domain-specific languages, heterogeneous systems, parallel processing, multi-core systems, RISC-V

Prošireni sažetak

Programski model za heterogene računalne sustave s prilagodljivim ubrzivačima

1. Uvod

Paralelno s rastom i razvojem znanosti te čovječanstva općenito, svakodnevno svjedočimo i povećanoj potrebi za računalnim resursima. Vršna performansa ponajboljih svjetskih superračunala trenutno je reda veličine 10^{15}, dok se ulazak u *exaFLOPS* domenu anticipira za skoriju budućnost. Tomu svjedoče i projekti koji postoje na razini Europske Unije, u Japanu i SAD-u. Postizanje *exaFLOPS* domene otvara mnoga pitanja, primarno u područjima arhitekture računala, heterogenosti te energetske učinkovitosti. Pojedini izvori navode da je ključna točka razvoja za postizanje *exaFLOPS* domene upravo heterogenost. Heterogenost sustava obično implicira postojanje više od jedne vrste procesne jezgre na relativno izoliranom čvoru, a praktične implementacije svode se na uparivanje generičkog procesora (CPU) s grafičkim procesorom opće namjene (GPGPU) ili s ubrzivačem specifičnim za pojedinu aplikaciju. Grafički procesori opće namjene koriste se za intenzivne podatkovno paralelne zadatke dok, s druge strane, specijalizirani ubrzivači postižu maksimalnu performansu i energetsku učinkovitost, ali uz cijenu i iskoristivosti samo u jednoj aplikaciji, uz razvoj na razini dizajna hardvera. Uzevši u obzir prednosti i mane grafičkih procesora opće namjene te specijaliziranih ubrzivača, primjećuje se golem potencijal za uporabu prilagodljivih ubrzivača. Takvi ubrzivači kombinirali bi prednosti te minimizirali nedostatke oba pristupa tako da izlože parametre za prilagodbu čime bi se izbjegla zaključanost na samo jednu vrstu aplikacije, uz postizanje sumjerljive performanse i energetske učinkovitosti potpuno specijaliziranim ubrzivačima.

I dok je većina računala s kojima smo okruženi višejezgrena, promatrajući paralelizam i heterogenost u edukacijskim sustavima, većina obrazovnih programa podrazumijeva podučavanje sekvencijalnom programiranju. Na taj način stvara se procjep između teorije i prakse, s obzirom na to da se paralelno razmišljanje ne stvara ondje gdje bi se trebalo formirati. Jedan od razloga jest i inherentno teži način programiranja, te kompliciraniji programski modeli potrebni za razvoj paralelnih aplikacija.

Cilj ove disertacije, koji se nalazi na tromeđi heterogenosti, paralelizma, te prilagodbe programskih modela, jest izlaganje programskog modela visoke razine apstrakcije, oblikovanog kao domenski specifičan jezik za efikasno programiranje heterogenih sustava s prilagodljivim

ubrzivačima. Temeljna hipoteza istraživanja je ta da će programski kôd napisan u izloženom programskom modelu biti barem jednako performantan i jednako energetski efikasan kao i programski kôd napisan u nativnom modelu za ciljnu platformu, a sve uz povećan stupanj programirljivosti koji nudi programski model obrađen u disertaciji. Kao ciljna platforma koristit će se heterogeni sustav GAP8, koji se sastoji od generičkog procesora te prilagodljivog ubrzivača s osam jednostavnih jezgri te hardverskim ubrzivačem operacije konvolucije.

Konačno, izvorni znanstveni doprinos disertacije jest sljedeći:

1. Programski model za raznorodne sustave s prilagodljivim ubrziva čima temeljen na domenski specifičnom jeziku
2. Adaptivni mehanizam prilagodbe izraza za iskorištavanje sklopovskog ubrziva ča primjenom domenskih transformacija na višim razinama neovisnim o detaljima sklopovske platforme s ciljem optimizacije performanci i energetske učinkovitosti.

Prvi dio doprinosa ostvaren je kroz prilagodbu programskog jezika visoke razine apstrakcije RISE, dodavanje potrebnih primitiva te integraciju s ciljnom heterogenom platformom. Drugi dio doprinosa razvijen je dodatno umjesto razvoja algoritama za prilagodbu ubrzivača. Naime, prilagodljivi ubrzivači čije je skoro postojanje bilo anticipirano na javnom razgovoru nisu još dostupni u formi prototipa ili komercijalnog proizvoda, što je razvoj drugog dijela doprinosa ne samo učinilo izlišnim, nego ga i u potpunosti onemogućilo. Adaptivni mehanizam prilagodbe kao drugi dio doprinosa demonstrira mogućnosti apstrahiranja arhitekture sustava te ultimativno povećanje efikasnosti sustava te procesa programiranja.

2. Teorijska pozadina

Mooreov zakon koji se može tumačiti kao de facto standard za projekciju broja tranzistora na silikonu, prema riječima njegovog autora prestat će vrijediti u bližoj budućnosti. Nadalje, percipirani eksponencijalni rast efektivne računalne snage opada radi dostizanja tri fizikalne barijere ili zida, a to su zid snage, zid paralelizma na razini instrukcije, te zid memorije. Iz svega navedenoga izravno slijedi da hardver budućnosti mora biti višejezgren, a radi postizanja veće performanse, te da se programi za takav hardver moraju pisati tako da iskorištavaju dostupne mogućnosti. Temelj svake klasifikacije arhitektura računala jest Flynnova klasifikacija koja dijeli računala s obzirom na pristup tokovima instrukcija i podataka. Podjela paralelizma kao koncepta obično se temelji na podatkovnom paralelizmu koji se bavi načinom na koji se podaci distribuiraju računalnim čvorovima, dok s druge strane stoji paralelizam zadataka čiji je fokus na razdvajanju i raspoređivanju zadataka, također na različitim računalnim čvorovima.

Iskorištavanje ovih dvaju navedenih tipova paralelizma moguće je na više razina. Paralelizam na razini instrukcije podrazumijeva preklapanje instrukcija tijekom njihovog izvođenja radi povećanja performansi. Tehnike za iskorištavanje ove vrste paralelizma su cjevovodi, odmotavanje petlji, predviđanje grananja te dinamičko raspoređivanje. Vektorske arhitekture predstavljaju drugi način iskorištavanja paralelizma, u kojem vektorski procesori kao prototipovi

SIMD arhitekture implementiraju ideju primjene jedne instrukcije paralelno nad više podataka. Paralelizam na razini dretve iskorištava se na razini operacijskog sustava, dok se paralelizam na razini zahtjeva obično iskorištava na visokim razinama gdje su zadaci koje računalo izvršava potpuno disjunktni. Programski modeli za paralelne sustave apstrahiraju paralelne arhitekture, a mogu se promatrati s obzirom na način na koji pristupaju dekompoziciji problema, te na način na koji procesi stupaju međusobno u interakciju. Dekompozicija problema svodi se na prethodno navedene podatkovni paralelizam i paralelizam zadataka, dok se modeli s obzirom na interakciju dijele na prosljeđivanje poruka, dijeljenu memoriju te implicitnu komunikaciju. Implementacije tih modela su obično Pthreads, OpenMP te MPI.

Heterogenost u kontekstu računalnih sustava implicira računalni sustav ili procesni čvor koji se sastoji od više jezgara različitih arhitektura. Implementacija takvih sustava najčešće znači korištenje procesora opće namjene uparenog s grafičkim procesorom opće namjene ili specijaliziranim ubrzivačem. Specijalizirani ubrzivači trenutno postižu najbolje performanse i energetsku učinkovitost, a trenutno se najčešće manifestiraju kao ubrzivači u domeni neuronskih mreža, kriptografskog procesiranja ili pojedinih matematičkih operacija. Prilagodljivi ubrzivači koji balansiraju između mogućnosti iskoristivosti u više domena, te performanse i energetske učinkovitosti trenutno su u razvoju. Također, ubrzivači te rekonfigurabilni hardver već je neko vrijeme dostupan u oblaku. Nasuprot ubrzivačima, grafički procesori opće namjene koriste se u podatkovno paralelnim aplikacijama te su već neko vrijeme razmjerno popularno rješenje u odgovarajućim problemskim domenama. Općenito, heterogenost se može postići i kombiniranjem različitih arhitektura konceptualno sličnih procesora.

3. Općeniti koncepti

Programiranje i izvođenje aplikacija zaseban su problem u domeni heterogenog računarstva. Više različitih vrsta procesnih jedinica, kao i potencijalno dijametralno suprotne aplikacije koje bi se mogle izvoditi u sustavu predstavljaju značajan problem za programere. Jedan od mogućih rješenja jest podizanje razine apstrakcije razvojem programskog modela koji apstrahira arhitekturalne detalje sustava, često kao domenski specifičan jezik. Domenski specifični jezici su programski jezici koji korištenjem prikladnih oznaka i apstrakcija nude mogućnost izražavanja u jednoj problemskoj domeni. Interni domenski specifični jezici ugrađeni su unutar postojećeg programskog jezika, dok oni eksterni zahtijevaju razvoj vlastite prevoditeljske ili izvedbene infrastrukture. Jedan od programskih jezika opće namjene koji nude konstrukte kojima se olakšava razvoj domenskih specifičnih jezika jest Scala, a programsko rješenje koje se koristilo za razvoj doprinosa ove disertacije upravo je jezik duboko ugrađen u programskom jeziku Scala. Samo područje domenski specifičnih jezika trenutno uživa značajnu pozornost istraživačke i industrijske zajednice te se prirodno nameće kao rješenje za problem opisan disertacijom.

Kada se govori o heterogenom hardveru, ali i o otvorenom hardveru općenito, kao polazna osnova odabrana je arhitektura skupa instrukcija RISC-V. Arhitekture skupa instrukcija

općenito apstrahiraju računalo na razini skupa asemblerskih instrukcija te predstavljaju sučelje između hardvera i softvera. RISC-V je jedan takav skup koji je trenutno uhvatio velik zamah u akademiji i industriji. Kako su argumentirali inicijalni autori koncepta, skup instrukcija procesora predstavlja najznačajnije sučelje u računalu općenito, te je logično da, s obzirom na to da ne postoji značajna tehnička zaprjeka, takvo sučelje bude otvoreno. Sam RISC-V procesor je zamišljen da bude jednostavan, ali da anticipira različite primjene, od ugradbenih računala, do poslužitelja. Iz perspektive arhitekture, procesor je vrste *load-store* s *little-endian* poretkom bajtova te operacijama kojima su operandi registri. RISC-V arhitektura je također modularna i tipično se sastoji od baznog i minimalnog skupa instrukcija te proširenjima koje odgovaraju potrebama računala koje se dizajnira. Bazni instrukcijski skupi podržavaju cjelobrojne skupove ili ugradbene skupove sa širinama adresa od 32, 64 ili 128 bita. Ekstenzije instrukcijskom skupu dodaju različite funkcionalnosti, od hardverskog množila, podrške za *floating-point* operacije različitih preciznosti, kompresirane instrukcije, vektorske operacije i sl. Za validnost službenog dijela instrukcijskog skupa brine se neprofitna udruga, a korisnici mogu samostalno proširivati instrukcijski skup korištenjem ekstenzije **X**. Heterogeni sustav GAP8 korišten kao hardverska platforma u ovoj disertaciji koristi procesorske jezgre RISC-V tipa RV32IMC.

Jedan od većih projekata proizašao iz RISC-V ekosustava jest PULP. PULP je krovni projekt dvaju europskih sveučilišta čiji je cilj razvoj otvorenih i slobodnih hardverskih platformi za potrebe istraživanja i industrije. Projekt za cilj ima optimiranje potrošnje energije na skali milivata kako bi zadovoljio računalne potrebe aplikacija u internetu stvari. Hardver koji proizlazi iz projekta je relativno modularan što omogućava razvoj i optimiranje hardvera za specifične aplikacije uz postizanje značajnih stupnjeva optimizacije i energetske učinkovitosti. Projekt obuhvaća procesore, jednojezgrene sustave, višejezgrene sustave, višeklasterske sustave, te akceleratore. Iz projekta je proizašla i platforma GAP8, korištena kao ciljni hardver u disertaciji. Konkretno, korišten je koncept višejezgrenog sustava (projekt *Mia Wallace*), višeklasterskog sustava (projekt *HERO*) te ubrzivač (projekt *HWCE*).

GAP8 je heterogena računalna platforma proizvođača GreenWaves Technologies, a primaran joj je cilj pružiti mogućnost relativno visokih performansi na rubnim uređajima koji radi svojih specifičnih zahtjeva, npr. baterijskog napajanja, moraju ostati u domeni niske potrošnje. GAP8 primarno se sastoji od dva disjunktna dijela, od *fabric* kontrolera koji gravitira oko procesora opće namjene, te oko prilagodljivog ubrzivača. *Fabric* kontroleru sadrži procesor opće namjene arhitekture RV32IMC, a primarna zadaća mu je pokretanje sustava, aplikacija te orkestriranje vanjskih jedinica. S druge strane, klaster se sastoji od osam RISC-V jezgri identične arhitekture kao i procesor opće namjene u *fabric* kontroleru, te dodatno sadrži i specijalizirani hardverski ubrzivač operacije konvolucije. Memorijski podsustav GAP8 ugrubo se može podijeliti na memoriju dostupnu isključivo *fabric* kontroleru, memoriju dostupnu isključivo jezgrama u klasteru, podijeljenu po segmentima za svaku procesorsku jezgru, te memoriju kojoj

mogu pristupiti oba podsustava. Upravo je potonja korištena za izvedbu doprinosa disertacije. Iako najsporija, ta je memorija najveća te za prednost ima mogućnost pristupa iz oba podsustava čime se omogućava jednostavan pristup bez sinkronizacijskih mehanizama. Hardverski ubrzivač operacije konvolucije sklopovski izvodi operaciju konvolucije nad dvodimenzionalnim ulaznim signalom, te filtrima dimenzija 3×3, 5×5, 7×7 te 7×4. Ubrzivač je izložen kroz zaseban API koji se pokazao relativno nestabilnim, pogotovo u kontekstu disjunktnog korištenja u odnosu na komponente SDK-a namijenjene procesiranju inferencije na neuronskim mrežama u klasteru.

4. Programski okvir RISE

RISE je podatkovno paralelan programski jezik temeljen na uzorcima, visoke razine apstrakcije te izveden kao domenski specifičan jezik, duboko ugrađen unutar programskog jezika Scala. Glavni naglasak programskog jezika RISE jest izlaganje sučelja koje će omogućiti programerima te znanstvenicima unutar specifičnih domena da izračune opišu odgovarajući na pitanje *što* treba izračunati, umjesto da nizom imperativnih naredbi opisuju *kako* se pojedina operacija treba izvesti. RISE je uparen s programskim jezikom ELEVATE, također izvedenim u obliku domenski specifičnog jezika unutar programskog jezika Scala, a čija je glavna uloga izražavanje optimizacijskih strategija. Oba jezika ulaz su u prevoditelj Shine koji primjenjuje transformacije te generira optimirani kôd niske razine. Cjeloviti RISE okvir razvija se na Sveučilištima u Edinburghu, Glasgowu i Münsteru, te je korišten za ostvarivanje doprinosa ove disertacije. Ulaz u prevoditelj jest izračun izražen u programskom jeziku RISE, te optimizacijska strategija u programskom jeziku ELEVATE. Prevoditelj u procesu prepisivanja primjenjuje transformacije te generira RISE izraz niske razine u kojemu su direktno enkodirane optimizacijske odluke. Nadalje, u procesu prepisivanja koristi se hibridni funkcijsko-imperativan međureprezentacijski jezik DPIA (*Data Parallel Idealised Algol*). RISE izraz niske razine prepisuje se u funkcijsku DPIA međureprezentaciju, a nakon toga u imperativnu DPIA međureprezentaciju. Ta se imperativna međureprezentacija na kraju prevodi u čvorove apstraktnog sintaksnog stabla za ciljni programski jezik ili platformu, te u konačnici u sam programski kôd niske razine, nativan za ciljnu platformu.

Programski jezik RISE sadrži standardne konstrukte tipične za (funkcijske) programske jezike općenito, primjerice identifikatore, lambda izraze ili literale. Posebna vrsta izraza su primitivi koji enkapsuliraju modularne operacije više ili niže razine. Primitivi mogu biti generički, odnosno primjenjivi na sve ciljne platforme, ili mogu biti specijalizirani za pojedinu platformu. Generički primitivi obuhvaćaju funkcije koje se mogu pronaći u drugim (funkcijskim) programskim jezicima, uključivo funkcije višeg reda, primjerice *map* i *reduce*, ali i *zip*, *join*, *slide* ili *pad*. S druge strane, optimizacijske strategije u programskom jeziku ELEVATE izražavaju se pravilima koja se onda ulančavaju odgovarajućim operatorima. Pravila prepisivanja mogu biti algoritamska, pa na taj način matematički dokazivim pravilima optimirati

izračun kombinacijom primitiva, npr. pravilo *mapFusion*. Također, pravila mogu biti i jednostavnija (*eng. lowering*) te se mogu koristiti za zamjenu generičkih primitiva konkretnim implementacijama. Jednostavan primjer jest zamjena generičkog primitiva *map* konkretnom paralelnom implementacijom *mapPar*. Pravila se primjenjuju ondje gdje je određeno kombinatorima koji rekurzivnim prolaskom kroz izraz u programskom jeziku RISE traže odgovarajuće uzorke te ih prepisuju kako pravilo nalaže. Konkretni kombinatori mogu tražiti uzorke izvana, iznutra ili unutar cijelog izraza.

5. Implementacija modela

Prvi dio očekivanog znanstvenog doprinosa ostvaren je korištenjem i odgovarajućim proširenjima programskog jezika RISE, te pripadajućeg radnog okvira. S obzirom na to da ciljna heterogena platforma GAP8 dijeli neke koncepte s platformama podržanima od strane jezika OpenCL, a primarno koncept domaćina (*eng. host*) i uređaja (*eng. device*) u sustavu koji idejno odgovaraju *fabric* kontroleru i klasteru na platformi GAP8, implementacija se u većim dijelovima oslanja upravo na komponente inicijalno predviđene za platformu OpenCL. Doprinos je u većoj mjeri ostvaren kroz komponente modula, generatora modula te generatora kôda. Prva razvijena komponenta je *GAP8 Module* koja enkapsulira validan dio kôda koji se može pokrenuti na ciljnoj platformi, na način da sadržava podmodul za domaćina te sekvencu podmodula za uređaj tj. klaster. Jedna aplikacija enkapsulirana na ovaj način može podržavati više funkcija koje će se izvršiti na klasteru. Oba podmodula su vrste modula za programski jezik C. Modul dodatno sadržava metodu koja ga prevodi u ciljni programski jezik te je zadužen za injektiranje odsječka kôda za raspakiravanje parametara unutar funkcije koja se izvodi na klasteru. S obzirom na to da API za GAP8 ne podržava slanje više od jednog parametra u trenutku pokretanja izvođenja na klasteru, injektiraju se odsječci kôda koji u strukturu pakiraju parametre prije slanja, *castaju* strukturu u pokazivač tipa *void*, te nakon zaprimanja takvog parametra na strani klastera, ponovno ga otpakiravaju u zasebne parametre. Modul je zadužen za otpakiravanje parametara dok se pakiranje parametara događa prilikom generiranja kôda za stranu domaćina.

Odgovornost generiranja kôda podijeljena je među komponentama zaduženima za stranu akceleratora, tj. klastera, te domaćina. Dodatno, strana domaćina sadržava i generator modula za domaćina. Strana klastera ponovno iskorištava ranije ugrađene mogućnosti radnog okvira te se na njih oslanja prilikom generiranja modula za klaster. Generiranje kôda za klaster izvedeno je proširenjem postojećeg generatora za model OpenMP te se na ovoj razini dodaje podrška za hardverski ubrzivač operacije konvolucije. U slučaju nailaska na odgovarajući imperativni DPIA primitiv, generator kôda generirat će seriju poziva koji na niskoj razini omogućuju izvedbu operacije konvolucije na hardverskom ubrzivaču. Ti koraci su: instanciranje i pokretanje ubrzivača operacije konvolucije, odgovarajuća konfiguracija, izvršavanje operacije te u konačnici isključivanje ubrzivača. S druge strane, s obzirom na to da paralelizam nije izvediv na strani domaćina, generator kôda za procesor domaćina oslanja se na generator za programski

jezik C. Primarna zadaća ovog generatora jest generiranje poziva za niske razine za izvršavanje izračuna na klasteru, pakiranje parametara koji se prosljeđuju klasteru te generiranje memorijskih sinkronizacijskih poziva. Iako se trenutno sinkronizacija memorije ne provodi nego se aplikacije oslanjaju na mogućnost izravnog pristupa podacima koji se nalaze u memoriji dostupnoj s obje strane sustava, preuzimanjem koncepta iz podrške za OpenCL platforme prevoditelj anticipira buduću mogućnost za takvom sinkronizacijom. Na taj način stvara se preduvjet za buduće optimiranje generiranog kôda s obzirom na prijenos podataka s domaćina na klaster i obrnuto. Generator modula domaćinskog kôda ulančava tok koji zadaje sučelje generatora modula na način da ponovno iskorištava dijelove generatora za platformu OpenCL te ih povezuje s direktivama za generiranje kôda i generiranje modula domaćinskog kôda. Produkt generatora modula domaćinskog kôda jest modul za programski jezik C koji enkapsulira strukturu s varijablom tipa *Kernel* za svaku funkciju koja se izvodi na klasteru, *typedef* deklaraciju za navedenu strukturu, direktivu uključivanja za odgovarajuću *header* datoteku s podrškom za izvedbenu okolinu, seriju funkcija za inicijaliziranje, pokretanje i uništavanje *kernela*, te u konačnici funkciju *main* kao ulaznu točku za izvršavanje aplikacije.

Ključna funkcionalnost sustava jest mehanizam za pokretanje izvođenja na klasteru, a ona je ostvarena kroz dodavanje odgovarajućih primitiva na svim razinama jezika, odnosno radnog okvira. Na razini programskog jezika RISE, dodan je primitiv *gap8run* koji kao parametar prima broj jezgri na kojem se izračun želi izvršiti, dok kao drugi parametar prima izraz u programskom jeziku RISE kojim je opisan izračun. Na razini funkcijske DPIA međureprezentacije, dodana su dva primitiva, *Run* te *KernelCall*. Primitiv *Run* je primitiv u koji se *gap8run* inicijalno prevodi, a koji se u procesu odvajanja razdvajanja inicijalnog izraza na dio koji će se izvršiti na domaćinu te na dio koji će se izvršiti na klasteru prevodi u primitiv *KernelCall*. Razdvajanje izraza na ta dva dijela odvija se u komponenti *SeparateHostAndAcceleratorCode* a svodi se na obilaženje izraza te konstrukciju funkcijske definicije kada se naiđe na dio izraza koji je za izvođenje na klasteru označen prethodno navedenim *gap8run* primitivom. Naposljetku, funkcijski DPIA primitiv *KernelCall* prevodi se u imperativni DPIA primitiv *KernelCallCmd* koji direktno uzrokuje generiranje konstrukata za pokretanje zadanog izraza u klasteru. Za podršku hardverskog ubrzivača operacije konvolucije dodana su po četiri primitiva na svakoj razini, tj. na razini programskog jezika RISE te na razinama imperativne odnosno funkcijske DPIA međureprezentacije. Svaki od dodanih primitiva odgovara jednoj od hardverski podržanih operacija konvolucije s filtrima različitih dimenzija.

Jedan od inicijalnih ciljeva ove disertacije bio je podići razinu apstrakcije za kompleksne heterogene sustave. Jedan od razloga takvog pristupa jest i taj što takve sustave često koriste znanstvenici u pojedinim domenama bez iskustva u području programiranja. Dodatno, kompleksni heterogeni sustavi za puno iskorištavanje njihovih mogućnosti često zahtijevaju poznavanje arhitekturalnih detalja na niskoj razini, te na taj način čine cjelokupni proces pro-

gramiranja takvog sustava inherentno teškim. Podizanje razine apstrakcije *de facto* zahtjeva prebacivanje odgovornosti za poznavanje tih detalja na prevoditelj. Uzevši sve u obzir, a u svrhu ostvarenja cilja disertacije, implementiran je mehanizam detekcije odgovarajućih uzoraka koji se mogu izvesti u hardverski implementiranim specijaliziranim ubrzivačima. Konkretno, za platformu GAP8, u programskom jeziku ELEVATE implementirano je pravilo koje prepoznaje slijed uzoraka operacije konvolucije te ga prevodi u jedan od prethodno opisanih odgovarajućih konvolucijskih primitiva u programskom jeziku RISE. Na taj način se omogućuje da se ispravno izražena operacija konvolucije izvrši u hardverskom ubrzivaču operacije konvolucije, a bez eksplicitnog navođenja poziva te bez potrebe za poznavanjem arhitekturalnih detalja ciljne platforme. Konkretni primitiv u koji se izraz prevodi ovisi o veličini susjedstva koji je zadan kao parametar primitivu *slide2D*.

Na niskoj razini, kao dio potporne infrastrukture za generiranje kôda, implementirana je i knjižnica za izvedbenu okolinu (*eng. runtime library*). Ta knjižnica izlaže sučelje koje je unificirano za ciljne platforme koje podržavaju koncept domaćina i uređaja, a cilj jest smanjiti količinu repetitivnog kôda koju generator kôda unutar RISE okvira mora generirati. Generator kôda generirat će pozive izložene ovom knjižnicom, a pojedine platforme implementiraju pozive na niskoj razini, uvažavajući vlastite specifičnosti. Knjižnica je izvorno nastala za OpenCL platforme, a na odgovarajući način je proširena te implementirana za platformu GAP8. Glavni koncepti koje knjižnica uvodi su kontekst, jezgra (*eng. kernel*) te *buffer*. Kontekst enkapsulira sve potrebne informacije za izvođenje akceleratorske funkcije na klasteru, dok jezgra (*kernel*) enkapsulira samu akceleratorsku funkciju koja će se izvesti na klasteru. *Buffer* predstavlja dio memorije u kojem se nalaze podaci nad kojima se izvršavaju operacije. Uvođenjem koncepta *buffera* anticipira se mehanizam transfera podataka s domaćina na uređaj i obrnuto, ali se ti transferi u ovom trenutku ne događaju, što je indicirano implementacijom u datoteci *nosync.c*.

Implementacija modela zaokružena je komponentom izvođača koja omogućava izvođenje programa u prirodnom okruženju okvira RISE, ali na ciljnoj hardverskoj platformi ili unutar simulatora. Izvođač enkapsulira generiranje kôda, prevođenje na niskoj razini nativnim prevoditeljem za ciljnu platformu, te izvršavanje na ciljnoj platformi. Kao parametre moguće je specificirati ciljnu platformu u vidu testne pločice ili simulatora, operacijski sustav, te kanal za komunikaciju s programerom. S obzirom na to da se na niskoj razini programi za GAP8 prevode korištenjem programa *make*, izvođač popunjava varijable u predefiniranoj datoteci *Makefile*.

6. Evaluacija modela

Kao dodatni doprinos, prethodno izloženi model ekstenzivno je evaluiran s obzirom na performanse, energetsku učinkovitost te programirljivost. Evaluacija je provedena na način uspoređivanja prethodno navedenih parametara za ručno optimiran kôd te kôd generiran proširenim programskim okvirom RISE. Metodologija mjerenja uvijek je bila identična, a sastojala se od pet uzastopnih mjerenja, od kojih je prvo odbačeno kako bi se eliminirali potencijalni negativni

efekti utitravanja ispitne okoline. Performansa i energetska učinkovitost mjerila se samo za izračun od interesa, dok podizanje sustava, učitavanje podataka te ispis rezultata nisu bile uzimane u obzir. U konačnici, dobiveni rezultati prikazani su kao prosjek i standardna devijacija uzorka. Za provođenje mjerenja pratile su se službene upute proizvođača ispitne okoline.

Mjerenje performanse mjerilo se dvama parametrima: praćenjem broja aktivnih ciklusa te stvarno proteklog vremena potrebnog za izvršavanje s obzirom na "zidni sat". Za oba parametra korišten je GAP8 API, a na odgovarajuća mjesta u kôdu ubačeni su odsječci koji pokreću mjerenje prije pokretanja ispitnog izračuna, te zaustavljaju mjerenje nakon izračuna. Energetska učinkovitost mjerena je kroz potrošnju energije za vrijeme trajanja izračuna od interesa. Prva diferencijalna sonda osciloskopa spojena je na izvode internog DC/DC regulatora GAP8 čipa. Između izvoda smješten je otpornik vrijednosti 1 Ohm čime je izravno omogućeno mjerenje struje kroz otpor te posljedično snage u diskretnim trenutcima izvođenja. Uzorkovanjem sonde, mjerene su vrijednosti napona te je izračunata snaga, a numeričkom integracijom snage s obzirom na vrijeme izvođenja, izračunata je potrošnja energije. Za verifikaciju, energija je izračunata i korištenjem prosječne snage za vrijeme izvođenja izračuna od interesa. Za označavanje izračuna od interesa korištena je druga diferencijalna sonda osciloskopa koja je bila spojena na GPIO priključnice sustava. Ta je priključnica na odgovarajućim mjestima u programskom kôdu bila postavljana u logičku jedinicu te spuštana u logičku nulu. Označavanje izračuna od interesa pokazalo se potrebnim iz razloga fluktuiranja napona na izvodima internog DC/DC regulatora čime je onemogućeno da se vršne vrijednosti tog napona smatraju početkom i krajem izračuna od interesa. Osciloskop je obje sonde uzorkovao frekvencijom od 1 kHz.

Za ispitne scenarije uzete su aplikacije iz stvarne primjene. Prvi ispitni scenarij je množenje matrica, a korištene su matrice dimenzija 250×250 elemenata. Drugi ispitni scenarij je Sobelov filter kojime se u slikama naglašavaju oštriji kontrastni prijelazi. Iako se ovaj filter svodi na operaciju konvolucije s dva različita filtra te geometrijsku sredinu kao konačan rezultat, prilikom izračuna radi nestabilnosti nije korišten hardverski ubrzivač operacije konvolucije. Testni podatak je monokromatska slika dimenzija 320×240 piksela. Treći i posljednji ispitni scenarij je klasteriranje k-sredinama, algoritam koji ima izravne primjene u strojnom učenju, a koristi se za klasteriranje m točaka s n dimenzija u k klastera. Za testiranje korišten je skup podataka o 250 točaka s dvije dimenzije u tri klastera, a postupak je provođen u 1000 iteracija algoritma. Barnes-Hut simulator n tijela izbačen je kao ispitni scenarij radi poteškoća u prilagodbi ručno optimiranog kôda za ispitnu platformu. Jedan ispitni scenarij korišten je u svrhu verifikacije, te za njega nisu provedena mjerenja. Taj scenarij jest verifikacija prevođenja operacije konvolucije izražene u programskom jeziku RISE za izvođenje na hardverskom ubrzivaču operacije konvolucije. Za ovaj slučaj prikazana je primjena jednostavne optimizacijske strategije koja izraženu konvoluciju prilagođava za izvođenje na hardverskom ubrzivaču.

Postignuti rezultati za performanse su zadovoljavajući. S obzirom na mjerenje protek-

log vremena ispitnog izračuna, generirani kôd postiže rezultate koji su marginalno bolji od ručno optimiranog kôda. Kada se promatraju rezultati s obzirom na broj aktivnih ciklusa, rezultati su također marginalno bolji od ručno optimiranog kôda, s izuzetkom klasteriranja k-sredinama gdje je performansa generiranog kôda značajnije veća od ručno optimiranog kôda. Potonji slučaj može se zanemariti jer je očekivano da rezultat ostvaren brojem aktivnih ciklusa odražava rezultat ostvaren proteklim vremenom. Moguće odstupanje, iako pozitivno za doprinos disertacije, vjerojatno je uzrokovano internim brojačima performansi unutar GAP8 API, a koji u ovom trenutku radi nedostatka resursa nije podrobnije analiziran. Evaluacija energetske učinkovitosti također je dala zadovoljavajuće rezultate. Za sva tri ispitna scenarija, te za obje metode mjerenja, rezultati generiranog kôda sumjerljivi su ručno optimiranom kôdu. Scenariji klasteriranja k-sredinama i Sobelovog filtra marginalno su bolji za generirani kôd, dok je rezultat množenja matrica marginalno lošiji, ali svejedno sumjerljiv ručno optimiranom kôdu.

Mjerenje programirljivosti izvršeno je implicitno, a s obzirom na to da su se metrike za ocjenu programirljivosti pojedinih programskih modela pokazale nepouzdanima ili subjektivnima, za ovu svrhu prebrojane su linije programskog kôda za ručno optimirani kôd, generirani kôd, te kôd u programskom jeziku RISE. Za sve slučajeve kôd u programskom jeziku RISE značajno je kraći od ručno optimiranog kôda. Generirani kôd je duži od ručno optimiranog kôda, ali se taj slučaj može zanemariti s obzirom na to da se generiranje kôda može percipirati kao jedan od koraka u prevoditeljskom lancu. Kôd u programskom jeziku RISE je kraći, čistiji i koncizniji te se može tvrditi da je uz postignute sumjerljive rezultate za postignute performanse i energetsku učinkovitost prikladan za korištenje u heterogenim sustavima.

7. Zaključak

Vršne performanse superračunala današnjice mjerene jedinicom FLOPS, trenutno su reda veličine *petaFLOPS* (10^{15}). Radi napretka čovječanstva, a posljedično i znanosti, anticipira se skorija potreba za ulaskom tzv. *exascale* domenu u kojoj će vršne performanse biti u razini *exaFLOPS* (10^{18}). Za dostizanje *exascale* domene, ključni faktori su efikasno iskorištavanje paralelizma te heterogenost. Jedan od problema s kojima se programeri i znanstvenici susreću u radu s takvim, visoko paralelnim i heterogenim, sustavima jest programiranje. Efikasno iskorištavanje dostupnih resursa često zahtjeva poznavanje arhitekturalnih detalja niske razine. Cilj ove disertacije bio je podići razinu apstrakcije, te izložiti jezik visoke razine za efikasnije programiranje kompleksnih heterogenih sustava. Taj cilj je i dostignut proširenjima programskog jezika RISE te pripadajućeg okvira. Usporedba performansi te energetske učinkovitosti postignutih generiranim kôdom u odnosu na ručno optimirani kôd pokazuje da je moguće barem zadržati, a često i postići bolje performanse i energetsku učinkovitost.

Buduće optimizacije na praktičnom dijelu doktorskog rada moguće su u više pravaca. Prvi mogući pravac jest optimizacija i bolje iskorištavanje memorijskog podsustava. Trenutno se svi podaci nad kojima se vrši obrada pohranjuju u memoriji koja je dostupna cijelom sustavu,

ali je ujedno i najsporija. Paralelizacijom obrade i transfera podataka u bržu memoriju rezerviranu isključivo za klaster, nedvojbeno bi se postigle više razine performansi i energetske učinkovitosti, uz zadržavanje iste razine programirljivosti. Optimizacijska strategija za detekciju uzoraka koji predstavljaju operaciju konvolucije je prespecifična te ne anticipira mogućnost da se konvolucija izrazi na moguće drugačiji način, te je strategiju u tom kontekstu moguće dodatno optimirati. S druge strane, strategiju se može generalizirati i na način da se implementira mogućnost operacija konvolucije za signale i filtre različitih dimenzionalnosti. Zadnji pravac u kojem je moguće dati doprinos jest dodatno pojednostavljenje modela. Trenutni model predstavlja napredak u odnosu na nativan programski model za ciljnu platformu, ponajprije radi izlaganja konciznijeg sučelja, ali i sakrivanja arhitekturalnih detalja niske razine. Ipak, funkcijska programska paradigma, još uvijek je relativno nepoznata te se model može prilagoditi dodatno kako bi se programiranje kompleksnih heterogenih sustava dodatno pojednostavnilo.

Ključne riječi: domenski specifični jezici, heterogeni sustavi, paralelno procesiranje, višejezgreni sustavi, RISC-V

Contents

Chapter 1

Introduction

Nowadays as we observe stable and constant growth in science and the general advancement of humanity, the increased need for computational resources is observed as well. Raw computing performance measured in FLOPS (Floating Point Operations Per Second), a de facto standard in measuring computational performance, especially in the high-performance computing domain, according to the Top500 list [1], is currently in Peta (10^{15}) order of magnitude. Although currently achieved peak performance levels satisfy today's needs for computing power, the need to enter the ExaFLOPS (10^{18}) domain with affordable energy consumption is anticipated and is currently one of the goals for computer engineers and scientists. In his opening keynote at the international high-performance computing conference SC20, Bjorn Stevens, a professor at the Max Planck Institute for Meteorology in Hamburg, Germany said that a real need for exascale computing power exists. As an example, he pointed out the massive data sets created by climate science, while stressing out that the computing power has to be accessible to a wide range of people [2].

The importance of achieving the Exascale domain is confirmed by the existence of the research projects in different parts of the World, e. g. "Exascale Computing Project" in USA [3], Japan [4] with allegations that China has already reached exascale on two separate systems [5]. Japan's Fugoku supercomputer built on ARM processor architecture and Tofu interconnect D already reached exascale domain in single or further-reduced precision [1]. European Union also heavily invests in High-performance computing, with aims towards Exascale as well [6], with a couple of notable projects like EuroHPC initiative [7], European Open Science Cloud (EOSC) [8], and ExaNode project [9].

The latter is heavily influenced by the current geopolitical perspective, due to the fact that most of the processing cores manufacturers are non-European. Intel, AMD, and NVIDIA are located in the USA, while ARM resides in the UK which can't be considered to be politically tied to the European Union any longer. Furthermore, NVIDIA is in process of acquiring ARM which will even more derogate the European intelligence on processing cores design.

Everything mentioned renders Europe and the European Union vulnerable, which is additionally amplified by the rise of tensions on the current geopolitical map of the world. In light of that, it is necessary to mention Europe Processor Initiative (EPI) [10] which aims to deliver a new family of low-power European processors for extreme-scale computing, high-performance Big-Data and a range of emerging applications [11], together with the processing cores design know-how, guaranteeing its processing sovereignty. EPI is a project currently implemented under the special sponsorship of the European Commission which further stresses the importance of the project in the context of European computing sovereignty, both in High-Performance and general computing domains.

The so-called Exascale domain opens up many challenges, mainly in the areas of computer architecture, heterogeneity, and energy efficiency, and it is exactly the heterogeneity that is one of the key development points with respect to reaching the Exascale domain [12]. Heterogeneity, in the most general case, purports the usage of multiple processing cores with different purposes and architectures inside one relatively isolated computational node.

Usually, considering a computational node, besides a general-purpose processor, heterogeneity implies the existence of the general-purpose graphics processing unit (GPGPU) or accelerator for specific applications or tasks. The concept of heterogeneity gained popularity exactly when graphic processing units' manufacturers exposed their cores to general computing by the appropriate APIs, CUDA [13] for NVIDIA chips and OpenCL [14] for AMD chips. General-purpose graphic processing units introduced a significant increase in performance in the intensive data-parallel applications. In these domains, they are significantly more energy-efficient than general-purpose processors. Also, they can be simply horizontally scaled by switching to newer models, which represents a perk being reasoned by the fact that Moore's law is still applicable to them. A major advantage is a considerably simpler programming model for the average programmer, exposed through the aforementioned CUDA and OpenCL, with the addition of OpenMP, when compared to programming and synthesizing custom accelerators.

The other important type of processing unit in the context of heterogeneity are customized accelerators which are implemented as locked, custom components, or customizable programmable logic implemented in FPGA chips. Such accelerators achieve higher performance and energy efficiency compared to conventional processing engines and graphic processing units, but that comes with a price: complicated programming model at the level of hardware design, lack of scalability, lack of portability, and usability only in application domain they were designed for. Tools that ease the implementation of customized accelerators exist, i.e. Chisel [15] or High-level synthesis techniques, but they haven't, due to multiple reasons, been caught on in general usage.

The combination of types of processing units in a heterogeneous environment is generally context-dependent. In embedded devices where energy efficiency and spatial conservativeness

prevail, heterogeneous devices usually combine a general-purpose processor with an application accelerator. On the other hand, High-performance computing usage scenarios utilize general-purpose graphic processing units and application-specific accelerators as well.

Intel's acquisition of Altera (today known under the name Intel FPGA), one of the two major FPGA suppliers, clearly indicates that future computing platforms will add more heterogeneity in the form of reconfigurable fabric, while in the high-performance computing domain, several initiatives already employ heterogeneous systems with CPUs, GPGPUs and large arrays of FPGA fabric, such as Amazon's EC2 F1 [16] and Microsoft's Catapult [17], with the latter being now deployed in nearly every new server across the more than a million machines that the Microsoft's hyperscale cloud consists of [18]. The trend is also visible in the Top500 list where nearly 30% of the systems use the accelerator or co-processor technology [19]. The main reason accelerators are not (yet) more widely employed in high-performance systems is the difficulty encountered in writing software that exploits accelerators' capabilities efficiently [20], a problem which persisted up until today.

Given the multiple advantages and drawbacks of both the general-purpose graphic processing units and customized acceleration engines, there exists a large gap for customizable accelerators which would combine advantages while minimizing flaws of both approaches. Such accelerators are flexible enough to avoid applicability-lock on just one type of application, and in every application domain they would achieve performance and energy efficiency commensurate with completely customized accelerators. Packed with the programming interface exposed through a domain-specific language, a complicated programming model could be avoided as well. In addition, their adjustability enables co-design and high-level architectural explorations for performance or energy efficiency optimization of the system running target application from one of the selected disruptive domains, such as machine learning, crypto processing, or multimedia processing.

When discussing parallelism and heterogeneity in education, most of the curricula that are being thought at the universities worldwide consist of teaching students sequential programming in the introductory courses. The problem arises because sequential programming heavily influences the sequential way of thinking. As stated before, virtually all of the computers that are surrounding us are multicores, and while we are surrounded by multicores, we still tend to teach students sequential programming and thinking [21]. Arguably, teaching the parallel way of thinking since the inception of one's education in computer science could be better than teaching imperative and sequential paradigms to only then teach how to parallelize algorithms or code in general. Furthermore, the latest guidelines published jointly by the ACM and the IEEE in 2013. recommended integration of the parallelism throughout the curriculum [22]. Parallel programming, together with the exploitation of heterogeneous hardware is inherently hard, starting from exploiting general-purpose graphic processing units to specific accelerators.

To ease programming and increase the exploitation of computational power of such systems we have to develop new methods and models.

Finally, following everything discussed so far, this thesis aims to cope with the emerging heterogeneity and the inherent need to increase level of parallelism, together with the need of easing parallel programming models by providing a novel programming model. The provided model is formed as a domain-specific language which with the provided compiler infrastructure compiles to the native model for the target in question.

1.1 Research goals

The main goal of this research is to provide an adequate programming model for complex heterogeneous systems with scalable and customizable acceleration engines based on the computational patterns that constitute the most compute-intensive kernels in selected application domains, as well as support computations that heavily rely on integer or fixed-point bit-manipulation with extreme producer-consumer parallelism. From the theoretical point of view, the research will provide a comprehensive review of the programming models used for heterogeneous systems with an emphasis on compilers and infrastructure which implicitly encourages and supports parallelization. The practical implication will be a programming language for heterogeneous systems with customizable accelerators. One of the outcomes is to evaluate the complete solution, namely the implemented programming model, to test it and compare it with the hand-tuned code both in domains of performance and energy efficiency.

The hypotheses of the research are:

1. It is possible to express complex problems stemming from various domains in a domain-specific language which can target heterogeneous systems
2. Performance of the code generated by the provided infrastructure is better or at least on par with the hand-tuned code
3. Energy efficiency of the hardware which executes code generated by the provided infrastructure is better or at least on par with the hand-tuned code
4. Programmability of the complex heterogeneous systems is higher when using the proposed programming model compared to the conventional programming model for the same platform

The starting point of this research is the efficient programming abstraction for programming complex heterogeneous systems with customizable accelerators. Given the context, the ground case for the research is a domain-specific language layer that will be modeled or adjusted to conform to the needs of such systems. The proposed language will be implemented together with the rest of the components defined by the expected scientific contribution.

Regarding the planned testing equipment, an adequate system has been detected in GAP8

[23] chip by GreenWaves Technologies. GAP8 is a heterogeneous system on chip which is consisted of a general-purpose processing core based on RISC-V ISA named Fabric Controller and an accelerator featuring 8 cores and a custom hardware convolution engine. Both fabric controller and cores available in the cluster implement the same RISC-V instruction set, that is RV32IMC. GAP8 stems from the PULP project [24], more specifically from *Mia Wallace* [25] and Open Heterogeneous Research Platform - HERO [26]. Concepts presented in both platforms can be found in GAP8. One such example is the bigPULP accelerator available in HERO being exactly the cluster in the GAP8 platform, with differences in the general-purpose processor which is ARM in HERO, and RISC-V based core in GAP8.

This thesis aimed to fulfill the expected scientific contribution proposed at the public thesis topic defense which is constituted through 2 parts:

1. Programming model for heterogeneous systems with customizable accelerators based on a domain-specific language
2. Algorithms for accelerator customization based on program features for performance and energy efficiency optimization

The first part of the expected scientific contribution is fulfilled through the extension of the RISE language and integration of the target platform in the Shine compiler infrastructure. Support for the target platform was carried out by the addition of the primitives necessary to support target-specific operations, together with primitives necessary to run expressions written in RISE on the target platform.

The development of the second part of the expected scientific contribution was heavily influenced by the eventual inexistence of the desired type of customizable accelerators that were anticipated by the project and the research described through this thesis. The idea proposed through public thesis topic defense anticipated the future existence of the accelerators that would expose tunable parameters, such as integer or floating-point precision, making them usable in more than one application domain. However, the concept of such accelerators, though promising, never ended with a usable prototype, let alone with a concrete and usable implementation. Some of the parameters can be set up, like clock frequency of the fabric controller and cluster, together with the voltage of the chip, but tuning those parameters can't be considered customizing the accelerator itself. However, given the final type of customizable accelerators that were used in this thesis, optimization techniques available in the Shine compiler were utilized. Furthermore, as an addition to the scientific contribution, support for the customized piece of hardware embodied through available Hardware Convolution Engine was added, together with optimization strategy which transforms one of the series of patterns that express convolution in RISE. The latter serves as a showcase of supporting customized hardware through pattern rewriting when a series of patterns that constitute a computation supported by the aforementioned customized hardware is detected. Also, additional optimizations are possible in the fu-

ture, regarding the detection of additional series of patterns that could constitute convolution operation, and thus be supported by the hardware convolution engine. Finally, this thesis also provides a number of benchmarks and example applications of the proposed model, which were not initially proposed by the expected scientific contribution.

1.2 Thesis outline

Besides the introduction which introduces a reader to the thesis topic, explains scientific contribution and hypotheses, this thesis is divided into seven chapters and two appendices which are organized as follows:

Chapter 2 Theoretical Background

This chapter additionally widens the motivation for research in this field by providing the necessary temporal context regarding the current state of the computing systems. Additionally, some fundamental concepts regarding heterogeneity, parallelism, and techniques to efficiently exploit parallelism on multiple levels are provided.

Chapter 3 General Concepts

This chapter covers a wide aspect of topics needed for the complete understanding of the thesis. Topics that are covered include domain-specific languages, RISC-V, PULP, and GAP8, with some of them packed together with the state-of-the-art in the respective topic.

Chapter 4 RISE Stack

Chapter **RISE Stack** gives a brief introduction to the data-parallel pattern-based RISE programming language and to ELEVATE, a language used to express optimization strategies. A brief overview of the programming framework used to utilize these two languages is provided, together with a description of the Shine compiler used to compile programs from expressions written in RISE to low-level code.

Chapter 5 Model Implementation

In **Model Implementation** chapter, every aspect of the extensions to the RISE framework is described, including code generation, module generation, code separation, expression transformation, and code execution.

Chapter 6 Model Evaluation

Chapter **Model Evaluation** evaluates the proposed solution on different benchmarks, including a benchmark that verifies code correctness for utilization of the HWCE, in terms of performance, energy efficiency, and programmability. The evaluation methodology is described, and the results are discussed.

Chapter 7 Conclusion

Finally, chapter **Conclusion** concludes this thesis with an overview of everything provided in it and some perspective proposals for future work.

Appendix A

This appendix provides an example of hand-tuned code, which was used in performance and energy efficiency comparisons. The hand-tuned code for the Sobel filter benchmark is provided.

Appendix B This appendix provides a couple of examples of the generated code used in comparisons. For consistency, generated code for the Sobel filter benchmark is provided. Additionally, an example of the generated code that utilizes HWCE is provided.

Chapter 2

Theoretical Background

This chapter provides the necessary background, in terms of concepts and literature overview necessary for the understanding of the thesis outcomes. The main concepts that are introduced are parallelism and heterogeneity with their description, techniques, and a brief overview of the concept's field.

2.1 General motivation

Moore's law, a de facto standard for approximation of transistor on integrated circuits stated that the aforementioned number doubles approximately every two years. The author of the law himself not so recently stated that he sees the "Law dying here in the next decade or so" [27, 28]. A similar conclusion came from the acting CEO of NVIDIA at the prestigious CES 2019 conference [29]. Although part of the chipmakers still does not perceive the law dead, it is indeed a fact that the on-chip transistor growth slowed down, mostly because the technology reached its physical limits.

Furthermore, the perceived exponential growth of the effective computing power faded away by hitting three types of "wall" [30], namely [31]:

- **Power wall** - As a result of the significant increase in frequency, the ability to dissipate heat has reached the physical limit. Furthermore, the ratio between energy consumption and increase in performance is not linear.
- **Instruction-Level parallelism (ILP) wall** - ILP causes a super-linear increase in complexity and power consumption of the processing unit without linear speed up in the application performance. To put it simply, one has to add complex pipelines that are not mirrored in an equal performance increase
- **Memory wall** - The mismatch between memory speed and computation speed

From the latter follows that augmented with the fact that Moore's law definitely sees its end at some point in the relatively near future, new hardware has to be designed and developed in

multicore or multichip fashion to deliver expected higher performance. Consequently, programs need to be developed in such a manner to exploit underlying multicore systems. Drive for the ever-lasting higher performance has to imply parallelism.

To achieve the desired increase in performance in multicore and heterogeneous environments, one has to exploit potential parallelism on every layer. Although most of today's languages offer explicit programming at the level of threads and locks [32], without modern programming constructs, such models are developer unfriendly unreasonably difficult to use on a wider scale. Parallel programming models offer additional abstraction over lower layers of parallelism with varying success.

Heterogeneous parallel programming which emerged with the popularization of GPGPUs and programming frameworks such as CUDA [13] backed up the increase in performances on various systems. Such frameworks, though serving their purpose, usually have steep learning curves. Minding Huang's law which states that the performance of GPUs will more than double in performance every two years [33], there exists a clear purpose for investigating into programming models which would target GPGPUs. On the other hand, heterogeneity can imply the usage of customized accelerators as well, for which programming models and frameworks for easy and efficient development are yet to be explored [34].

2.2 Parallelism

The ground for any classification of computer architectures, which implies parallel architectures as well, starts with Flynn's classification [35]. Flynn proposed four main ways in which computer approaches instruction and memory streams:

- **SISD** - Single Instruction stream, Single Data stream
- **SIMD** - Single Instruction stream, Multiple Data streams
- **MISD** - Multiple instruction streams, Single Data stream
- **MIMD** - Multiple instruction streams, Multiple Data streams

Various other sources extended the original Flynn's classification with models such as **SIMT** (Single instruction stream, Multiple Threads), an execution model in which **SIMD** model is combined with multithreading techniques, and with **SPMD** (Single Program, Multiple Data streams) [36].

On the other hand, Hennessy and Patterson in their notable book Computer Architecture: a quantitative approach classify parallelism with respect to applications on two different levels [37]:

- **Data-Level parallelism** copes with parallelization problems by focusing on ways of distributing data across different computation nodes or devices which then operate on that data in parallel. The scale on which it can be applied varies greatly, from multicores to

heterogeneous clusters. Data containers for data that is operated on can be in form of specialized data structures or any regular data structures like arrays or matrices that are being divided and distributed

- **Task-Level parallelism** focuses on separating and scheduling different tasks for execution on different processing units. Those tasks can operate on shared or distributed data.

When it comes to means of exploiting those two types of parallelism, there are four major modes, namely Instruction level, Vector architectures and GPUs, Thread level, and Request level [37].

2.2.1 Levels of parallelism

Instruction-Level parallelism

Instruction-Level parallelism (ILP) in its most common case implies overlapping of instructions during their execution on a processing device to achieve an increase in performance. ILP can be exploited either dynamically on the hardware side, or statically on the software side. Dynamic exploitation relies on the capability of processing units to detect a possibility of instructions being executed in parallel, while static exploitation relies on compilers to detect parallelizable instructions at compile time.

Pipelining

The most common technique used to exploit ILP is pipelining. Given a processing unit with n-stage pipeline where every stage represents one phase in the execution of an instruction, one could theoretically execute different phases of n instructions in parallel, since they could partially overlap. A school example of a pipeline can be seen in figure 2.1, with different stages being coloured differently. Keeping a pipeline full requires finding sequences of unrelated instructions either during runtime or compile time.

Loop Unrolling

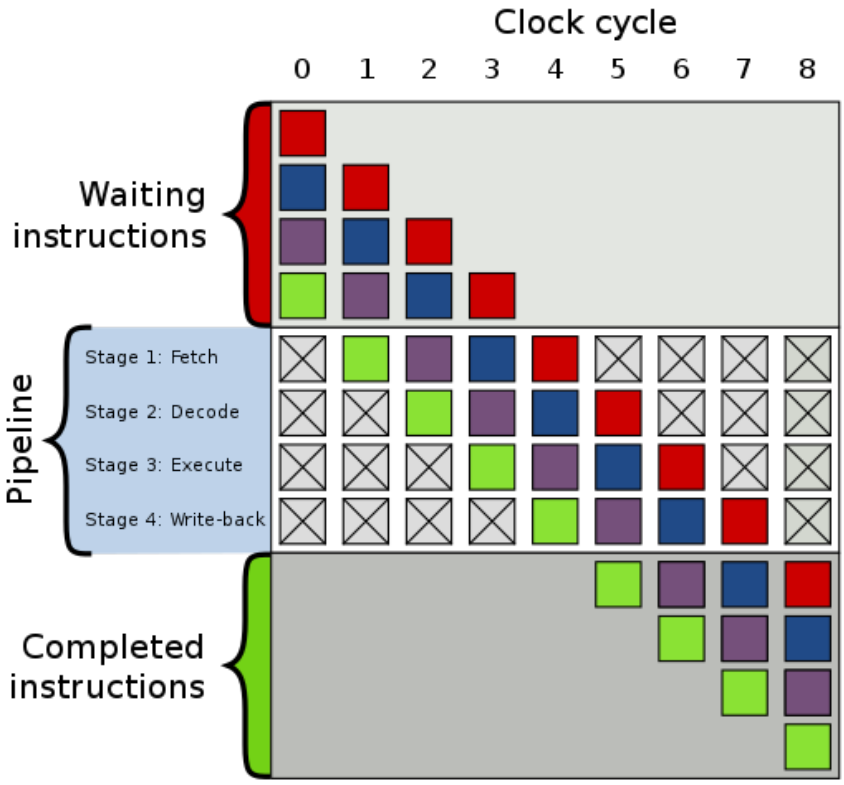
Loop Unrolling, used later in some parts of this thesis as a compiler quasi optimization, as a technique to exploit ILP is a loop transformation technique which exploits space-time tradeoff eliminating instructions that control the loop, such as pointer arithmetic, tests at the loop ending on each iteration [38] together with the respective jumps. This effectively reduces branch penalties and delays in memory data fetches. Unrolling itself is performed in such a way that the loop is rewritten as a repeated sequence of independent statements [39].

Branch Prediction

The aforementioned branching necessarily leads to branch penalties which can, in turn, be reduced by using Branch Prediction techniques. Branch prediction aims to predict which branch

[1] CC BY-SA 3.0 Cburnett, Licence at: `https://creativecommons.org/licenses/by-sa/3.0/deed.en`, File available at: `https://commons.wikimedia.org/wiki/File:Pipeline,_4_stage.svg`

Figure 2.1: 4 stage pipeline[1]



the processing unit has to take after the execution of a conditional jump. Static branch prediction uses information collected from earlier runs, key observation being that individual branch is often highly biased towards being taken or untaken [37]. Dynamic branch prediction uses information collected on the fly. Such techniques include branch prediction buffers, such as two-level adaptive predictor [40] as an example, or branch history tables.

Dynamic Scheduling

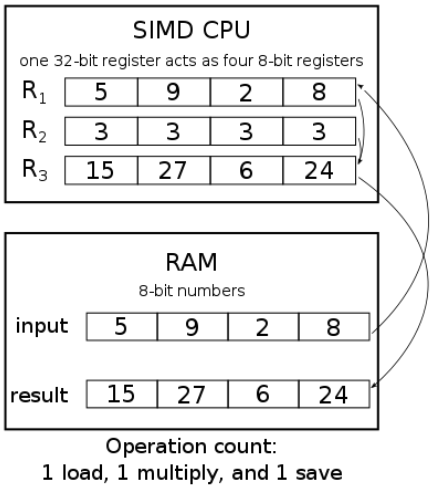
A major limitation of pipelining is a potential data dependency between instructions in various phases of execution. If currently executing instruction requires the result of an instruction that is also in pipeline, data hazard will happen and execution will be stalled. By utilizing the dynamic scheduling technique, the processing unit rearranges the order of instruction execution while maintaining data flow and consistency [37].

Vector architectures

Vector processing units and vector architectures are prototypes of SIMD architecture, as they are based on the idea of applying a single instruction to multiple data in parallel, thus exploiting data-level parallelism. One of the data structures that the vector processors can naturally be applied to are one-dimensional arrays (vectors), which does not exclude an option of collecting a set of in-memory scattered data, organizing that data in a relatively large vector registers,

and processing it from there. Mass production of vectorized architectures can be found in GPUs which lately emerged as purely computational devices through interfaces (e.g. CUDA [13]) that enabled easier development of programs treating them partially as general-purpose processing units. CUDA uses a modified version of the C programming language to enable the programmer to utilize the full potential of GPUs and GPU-like architectures with many parallel floating-point units.

Figure 2.2: SIMD CPU[2]



SIMD extensions for multimedia, such as SSEx or AVX [41], can be considered as other types of vector architectures as they exploit the fact that multimedia data is usually narrower than internal processor buses, registers, and computational units, which enables them to utilize the aforementioned components more efficiently by filling them with more data that is being processed. This concept is depicted on figure 2.2

Thread-Level parallelism

Thread-Level parallelism is being exploited from higher levels of abstractions relative to ILP and Vector architectures. Threads, being the smallest sequence of instructions that can be managed independently [42], imply the existence of multiple program counters and hence are being exploited primarily through MIMD computers [37]. Thread-Level parallelism is being exploited on computers of various scales, from embedded computers to servers. By running many threads at once, one can decouple various parts of the program of which some could be prone to high latencies, and thus better exploit assigned processing time. On the server scale, exploitation of Thread-Level parallelism leads to an increase in performance and availability when it comes to high amounts of Input/Output operations and memory latencies.

[2]CC BY-SA 3.0 Decora, Licence at: `https://creativecommons.org/licenses/by-sa/3.0/deed.en`, File available at `https://commons.wikimedia.org/wiki/File:SIMD_cpu_diagram1.svg`

Request-Level parallelism

Being implemented mostly on larger scales, Request-level parallelism exploits parallelism among largely decoupled tasks specified by the programmer or the operating system [37]. Such tasks are e. g., user requests to a web service hosted on a server, or multiple queries to a database on a server that nowadays handle millions of requests on the scale of hours. In general, those requests are largely independent and mostly involve read-only data operations. Computations among those requests are also easily partitioned within a request and across different requests.

2.2.2 Programming models

Programming model is an abstraction of architecture used to express programs for the abstracted platform which exposes interfaces of some subset of its functionalities. Parallel programming model abstract parallel architecture and can be approached with respect to process interaction and with respect to problem decomposition [43]. Problem decomposition approach is analogous to aforementioned classification of parallelism with respect to applications, namely Data-Level and Task-Level parallelism) with the addition of implicit parallelism which resembles everything that the compiler, the runtime, or the hardware are responsible for. On the other hand, the process interaction approach takes into consideration ways of unavoidable communication between parallel processes. These are:

- **Message passing** - processes exchange data through messages. Communication can be either synchronous or asynchronous
- **Shared memory** - processes communicate through part of memory which can be accessed by all participants. To avoid race conditions, one has to use means of explicit synchronization
- **Implicit communication** - Covers every aspect of communication hidden from the programmer

With the addition of the basic threading model, the aforementioned classification partially fits the classification of pure parallel programming proposed by [44]. As it is widely known, as implementations of those models, Pthreads, OpenMP, and MPI are proposed. For the sake of completeness, a brief review follows.

Pthreads

POSIX threads [45] (Pthreads) is a standardized programming interface, a parallel programming model for exploiting threading mechanisms implemented by various processor architectures, and de-facto reference implementation of the **threading model**. Although Pthreads originated as a UNIX standard, it eventually settled down as an overall and official standard. Pthreads have access to their private and shared process memory, and a rich set of API calls with the purpose

of thread management, synchronization, operation on mutexes, and condition variables. Unlike processes, threads are more lightweight when it comes to creation and memory requirements. On the other hand, they require manual synchronization through mutexes and can't be easily ported to non-POSIX compliant systems. The threading model could be generalized to a parallelized model which requires manual synchronization, and in that context partially resembles one of the multiprocessing APIs available for the GAP8 platform, which will be covered later in the text.

OpenMP

Open Multi-Processing [46], is an API and compiler extension, a prototype of **shared memory** model, that supports multiprocessing and parallelization on multiple platforms through the utilization of shared memory concept. The standard is managed by the "OpenMP Architecture Review Board" consortium. It is implemented as an extension in C, C++, and Fortran programming languages and thus requires a compiler for the respective languages with OpenMP support. Generated code is generally portable, with support for GPUs and accelerators as of version 4.0., with the currently active version being 5.2. Since it relies on the shared memory model, the communication model is simple without the need for messaging incorporation. Throughout this thesis, OpenMP was widely used as a programming model targeted by the compiler extended for the purposes of this thesis.

MPI

Message passing interface [47] is an prototypical interface to the **shared memory** model. The standard is managed by the "Message Passing Interface Forum", and as the name suggests, implementations of the interface rely purely on a message exchange to enable programming of parallel computers. The absence of shared memory (with limited distributed shared memory in MPI-2) makes it global memory independent and more scalable than OpenMP. To date, one of MPI implementations, namely Open MPI [48] is used by many supercomputers from the Top500 list [49].

2.3 Heterogeneous systems

2.3.1 Heterogeneity as a concept

One of the core aspects this thesis tackles is heterogeneity, and it is thus described in this subsection. Heterogeneous as a word, stemming both from latin and greek, by definition means *consisting of dissimilar or diverse ingredients or constituents*[3]. When considered in the context of computing systems, it implies a computing system or processing node *consisting of dissimilar*

or diverse compoents. Diverse or dissimilar components usually imply processing units of different purposes. When it comes to the concrete realization of heterogeneous systems, in practice the most common combinations are that consisting of one general-purpose processor paired with a general-purpose graphics processing unit or a customized accelerator for a specific application. Systems containing all three types of processing units are not excluded. Other types of accelerators, although not prevalent in academic projects or industry, are also possible. in the following subsection, there will be provided a brief overview of each type of accelerator that is objectively often used in practice.

2.3.2 Accelerators

Accelerators are either integrated or disjunct parts of the system which are programmed at the hardware level for one specific operation. Such accelerators, when observed independently, excel in processing the operation they are specified for, and additionally achieve higher scores in energy efficiency measured in the number of individual calculations per second per watt of power. However, the price of high performance and technically best energy efficiency is double-natured. Highly-specified hardware cannot perform any computation but the one they are designed for. Furthermore, designing a customized piece of hardware is an expensive process regarding engineer-hours, as it requires specific skills at the level of hardware and digital design.

Customizable accelerators posed a promising solution to compromise between gain in performance and energy efficiency, and reusability in multiple application domains. Such accelerators ideally would expose parameters that could allow customization of the underlying hardware, such as integer operation precision, floating-point operation precision, size of the vectors being processed, etc. However, customizability to such an extent has not yet, up to date, been achieved. From the other point of view, another type of customizability emerged, through accelerators containing multiple simple processing cores, usually of RISC type, and supporting only relatively simple operations, like integer operations including hardware multiplication, but excluding floating point operations. That type of customizability enabled achieving increases in performance and energy efficiency but due to the nature of general-programmability of the aforementioned simple accelerator cores, retained a wide spectrum of applications supported. Examples of such accelerators are Epiphany which features arrays of simple RISC cores [50], and accelerators used by the PULP platform, described as a part of the following section (3.3)

The current state-of-the-art in the field of hardware acceleration mostly leans towards accelerators for deep learning, inference in deep or convolution neural networks, machine learning, and brain-inspired computing in general. That FPGA-Based neural network accelerators are

[3] `https://www.merriam-webster.com/dictionary/heterogeneous`

a prominent area of study is witnessed by a survey available in [51]. There are other notable examples in this area. DLAU is a deep learning accelerator unit that employs three pipelined processing units to improve throughput and utilize tile techniques to explore locality for deep learning applications [52]. Regarding convolution neural networks, authors in [53] propose a scalable high performance depthwise separable convolution optimized convolution neural networks accelerator. A solution proposed by authors in [54] demonstrates that FPGA acceleration can be a superior solution compared to GPUs in terms of throughput and energy efficiency when a convolution neural network is trained with binary constraints on weights and activations. Regarding deep convolution neural networks, work in [55] proposed a scalable parallel framework that exploits four levels of parallelism in hardware acceleration. Systematic design space exploratio nmethodology is then exploited to maximaze accelerator throughput under FPGA constraints. Similar to the latter work, authors in [56] propose an analytical design scheme using roofline model, by quantitatively analyzing solution's computing throughput and required memory bandwidth using various optimization techniques, such as loop tiling and transformation. Regarding the research which includes the PULP platform, there is an example of an integration of binarized neural network accelerator into PULPissimo SoC [57]. More detailed description of the PULPissimo platform is available further in text (subsection 3.3.2).

Accelerators and reconfigurable hardware in cloud

In the field of domain-specific acceleration, accelerators and reconfigurable hardware in the cloud gained some traction. A most prominent example is the availability of reconfigurable hardware exposed as cloud service at Amazon web services [16]. The same cloud is used as a driver for FireSim, a simulation platform that enables cycle-exact microarhitectural simulation of large scale-out clusters by combining FPGA-accelerated simulation of silicon-proven RTL designs with a scalable, distributed network simulation [58]. Another prominent example of the programmable hardware in the cloud is the Catapult project [18], with a notable example of accelerating deep convolution neural networks using servers augmented with FPGAs [59].

As reconfigurable hardware's resources are limited, [60] proposes a flow to provision FPGAs from a pool of cloud resources by enabling FPGA development and simulation in virtual machines. Programmability of FPGA fabric available in the cloud poses another issue, as reconfigurable hardware is inherently hard to utilize, regardless of its location. To increase the exploitation of such resources, various frameworks exist, with many of them listed in a survey available in [61]. The survey also provides a list of hardware accelerators that have been implemented for many cloud computing-based applications and a qualitative comparison of the proposed schemes. Regarding physicality of data centers that implement cloud infrastructure, [62] proposed a platform that decouples the FPGA from the processing units of servers by connecting FPGA directly to the datacenter's network. That way, an FPGA can be turned into a

disjunct computing resource that can be deployed at a large scale into emerging hyperscale data centers.

2.3.3 GPGPUs

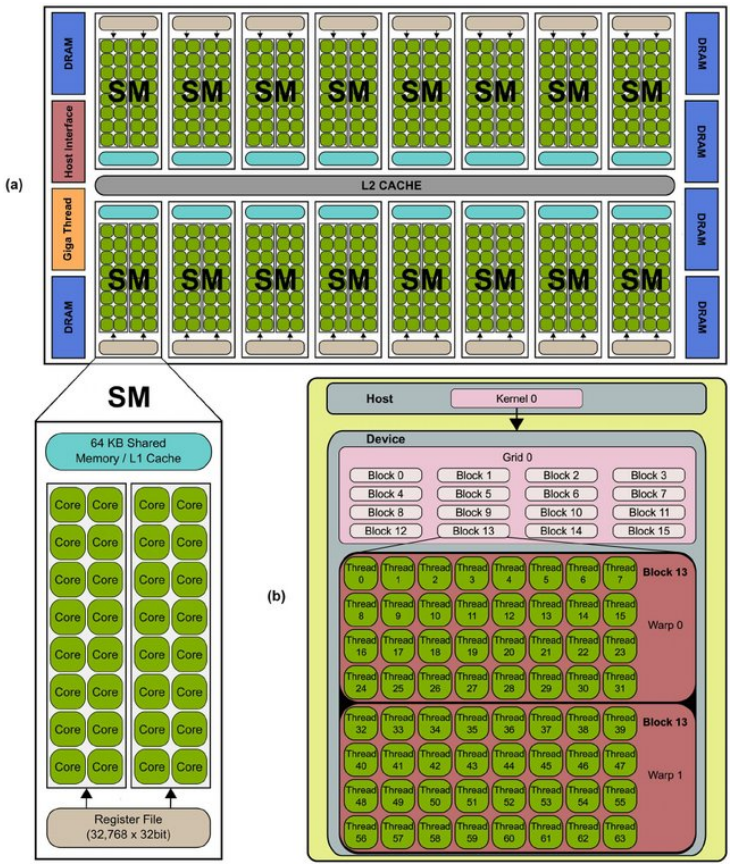
General-purpose graphics processing units are technically the first accelerators that emerged in the era of specialized hardware. Initially used as accelerators for graphics operations, by the exposition of the interfaces for their general-purpose programming, they again emerged as highly performant processing units for data-parallel and vector operations. Architecture of a generalized GPU is available on figure 2.3. As can be seen in the upper part of the figure, GPU features a series of Streaming Multiprocessors (SM), mutual L2 cache, memory (DRAM), and adequate interfaces. Each streaming multiprocessor further contains multiple processing cores with shared memory and L1 cache. What is visible to the programmer is a device with a series of threads divided into blocks and warps. Such architectures are usually encapsulated by the programming models, which are currently CUDA [13] for NVIDIA-based devices and OpenCL [14] for AMD-based devices, at least for two of the most represented manufacturers of the GPUs. Bindings exist for widely popular Python programming language for CUDA-based GPUs. What can be intuitivelly concluded from the depicted architecture is that GPU architectures provide high throughput when applying operations on the same data. The cost of the high throughput regarding computation is paid with costly transfers of the data to and from GPU.

GPUs are nowadays widely used for various artificial intelligence paradigms. Models of deep neural networks are trained on GPUs before running inference on other devices [64]. Another popular application of GPUs is cryptocurrency mining, which is one of the reasons for shortages and price increases on the market, at least during the making of this thesis (early 2022.). Since their inception as general-purpose accelerators, they have been widely applied in cryptography. Paper [65] presents an efficient impelmentation of the Advanced Encryption Standard (AES), while [66] presents a GPU implementatino of a 1024-bit RSA decrypt primitive. Other usages of GPUs include, but are not limited to, pysical simulations, e. g. paper [67] describes an efficient CUDA implementation of the Barnes-Hut n-body interaction simulator.

There are other types of devices built around GPUs with different purposes, such as NVIDIA's Jetson [68] which is advertised as a platform for autonomus machines and other embedded applications. Jetsons feature CPU, GPU, memory, power management, and high-speed interfaces.

[4]CC BY Hernández et al.

Figure 2.3: Architecture of a GPU[4][63]



2.3.4 Other

Outside of the so-far conventional heterogeneity which implied a general-purpose processing unit coupled with an accelerator, there are other examples of the utilization of the concept. Authors in [69] utilized two different processing units of the same type, i. e. general-purpose processing units. One of the processing units is relatively weak regarding processing capabilities but is extremely energy-saving, while the other is more performant, but more power hungry as well. Combining two of them, with mutual sleep and wake strategies, enabled optimization of energy consumption while retaining processing capabilities. Another example of heterogeneity driving energy efficiency is available in [70], which presents a fast heterogeneous and distributed cluster for efficient calculation of bcrypt password hashes. The cluster contains computational nodes featuring a general-purpose processing unit coupled with FPGA fabric implementing a custom accelerator for the most costly parts of the bcrypt loop calculation. The improved version of the cluster is presented in [71] containing an improved version of the bcrypt accelerator, heterogeneity on a higher level where nodes have different processing capabilities, and an implementation of the password candidate distribution scheme based on the passwords' probability distribution.

Chapter 3

General Concepts

This chapter provides some fundamental concepts used throughout this thesis, such as domain-specific languages, together with a brief review of state of the art in each respective field. Furthermore, this chapter provides an introduction to RISC-V instruction set architecture, the PULP project, and finally to the GAP8 platform, the platform used as hardware support for delivering outcomes of this thesis.

3.1 Domain-specific languages

3.1.1 General

The programming and application execution presents a special issue in the domain of heterogeneous computing. Multiple types of processing units, as well as potentially diametrically opposite applications which can be executed under the system present a significant challenge for the programmers. One of the possible means of coping with the aforementioned challenges is the development of the programming models which abstract away the implementation details of the system from the programmer. The solution comes up often in a form of the Domain-Specific language which is defined as a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [72].

The main differences between general-purpose programming languages and Domain-Specific languages are embodied in the fact that DSLs target a specific problem area and that DSLs contain syntax and semantics that model concepts at the same level of abstraction as the problem domain does [73]. Domain-specific languages have always been around as means of solving complex, domain-bound problems. Deursen, Lint and Visser in 2000. published a survey on DSLs, citing 75 key publications in the area [72].

When it comes to development of DSLs, they are mostly clasiffied as *internal* and *external*

[73, 74]. *Internal* DSLs are embedded inside of the *host* programming-language and reuse its syntax, toolchains, and compilers. *External* DSLs are developed independently, and have their own infrastructure in terms of parsing, interpretation, compilation and code generation [73].

One of the general-purpose programming languages that have native support for embedding DSLs is Scala. Since the framework used to deliver contributions of this thesis utilizes Scala, DSLs developed with Scala, are of special interest. In this overview [74], authors present five approaches for implementing Domain-Specific languages in Scala as *external* which can use either a *parser generator* or a *parser library*, and *internal*, which can use *annotations of existing language*, *deep embedding*, or *shallow embedding*. Besides providing examples for every case in sense of an example of each type of DSL development even emulating *external DSLs*, authors grade each approach with regards to traits of the programmer who will use language, which can be seen in table 3.1.

Table 3.1: Pros and cons of DSLs with Scala[74]

	External		Internal		
	tool	library	annotations	deep	shallow
Ease of development	-1	0	2	1	2
Flexibility of syntax	2	2	-1	0	-1
Quality of syntax error messages	1	1	2	-1	0
Ease of use	2	2	1	0	-1

Those traits stemming from the features of the particular approash, namely *Ease of development*, *Flexibility of syntax*, *Quality of syntax error messages*, and *Ease of use*, are graded with scores between -1 and 2. It can freely be concluded that each approach has its own pros and cons, and that selection of a particular approach heavily depends on requirements regarding the DSL being developed.

3.1.2 Overview of the field

There are other notable examples in the world of DSLs. **Halide**, for example, is a DSL for expression of the high-performance image processing pipelines [75]. The language itself is embedded in C++ and targets many modern CPU architectures and GPU compute APIs. Although Halide was designed and implemented as a DSL primarily for image processing, it eventually evolved into supporting heterogeneus computing. Work available in [76] demonstrates the extensions to the language which enable users to specify portions of their applications that should be hardware accelerated. The provided compiler then automatically generates the accelerator,

together with the glue logic necessary for the programmer to utilize it. Halide is of special interest because it is de facto standard for expressing image processing pipelines, and yet one of the benchmarks later in the text (section 6.2.2) is exactly a part of the image processing pipeline, meaning that Halide and outcomes of this thesis could be compared as a part of future work. Furthermore, the RISE team recently published a paper comparing RISE-generated code with Halide on mobile CPUs [77].

In light of the ever-lasting run for better performances, it was previously mentioned that one of the key aspects for achieving them is heterogeneity. Since the programming of heterogeneous systems is inherently hard, DSLs and custom infrastructures that often come packed with them could present a solution. However, developing a DSL, be it internal or external is again a process that cannot be easily perceived as mainstream or easy. **Delite** is a framework that eases the creation of DSLs by providing common components like parallel patterns, optimizations, and code generators that can be reused in DSL implementations [78, 79, 80]. The framework includes lifting of embedded DSLs into an internal representation, optimizations, and code generators that compile DSLs to C++, CUDA, or OpenCL. Programs are automatically parallelized, and different parts of the program can be run simultaneously on CPU and GPUs.

Figure 3.1: High-level overview of Delite framework[1] [79]

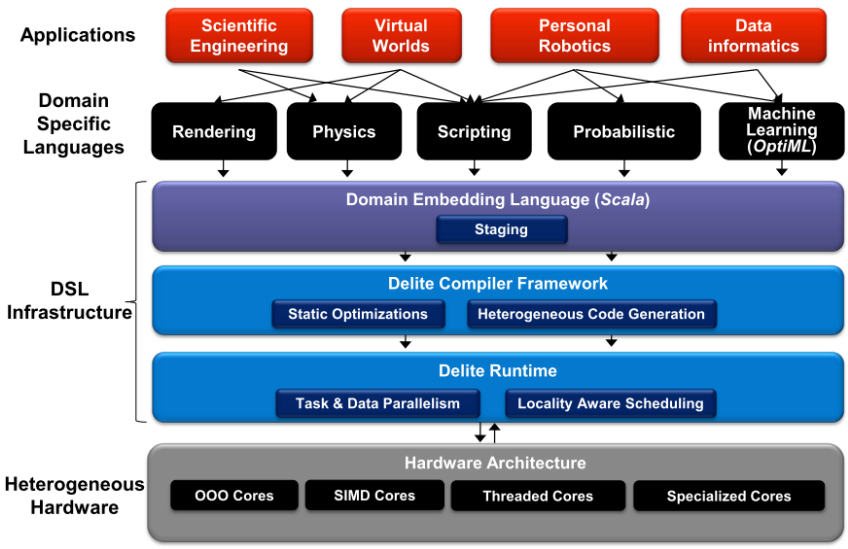


Figure 3.1 depicts a high-level overview of Delite framework. As can be seen in the figure, DSL infrastructure consists of a domain embedding language embedded in Scala, compiler framework which incorporates static optimizations and heterogeneous code generation, and runtime part incorporating task and data parallelism together with locality aware scheduling. Delite development team so far proposed four DSLs built on top of the framework, and those are:

- **OptiML** [80, 81] - a DSL for machine learning

[1] ©2011 IEEE

- **OptiQL** [80] - a DSL for data querying
- **OptiGraph** [80] - a DSL for graph analysis
- **OptiMesh** [80] - a DSL for scientific computing

Although Delite can be perceived as an outdated project, it is important to mention it, as it is one of the first modern and complete frameworks for DSL development. Furthermore, the RISE framework includes similar concepts, such as compilation of a DSL to relatively lower programming language, use of internal representation, employing transformations and optimization, down to the sole fact that it is embedded in Scala.

A team of researchers that worked on Delite, later proposed a novel intermediate language, named **Distributed Multiloop Language** (DMLL) [82]. DMLL utilizes data-parallel patterns, a concept that RISE utilizes as well, that capture necessary semantic knowledge that restructures computations for heterogeneous devices. DMLL is wrapped with a compiler and runtime environment that can execute parallel applications across a heterogeneous cluster with non-uniform memory and accelerators.

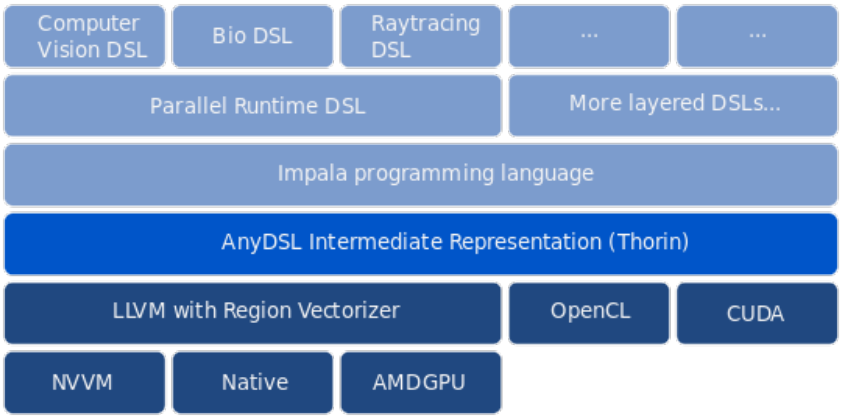
RISE is a data-parallel and pattern-based language embedded in Scala for high-level computation expression. The main idea behind the project is to express *what* is being computed, instead of *how* is something computed. Expressing computation using patterns enables transformations using rewrite rules that encode implementation and optimization choices[2]. As authors put it, RISE is a spiritual successor to **LIFT** [83] project. Unlike in LIFT which explores a large space of possibilities in optimizing the provided high-level program, the RISE-centered framework features another language named **ELEVATE** [84, 85] for expressing optimization strategies. Since the contributions of this thesis completely rely on the RISE framework, further description of these two languages and their compiler named Shine is provided in chapter 4.

When considering novelty, newer solutions, and wider acceptance, one has to mention **AnyDSL**. AnyDSL is a domain-specific libraries stack and compiler framework whose main forte is achieving high-performance code using partial evaluation. It extends the continuation-passing style intermediate representation named Thorin with a simple online partial evaluator [86]. DSLs supported by the framework are implemented in a front-end language named Impala, while the optimizations are performed on the aforementioned framework's intermediate representation. AnyDSL somewhat resembles the previously describe Delite framework, which can be seen in the figure depicting the architecture of the framework (figure 3.2).

At the end of the compilation chain, the framework supports code generation with parallelization and vectorization for contemporary CPU and GPU architectures. Examples of DSLs implemented using AnyDSL framework are Stincilla[4], a DSL for stencil codes, and Ra-

[2] `https://rise-lang.org/`
[3] ©AnyDSL project under Apache 2.0, available at: `https://anydsl.github.io/`
[4] `https://github.com/anydsl/stincilla`

Figure 3.2: Architecture of the AnyDSL framework[3]



Trace[5] which is a DSL for ray traversal.

One of the most prominent and relatively recent examples when it comes to interleaving concepts of domain-specific languages and hardware accelerators is **Spatial**. Spatial is a domain-specific language with compiler infrastructure for the higher-level description of applications accelerators. Spatial provides hardware-centric abstractions which increase programmability and performance by providing compiler passes that support these abstractions, namely pipeline scheduling, automatic memory banking, and automated design tuning driven by active machine learning [87]. Authors claim that Spatial can target a wide range of architectures, achieving significant speedups with less code.

Regarding additional DSL examples that utilize code generation or various transformations, there are more notable examples that are either historically meaningful or interesting in the context of this thesis. **Lime** [88] is a programming language executed on the Java virtual machine targeting heterogeneous systems. Language features an optimizing compiler that generates high-quality GPU code by generating OpenCL [14] programming constructs. The authors of the paper explicitly stated that the main motivation was rising the level of abstraction regarding general-purpose GPU programming which to some extent matches this thesis confirming that the easiness and learning curve of a programming model can be cumbersome for engineers not familiar with low-level architectural details. Unlike the rest of the DSLs that are rather implicitly executed on Java virtual machine, mostly due to the fact that some of them are embedded in Scala, Lime is executed on the JVM but as an independent language.

GraphIt is a high-performance DSL for graph computations that generates fast implementations for algorithms with different performance characteristics running on graphs with different sizes and structures [89]. GraphIt separates the algorithm from execution by providing a separate scheduling language. It also features an autotuner that automatically finds high-performance schedules. A similar separation of concerns is present in **ANTAREX** project

[5] `https://github.com/anydsl/traversal`

which enables expressing functionalities in C++ but makes it possible to express the adaptivity, energy, and performance strategies at compile-time [90], together with runtime autotuning and resource and power management. Strategies are written in an aspect-oriented programming language **LARA**, which allows the specification of compilation strategies to enable efficient generation of software code and hardware cores for alternative target architectures [91].

Although not strictly tied to DSLs, regarding runtime environments mentioned so far in the context of Delite and ANTAREX, there are some additional notable projects which demonstrate the existence of the concept for a relatively long period of time. **Harmony** is a runtime supported programming and execution model that provides semantics for simplifying parallelism management, and dynamic scheduling of compute intensive kernels to heterogeneous resources [92]. **Dandelion** is another system designed to address the problem of programming heterogeneous systems. It provides a unified programming model for underlying processing elements of heterogeneous systems, integrating data-parallel operators into general purpose programming languages. Additionally, it is packed with a scheduler that distributes data-parallel portions to available resources [93]. Execution on GPUs is supported by compiling the code written for it to CUDA kernels.

Hardware-Description Languages (HDL) like (System)Verilog or VHDL belong to a completely different cluster of programming languages. They provide constructs that engineers use to describe hardware that will eventually be implemented in appropriate circuitry. **Chisel** is a Hardware-Description Language (HDL) implemented as a DSL in Scala [15], aims to ease the process of digital design. Chisel can generate Verilog aimed for synthesis on FPGAs or ASICs, or high-speed C++-based software simulator. By compiling to another programming language, Chisel falls in the category of code generators. Being implemented as a DSL in scala, it could ease the generation of hardware-implemented custom accelerators if well-integrated in one of the aforementioned stacks.

At the level of libraries, a notable example is **TACO**, being implemented as a library in C++. TACO, which stands for The Tensor Algebra Compiler, introduces a compiler technique that automatically generates kernels for any compound tensor algebra operation on dense and sparse tensors [94]. **HPX** as a parallel runtime system also extends C++ to facilitate distributed operations, enable fine-grained constraint-based parallelism, and support runtime adaptive resource management [95]. **Spiral** is a library generator for linear transformations. Authors in [96] implemented it as a DSL in Scala by using Lightweight modular staging (LMS), a generative programming approach that lowers the effort needed to write a high-quality program generator [97]. Based on LMS with additional features, an experimental framework for creating staged DSLs named Argon was developed [98].

In this section, a plethora of DSLs was displayed, of which many rely on compilation chains with optimizations at the level of intermediate representation which eventually end with gen-

erating code for a widely accepted general-purpose programming language used by a target platform, be that target platform a CPU, GPU, accelerator of any kind, or any combination of the previously listed targets. This leads to the conclusion that code generation for native platforms indeed is a plausible way of raising the level of abstraction for any kind of system.

3.2 RISC-V

3.2.1 General

Instruction-Set Architecture (ISA) is an abstract definition of a computer at the level of, as the name suggests, an assembly instruction set. Fundamentally, ISA presents an interface, a contract between hardware and software, abstracting hardware implementation details for software developers, and allowing hardware desingers to design new hardware which will be supported by the preexisting software. In that context, **RISC-V** is an open, royalty-free ISA initially developed at the University of California at Berkeley.

In their case for RISC-V, main collaborators (or creators) of the standard stated that processors are just a small fraction of the design which justifiably raises the question of why the most important interface is usually proprietary. They further argue that there is no good technical reason not to have free and open instruction sets and that among the existing RISC free open instruction sets, RISC-V is the best and safest choice [99]. Started as an academic project as RISC-V Foundation with 29 members, RISC-V is now a non-profit based in Switzerland, governed by a board of directors elected from all classes of members, and with more than 2,000 members from more than 70 countries (as of January 2022) [100].

The processor itself was meant to be kept simple, yet anticipated everything, from embedded devices to vector processors and high-performance computing. From the architectural point of view, RISC-V is of type *Load-Store* with little-endian byte ordering and register-register operations. The standard is specified through Instruction Set Manuals, namely Volume I specifying Unpriviledged (user-mode) ISA [101], and Volume II specifying Priviledged (supervisor mode) ISA [102].

3.2.2 Extensions

RIS-V ISA is modular, and typically consists of a minimal base instruction set and extensions appended to base set with compliance to the needs of the computer that is being designed. Every part of the ISA exists in one of three statuses: *draft*, *frozen*, *ratified*, and is versioned appropriately. Only *ratified* set are fully operational and safe to use. On the other hand, *frozen* sets are expected to undergo minor changes before ratification, while *draft* sets could undergo further major changes. There exists five base instruction sets, all of them being displayed in

upper part of table 3.2. Base instruction set can support either fully-sized integer sets (**I**) or embedded sets (**E**) with address sizes of 32, 64, and 128 bits. Base instruction set specifies number of registers. For **I** set, architecture features 32 registers, while **E** set features 16 registers.

On the other hand, the lower part of table 3.2 displays extensions to the base instruction sets. Those extensions imply the existence of the additions to the architecture necessary to execute them. **M** extension adds hardware support for integer multiplication and division, while **F**, **D**, and **Q** extensions add hardware support for singe, double, and quad precision floating-point operations respectively. Out of the rest of the available extensions, it is especially worth to mention **C**ompressed instructions with shrunk instruction operational code, which are somewhat similar to ARM Thumb instruction set, particularly useful in embedded systems. Furthermore **C** extension is used in RISC-V cores featured by GAP8 platform (section 3.4) used in this thesis. Extensions can be used with any base and without conflict regardless of the combination used.

Z prefixed extensions were a pragmatic solution to the anticipated growth of extension sets. Those extensions are added to the end of the architecture abbreviation, sorted alphabetically, and most important of all, prefixed with the small letter of the closest set of preexisting extensions. For example **Zam** which is extension for misaligned atomics, is closest to **A** extension for atomic instructions, while **Ztso** for total store ordering is closest to **T** extensions for transactional memory.

Versions of utilized extensions in instruction sets are stated explicitly, by adding numbers that represent major and minor version parts of the version. The first number after the letter abbreviating the extension indicates the major version. The minor version is stated right after the major version followed by a small letter **p**. If ommited, assumed major version is 1, while ommited minor version assumes 0. For example, **M2p0** indicates usage of **M** extension of version 2.0

As an example of an actively-used instruction set, the GAP8 platform uses RISC-V cores with **RV32IMC**, which indicates an instruction set consisting of 32-bit integer core with hardware multiplication and division and with the compressed instruction set.

There is additional extension **G** not mentioned in table 3.2. **G** is abbreviation given to a instruction set consisting of: integer base, **RV32I** or **RV64I**, with six standard extensions, namely **M**, **A**, **F**, **D**, **Zicsr**, **Zifencei**, which ultimately sums to **IMAFDZicsr_Zifencei** instruction set. That set resembles a **G**eneral-purpose scalar instruction set [101]. **RV32G** and **RV64G** are currently default target of compiler chains. RISC-V is currently supported by major compiler stacks, namely GCC and LLVM.

Custom or non-standard extension addition is enabled through **X** extension prefix. For example **Xpulp** is a non-standard extension set which incorporates increasing computational density and minimizing pressure toward the shared memory hierarchy [103] (according to [104]).

Table 3.2: RISC-V extension sets as of December 2021 (unpriviledged ISA) [101]

Base	Description	Version	Status
RVWMO	Memory Consistency Model	2.0	Ratified
RV32I	Base 32-bit Integer Instruction Set	2.1	Ratified
RV64I	Base 64-bit Integer Instruction Set	2.1	Ratified
RV32E	Base 32-bit Integer Instruction Set, Embedded	1.9	Draft
RV128I	Base 128-bit Integer Instruction Set	1.7	Draft
Extension	**Description**	**Version**	**Status**
M	Integer Multiplication and Division	2.0	Ratified
A	Atomic Instructions	2.1	Ratified
F	Single-Precision Floating-Point	2.2	Ratified
D	Double-Precision Floating-Point	2.2	Ratified
Q	Quad-Precision Floating-Point	2.2	Ratified
C	Compressed Instructions	2.0	Ratified
Counters	Performance counters and timers	2.0	Draft
L	Decimal Floating-Point	0.0	Draft
B	Bit Manipulation	0.0	Draft
J	Dynamically Translated Languages	0.0	Draft
T	Transactional Memory	0.0	Draft
P	Packed-SIMD Instructions	0.2	Draft
V	Vector Operations	0.7	Draft
Zicsr	Control and Status Register (CSR) Instructions	2.0	Ratified
Zifencei	Instruction-Ferch Fence	2.0	Ratified
Zam	Misaligned Atomics	0.1	Draft
Ztso	Total Store Ordering	0.1	Frozen

3.2.3 Notable projects

So far, RISC-V ISA attracted a large number of users, both from academia and industry. One of the biggest engagements of the RISC-V ISA is the European Processor Initiative (EPI) which adopted it as a base for the development of a fully European developed processor [10]. First power-efficient and high-throughput accelerator chips were taped out in late 2021. under the name EPAC (European Processor Accelerators) [105]. Another notable industrial-grade example is SiFive, a fabless semiconductor company founded by collaborators working on the RISC-V specification. SiFive develops essential, machine learning, and performance-aiming cores, appropriate software, and hardware support for development and prototyping [106].

A couple of notable projects come straight from the laboratory that initiated the RISC-V standard. Berkeley Out-of-Order Machine is, as authors describe it, a synthesizable, parametrized, superscalar, out-of-order RISC-V core designed to serve as the prototypical baseline processor for muture micro-architectural studies of processors of such type [107]. It aims to be competitive in fields of performance and area for energy-efficient, out-of-order cores. The core is comparable to other similar proprietary designs. More recent updates include BOOM v2 which include an updated 3-stage front-end design with a bigger set-associative Branch Target Buffer, a pipelined register rename stage, split floating point and register files, a dedicated floating point pipeline, a separate issue windows for floating point, integer, and memory micro-operations, and separate stages for issue-select and register read [108].

From the same team comes an open-source SoC design generator that emits synthesizable RTL by leveraging the Chisel hardware construction language to compose a library of sophisticated generators for cores, caches, and interconnects into an integrated SoC [109]. Rocket chip generator can provide both an in-order core generator (Rocket) and an out-of-order core generator (BOOM).

An interesting community project is SERV - The SErial RISC-V CPU [110]. From the same author comes an interesting benchmark for FPGAs and parts of their toolchains, namely synthesis and place and route, called CoreScore which tests how many SERV cores can be put into a particular FPGA [111]. The current leader (as of January 2022.) is Xilinx VCU128 which can fit 6000 SERV cores[6].

RISC-V is also used as ISA for many cores developed by the PULP platform, as well in GAP8. Both platforms will be described in their respective and independent sections (section 3.3 for PULP, section 3.4 for GAP8), as both of them are either directly or indirectly used as hardware-support for delivering the contributions of this thesis.

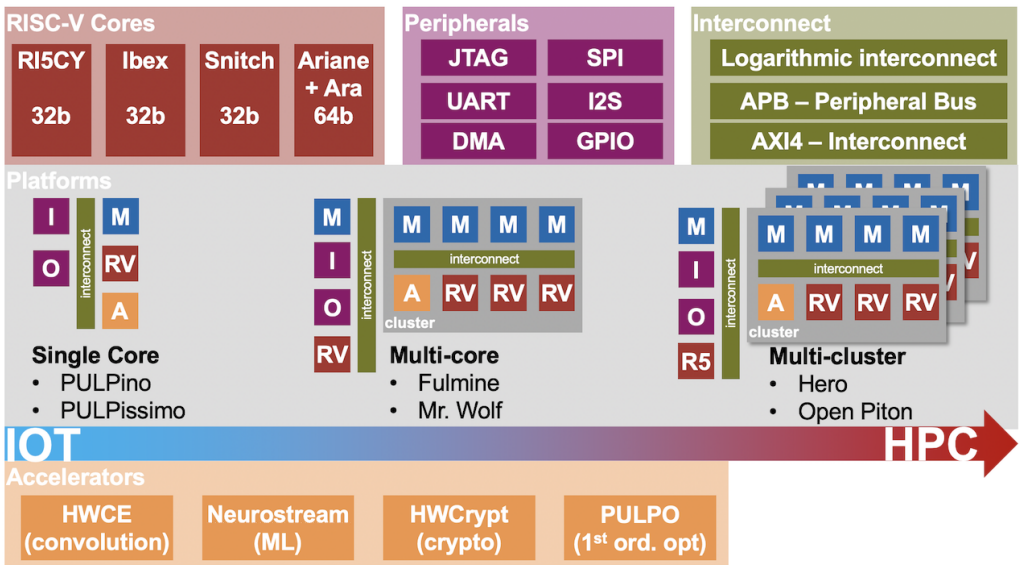
[6] `https://corescore.store/`

3.3 PULP

The **P**arallel **U**ltra **L**ow **P**ower (PULP) Platform is a research project jointly led by ETH Zürich and the University of Bologna which aims to deliver open and free hardware platforms both for research and industry-grade purposes. As the name suggests, the scale on which projects aims to deliver its outcomes is of ultra-low power, breaking the energy efficiency barrier within a power envelope of a few milliwatts to satisfy computational demands of IoT applications requiring flexible processing of data streams generated by various sensors [112], from relatively primitive sensors such to cameras and other complex sensors. To develop its open hardware, the PULP platform heavily utilizes and builds on RISC-V ISA previously described in section 3.2. In the context of this thesis, the PULP platform is of particular importance as heterogeneous systems that are targeted by the programming model being developed are embodied in some of the projects that stem from it. Furthermore, it is anticipated that the growth for the need for such devices being developed by the platform will need a modern programming model with increased expressibility, and utilizing the PULP platform as a target for the model proposed by this thesis makes a perfect use-case for exploring the potential positive outcomes of such approach. Modularity of the design of systems developed under the project enables a top-down approach where system and hardware are developed and tailored for specific applications which allow for heavy optimizations mirrored on increased energy efficiency.

As a high-level representation of the platform suggests available in figure 3.3, projects developed under the umbrella of the PULP platform can generally be separated into five fields: processors, single-core platforms, multi-core platforms, multi-cluster systems, and accelerators. For the sake of clarity, a brief overview of every field is provided further in the text.

Figure 3.3: High-level overview of the PULP platform[7]



[7]©PULP platform, 2022, Permission granted to reuse, available at: `https://pulp-platform.org/`

3.3.1 Processors

RI5CY [103], according to [113] is an in-order RISC-V core with four pipeline stages, implementing an RV32IMC subset of the RISC-V ISA. RI5CY includes extensions which include hardware loops handling up to two finite loops, auto-increment load/store instructions, and bit-manipulation instructions to handle bit fields in registers. It also includes DSP extensions including MAC instructions, basic fixed-point arithmetic instructions, packed-SIMD instructions on vectors of 8-bit elements packed into registers of bigger size, and a dot product unit [113].

Ibex [114], previously known as Zero-riscy, is a core now contributed to lowRISC, a non-profit organization. It is an area-optimized RISC-V core, implementing an RV32IMC subset of the RISC-V ISA. The pipeline is consisted of two stages, namely instruction fetch, and instruction decode and execute. The ALU is minimal, while the core's multiplier unit contains one MAC unit capable of sequentially multiplying two 16-bit operands and accumulating the result in a 32-bit register. The core implements a minimum set of control and status registers defined by the privileged ISA [113].

Snitch is a general-purpose, single-stage, single-issue core tuned for high energy efficiency which, paired with a double-precision floating-point unit, aims at the maximization of the compute and control ratio by enhancing the ISA with two minimally intrusive extensions: stream semantic registers and a floating-point repetition instruction [115]. Ariane [116], now listed as OpenHW's CVA6 CPU is a 64-bit, single, in-order issue and out-of-order execute RISC-V core, implementing RV64GC instruction set. By implementing both the unprivileged and privileged instruction set, Ariane is a fully-fledged processor capable of running Linux OS and targeting a baseline application running environment.

3.3.2 Single core platforms

Regarding single-core platforms, the PULP platform currently provides two of them: PULPino and PULPissimo. PULPino is a single-core SoC built around RI5CY or Zero-riscy cores [117] mentioned in the previous subsection. It is a minimal system, focused on simplicity without caches, memory hierarchy or DMA circuitry [118].

PULPissimo with code name Quentin regarding its implementation, a more powerful variant of a single core platform features a 32-bit in-order RISC-V processor with four stages pipeline. The processor implements an RV32IMFC RISC-V ISA subset, with extensions targeting energy-efficient DSP such as hardware-loops, automatic increment of addresses during load and store operations, bit manipulation instructions, fixed-point, and packed SIMD operations. The SoC includes a full set of peripherals, namely a Quad SPI, I^2C, UART, GPIOs, JTAG, and a HyperBus interface [119]. All of the peripherals' data transfers are managed by μDMA to minimize the amount of heavy-lifting needed by the processor.

3.3.3 Multi-core platforms

Multi-core platforms or cluster-based systems are the core of interest for this thesis, as they introduce a concept of a system consisting of two relatively loosely coupled parts: a general-purpose processor, and a cluster of simple RISC-V cores packed with memory utilized by those cores.

Mr. Wolf is an SoC featuring a relatively small microcontroller based on a RISC-V core, coupled with an IO subsystem with a wide range of peripherals. The microcontroller can offload compute-intensive kernels to an eight-core floating-point capable processing engine available on-demand [120]. The cluster resides on a dedicated voltage and frequency domain and contains eight RISC-V cores implementing RV32IMC instruction set with extensions for DSP. The cluster also features two floating-point units shared among the cores in the cluster which implement common floating-point operations, including FMAC. The cluster can access a banked L1 memory which enables usage of shared memory programming models. Event management, parallel thread dispatch, and synchronization are supported by a dedicated hardware block [121].

A more advanced SoC architecture can be found in Fulmine, an SoC based on a multi-core cluster coupled with specialized hardware for compute-intensive data processing. Fulmine's architecture again distincts between two voltage and frequency domains, namely between the microcontroller and cluster. The cluster contains four processing general-purpose in-order, single-issue, four-stage pipeline, OpenRISC [122] ISA implementing processing cores called OR10N. Additionally, the cluster contains Hardware Cryptography Engine (HWCRYPT) and Hardware Convolution Engine (HWCE). Every core in the cluster can access a banked L1 Tightly-Coupled Data Memory [123].

Mia Wallace is an SoC aimed at energy-efficient brain-inspired computing, by tightly integrating convolution engine used to accelerate convolution neural networks inference in edge IoT nodes [25]. Regarding its architecture, it is similar to that of the aforementioned Fulmine SoC, but due to the nature of the applications that are meant to be run on it, lacks hardware cryptography support. The cluster contains four OpenRISC ISA OR10N cores with extensions for energy-efficient DSP operations, explicitly managed and banked TCDM.

The most powerful multi-core single-cluster SoC is Vega which is consisted of a single core that manages SoC and IO, and nine core cluster supporting multi-precision SIMD computation for both integers and floating-point data. Vega SoC aims towards highly energy-efficient always-on IoT edge nodes which support the acceleration of deep neural network inference packed with cognitive wake-up from MRAM-based state-retentive sleep mode [124].

3.3.4 Multi-cluster systems

Unlike SoCs described in the previous section (3.3.3) whose high-level architectural overview generally consisted out of a general-purpose controller and a cluster with multiple general-purpose processing cores and potentially accelerators for specific applications, multi-cluster systems described in this section anticipate existence of multiple clusters attached via interconnect to a single general-purpose controller. A core exemplar of such a system is HERO - Open **He**terogeneous **Re**seach Platform [26] which combines a hard-IP ARM Cortex-A host CPU with a scalable, configurable, and extensible FPGA implementation of a cluster-based programmable manycore accelerator, colloquially cluster. While HERO's goals were that of primarily research type, authors claim that implementing up to 64 of RISC-V cores running at 30 MHz can yield around 1.9 GIPS of raw performance. The platform introduced OpenMP programming model with support to offload computation to the cluster directly from the source executed on a general-purpose controller. Multiple hardware platforms are supported, such as Juno ARM Development Platform and Xilinx Zynq ZC706 Evaluation Kit. The latter was tried in practice as a potential hardware platform for the development of outcomes of this thesis but was discarded as tryout tests could not guarantee stability to successfully support outcomes on a hardware layer.

More recently, the second iteration of the HERO platform was released which, unlike the initial version, offered support for both ARM (ARMv8) and RISC-V (RV64) as a general-purpose controller, both of them being 64-bit application-class processors. HEROv2 offers seamless sharing of data between 64-bit controller and 32-bit accelerator cores, a fully open-source on-chip network, unified heterogeneous programming interface, and a mixed-data-model mixed-ISA heterogeneous compiler based on LLVM. The aforementioned compiler allows for single-source single-binary development of heterogeneous applications, utilizing OpenMP with support for offloading code to accelerator directly from the source [125].

There are other projects aiming to provide academic and research-level platforms for the exploration of heterogeneous and manycore architectures. One of them not stemming from the PULP project is OpenPiton. OpenPiton is a general-purpose, multithreaded manycore processor and framework which leverages OpenSPARC T1 core and creates upon it a flexible, modern manycore design. OpenPiton's many-core design consists of 64-bit cores implementing SPARC v9 ISA with a distributed, directory-based cache coherence protocol. Furthermore, it contains a pipelined dual-precision floating-point unit per core and supports native multithreading [126]. In a recent collaboration, previously described Ariane core [116] was integrated into the OpenPiton framework. The resulting platform is an open-source, RISC-V-based, Linux-booting, symmetric multiprocessing framework designed to enable scalable architecture research prototypes [127].

3.3.5 Accelerators

Accelerators for specific applications are non-standalone components, usually depending on an SoC which will utilize their processing power to deliver a result within a larger computing system. It is thus relatively hard to observe them independently of the platforms they are integrated to. Bearing that context, this subsection will present a couple of accelerators implemented under the umbrella of the PULP platform.

HWCRYPT is a cryptographic hardware accelerator capable of supporting multiple encryption and decryption modes for AES and Keccak cryptographic algorithms [128]. The accelerator balances between the need to provide computing power in the respective domain, while running in a heavily energy-constrained IoT domain. HWCRYPT is integrated into previously mentioned Fulmine SoC [123].

Regarding one of the most disruptive domains of the current temporal moment, that is deep learning and brain-inspired computing, authors in [129] proposed NeuroStream coprocessors as an alternative to vector-processing, providing a flexible form of parallel execution without the need for fine-grained synchronization. NeuroStream coprocessors are packed with energy-efficient RISC-V processing elements in a cluster named NeuroCluster, where each cluster contains four processing elements and eight NeuroStream coprocessors.

Lastly, HWCE - Hardware Convolution Engine will be described in a dedicated section (3.4), as it is heavily used to deliver some of the outcomes of this thesis.

3.4 GAP8

Global growth of the concept of the Internet of Things implies the increase of data acquired by the nodes that the IoT consists of, which again implies the increase of the need to process at least parts of the acquired data. In most of the current cases, data is transferred to a remote server, which processes the data and returns it back to a node which in that case, according to a use-case can act as an actuator. With the increase of the data acquired, a need to process some of that data on the end node emerged. In some scenarios, servers became overwhelmed by the amount of data that they need to process, and in some cases the result has to be instantaneous because the actuator decision depends on a result that cannot pay the latency implied. The concept of federated learning emerged as well, further driving the need for edge-node data processing. The main issue of edge-node data processing is the fact that edge-nodes are by design of ultra-low-power as they need to provide a small energy footprint, due to the fact that they are mostly battery powered.

To compromise between the need to stay in the domain of ultra-low-power but provide necessary computational power on demand, based on the previously described platforms such as single-cluster (section 3.3.3) or multi-cluster HERO platform (section 3.3.4), a GAP8 chip was

designed. GAP's main forte is high-performance on edge devices that are heavily constrained by energy availability. GAP8 is a heterogeneous system featuring a total of nine RISC-V-based processing cores. As their authors describe it, it is a fully programmable RISC-V IoT-edge computing engine, featuring an 8-core cluster with hardware convolution engine and ultra-low power memory control unit [23]. GAP8 is designed and manufactured by GreenWaves Technologies[8], which further explains the fact that GAP is an abbreviation for **G**reenWaves **A**pplication **P**rocessor.

3.4.1 Architecture of the platform

GAP8 can roughly be separated into two loosely coupled parts: **Fabric controler** and **PULP cluster**. Fabric controller is a RISC-V general-purpose processing core with its own instruction cache and fast L1 memory which is of 16 kB in size and solely accessible by the fabric controller. The platform features a rich set of peripherals, such as RTC, UART, SPI, I^2C, HyperBus, GPIO, and JTAG. Data transfers to and from peripherals are managed by a multi-channel Input/Output μDMA to minimize the number of interactions and the workload of the fabric controller when performing Input/Output operations [23].

The second part is of the platform is **PULP cluster**, colloquially called the **cluster**. The cluster features a total of eight RISC-V processing cores with custom hardware convolution engine (HWCE) designed for convolution neural network inference, a DMA unit for data transfers between disjunct memories in the system, instruction cache shared by all of the cores in the cluster, and shared L1 memory of the size of 64 kB. L1 memory of the cluster is banked so that each processing core has its own part. High-level block diagram of GA8 architecture is provided on figure 3.4. Note that HWCE, though featured by the platform is not depicted on the figure. Furthermore, HWCE though powerful lacks serious support and its design is somewhat flawed which will be covered in the API section (3.4.2), as most of those limitations are mirrored through API.

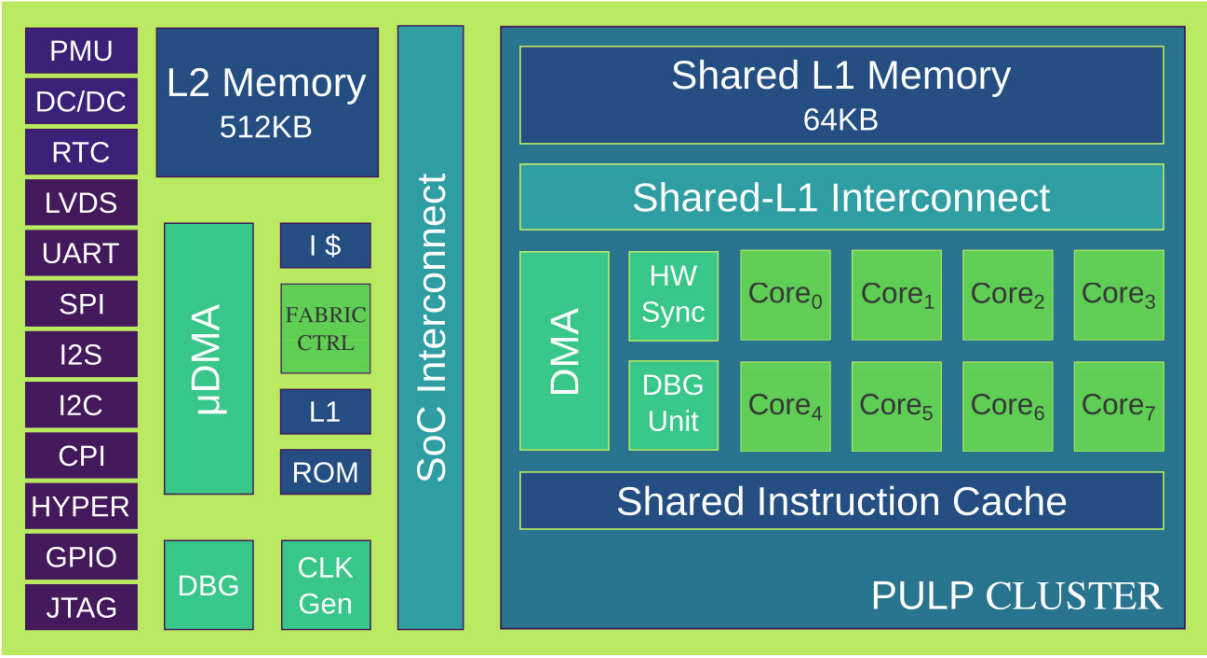
GAP8 SoC features additional L2 Memory of size of 512 kB which is directly accessible by both the fabric controller and all cores, including HWCE, in the cluster.

GAPuino prototyping board

The concrete implementation of the GAP8 platform used in this thesis came in form of the GAPuino development board. GAPuino, which can be observed on figure 3.5 is a board made in the form factor of Arduino Uno that includes GAP8 and peripherals needed to prototype applications for GAP8 [131]. It is stated that the board is compatible with most of the Arduino

[8] `https://greenwaves-technologies.com/`
[9] ©2019 IEEE

Figure 3.4: Block diagram of the GAP8 architecture[9][130]



shields. Among all peripherals, it is worth mentioning that the board features an additional portion of L3 flash memory as large as 512Mbits connected to the GAP8 SoC through the HyperBus interface.

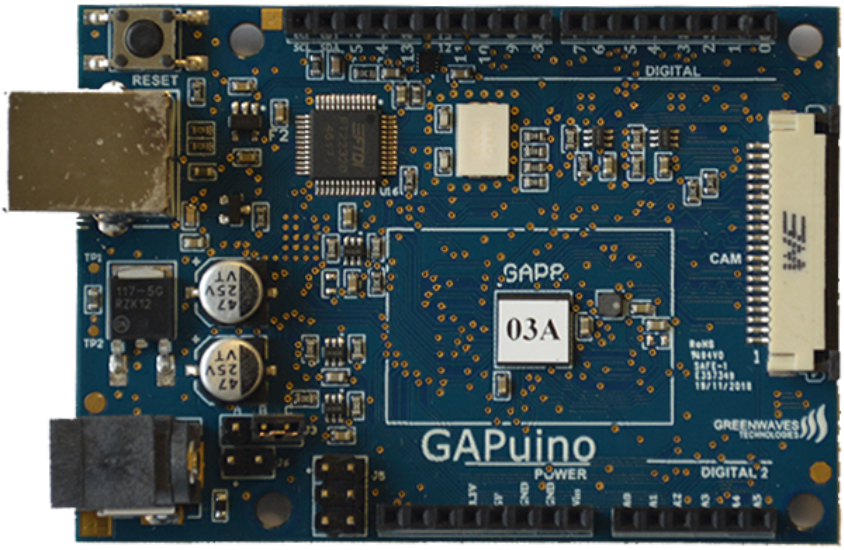
3.4.2 API

Native GAP8 programming API is exposed through programming language C with an option of utilizing OpenMP extension for parallelization of computations on the cluster. GAP8 natively supports two operating systems, PulpOS developed for the purposes of the PULP platform (section 3.3), and FreeRTOS[11], a popular and free real-time operating system for which the GAP8 API has been implemented. Furthermore, GAP8 API exposes parts of the functionalities through PMSIS API which can be implemented technically by any operating system to provide a common layer for the application programming [132]. PMSIS API in the context of GAP8 acts as a wrapper for FreeRTOS and PulpOS native API calls. This thesis for code generation that is described further in the text mostly utilizes PMSIS API calls with bits of PulpOS API calls. While it would be impossible and counterproductive to provide a complete overview of the API, for the sake of better understanding both the hand-tuned code and code that is being generated, utilized parts of the APIs will be described in this subsection. Most of the API is well described in official documentation, which although being generally good, sometimes lacks clarity and coherency [133].

[10]©GreenWaves Technologies, 2022, Permission granted to reuse, available at: `https://greenwaves-technologies.com/product/gapuino/`

[11]`https://www.freertos.org/`

Figure 3.5: GAPuino development board[10]



Listing 3.1 displays interfaces of initialization and deinitialization functions. Kickoff function (line 1) as a parameter accepts a function which will be the entry point for program on the fabric controller and is invoked right after the start of the program, while the exit function (line 2) exits platform with exit code as a parameter.

```
1 static inline int pmsis_kickoff(void *arg);
2 static inline void pmsis_exit(int err);
```

Listing 3.1: Kickoff and exit functions[12]

Cluster execution API

Listing 3.2 provides an overview of the function used to instantiate the cluster, run computation on it, and then close the cluster. Initialization function on line 1 accepts cluster configuration structure, available in listing 3.5 on line 1 as a parameter and initializes it with default values. In the API, the cluster is treated like a generic device which implies fetching it with a generic device-fetching function, listed in listing 3.3. Device-opening function using the aforementioned cluster configuration, initializes device descriptor embodied as *pi_device* structure available in listing 3.4. After device descriptor instantiation and initialization, one can use cluster-opening function (listing 3.2, line 2) to open the cluster. Analogous function with previously mentioned device descriptor can be used to close the cluster (line 3).

```
1 void pi_cluster_conf_init(struct pi_cluster_conf *conf);
2 int pi_cluster_open(struct pi_device *device);
3 int pi_cluster_close(struct pi_device *device);
```

[12] https://github.com/GreenWaves-Technologies/gap_sdk/blob/master/rtos/pmsis/pmsis_api/include/pmsis/rtos/os_frontend_api/os.h

```
intpi_cluster_send_task_to_cl(
    structpi_device    *device,
    structpi_cluster_task    *task
);
```

Listing 3.2: Cluster handling functions[13]

```
voidpi_open_from_conf(structpi_device      *device,void    *conf);
```

Listing 3.3: Device handling functions[14]

```
typedefstructpi_device {
    structpi_device_api    *api;
    void *config;
    void *data;
} pi_device_t;
```

Listing 3.4: Device encapsulating structure[15]

To send task for execution on cluster, a cluster configuration descriptor has to be instantiated and initialized. Unlike cluster configuration descriptor and device desriptor, memory allocation for cluster task descriptor is done manually through *malloc*. Task descriptor, listed in listing 3.5 on line 10 wraps configuration settings needed for a task to be executed on cluster, including pointer to a function which acts as an entry point upon inception of cluster execution (line 11) and arguments that will be passed to that function (line 12).

```
1  structpi_cluster_conf {
2      pi_device_e device_type;
3      intid;
4      void *heap_start;
5      uint32_t heap_size;
6      structpmsis_event_kernel_wrap * event_kernel;
7      pi_cluster_flags_e flags;
8  };
9
10 structpi_cluster_task {
11     void( *entry)(void *);
12     void *arg;
13     void *stacks;
14     uint32_t stack_size;
15     uint32_t slave_stack_size;
```

[13] https://github.com/GreenWaves-Technologies/gap_sdk/blob/master/rtos/pmsis/pmsis_api/include/pmsis/cluster/cluster_sync/fc_to_cl_delegate.h

[14] https://github.com/GreenWaves-Technologies/gap_sdk/blob/master/rtos/pmsis/pmsis_api/include/pmsis/device.h

[15] https://github.com/GreenWaves-Technologies/gap_sdk/blob/master/rtos/pmsis/pmsis_api/include/pmsis/pmsis_types.h

```
16      int nb_cores;
17      pi_task_t *completion_callback;
18      int stack_allocated;
19      struct pi_cluster_task    *next;
20      CLUSTER_TASK_IMPLEM;
21  };
```

Listing 3.5: Cluster encapsulating structures[16]

Finally, *pi_cluster_send_task_to_cl* back from listing 3.2, line 4, with cluster device descriptor and task descriptor as parameters can be used to start the computation on cluster. While the aforementioned function blocks the execution on the fabric controller until the computation finishes on the cluster, it has its own asynchronous analogon which upon calling does not block the execution on the fabric controller.

Benchmarking API

For benchmarking purposes of the contributions of this thesis, parts of API used for measurements are introduced in this subsection. First, a function which returns time elapsed since the startup of the system in milliseconds is available in listing 3.6.

```
1  unsigned int rt_time_get_us();
```

Listing 3.6: Wall-time fetch[17]

Next, here in listing 3.7 five functions for accessing and manipulating performance counters are given. GAP8 has many performance counters which can be configured to count one or more events [133]. Particularly interesting for benchmarking and evaluation will be counter of active cycles which, according to a comment in source file[18], counts cycles the core was active, i. e. not sleeping. Counter of active cycles is configured by shifting 1 to the left for *PI_PERF_ACTIVE_CYCLES*[18] and passing the result integer as a parameter to configuration function (line 1). The rest of the functions are self-descriptive. Reset function on line 2 resets the counter, start function on line 3 starts the counter and should be called prior to starting the computation whose performance is being measured, while stop function on line 4 stops the counter if called after the measured computation finishes with execution. State of the counter is then read with the reading function on line 5, again by passing a parameter which denotes which counter is being read. In the case of measuring active cycles, 1 shifted to the left for *PI_PERF_ACTIVE_CYCLES* should be passed.

[16] `https://github.com/GreenWaves-Technologies/gap_sdk/blob/master/rtos/pmsis/pmsis_api/include/pmsis/cluster/cl_pmsis_types.h`

[17] File cannot be referenced as it eventually does not exist in the official repository

[18] `https://github.com/GreenWaves-Technologies/gap_sdk/blob/master/rtos/pmsis/pmsis_api/include/pmsis/chips/gap8/perf.h`

```
1  int pi_gpio_pin_configure(
2      struct pi_device   *device,
3      pi_gpio_e gpio,
4      pi_gpio_flags_e flags
5  );
6  int pi_gpio_pin_write(
7      struct pi_device   *device,
8      uint32_t pin,
9      uint32_t value
10 );
```

Listing 3.9: GPIO functions[21]

```
static inline void pi_perf_conf(unsigned events);
static inline void pi_perf_reset();
static inline void pi_perf_start();
static inline void pi_perf_stop();
static inline unsigned int pi_perf_read(int event);
```

Listing 3.7: Performance calls[19]

GPIO API

GPIO API exposes a function for controlling the GPIO circuit which is in the context of this thesis used for benchmarking purposes. When setting up GPIO on GAP8, the first thing that has to be done is to configure the function of the specific pad in case it supports more than one function [133]. That can be achieved by invoking the pad set function provided in listing 3.8. The function accepts two parameters, an identifier of the pad, and an identifier of the desired pad function.

```
void pi_pad_set_function(pi_pad_e pad, pi_pad_func_e function);
```

Listing 3.8: Pad function[20]

Similar to the API part related to the cluster, described in section 3.4.2, GPIO is from the API perspective treated as a device, and thus requires appropriate initialization of a device descriptor. According to the documentation, this is done by invoking a configuration function available in listing 3.9 on line 1 with parameters being a device descriptor structure, an identifier of a concrete pin within GPIO port, and configuration flags which in most cases just configure the pin as an input or output. Driving the pin to logical 1 or logical 0 is then done by invoking the write function on line 6 with device descriptor, GPIO pin, and desired value as parameters.

[19] https://github.com/GreenWaves-Technologies/gap_sdk/blob/master/rtos/pmsis/pmsis_api/include/pmsis/drivers/perf.h

[20] https://github.com/GreenWaves-Technologies/gap_sdk/blob/master/rtos/pmsis/pmsis_api/include/pmsis/drivers/pad.h

Hardware convolution engine API

The special part of the API overview is dedicated to the API of the Hardware convolution engine. HWCE API is part of the autotiler library [134] and thus requires its compilation during the SDK build process. Although this part of the GAP8 SoC was normally advertised, up to date remains unclear if it was meant to be used independently of the autotiler library and regardless of the use-case of convolution neural networks inference. The API exists in specific header files which expose the interface to the HWCE, but it lacks any kind of documentation for its utilization. Furthermore, developers of the GAP8 SDK admitted that this part of the independent usage of this part of SoC might be unreasonably cumbersome[22]. However, regarding the outcomes of this thesis and despite all limitations, HWCE was successfully set up. Specifics of utilizing HWCE will be described accordingly and with respect to a particular function that limitation is referred to.

Listing 3.10 provides two of the most basic functions which enable (line 1) or disable (line 2) the HWCE.

```
1 voidHWCE_Enable();
2 voidHWCE_Disable();
```

Listing 3.10: Enable and disable functions

Listing 3.11 provides generic functions which set up or in other ways influence the behaviour of HWCE. Software reset function on line 1 performs a software reset of the HWCE. Generic initialization function on line 3 initializes HWCE with respect to provided parameters:

- **ConvType** - 0 in case of convolution with filter of size 5×5, 1 in case of convolution with filter of size 3×3, 2 in cases of convolutions with filters of sizes 7×4 or 7×7
- **WStride** - Width of the input matrix
- **Norm** - Qnorm of the fixed-point aritmetic

By executing experimental tryouts of the HWCE, it was concluded that the **WStride** parameter has to be set to 0, regardless of the dimensions of the input matrix, and that **Norm** parameter should also be set to 0. Up to today, it remains unclear why is it so.

Function for setting input bias on line 9 sets the global bias for the convolution result. As in the case of parameters for the initialization function, to date remains unclear what are the actual repercussions of invoking this function. Function for setting Y in mode on line 11 should accumulate convolution results with the previous result in case of the input parameter set to 1, or use the input bias set by the aforementioned bias setting function (line 9) in case of the input parameter set to 0. Again, to date, the alleged behaviour cannot be experimentally confirmed to work in the use-cases this thesis works with.

[21] https://github.com/GreenWaves-Technologies/gap_sdk/blob/master/rtos/pmsis/pmsis_api/include/pmsis/drivers/gpio.h

[22] https://github.com/GreenWaves-Technologies/gap_sdk/issues/246

```
void HwCE_SoftReset();

void HWCE_GenericInit(
    unsigned int ConvType,
    unsigned int WStride,
    unsigned int Norm
);

void HwCE_SetInputBias(int Bias);

void HwCE_SetYinMode(unsigned int Disable);
```

Listing 3.11: Setup functions

Listing 3.12 provides functions which ultimately execute a convolution operation on HWCE. Since HWCE supports four different convolution modes, there are four different functions mirroring those four modes. All of the input arrays to the HWCE have to be one-dimensional. Each function takes a pointer to an input array, a pointer to an output array, a pointer to convolution filter, convolution bias, and dimensions of the input matrix. The exception is convolution with a filter of size 3×3 which is something that will be tackled further in text. The first constraint set by the HWCE is mutual for every convolution mode, and that is that width of the input matrix has to be a multiple of 2.

```
void HWCE_ProcessOneTile3x3_MultiOut(
    short int *In, short int *Out0, short int *Out1, short int *Out2,
    short int *Filter,
    short int Bias,
    unsigned int W, unsigned int H,
    unsigned int OutMask
);

void HWCE_ProcessOneTile5x5(
    short int *In, short int *Out, short int *Filter,
    short int Bias,
    unsigned int W, unsigned int H
);

void HWCE_ProcessOneTile7x7(
    short int *In, short int *Out, short int *Filter,
    short int Bias,
    unsigned int W, unsigned int H
);
```

```
22 voidHWCE_ProcessOneTile7x4(
23     shortint    *In,shortint      *Out,shortint      *Filter,
24     shortintBias,
25     unsignedintW,unsignedintH
26 );
```

Listing 3.12: Convolution kickoff functions

Function for executing convolution with filter of size 3×3 (line 1) is of different signature, because HWCE supports multicycle output for that type of convolution. In the case of a single output cycle, each filter group has to be padded with one zero, effectively meaning that the convolution filter is a one-dimensional array with 10 elements. In the case of two outputs per cycle, there are no constraints on the sizes of input data, but in the case of three outputs per cycle, input data has to be padded with one zero after every 27 coefficients. Besides parameters that are the same for every function handling different convolution modes, 3×3 convolution function takes two additional pointers to two output arrays, one for each additional output cycle. Furthermore, the function takes a parameter *OutMask* which specifies the number of outputs per cycle that is going to be used. If that parameter is set to $0x7$, one output per cycle is produced, if set to $0x3$, two outputs per cycle are produced, and if set to $0x1$, three outputs per cycle are produced. It is worth mentioning that experimental tryouts of the HWCE yielded no result in cases of more than one output per cycle.

Function for convolution with filter of size 5×5 (line 9) has a constraint on filter group being padded with one zero, while filter for convolution with filter of size 7×7 (line 15) has to be padded with seven zeros, reason for that being HWCE's utilization of 7×4 convolution mode, by performing the latter convolution operation two times in a row. Convolution with a filter of size 7×4 does not have any constraints on the input matrix or filter. It is important to mention that the convolution does not pad input matrices, which implies that the output arrays are smaller, depending on the filter size. Given an input matrix of size $W \times H$, the output array will be of size:

- 3×3 — $(W-2) \times (W-2)$
- 5×5 — $(W-4) \times (W-4)$
- 7×7 — $(W-6) \times (W-6)$
- 7×4 — $(W-6) \times (W-3)$

Chapter 4

RISE Stack

This chapter describes the programming language RISE, together with its framework which was used to deliver the research contributions proposed by the thesis. The framework consists of the aforementioned programming language RISE used to express computations on a high-level, programming language ELEVATE used to express optimization strategies, and Shine compiler which implements compilation chain. The framework is currently under active development by researchers at the Universities of Edinburgh, Glasgow, and Münster. This chapter also provides a brief introduction to the main concepts of functional programming, that are both applicable to RISE language as well to generic functional programming languages.

4.1 General concepts

As stated before in the text, RISE is a functional programming language, which implicitly means that it inherits concepts from general functional programming languages. In such languages, functions as building blocks are first-class citizens which allows their almost-to equal treatment compared to other building blocks, such as data structures. One of these concepts of functional programming is higher-order functions which take functions as parameters or return them as return values. This section partially provides an overview of such functions which exist as patterns in RISE but are applicable to functional programming in general. Mind, not all of the listed functions are strictly higher-order, but they certainly were popularized by functional programming concepts.

Map

Map is a higher-order function, typically available in functional programming languages, as well in some that cannot strictly be defined as functional, like Javascript. Map function takes two parameters, the first one is a collection of elements of type T, and the second one is a function of type $T \rightarrow U$. As shown in relation 4.1, map transforms the input collection by applying the function provided as a parameter to every element of the input collection. The

result is a transformed collection of equal size as the input collection.

$$map\left([i_1, i_2, \ldots, i_n], f : T \rightarrow U\right) = [f(i_1), f(i_2), \ldots, f(i_n)] \tag{4.1}$$

The alternative notation available in relation 4.2 suggests that the map function is applied on the collection, instead of the collection being a parameter of the map. Both notations are semantically equivalent, with the second one being somewhat more concise.

$$[i_1, i_2, \ldots, i_n].map\left(f : T \rightarrow U\right) = [f(i_1), f(i_2), \ldots, f(i_n)] \tag{4.2}$$

Reduce

Reduce is a higher-order function that, in general, accepts 3 parameters: a collection of elements of type T, a function of type $(U, T) \rightarrow U$, and an initial element of type U. Reduce reduces given collection by repeatedly applying the given function to the accumulator which is initially set to the provided initial element, and to one of the elements of the collection, as depicted in relation 4.3

$$[i_1, i_2, \ldots, i_n].reduce\left(f : (U, T) \rightarrow U, accu : U\right) = f(f(f(f(accu, i_1), i_2), ...), i_n) \tag{4.3}$$

To preserve result correctness, the reduction operator has to be associative. Similar to the second notation of map function demonstrated in relation 4.2, a collection of elements can in the context of the reduce function be observed as if the reduce is executed on in, rather than taking it as a parameter. In some programming languages, a higher-order function with semantics as described here is available under the name *fold*.

The two previously mentioned patterns, **Map** and **Reduce** are as described used in a once-popular programming paradigm **MapReduce** used for the analysis of large data sets.

Zip

Zip is a function that takes two collections of the same sizes and produces a collection of pairs of respective elements from the input collections, as demonstrated in relation 4.4.

$$zip\left([i_1, i_2, \ldots, i_n], [j_1, j_2, \ldots, j_n]\right) = [(i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n)] \tag{4.4}$$

Join

Join is a function that takes a collection and reduces its dimensionality by 1, effectively *joining* the outermost layer of collections in the provided collection. In some functional programming languages like Scala, function *flatten* can be considered *join*'s counterpart. A common use-case

for join function is flattening matrix as an array of arrays to a 1-dimensional array, as given in relation 4.5.

$$join\left(\left[\left[i_1, i_2, \ldots, l_n\right], \ldots, \left[i_m, i_{m+1}, \ldots, l_{n+m}\right]\right]\right) = \left[i_1, i_2, \ldots, l_n, \ldots, i_m, i_{m+1}, \ldots, l_{n+m}\right] \tag{4.5}$$

Fst and **Snd**

Fst is a function which returns the first element of a pair or tuple (relation 4.6). Analogously, Snd returns the second element of a pair or tuple (relation 4.7).

$$fst\left(x_1, x_2\right) = x_1 \tag{4.6}$$

$$snd\left(x_1, x_2\right) = x_2 \tag{4.7}$$

Various programming languages that support simple and instant encapsulation of data in tuples, provide analogous constructs for accessing the underlying elements. For example, Scala provides functions _1 and _2 which can be invoked on tuple structures.

Slide

Slide is a function or primitive which virtually *slides* a window over a collection, returning a collection of collections created by the sliding window passing over an input collection. It usually takes two parameters, step which indicates how many elements will be skipped passing the collection, and size indicating how large the sliding window will be. An example for *slide* primitive applied to an array of size n with parameters 3 for size and 1 for step can be seen in relation 4.8.

$$\left[i_1, i_2, i_3, i_4, \ldots, l_n\right].slide(size = 3, step = 1) = \left[\left[i_1, i_2, i_3\right], \left[i_2, i_3, i_4\right], \ldots \left[l_{n-2}, l_{n-1}, l_n\right]\right] \tag{4.8}$$

Generally, in the context of RISE, simple primitives like *slide* can be composed to create complex primitives. Two-dimensional slide (*slide2D*) is exactly an example of such composition, as described in [135].

Pad

Pad is a primitive which *pads* an input collection with a provided element, or according to a provided function, adding a number of elements usually also provided as a parameter. In context of RISE, there are two implementations of *pad* primitive which are of special interest, namely *padCst* which pads the collection with a constant provided as an parameter, and *padClamp* which repeats the margin element necessary number of times. An example of *padCst*, padding

an array with constant 0, 3 times from both left and right size is avilable in relation 4.9, wile an example of *padClamp* with the same number of repeats from both sides is avilable in relation 4.10.

$$[i_1, i_2, \ldots, l_n]\,.padCst(constant = 0, number = 3) = [0,0,0,i_1,i_2,\ldots,l_n,0,0,0] \tag{4.9}$$

$$[i_1, i_2, \ldots, l_n]\,.padClamp(number = 3) = [i_1,i_1,i_1,i_1,i_2,\ldots,l_n,l_n,l_n,l_n] \tag{4.10}$$

Both of the aforementioned primitives exist in higher-dimensional variations (*padCst2D* and *padClamp2D*), build up from their one dimensional counterparts. In RISE, primitive constructors are available which allow for more detailed specification of the behaviour of *pad* primitives, e. g. specifying the number of repeats on the left and right side of a one-dimensional array separately etc.

4.2 RISE & Shine

4.2.1 General

RISE is a functional data-parallel and pattern-based programming language implemented as a deeply embedded domain-specific language in Scala. Its main idea is the paradigm shift towards expressing computations with respect to *what* is being computed, instead of explaining *how* is something computed, as is the usual case of the imperative programming languages. RISE is packed with a compiler named Shine which handles the compilation of the expressions written in RISE to the supported low-level languages and applies transformations expressed in ELEVATE. ELEVATE is a domain-specific language for expressing optimization strategies, and is a part of the RISE stack. The language itself is further described in subsection 4.3. RISE & Shine stack is the successor to the LIFT project [83], and as such cherishes the same ideas, though they are somewhat differently expressed and internally implemented.

An example of a RISE expression representing a dot product of two arrays is provided in listing 4.1. Function is declared on line 1 by the provided lambda. Lambda variables *x* and *y* represent elements of the arrays being multiplied. Elements are first zipped, creating a new array of element pairs (line 3), then mutually multiplied (line 4). Finally, the resulting collection is reduced (line 5) with *add* as reduction operator and *0* as the initial element. It is important to note that this function is size-independent with respect to arrays it works with, but is not datatype-agnostic due to the initial element of the reduction operator (line 5) being declared *0*

as a signed integer. This effectively and unfortunately ties function and arrays it works with as containers of data of the same type. Additionally, the given expression is agnostic with respect to concrete implementations of patterns carrying out parts of the computation. For example reduce on line 5 can be either carried out sequentially (*reduceSeq*) or sequentially but with controlling loop unrolled (*reduceSeqUnroll*). In both cases, the abstract pattern should be replaced with the concrete implementation in later stages of compilation.

Listing 4.1: Dot product in RISE

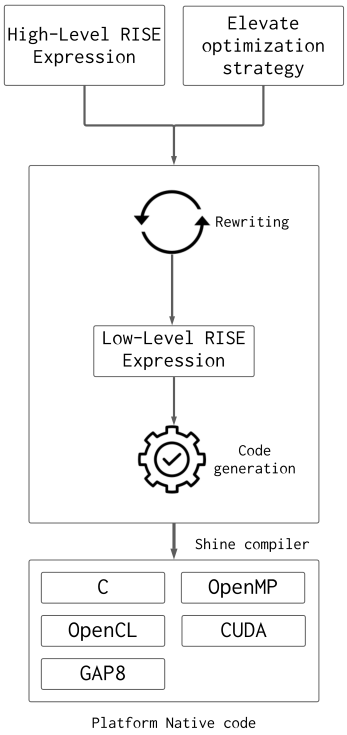
```
valexpr: ToBeTyped[Rise]=fun(x=>
  fun(y=>
    zip(x)(y) |>
      map(fun(f=>fst(f) * snd(f))) |>
      reduce(add)(li32(0))
))
```

The RISE compilation flow is depicted on figure 4.1. Shine compiler takes a computation as an expression written in RISE, and an optimization strategy written in ELEVATE. The compiler then applies an optimization strategy to the input expression, transforming it accordingly, and encoding any low-level optimizations directly into it, yielding a Low-Level RISE Expression. Expression is from RISE translated into lower, imperative-functional hybrid language DPIA [136], first into its functional counterpart and then into its imperative counterpart. The imperative DPIA constructs are then passed to code generation which generates nodes for the abstract syntax tree (AST) of the targeted backend language. Ultimately, AST is printed to code represented as an ordinary string.

4.2.2 Important constructs

RISE itself is a complex domain-specific language with a wide range of constructs for the expression of the data-parallel computations. Providing a complete manual for the language would not be appropriate in the context of this thesis, as it is not the main but auxiliary topic of it. Instead, this subsection provides a subset of the RISE language constructs needed for understanding the rest of the text. The synopsis was made by the existing experience of working with RISE and by observing the RISE codebase. The rest of the statements will be cited accordingly. A complete overview of the language and the framework is available in a recently published paper [137].

Figure 4.1: RISE stack



Expressions

Expression is the core construct of the RISE language, which represent other values or representations of computations of the system. Expressions contain their *type* and according to the codebase and [137] can be concretized to:

- **Identifier** - represents identifiers across the language, e. g. in functions and lambdas
- **Literal** - represents data-level literals
- **Lambda** - models lambda expressions, containing an identifier and an expression
- **Apply** - models function applications
- **DepLambda** - models dependable lambdas
- **DepApply** - models dependable function applications
- **Primitive** - serves as a superclass for every primitive available in the language.

Types

As a standard programming language, RISE has its own typing system. According to [137], core type can be conretized to *TypeIdentifier*, *FunType*, *DepFunType*, and *DataType*. *DataType* represents the typing subsystem for data values, which can further be dissassembled to *ScalarType*, *ArrayType*, and *PairType*. *ArrayType* and *PairType* are complex data types build as collections of values of *ScalarType*. RISE supports following scalar types:

- **Integers** - *i8*, *i16*, *i32*, *i64*

- **Unsigned integers** - *u8*, *u16*, *u32*, *u64*
- **Floating-point numbers** - *f16*, *f32*, *f64*
- **Standard types** - *bool*, *int*

Operators

RISE offers a variety of operators which are used to construct simple and complex data types.

Array construction

Arrays are constructed by using dot operator with type of the underlying data and size of the array (listing 4.2).

Listing 4.2: Array construction

```
1 p‘.‘u32 ↔ ArrayType(p, u32)
2 10‘.‘i32 ↔ ArrayType(10, u32)
3 c‘.‘f‘.‘u32 ↔ ArrayType(c, ArrayType(u32, f))
```

Examples of the one dimensional array declaration are on line 1 which declares an array of *p* elements of type *u32*, and on line 2 which declares an array of *10* elements of type *i32*. Dot operator can be chained to declare multidimensional arrays, as provided in example on line 3 which declares a matrix of dimensions $c \times f$

Tuple construction

Tuples as pair types can be constructed by invoking the *x* helper (listing 4.3). The first example creates a tuple of two elements of type $u32$ (line 1), while the second example (line 2) creates a tuple of an element of type $i32$ and an element of type $i16$.

Listing 4.3: Tuple construction

```
1 u32 x u32 ↔ PairType(u32, u32)
2 i32 x i16 ↔ PairType(i32, i16)
```

Literal declaration and casting

Data literals are constructed by invoking the appropriate helper methods. Listing 4.4 depicts 0 literal creation.

Listing 4.4: Literals

```
1 l(0)//Createsliteralofintegertype
2 lf32(0.0)//Createsliteraloffloattype
3 lf64(0.0)//Createsliteralofdoubletype
```

Literals of other types can generally be constructed to another type by invoking the *cast* helper paired with *::* operator. Casting examples are provided in the following listing (4.5):

Listing 4.5: Literal cast

```
cast(l(0)) :: f16//Createsliteralofintegertypeandcastsittof16
cast(l(1)) :: u64//Createsliteralofintegertypeandcastsittou64
cast(l(2)) :: i32//Createsliteralofintegertypeandcastsittoi32
cast(l(3)) :: u32//Createsliteralofintegertypeandcastsittou32
```

For example, the last case on line 4 creates a literal with value 3 of type *IntData* and then immediately casts it to *u32* representing unsigned 32-bit integers, often translated to *uint32_t*. Some constructs exist for direct creation of literals of frequently used types. Language could be easily extended to support direct creation of literals of every data type supported. The existing direct literal constructors are:

Listing 4.6: Frequent constructors

```
li16(value: Int)//Createsliteralofi16type
li32(value: Int)//Createsliteralofi32type
lu8(value: Int)//Createsliteralofu8type
```

Type declaration

Type construction is achieved by utilizing *->:* operator, and is later used in function construction. The operator can be chained to construct complex types. Examples are available in listing 4.7.

Listing 4.7: Type construction

```
//Denotesatypewhichacceptsau32andreturnsu8
u32 ->:u8  ↔ FunType(u32, u32)

//Denotesatypewhichacceptstwovaluesoftypeu32and
//returnsasinglevalueoftypei64
u32 ->:u32 ->:i64   ↔ FunType(u32, FunType(u32, i64))

//Denotesatypethatacceptsamatrixofsizes6x6ofu8,twomatricesof
    sizes3x3ofint,andreturnsamatrixofsize4x4ofu8
(6‘.‘6‘.‘u8) ->:
  (3‘.‘3‘.‘int) ->:
    (3‘.‘3‘.‘int) ->:
      (4‘.‘4‘.‘u8) ↔ FunType(u8, FunType(int, FunType(int, u8)))
```

Function declaration

Functions can be declared both by providing the function type explicitly and by ommitting it. In both cases, a lambda expression of type *ToBeTyped[Identifier] => ToBeTyped[Expr]* has to

be applied to *fun* object.

Listing 4.8: Function declaration

```
1 valfunction=fun(tuple=>tuple._2)
2
3 valfunctionWithType=fun((10‘.‘u32) ->:(10‘.‘u32))(elems=>
4   mapSeq(fun(elem=>elem * cast(l(2)) :: u32))
5 )
```

Function on line 1 will return a second element of the tuple, while the function on line 3 will multiply every element of the array with 2.

Dependable function declaration

Dependable functions introduce size variables that can be used for size-dependant data containers, e. g. arrays. A list of size variables with types is provided in the lambda parameters list, which can then be used in lambda body.

Listing 4.9: Dependable function declaration

```
1 valsizeFun=depFun((n: Nat)=>fun((n‘.‘int) ->:(n‘.‘int))(elems=>
2   mapSeq(fun(elem=>elem * l(2)))
3 ))
4
5 valmatrixSize=depFun((n: Nat, m: Nat)=>
6   fun((n‘.‘m‘.‘int) ->:((n * m)‘.‘int)(matrix=>
7     matrix > join > map(fun(elem=>elem * l(2)))
8   )
9 )
```

An example provided in listing 4.9 on line 1 is analogous to function on line 3 in listing 4.8 with difference in size of the array being processed. In this case, size of the array is given by the size variable n and will be translated to size parameter of the low-level generated function. Example on line 5 declares a function which accepts a matrix of size $n \times m$, and returns it as an one dimensional array of size $n * m$ after multiplying each element with 2.

Pipe operator

Pipe operator is a language-level helper for expressing function application in a reverse-fashion way, giving a notion of parameter passing through a function. Given a function $f(x)$, one can apply x to the body of the function f by writing:

Listing 4.10: Pipe operator in RISE

```
1 f(x) ↔ x |> f
```

Pipe operator increases code readability, as it enables reading of expressions from top to down, and from left to right.

4.3 ELEVATE

ELEVATE is a domain-specific language for expressing optimization strategies [84, 85]. An example from the official website of the project offers a simple demonstrative strategy, available in listing 4.11. By applying a top-down traversal, when applied the strategy exchanges the most outermost *map* primitive with *mapPar* primitive, and the next following *map* primitive with *mapSeq* primitive. On the other hand, *reduce* primitive is everywhere in expression exchanged with its sequential counterpart *reduceSeq*.

Listing 4.11: Example of a strategy in ELEVATE[1]

```
defoptimizationStrategy: Strategy[Rise]=
  (‘map |-> mapPar‘     ‘@‘ outermost(isMap)) ‘;‘
  (‘map |-> mapSeq‘     ‘@‘ outermost(isMap)) ‘;‘
  (‘reduce |-> reduceSeq‘ ‘@‘ everywhere)
```

Optimization strategies in ELEVATE are built up of *rules* which can, together with other strategies, be chained with *;* operator. Types of *rules* range widely, from algorithmic rules like *map fusion* which fuses composition of two map patterns to one map pattern, to simple lowering rules which interchange abstract patterns with their concrete coutnerparts, like *map* |-> *mapSeq*, *map* |-> *mapPar*, *reduce* |-> *reduceSeq*, or *reduce* |-> *reduceSeqUnroll*. Examples of more complex strategies and their expressions are available in the codebase and in research related to the RISE stack, e. g. [77].

Custom rules can be defined by pattern-matching against RISE expression types like *App* for function application, *Lambda* for lambda expressions, *DepApp* for dependable function application etc. Example of custom optimization strategy is given further in text (subsection 5.5.1)

Since the simple application of the optimization strategy would be applied only to immediate expression, ELEVATE defines recursive strategies which enable traversing entire expressions, namely *topDown*, *bottomUp*, *allTopDown*, *allBottomUp*, and *tryAll* [84]. Strategies are applied to expressions with respect to provided combinator which concretizes traversals, after the @ operator, e. g. *outermost*, *innermost*, or *everywhere*, which apply strategies to outermost matching pattern, innermost matching pattern, or simply everywhere where pattern is matched respectively.

[1] Available at: `https://rise-lang.org/`

4.4 Notable research

So far, RISE has become a solid foundation for wide research and has spawned a number of projects in domains of high-performance computing, signal processing, modeling in general, etc. In this subsection, a brief overview of the research state is presented.

In [77], authors demonstrated the increase of performance regarding expressing image processing pipelines, namely Harris corner detection, in comparison to OpenCV library and Halide compiler, by extending the compiler with appropriate domain and hardware-specific optimizations. Paper [138] demonstrates performance-portable modeling of complex room acoustic simulations with complex boundaries, while the [139] presents expressing FFT in Lift, which generates high-performance GPU code. Regarding accelerators, authors in[140] demonstrate potential for targeting FPGA-based platforms, describing the implementation of Lift VHDL backend. Usage of the RISE / LIFT stack in the context of accelerators is present in [141] which demonstrates work regarding applying Lift to Deep neural network accelerators by mapping expressions to coarse-grained ISA primitives. An example of an integration with other intermediate representations, namely MLIR is available in [142].

Chapter 5

Model Implementation

This chapter provides an in-depth overview of the features implemented in the RISE framework as contributions of this thesis. Everything covered in this chapter is accepted or proposed for acceptance in the official repository of the RISE framework available at [143].

5.1 General

Thesis outcomes were satisfied with the extension and adjustment of the RISE framework. RISE framework, which already supported compilation to C, OpenMP, CUDA, and OpenCL, was extended with the backend for the GAP8 platform. Due to some similarities between OpenCL-supported platforms and the GAP8 platform itself, primarily regarding the separation of concerns between a **host** and a **device**, some parts of the GAP8 backend reused components from the OpenCL backend, while some parts heavily relied on, or adapted OpenCL components.

When considering backend support for a platform or target in Shine compiler, one has to introduce a few concepts:

Module is a wrapper that encapsulates all of the constructs necessary to run a meaningful piece of code. In the case of the C module, it encapsulates include directives, declarations, and functions, while in the case of the GAP8 module, it encapsulates more than one of the aforementioned C modules.

ModuleGenerator is a component which through a chain of method calls set by *ModuleGenerator* trait generates a viable module of a certain type. Module type is set by overriding type Module in the respective ModuleGenerator.

CodeGenerator is a component that maps imperative DPIA primitives to AST for a low-level language.

5.2 GAP8 Module

GAP8 module, which represents and encapsulates a viable and runnable piece of code on the GAP8 platform, is represented by a case class in *Module*[1] file. As can be seen in listing 5.1, a module contains a C submodule that represents host code and a sequence of C submodules which represent potentially multiple accelerator functions, which are going to be executed on the cluster. Module provides a method *compose* (line 2) which enables composition with other GAP8 modules.

Listing 5.1: GAP8 Module

```
1 caseclassModule(hostCode: C.Module, acceleratorFunctions: Seq[C.Module]) {
2   defcompose(other: Module): Module=
3     Module(
4       hostCode.compose(other.hostCode),
5       acceleratorFunctions ++ other.acceleratorFunctions
6     )
7 }
```

Besides fundamental encapsulation of the host and accelerator codes, GAP8 Module's companion object exposes a method for translating the module to code in plain string representation, available in listing 5.2.

Listing 5.2: GAP8 Module

```
deftranslateToString(m: Module): String={
  valaccFunctions=m.acceleratorFunctions
    .map(injectUnpacking)
    .map(C.Module.translateToString)

  valhostCode=C.Module.translateToString(m.hostCode)

  s"""
      |\${accFunctions.mkString("\n\n")}
      |\$hostCode
      |""".stripMargin
}
```

Translating a GAP8 module to string is performed by disjunctly mapping *translateToString* method of the C module to underlying modules encapsulating host code and accelerator functions. Prior to mapping that method to accelerator functions, the function which injects un-

[1] https://github.com/rise-lang/shine/blob/main/src/main/scala/shine/GAP8/Module.scala

packing code is mapped to them. This is done because the sole parameter to the low-level C function that acts as an entry point executed on the cluster is of type *void* *. To conform with the low-level interface, parameters passed from the **host** to the **device**, i. e. cluster, are packed in a structure, one for each accelerator function, and stored in a sequence of declarations held by the respective C Module for that accelerator function. Function *injectUnpacking* injects appropriate directives as C AST nodes which unpack the structure passed to the cluster as a parameter that encapsulates multiple parameters. Those directives include casting argument of type *void* * to type *struct cluster_params*, and generating local variable for each member of the structure. Names of those local variables correspond to the names within the structure to keep the consistency with variables of the same names being used within computation constructs in the function that performs computations on the cluster.

5.3 Code generation

Code generation is the last step in the Shine compiler's compilation pipeline. It maps imperative DPIA primitives to C AST nodes, which ultimately get printed to code represented as a string. Shine compiler already features code generators for C, OpenMP, CUDA, and OpenCL. GAP8's code generator reused C and OpenMP generators for host code generation and accelerator code generation respectively, which is described in detail in separate subsections.

5.3.1 Accelerator code generation

Since the OpenMP extension is used natively by the GAP8 SDK to parallelize computations on the cluster, accelerator code generation is built by extending the previously existing OpenMP code generator. Accelerator code generator is constituted of one class, *AcceleratorCodeGenerator*[2], and its companion object. The code generator overrides *cmd* method, adding support for mapping HWCE imperative DPIA primitives to the C language abstract syntax tree while passing other primitives to the *OpenMP.CodeGenerator* object.

Each GAP8 HWCE native API call has its own counterpart in the accelerator code generator that returns C AST nodes that can be easily composed. The counterpart pairs are displayed in table 5.1.

The C AST nodes for HWCE support are generated by invoking *generateCalls* method after matching the appropriate imperative DPIA pattern. Method *generateCalls*, which accepts the following parameters:

- **fs** - an instance of a ConvolutionFilterSize object, available in listing 5.3
- **w** - width of the input matrix

[2] `https://github.com/rise-lang/shine/blob/gap8-hwce/src/main/scala/shine/GAP8/Compilation/AcceleratorCodeGenerator.scala`

Table 5.1: Shine vs. GAP8 native API HWCE calls[2]

Shine's GAP8 code generator	GAP8 native API
hwceEnableCall	HWCE_Enable
hwceDisableCall	HWCE_Disable
hwceSetYinModeCall	HwCE_SetYinMode
hwceGenericInitCall	HWCE_GenericInit
generateHwceCallFunction	HWCE_ProcessOneTile3x3_MultiOut
	HWCE_ProcessOneTile5x5
	HWCE_ProcessOneTile7x7
	HWCE_ProcessOneTile7x4
hwceSetInputBiasCall	HwCE_SetInputBias
hwceSoftResetCall	HwCE_SoftReset

Listing 5.3: ConvolutionFilterSize[3]trait

```
sealedtraitConvolutionFilterSize {
  deftoBackendConst: String
  deffunctionName: String
}
```

- **h** - height of the input Matrix
- **bias** - a value added to the resulting element of the convolution operation
- **in** - C AST expression representing the input matrix
- **filter** - C AST expression representing the filter Matrix
- **output** - C AST expression representing the output matrix

generates a sequence of statements, or to put it straight, a block of C code represented by node *C.AST.Block*. The complete method is available in listing 5.4. The aforementioned *ConvolutionFilterSize* trait encapsulates different low-level API constants, namely a constant passed to generic initialization function which can be:

•HWCE_CONV3x3 in case of a convolution with filter of size 3×3

•HWCE_CONV5x5 in case of a convolution with filter of size 5×5

•HWCE_CONV7x7 in case of convolutions with filters of sizes 7×7 and 7×4 since convolution with filter of size 7×7 relies on the 7×4 mechanisms internally

specifying the convolution operation mode, and name of the low-level API call which corresponds to one of the functions which invoke convolution processing already mentioned in table 5.1.

Listing 5.4: HWCE call sequence

```
privatedefgenerateCalls(fs: ConvolutionFilterSize, w: Nat, h: Nat,
  bias: Nat, in: Expr, filter: Expr, output: Expr): Stmt={
    C.AST.Block(Seq(
      hwceEnableCall,
      hwceGenericInitCall(fs),
      hwceSetYinModeCall(),
      generateHwceCallFunction(fs, w, h, bias, in, filter, output),
      hwceDisableCall
    ))
}
```

Available and supported, yet unused API calls are *hwceSoftResetCall* and *hwceSetInputBiasCall*.

Function *generateHwceCallFunction* which generates a low-level call to a function which starts the convolution takes same parameters as the previously described *generateCalls* functions. It is important to mention that *generateHwceCallFunction* distinguishes between the convolution with a filter of size 3×3 and the other convolution modes, because that particular type uses different low-level API call, with a different argument list. Furthermore, although convolution with a filter of size 3×3 supports multi-out mode in sense of the ability to produce one, two, or three outputs per cycle, for simplicity, only one output per cycle is currently supported, meaning that the low-level API call generated currently fixes *OutMask* to $0x7$ ($0x3$ in case of 2 outputs per cycle, or $0x1$ in case of 3 outputs per cycle). A detailed description of the low-level API is available in one of the previous subsections (3.4).

5.3.2 Host side

Host code generation

Code generator for the **host** side of the backend is placed in *HostCodeGenerator*[4] source file. It extends the C code generator, as current parallelization with the OpenMP extension is supported only on the cluster side of the platform. The main purpose of the specific host code generation is to map *KernelCallCmd* imperative DPIA platform to low-level code which handles running computations on the cluster, which is achieved by overriding the *cmd* function and adding appropriate match clause.

Generate buffer syncronization calls

Since the GAP8 was built upon or reused some components of the OpenCL backend, it, there-

[3] https://github.com/rise-lang/shine/blob/gap8-hwce/src/main/scala/shine/GAP8/ConvolutionFilterSize.scala

[4] https://github.com/rise-lang/shine/blob/main/src/main/scala/shine/GAP8/Compilation/HostCodeGenerator.scala

Table 5.2: Parameters of the low-level counterparts

Method	Parameter name	Parameter description
hwceSetYinModeCall	mode: Int = 1	When set to 1, convolution result will be accumulated with the input bias previously set by hwceSetInputBiasCall. If set to 0, convolution result is accumulated with the previous result
hwceSetInputBiasCall	bias: Int = 0	Sets the default input bias
hwceGenericInitCall	fs: ConvolutionFilterSize	Instance of the ConvolutionFilterSize object specifying convolution size
	wstride: Int = 0	Since input matrix is internally represented as a 1-dimensional array, this parameter specifies width of the input data
	qnorm: Int = 0	Sets the fixed-point arithmetic format

fore, inherited concepts introduced by that package. One of those concepts is a concept of a *buffer*, which is defined as a chunk of data being transferred between the **host** and **device**. Data transfers in heterogeneous systems usually involve non-trivial approaches, often employing additional circuitry like DMAs. That transfers are in RISE modeled by buffer synchronizations. The first task that the GAP8 host code generator does is generating C AST nodes that synchronize the output parameter and every input parameter to the cluster prior to beginning the computation.

Allocate wrapper structure and pack parameters

As stated in the text prior to this subsection, low-level API mandates that the function which acts as the entry point for the computation on the cluster accept only one parameter of type *void* *. Since one parameter usually is not enough to satisfy modern programming needs, multiple parameters are packed inside a structure, and that single structure is passed to the function acting as the entry point. That is why the next steps are:

1. allocating memory for the appropriate *cluster_params* structure by invoking the low-level *PMSIS* API call,
2. generating assignments which map elements within the previously created structure and synchronized buffers.

Generate launch kernel call

Finally, a call to the runtime function *launchKernel* gets generated. The call is generated with the number of cores parameter passed to it, previously extracted from the *KernelCallCmd* primitive.

Host code module generation

HostCodeModuleGenerator[5] acts as a module generator, stitching together flow set by the *ModuleGenerator* trait, reusing *imperativePasses* from the OpenCL instance of the module generator, and chaining it with *generateCode* and *makeHostCodeModule*. Since the host code module for the GAP8 platofrm is of type C Module, this module generator generates modules of the same type.

Generated module is returned from the aforementioned *makeHostCodeModule* function which wraps together multiple constructs:

- structure containing variable of type *Kernel* for each accelerator function or module that is going to be executed on the cluster within the respective GAP8 module
- typedef declaration for the aforementioned structure
- include directive which includes *gap8.h* include header, needed by the GAP8 native runtime

[5] `https://github.com/rise-lang/shine/blob/main/src/main/scala/shine/GAP8/Compilation/HostCodeModuleGenerator.scala`

•a series of the functions for initializing, running, and destroying the kernel

•function which serially invokes the previous 3 functions

•function which generates *main*, i. e. program entry point executed on the fabric controller of the GAP8

Besides everything mentioned, *HostCodeModuleGenerator* provides methods for generating constructs listed previously under points 1, 2, 4, and 5.

5.4 Expression running mechanism

The main idea behind this thesis was to deliver a programming model for heterogeneous systems with capabilities of expressing computations that can relatively easily be run on the accelerator part of the system. To satisfy that condition, a series of primitives were added, adding support for running expressions on the cluster. On the RISE side, *gap8run* primitive is added, as shown in listing 5.5. The primitive accepts two parameters, the first one being a natural number representing the number of cores in the customizable accelerator calculation will be executed on, and the second one being the expression describing computation in question.

Listing 5.5: GAP8 cluster running primitive in RISE

```
gap8run(numCores: Nat)(expression: Rise)
```

After type inference and first part of translation, *gap8run* primitive is translated to *Run*, a functional DPIA primitive which again encodes a number of cores that the computation will be executed on, together with type *dt* of the phrase *input* acquired by translating the expression to phrase *Run* primitive is translated to *KernelCall* primitive in process of separation of host code and accelerator code, which will be showed in subsection 5.4.1. *KernelCall* primitive encodes a series of parameters:

- **funName** - name of the main function that is going to be executed on the cluster
- **cores** - number of cores in the customizable accelerator that the computation specified by the input expression is going to be run on
- **numArgs** - number of arguments to the main function executed on the cluster
- **inTs** - a sequence of types of the input parameters to the main accelerator function
- **outT** - output type of the main accelerator function
- **args** - a sequence of arguments of the main accelerator function

Finally, *KernelCall* functional DPIA primitive is translated to *KernelCallCmd* imperative DPIA primitive available in listing 5.7, which is later in code generation process directly mapped to directives and API calls which run the computation on cluster. *KernelCallCmd* primitive encodes one additional parameter, *out* of type *Phrase[AccType]* which represents a data container, *an accumulator* which contains the result of the computation.

Listing 5.6: GAP8 cluster running DPIA functional primitive

```
Run(cores: Nat)(valdt: DataType,valinput: Phrase[ExpType])
KernelCall(funName: String, cores: Int, numArgs: Int)
  (valinTs: Seq[DataType],valoutT: DataType,
    valargs: Seq[Phrase[ExpType]])
```

Listing 5.7: GAP8 cluster running DPIA imperative primitive

```
KernelCallCmd(funName: String, cores: Int, numArgs: Int)(
  inTypes: Seq[DataType], outType: DataType,
  args: Seq[Phrase[ExpType]], out: Phrase[AccType]
)
```

5.4.1 Host and accelerator code separation

One of the problems that arise regarding compiling code for complex heterogeneous systems from relatively simple expressions is the separation of the code regarding the execution environment. For example, part of the expression can be run on the general-purpose processing unit, i.e. the host, while the other part can be run on the accelerator. Previously described expression running mechanism in section 5.4 explained how to mark parts of the expression that are meant to be run on the cluster, i.e. accelerator, while this subsection describes the process of separation of respective parts of the code with its implementation carried out in object *SeparateHostAndAcceleratorCode*[6].

The compilation and thus separation process starts in the utility file *gen.scala*[7], in the object *gap8*, where a compiler is composed from partial host compiler (see [144], according to [8]) and module generators for both the accelerator module and host module. Partial host compiler employs the aforementioned host and accelerator code separator which traverses a *Phrase* transformed from the initial *Expression*, and matches the functional *Run* primitive (listing 5.6, line 1). Part of the expression that is wrapped by the *Run* primitive is extracted and a function definition is created from it, adding it to a sequence of the accelerator function definitions. Traversal returns the *KernelCall* (listing 5.6, line 2) functional DPIA primitive which wraps data extracted from the *Run* primitive. *KernelCall* primitive further through compilation process gets mapped to imperative primitive *KernelCallCmd* which in code generator triggers generation of the calls needed to launch a computation on the cluster.

This concept theoretically enables multiple disjunct computations to be run on the cluster

[6] https://github.com/rise-lang/shine/blob/main/src/main/scala/shine/GAP8/Compilation/SeparateHostAndAcceleratorCode.scala

[7] https://github.com/rise-lang/shine/blob/main/src/main/scala/util/gen.scala

within the same program execution. However, in this thesis and experiments, only one computation has been tested and proved to work.

5.5 Hardware convolution engine support

As stated previously in the text, GAP8 features a hardware convolution engine that supports 4 different operation modes, exposed through 4 different API calls. For each of those operation modes, an analogous pattern was added in RISE, which is displayed in listing 5.8. Each of the patterns accepts a *bias*, *input*, and *filter*. *Bias* is a parameter that specifies how much will the coefficients in the output matrix be *biased*, by adding *bias* to the result of the convolution operation for the respective element. *Input* is a matrix of size $w \times h$, while *filter* is a matrix dimensioned with respect to the convolution operation mode. The output matrix is defined by the parameters of the input matrix and the filter size:

- **gap8hwConv3x3** - Filter is a matrix of size 3×3, output matrix is of size $(w-2) \times (h-2)$(line 1)
- **gap8hwConv5x5** - Filter is a matrix of size 5×5, output matrix is of size $(w-4) \times (h-4)$(line 4)
- **gap8hwConv7x7** - Filter is a matrix of size 7×7, output matrix is of size $(w-6) \times (h-6)$(line 7)
- **gap8hwConv7x4** - Filter is a matrix of size 7×4, output matrix is of size $(w-6) \times (h-3)$(line 10)

The type system ensures that dimensions of the output matrices are correct with respect to input and filter matrices.

Listing 5.8: RISE primitives for HWCE

```
1  gap8hwConv3x3(bias: Nat)
2    (input: ToBeTyped[Identifier], filter: ToBeTyped[Identifier])
3
4  gap8hwConv5x5(bias: Nat)
5    (input: ToBeTyped[Identifier], filter: ToBeTyped[Identifier])
6
7  gap8hwConv7x7(bias: Nat)
8    (input: ToBeTyped[Identifier], filter: ToBeTyped[Identifier])
9
10 gap8hwConv7x4(bias: Nat)
11   (input: ToBeTyped[Identifier], filter: ToBeTyped[Identifier])
```

[8] https://github.com/rise-lang/shine/blob/main/src/main/scala/util/compiler/package.scala

As a part of the standard compilation flow in Shine, RISE primitives are after type inference translated to respective functional DPIA primitives available in listing 5.9.

Listing 5.9: Functional DPIA primitives for HWCE

```
FunConv3x3(w: Nat, h: Nat, bias: Nat,
  inType: DataType, input: Phrase[ExpType], filter: Phrase[ExpType]
)
FunConv5x5(w: Nat, h: Nat, bias: Nat,
  inType: DataType, input: Phrase[ExpType], filter: Phrase[ExpType]
)
FunConv7x7(w: Nat, h: Nat, bias: Nat,
  inType: DataType, input: Phrase[ExpType], filter: Phrase[ExpType]
)
FunConv7x4(w: Nat, h: Nat, bias: Nat,
  inType: DataType, input: Phrase[ExpType], filter: Phrase[ExpType]
)
```

Further down the compilation flow, functional DPIA primitives are translated to imperative DPIA primitives available in listing 5.10. During this compilation step, filter matrices are serialized to one-dimensional arrays with respect to rows. This is done because such structure fits better low-level GAP8 HWCE API which accepts filter matrices as 1-dimensional arrays. Furthermore, during this compilation step, the filter array is padded if necessary, as it is a constraint set by the low-level API.

Listing 5.10: Imperative DPIA primitives for HWCE

```
Conv3x3(w: Nat, h: Nat, bias: Nat, inType: DataType,
  input: Phrase[ExpType], filter: Phrase[ExpType],
  output: Phrase[AccType]
)

Conv5x5(w: Nat, h: Nat, bias: Nat, inType: DataType,
  input: Phrase[ExpType], filter: Phrase[ExpType],
  output: Phrase[AccType]
)

Conv7x7(w: Nat, h: Nat, bias: Nat, inType: DataType,
  input: Phrase[ExpType], filter: Phrase[ExpType],
  output: Phrase[AccType]
)

```

```
Conv7x4(w: Nat, h: Nat, bias: Nat, inType: DataType,
  input: Phrase[ExpType], filter: Phrase[ExpType],
  output: Phrase[AccType]
)
```

From this point, imperative DPIA primitives are directly mapped to low-level API calls in the process of code generation.

5.5.1 Optimization strategy

Heterogeneous systems own their performance and energy efficiency to specialized pieces of hardware within them which are able to carry out heavy parts of computations they were meant for efficiently. However, as stated previously in the text, the existence of such hardware, not to mention its utilization, often implies and requires knowledge on specific implementation details, communication protocols, and low-level API exposed to the programmer. One of the main ideas of this thesis is not only to encapsulate and make APIs of such hardware easier but to provide a language that would expose high-level primitives and a compiler that would through compilation steps detect patterns constituting computations that are executable on the available specialized hardware, making it somewhat opaque.

From the programmer's point of view, such hardware and accelerators are often used by domain scientists who are not and cannot be experts in each and every accelerator which accelerates computations they want to make. Simply put, one could easily argue that accelerators in heterogeneous systems cannot be exploited efficiently by the ones that they are aimed for. As stated before, the GAP8 platform features a limited but working hardware convolution engine. Convolution itself is an operation that is widely used in the domain of signal processing.

Listing 5.11 displays an optimization rule written in ELEVATE (subsection 4.3) which transforms a series of primitives, roughly displayed in relation 5.1, which constitute a convolution operation in RISE, into a single primitive which denotes concrete convolution with existing implementation on GAP8 platform. The rule is basically tailored for example in subsection 6.2.4, listing 6.11.

$$Input \rightarrow slide \rightarrow map \rightarrow map \rightarrow zip \rightarrow map \rightarrow reduce \tag{5.1}$$

The optimization rule matches appropriate function applications (*App* pattern), dependable function applications (*DepApp* pattern), and lambda expressions (*Lambda* pattern). The rule transforms the aforementioned series of patterns into an appropriate convolution based on the size of the neighbourhood created by the *slide* pattern:

Listing 5.11: Convolution optimization rule in ELEVATE

```
@ruledefgap8hwConvMerge: Strategy[Rise]={
  casee@
    App(
      App(mapSeq(),
        App(mapSeq(), Lambda(_,
          App(
            App(
              App(reduceSeq(), add()),
                App(cast(), _)),
                  App(
                    App(map(), Lambda(_,
                      App(App(mul(), App(fst(), _)), App(snd(), _))
                      )
                    ),
                    App(App(zip(), App(join(), _)), App(join(), filter))
                  )
            )
          )
        )
      ),
      App(Lambda(_,
        App(_,
          App(Lambda(_,
            App(
              DepApp(NatKind, DepApp(NatKind, slide(), size), step), _
            )
          ), _)
        )
      ), in)
    )=>
    (size, step)match{
      case(Cst(iSize), Cst(iStep))=>
        if(1==iStep && 3==iSize) Success(gap8hwConv3x3(0)(in)(filter) !:
            e.t)
        elseif(1==iStep && 5==iSize) Success(gap8hwConv5x5(0)(in)(filter)
            !: e.t)
        elseif(1==iStep && 7==iSize) Success(gap8hwConv7x7(0)(in)(filter)
            !: e.t)
        elseFailure(gap8hwConvMerge)
      case_=>Failure(gap8hwConvMerge)
    }
}
```

•if $size = 3$, *gap8hwConv3x3* pattern is yielded

•if $size = 5$, *gap8hwConv5x5* pattern is yielded

•if $size = 7$, *gap8hwConv7x7* pattern is yielded

All of the cases assume that $step = 1$ when creating a matrix of neighbourhoods. If none of the cases was matched, the strategy fails. It is important to mention that this rule, out of simplicity, does not detect a convolution operation with a filter of size 7×4 supported by the hardware convolution engine.

While the aforementioned strategy yields correct results, as presented later in text in evaluation (subsection 6.2.4), there are a few objections. One could easily argue that the provided optimization rule is too specific, and that is true. First step towards abstracting this rule is accepting generic *maps* and *reduces* instead of matching their counterparts which imply concrete implementations, like *mapSeq* and *reduceSeq*. Also, *cast* pattern should be generalized as its necessity depends on the expression type. Second, higher-level convolutions should be decomposed into simpler, one-dimensional convolutions, with an optimization strategy taking such an approach into account when matching an input expression.

Also, a platform-agnostic convolution primitive could be easily added to RISE, which would with an appropriate optimization strategy transform the platform-agnostic primitive into one that indicates a concrete implementation. Furthermore, this rule fails to detect or will result in failure if applied, in the case of chained convolutions, which is a use-case for the convolution operation in the domain of signal processing, in which it's heavily exploited. However, the way the rule detects convolution is completely legal in the sense that it is the right way to express it in RISE, thus making it relatively plausible.

5.6 Runtime environment

The runtime environment was initially developed by the RISE team for the purposes of the OpenCL code generator. It encapsulates concepts that are characteristic for heterogeneous systems on the low level, i.e. at the level of the system's native programming model, thus liberating Shine's code generator of generating repetitive boilerplate code. Shine's code generator generates code that utilizes the runtime environment exposed through a set of C language *header* files. Since the good practice of software engineering dictates that the wheel should not be reinvented, parts of the OpenCL code generator were reused to develop the GAP8 code generator, which implied the necessity to implement the interface of the runtime environment for the GAP8 platform on a low-level. The initial version was thus adapted to fit the needs of both the OpenCL and GAP8 backends.

Main component and the starting point of the runtime environment is **runtime.h** C header file, available in listing 5.12. The header defines 3 types: **Context**, **Kernel**, and **Buffer**.

The first concept runtime environment re-introduces are:

- **Host** - part of the heterogeneous system which acts as the controller exposed directly to a programmer. It can usually be identified as a general-purpose processor on which the program initially runs, bootstrapping the rest of the system
- **Device** - part of the system which is not general-purpose, an accelerator, or analogous device. Conceptually, one **host** can have multiple **devices**.

Furthermore, the environment introduces concepts of **context**, **kernel**, and **buffer**. Description of the aforementioned concepts are:

- **Context** - based on dichotomy of having a **host** and a **device** in a system with their respective roles, it is clear that API has to hold setting needed to run the computations, primarily on the **device**. **Context** wraps native configuration directives and function calls needed to instantiate **devices** and run computations on them.
- **Kernel** - a part of the program, series of instructions, or programming code that eventually will be executed on a **device**.
- **Buffer** - a chunk of data being moved between a **host** and a **device**.

The *runtime.h* part of the API provides functions for creating **context**, creating **buffer**, destroying **context**, destroying **buffer**, and destroying **kernel**. **Kernel** creation, even at the level of the interface is platform-dependant without a common denominator, hence it is not exposed in the main *runtime* header file, but in platform-specific header.

Though the idea of a uniform interface for the heterogeneous backend platforms holds, there are some minor differences that yielded implementations that could be interpreted as workarounds. For example, *waitFinished* function on line 22 in listing 5.12 in case of OpenCL backend blocks the execution of the program until all of the scheduled computations are finished. However, GAP8 backend simply returns *NULL* because a single **kernel** execution is currently assumed. GAP8 code generation currently will not emit *waitFinished* function call, at least for now. Additionally, *AccessFlags* defined by the structure on line 18 is currently not used in the context of the GAP8 backend.

```
1  #ifndefSHINE_RUNTIME_H
2  #defineSHINE_RUNTIME_H
3
4  #include<stdlib.h>
5  #include<stdint.h>
6  #include<stdbool.h>
7  #include<stdio.h>
8
9  typedefstructContextImpl    * Context;
10 typedefstructKernelImpl    * Kernel;
11 typedefstructBufferImpl    * Buffer;
12
```

```
13 typedefenum{
14     HOST_READ = 1 << 0,
15     HOST_WRITE = 1 << 1,
16     DEVICE_READ = 1 << 2,
17     DEVICE_WRITE = 1 << 3,
18 } AccessFlags;
19
20 Context createDefaultContext();
21 voiddestroyContext(Context ctx);
22 voidwaitFinished(Context ctx);
23
24 Buffer createBuffer(Context ctx, size_t byte_size, AccessFlags access);
25 voiddestroyBuffer(Context ctx, Buffer b);
26 void* hostBufferSync(Context ctx, Buffer b, size_t byte_size,
27     AccessFlags access);
28 DeviceBuffer deviceBufferSync(Context ctx, Buffer b, size_t byte_size,
29     AccessFlags access);
30
31 voiddestroyKernel(Context ctx, Kernel k);
32
33 #endif
```

Listing 5.12: runtime.h[9]

Syncing functions, namely *hostBufferSync* (line 27) and *deviceBufferSync* (line 29) function introduced in *runtime.h* in context of the GAP8 platform are actually not performing any concrete syncing, which leads to a display of an additional advantage of the separation of the code generation and runtime environment, which is the ability to change the implementation of disjunct parts of the both, without the impacting the significant other. For example, the code generation process should be agnostic regarding the data transfers between **host** and **device**, and that is exactly what is provided in this case. Currently, code being generated for the GAP8 platform persists data in L2 memory (see section 3.4) which is directly accessible both by fabric controller (**host**) and cluster (**device**). This enabled simple returning of the inner pointer to the block of memory wrapped by the **Buffer** type, which can be seen on lines 17 and 22 in listing 5.13. Future optimization regarding transfers of chunks of data currently being processed from mutually accessible L2 memory to faster shared L1 memory which will employ DMA circuitry are going to be encapsulated by the aforementioned functions.

```
1 #include"gap8.h"
2
3 Buffer createBuffer(Context ctx, size_t byte_size, AccessFlags access){
4     Buffer buffer = (Buffer) pi_l2_malloc(sizeof(structBufferImpl));
```

[9] https://github.com/rise-lang/shine/blob/main/runtime/runtime.h

```
5       buffer->inner = (void *) pi_l2_malloc(byte_size);
6       buffer->byte_size = byte_size;
7       returnbuffer;
8   }
9
10  voiddestroyBuffer(Context ctx, Buffer b){
11      pmsis_l2_malloc_free(b->inner, b->byte_size);
12      pmsis_l2_malloc_free(b,sizeof(structBufferImpl));
13  }
14
15  void* hostBufferSync(Context ctx, Buffer b, size_t byte_size,
16      AccessFlags access){
17          returnb->inner;
18  }
19
20  DeviceBuffer deviceBufferSync(Context ctx, Buffer b, size_t byte_size,
21      AccessFlags access){
22          returnb->inner;
23  }
```

Listing 5.13: nosync.c

Listing 5.14 provides part of the environment implementation for GAP8 platform. *DeviceBuffer* is implemented as *void pointer*, while implementations of **context** and **kernel** wrap PMSIS API structures that hold cluster instance data and cluster configuration, or structure that holds data about the task that is about to be executed on cluster respectively. This part of the interface exposes functions for context creation with specific device identificator (line 26), and function for loading (line 27) and launching (line 28) kernels. Function for loading kernel accepts a pointer to a function that will be entry point upon cluster execution. It is important to notice that current GAP8 platform instances are equipped with only one cluster or device, while the API theoretically supports multiple clusters attached to one fabric controller. The main part of the runtime environment implementation for the GAP8 platform is available in the appendix.

```
1   #ifndefSHINE_GAP8_H
2   #defineSHINE_GAP8_H
3
4   #include"pmsis.h"
5   #include"gaplib/ImgIO.h"
6
7   typedefvoid * DeviceBuffer;
8
9   #include"../runtime.h"
10
11
```

```
12 structContextImpl {
13     structpi_device cl_device;
14     structpi_cluster_conf cl_configuration;
15 };
16
17 structKernelImpl {
18     structpi_cluster_task * cl_task;
19 };
20
21 structBufferImpl {
22     void* inner;
23     size_t byte_size;
24 };
25
26 Context createContext(intdevice_id);
27 Kernel loadKernel(void( *handler)(void *), uint32_t stack_size);
28 voidlaunchKernel(Context ctx, Kernel k,intnum_threads,void * args);
29
30 #endif
```

Listing 5.14: gap8.h

5.7 Executor

Executor is a component of the framework that takes care of the program execution in its native environment but within the context of the framework. The goal of the executor component is to provide a convenient way of running the RISE expression on a testing platform without the explicit need to copy the generated code printed to the console to a C source file, followed by building and running code using appropriate makefile.

Executor encapsulates the aforementioned steps and provides convenient interface, mitigating the drawbacks. Executor is instantiated by providing the following parameters:

- Local filesystem path to the GAP SDK
- Execution target: **GVSoC** simulator [145] or development **board**
- Target operating system: **FreeRTOS** or **PulpOS**
- Input and output channel: Selects wheter the user communicates with program in execution through **UART** or trough **host** computer.

Upon instantiation, the client explicitly invokes *execute* method with code in the native GAP8 programming model, which launches the following chain of events:

1. A new temporary directory and file are created
2. Generated code is stored in the newly created temporary file

3. Files from the prototype directory within the framework are copied to the temporary folder. These include the runtime library described in subsection 5.6 and a prototype of Makefile used to build and run the program. An excerpt from the Makefile in question is available in listing 5.15
4. Placeholders for the application name (line 1) and name of the main source file (line 3) are replaced with actual corresponding data
5. Shell command is created. The command sources the appropriate configuration script depending on the underlying hardware, enters the created temporary directory, and then executes clean, build and finally run jobs from the Makefile
6. Previously described shell command gets executed as an independent process which returns to the process Executor is executed in.

Listing 5.15: Makefile excerpt

```
APP = #APP_NAME_PLACEHOLDER

APP_SRCS = #APP_MAIN_SRC_PLACEHOLDER
APP_SRCS += gap8/gap8.c
APP_SRCS += gap8/nosync.c

APP_INC  = $(GAP_LIB_PATH)/include
APP_INC += gap8

CONFIG_OPENMP = 1

APP_CFLAGS += -O3 -DFROM_FILE

include $(GAP_SDK_HOME)/tools/rules/pmsis_rules.mk
```

Chapter 6

Model Evaluation

This chapter covers every aspect of the evaluation of the proposed solution. To test the solution in simulated real-world scenarios a physical implementation of the GAP8 SoC was used on GAPuino board [131]. Keeping good scientific practices required ensuring a controlled environment with strictly and clearly defined criteria. Raw system performance, consumption of electrical energy, and programmability were measured.

Examples of the produced testing codes, are available in the appendix, both the hand-tuned code (appendix A) and the generated code appendix(B).

6.1 Methodology

The ground case testing idea is constituted around comparing performance, energy efficiency, and programmability of the hand-tuned code with code generated by Shine compiler from a RISE expression. Multiple benchmarks which demonstrate various aspects of parallelism and heterogeneity were used, every one of them described in section 6.2.

Testing was conducted in a controlled environment with strictly and clearly defined criteria minding the elimination of possible instability effects which can't be easily managed (i.e. cold-start effect). Every measurement was conducted five times: first time to eliminate every potential temporal effect, and every other time to implicitly filter out potential artifacts, both for generated code and hand-tuned code. Collected data were processed by the standard statistic models and tools. The previously described procedure applied both to the hand-tuned code and generated code. Attributes that were measured are raw system performance in the means of active processor cycles, wall clock time of the execution, and consumption of electrical energy. Implicitly, the programmability of the system was measured as well. Collecting five consecutive measurements can be seen on a screenshot of the oscilloscope used to measure the voltage of the GAP8 chip on figure 6.1. Each pulse of the yellow or blue line represents one measurement. A deeper explanation of the measurement collecting procedures will be given further in the text.

Figure 6.1: Five consecutive measurements

Regarding generated code, it is important to mention that even in that case, a small amount of code had to be written manually to stitch the generated code with input and output parameters. That part of code contained by the function *__main*() took care of loading testing data, kicking the computation off on the cluster, and writing results to check the correctness of the solution. Example of such a function is available in appendix B, line 211.

Every measurement was run in the same way, cleaning, building, and finally running the code with the same set of parameters:

- **PMSIS_OS** - Allows selecting a concrete OS that implements PMSIS API, that the solution will be run on. For measurements, PulpOS was used;
- **CONFIG_OPENMP** - Enables OpenMP extension in the compiler;
- **platform** - Selects whether the program will be run on the board or in the simulator (GVSoC);
- **io** - Selects input and output interface of the board, which can be *host*, meaning the communication with the board will be done through the host computer, or *uart* which utilizes UART controller and lines to channel the communication through it.

The complete shell command used is available in listing 6.1.

Listing 6.1: Run command

```
#!/bin/bash
make clean && \
  make \
    PMSIS_OS=pulpos \
    CONFIG_OPENMP=1 \
    platform=board \
```

```
        io=host \
        all && \
    make run
```

6.1.1 Performance measuring

As suggested by the online GAP8 SDK manual [133], performance was primarily measured by counting active processing cycles, which provided insight into the clock-wise performance of the code. Additionally, the wall clock duration of the code execution was measured as well. Both types of measurements were conducted by using performance and clock function APIs implemented by the GAP SDK. Measurements were conducted by wrapping the code that was being observed with functions that started the counters just before the computation began, and stopped them as soon as the computation finished. Code snippet depicting performance measuring is provided in listing 6.2 with line 3 starting the active cycle counter, and 8 stopping the counter.

```
1 pi_perf_conf(1 << PI_PERF_ACTIVE_CYCLES);
2 pi_perf_reset();
3 pi_perf_start();
4 inttime_cycles1 = pi_perf_read(PI_PERF_ACTIVE_CYCLES);
5
6 /** Coderunningonclusterhere */
7
8 pi_perf_stop();
9 inttime_cycles2 = pi_perf_read(PI_PERF_ACTIVE_CYCLES);
```

Listing 6.2: Code snippet which does active cycle counting

Measuring wall clock time was conducted by recording a timestamp prior to starting a computation and just after the computation finished. A function was used that returns the total amount of time that has elapsed since the start of the runtime in microseconds. Invoking of the appropriate function can be seen in listings 6.3 (lines 2 and 2) and 6.4 (lines 2 and 8), for hand-tuned code and generated code respectively, which differ only by the function invoking the computation.

```
1 /* Getcurrentsystemtime */
2 longtime_usec1 = rt_time_get_us();
3
4 /* Sendtasktocluster,blockuntilcompletion */
5 pi_cluster_send_task_to_cl(&cl_device, cl_task);
6
7 longtime_usec2 = rt_time_get_us();
```

Listing 6.3: Wall clock time measurment – Hand-tuned code

```
/* Getcurrentsystemtime        */
longtime_usec1 = rt_time_get_us();

/* Sendtasktoclusterbycallingwrapperfunction
    whichfurtherinvokesruntimelibrary        */
foo_init_run(ctx, out, ROWS, COLS, ROWS, in, in);

longtime_usec2 = rt_time_get_us();
```

Listing 6.4: Wall clock time measurment – Generated code

6.1.2 Measuring energy consumption

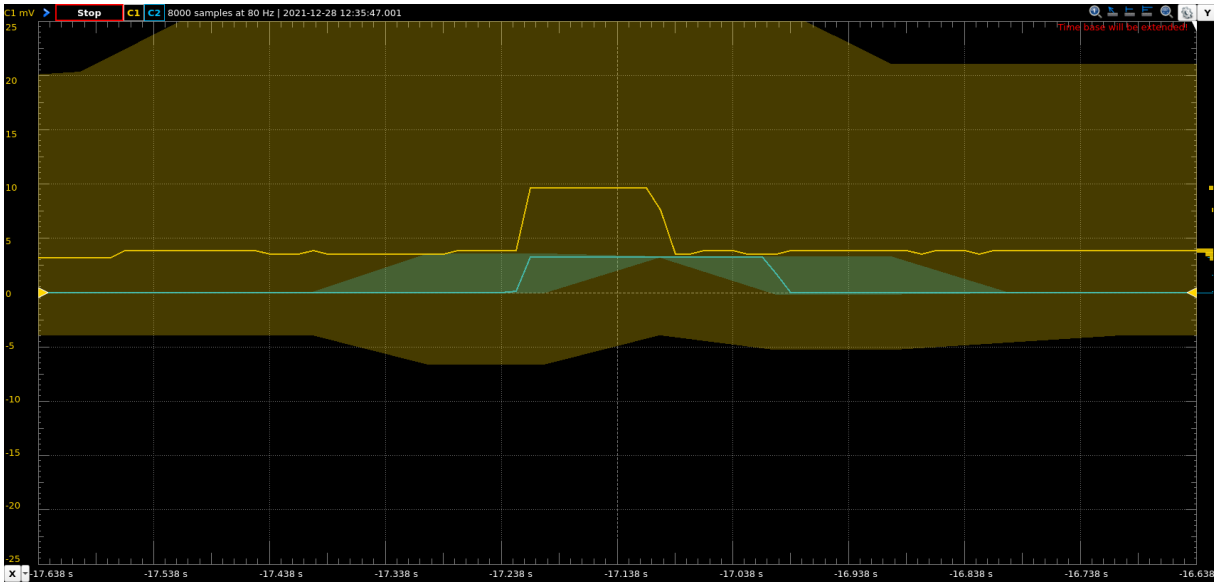
Energy consumption was measured using Digilent Analog Discovery 2, a digital oscilloscope and logic analyzer. There are a couple of reasons for using oscilloscope instead of wattmeter or other similar tools. The first reason is that the energy consumption for the observed computation portions executed on the cluster would not necessarily be uniform, i. e. different parts of the observed computation could consume different amounts of electrical energy, making energy consumption relatively unstable. The second reason for using oscilloscope lies in the fact that the observed computations on the cluster are relatively short, which would make readings fetched from the ordinary voltmeters or wattmeters hard to get, thus making them unstable and unreliable.

An example depicting reasoning for using an oscilloscope is provided in figure 6.2. The blue line in the state of logical 1 represents the ongoing execution of the observed computation, while the yellow line represents the voltage on the differential probes. It is clear that the voltage level and thus energy consumption is not constant during the observed computation, which effectively disables the opportunity to measure a single power surge which could in some cases represent a calculation of interest kicking in. Furthermore, the observed computation is as short as half of a second, making the use of a simple voltmeter of wattmeter impractical and error-prone.

The oscilloscope was connected to the GAPuino testing board with two differential probes, and one line connecting the ground of the oscilloscope and the ground of the board. The first differential probe was connected to connector J11 on the board which exposes the electrical interface to the internal DC/DC regulator solely of the GAP8 chip featured by the board. A 1 Ohm resistor is placed between the testing points which allows for easy measurement of the current flowing in the regulator. Block diagram of the connected testing equipment can be found in figure 6.3.

By utilizing Ohm's law:

Figure 6.2: Single zoomed measurement



$$U = R \times I \tag{6.1}$$

it is easy to calculate the power consumed by the chip:

$$P = U \times I \rightarrow P = U^2 \, [W] \tag{6.2}$$

Given average power of the system with respect to time, or given that the power consumption is constant, consumed energy is then defined by the relation:

$$E_{total} = P_{average} \times \Delta t \, [J] \tag{6.3}$$

where Δt denotes the duration of the chip working, with average power denoted by $P_{average}$ throughout the aforementioned time span. While by using this method one can obtain relatively accurate results, depending on the application itself, it would be even more precise to plot the observed voltage as a function and calculate total energy consumption by calculating the area beneath the power curve with respect to time:

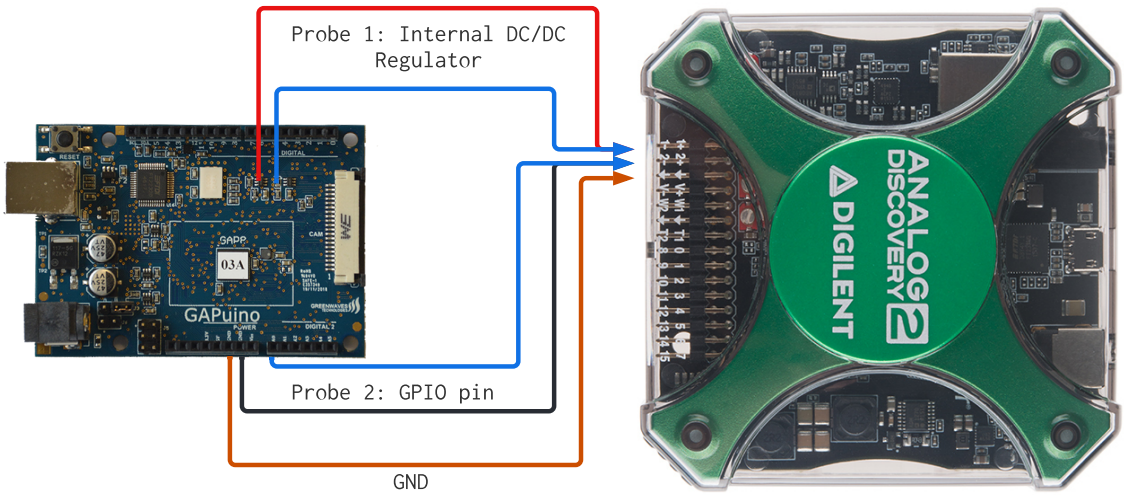
$$E = \int_{t}^{t+\tau} P dt \tag{6.4}$$

The oscilloscope used to measure the voltage on the internal DC/DC regulator was set up

to record measurements with a sampling frequency of 1 kHz, thus producing 1,000 sampled voltages from both probes for each second of each measurement. A subset of voltage samples from the internal DC/DC regulator was then extracted, indicated by samples of voltage recorded on the GPIO pin used to signal the ongoing computation of interest, which will be explained further in the text. The extracted samples were then used to calculate energy consumption in discrete points of time during the calculation, effectively numerically integrating the power function with respect to time:

$$E = \sum_{t}^{t+\tau} (U[t])^2 \times 0.001 \tag{6.5}$$

Figure 6.3: Block diagram of the measurement equipment



To indicate starting and ending point of the calculation for which the energy consumption is measured, the second differential probe of the oscilloscope was connected to one of the GPIO ports of the GAPuino board, namely GPIO12, exposed through connector A3. The observed calculation was then manually wrapped by code snippets which programmatically drove the aforementioned GPIO port to states of logical 1 and logical 0. The setup code for the port is available in listing 6.5.

```
/* Setfunctionoftheboardpintoo GPIO */
pi_pad_set_function(PI_PAD_12_A3_RF_PACTRL0, PI_PAD_12_A3_GPIO_A0_FUNC1);
pi_gpio_e gpio_out_a1 = PI_GPIO_A0_PAD_12_A3;

/* Setpindirection */
pi_gpio_flags_e cfg_flags = PI_GPIO_OUTPUT;

/* Writeconfiguration */
```

```
9   pi_gpio_pin_configure(&gpio_a1, gpio_out_a1, cfg_flags);
10
11  /* Initiallydeassertpinto0          */
12  pi_gpio_pin_write(&gpio_a1, gpio_out_a1, 0);
```

Listing 6.5: GPIO pin setup

Listings 6.6 and 6.7 depict code snippets which wrap the observed computations run on cluster. Snippets are fairly similar and differ only by the function which executes computation on cluster. Setting the GPIO port to logical 1 can be observed on lines 2 and 2, while setting the GPIO port to logical 0 can be observed on lines 8 and 9. Computation is executed by invoking respective functions on lines 5 and 6.

```
1   /* AssertGPIOpin     */
2   pi_gpio_pin_write(&gpio_a1, gpio_out_a1, 1);
3
4   /* Sendtasktocluster,blockuntilcompletion              */
5   pi_cluster_send_task_to_cl(&cl_device, cl_task);
6
7   /* DeassertGPIOpin     */
8   pi_gpio_pin_write(&gpio_a1, gpio_out_a1, 0);
```

Listing 6.6: Computation marking – Hand-tuned code

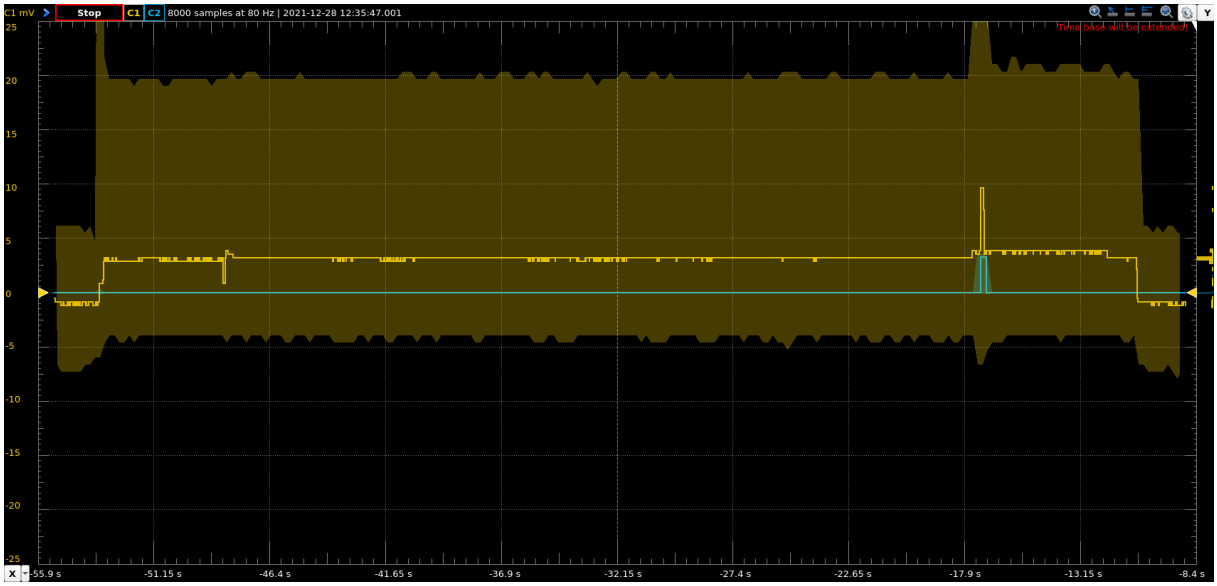
```
1   /* AssertGPIOpin     */
2   pi_gpio_pin_write(&gpio_a1, gpio_out_a1, 1);
3
4   /* Sendtasktoclusterbycallingwrapperfunction
5       whichfurtherinvokesruntimelibrary          */
6   foo_init_run(ctx, out, ROWS, COLS, ROWS, in, in);
7
8   /* DeassertGPIOpin     */
9   pi_gpio_pin_write(&gpio_a1, gpio_out_a1, 0);
```

Listing 6.7: Computation marking – Generated code

Both of the probes were connected and the measurements were conducted as it was suggested by the official GAPuino User's manual [146].

Figure 6.4 focuses on a single measurement and depicts the voltage on the internal DC/DC regulator of the GAP8 chip in various stages of the program execution lifecycle. First, the testing board is disconnected from the power source (1.), then, we observe an obvious surge when the board is connected to the power source (2.). A jitter of voltage (3.) denotes that the program execution started, in this case on fabric controller. Pulse (4.) on the GPIO pin visible on both the blue and yellow line marks the observed computation, and finally, the board is again disconnected from the power source (5.).

Figure 6.4: Single measurement



In real testing scenarios, code snippets that measured wall-clock were interleaved with snippets taking care of rising specified GPIO port to logical 1 and logical 0, as can be seen in appendices A and B. Listings 6.3, 6.4, 6.6, and 6.7 depicting those snippets separately in this section were given for clarity.

6.2 Benchmarks

6.2.1 Matrix multiplication

The first benchmark the proposed setup is tested against is standard matrix-matrix multiplication. Expression given in listing 6.8 introduces 3 size variables, n, m, and o, which represent dimensions of the input matrices and the output matrix. First matrix is of size $n \times o$, second matrix is of size $o \times m$, while the result matrix is of size $n \times m$. Matrix multiplication is defined by lambda expression passed to fun (second pair of parentheses), along with function type (first pair of parentheses). Lambda parameters, which turn out to be matrices that are being multiplied are denoted with a and b. Lambda body is wrapped with $gap8run$ primitive with parameter 8 which denotes that the expression will be run on the cluster on the GAP8 board. The first matrix (a) is first piped through *mapPar* primitive which applies provided function to every row of the first matrix. Analogously, the second matrix is first piped through $transpose$ primitive which is then piped through *mapPar* primitive, applying function body to every column of the second matrix. A row of the first matrix and column of the second matrix are then zipped [RefSlikaZip] through zip primitive, yielding a sequence of pairs of corresponding elements. Every pair is then multiplied and piped through *reduceSeq* primitive which reduces the sequence with operator *add* as reduction function and 0 of type $u32$ as the initial element of the

reduction.

Listing 6.8: Matrix multiplication expression in RISE

```
valexpr: ToBeTyped[Expr]=depFun((n: Nat, m: Nat, o: Nat)=>
  fun((n‘.‘o‘.‘u32) ->:(o‘.‘m‘.‘u32) ->:(n‘.‘m‘.‘u32))(
    (a, b)=>
      gap8Run(8)(
        a |> mapPar(fun(rowa=>
          b |> transpose |> mapPar(fun(colb=>
            zip(rowa)(colb) |>
              map(fun(x=>fst(x) * snd(x))) |>
              reduceSeq(add)(cast(l(0)) :: u32)
          ))
        ))
      )
  )
)
```

Benchmark was executed by multiplying 2 square matrices of size 250×250 elements.

6.2.2 Sobel filter

The second selected benchmark selected for benchmarking purposes is Sobel filter [147]. Sobel filter, named after its creator Irwin Sobel with the unfair omission of its second creator Gary Feldman, is an image filtering algorithm used in edge detection which, by separably convolving two small matrices of sizes 3×3 over the input image in the horizontal and vertical direction, resulting in an output image with emphasized edges of the objects on picture. Edges are parts of the picture which relatively differ in intensity one from another. The algorithm is easily parallelizable and cost-effective which makes it widely popular in image processing pipelines, especially in computer vision, object detection, and object recognition applications.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Listing 6.9 contains the Sobel filtering expression written in RISE. Expression introduces 2 size variables n, and m which represent dimensions of the input picture. Type of the expression denotes that it accepts one matrix of size $n \times m$ and two matrices of size 3×3. The resulting

matrix representing the output image from the algorithm is of size $n \times m$ as well. Input matrices map to lambda parameters *pic*, h_w, and, v_w, representing input image, horizontal filter, and vertical filter respectively. Expression is first wrapped in *gap8Run* pattern indicating that it will be run on the cluster. The input image is first piped through *padCst2D* primitive which, as specified by the input parameters to the primitive, pads the image on every edge with zeros. Next, padded image is piped through *slide2D* primitive which creates a matrix of neighbourhoods of each element of the input image. The neighbourhood of an element is a submatrix of the input matrix and is defined as all of the elements that surround the element currently being processed. The neighbourhood matrix is thus a 3×3 matrix with the element that is currently being processed in the center. The matrix of neighbourhoods is then mapped twice, one time for each dimension. First, the input matrix which is at this point a neighbourhood of a single element, represented by lambda parameter *submat* and filter represented by h_w and v_w are piped through pattern *join*, which joins a two-dimensional array and turns it into a one-dimensional array. Joined arrays are then effectively multiplied using dot product, by being zipped with *zip* pattern, elementwise multiplied and then reduced with *add* operator and 0 as the initial accumulator. Finally, the geometric average is calculated from corresponding elements both from the horizontal and vertical components. Since GAP8 doesn't implement a standard math library, a custom embedded-appropriate implementation is provided in form of a *foreignFun* pattern which is then integrated into expression.

$$\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & a_1 & b_1 & 0 \\ 0 & a_2 & b_2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & a_1 & b_1 \\ 0 & a_2 & b_2 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ a_1 & b_1 & 0 \\ a_2 & b_2 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & a_1 & b_1 \\ 0 & a_2 & b_2 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_1 & b_1 & 0 \\ a_2 & b_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix} \rightarrow \tag{6.6}$$

$$\begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 & a_1 & b_1 & 0 & a_2 & b_2 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & a_1 & b_1 & 0 & a_2 & b_2 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & a_1 & b_1 & 0 & a_2 & b_2 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} a_1 & b_1 & 0 & a_2 & b_2 & 0 & 0 & 0 & 0 \end{bmatrix} \end{bmatrix} \tag{6.7}$$

Listing 6.9: Sobel filter expression in RISE

```
valgapSqrt=foreignFun("gap_sqrt",
  Seq("a_nInput"),
  """
    |{
    |uint32_top=a_nInput;
    |uint32_tres=0;
    |
    |uint32_tone=1uL<<30;
    |while(one>op){
    |one>>=2;
    |}
    |while(one!=0){
    |if(op>=res+one){
    |op=op-(res+one);
    |res=res+2*one;
    |}
    |res>>=1;
    |one>>=2;
    |}
    |returnres;
    |}
    |""".stripMargin,
  u32 ->:u32
)

valexpr: ToBeTyped[Rise]=depFun((n: Nat, m: Nat)=>
  fun((n`.`m`.`u8) ->:(3`.`3`.`int) ->:(3`.`3`.`int) ->:(n`.`m`.`u8))(
    (pic, h_w, v_w)=>
      gap8Run(8)(
        pic |>
          padCst2D(1, 1)(cast(l(0)) :: u8) |>
          slide2D(sz=3, st=1) |>
          mapPar(mapPar(fun(submat=>{
          zip(submat |> join)(h_w |> join) |>
            map(fun(x=>(cast(fst(x)) :: u32) * cast(snd(x)) :: u32)) |>
              reduceSeqUnroll(add)(cast(l(0)) :: u32) |>
                letf(h=>
                  zip(submat |> join)(v_w |> join) |>
                    map(fun(x=>
                      (cast(fst(x)) :: u32) * cast(snd(x)) :: u32)) |>
                        reduceSeqUnroll(add)(cast(l(0)) :: u32) |>
                          letf(v=>
                            cast(gapSqrt(h * h + v * v)) :: u8
                          )
                )
            }))
      )
  )
)
```

6.2.3 k-means clustering

k-means clustering is an algorithm first described by John A. Hartigan [148]. Given m points in generalized n-dimensional space, the algorithm aims to partition given points into k clusters so that the within-cluster sum of squares is minimized [149]. The algorithm initially selects k centroids, one for every cluster, which can be done randomly within boundaries of the data being partitioned, or by using different strategies, e. g. average or mean of points in the initial set. Each of m points is then associated with a cluster defined by the centroid the point is closest to. After the association of every point with one of the clusters, centroids are recalculated so that the square distance to each of the points in the cluster is minimized. The process repeats for a predefined number of iterations until the algorithm eventually converges.

An expression of the algorithm in RISE, adapted from the RISE's GitHub repository, is provided in listing 6.10. Since the expression itself is relatively complex it defines a few auxiliary functions. Function *update* accepts an integer of type *u32*, a pair of integers of the same type, and produces again an integer of the same type. The semantics of the function within the algorithm is to *update* the distance between a point in 2-dimensional space and a centroid. Function *select* accepts a tuple whose second element is again a tuple and returns the second individual element of that tuple. Function *testF* is defined as a *foreignFun*, meaning that the body of the function is given in a programming model that is native for the target platform. The function tests how the distance to a centroid provided by the first parameter to it compares to the minimal distance to any centroids so far found. Minimal distance is provided as the first element of a tuple passed as a second parameter to the function. The second element of that tuple is again a tuple consisting of an index of the centroid currently being tested and an index of a centroid for which the distance is currently at the minimum. It is important to mention that record construction on returning from the function on lines 8 and 12 directly mimics type of the function, i. e. *uint32_t* relates to *u32* in the expression defining the type of the function. This *de facto* ties function to a concrete implementation regarding type and should probably be avoided.

The core expression starting on line 26 defines three size variables, p as number of points, c as number of clusters, and f as number of features or dimensions of each point. The expression generates a function accepting a matrix of size $f \times p$, a matrix of size $c \times f$, and returns an array of p elements containing element-wise index of the centroid for every point in the array. The underlying lambda defines two lambda variables, *features*, and *clusters*. The body of the lambda is wrapped with *gap8run* pattern, indicating that the computation should be run on cluster. First, *features* is piped through a *mapPar* pattern which maps a function on each oelement of the collection represented by *features*. Second, *clusters* is piped through *reduceSeq* pattern, reducing a collection with the underyling function and a pair provided as an initial element for reduction operation (line 35). The reduction function further decomposes *clusters* collection, zips each *feature* with a *cluster* and reduces the resulting collection with *update* as a reduction

function and 0 as an initial element effectively calculating distance (line 32). Distance structure is then passed to the *testF* function with the tuple consisting of up to point in time calculated minimal distance, indexe of the centroid currently closest to the point for which the distance is being calculated, and index of the centroid for which the distance is currently being calculated. The reduction operation is finally piped through previously described *select* auxiliary function.

Listing 6.10: k-means clustering expression in RISE[1]

```
valtestF=foreignFun("test",
  Seq("dist","tuple"),
  """{
    |uint32_tmin_dist=tuple._fst;
    |uint32_ti=tuple._snd._fst;
    |uint32_tindex=tuple._snd._snd;
    |if(dist<min_dist){
    |return(structRecord_uint32_t__uint32_t_uint32_t_){
    |dist,{i+1,i}
    |};
    |}else{
    |return(structRecord_uint32_t__uint32_t_uint32_t_){
    |min_dist,{i+1,index}
    |};
    |}
  }""".stripMargin,
  u32 ->:(u32 x (u32 x u32)) ->:(u32 x (u32 x u32))
)

valupdate=fun(u32 ->:(u32 x u32) ->:u32)((dist, pair)=>
  dist + (pair._1 - pair._2) * (pair._1 - pair._2)
)

valselect=fun(tuple=>tuple._2._2)

valexpr: ToBeTyped[Rise]=depFun((p: Nat, c: Nat, f: Nat)=>
  fun((p`.`f`.`u32) ->:(c`.`f`.`u32) ->:(p`.`u32))(
    (features, clusters)=>
      gap8Run(8)(
        features |> mapPar(fun(feature=>
          clusters |> reduceSeq(fun(tuple=>fun(cluster=>{
            valdist=zip(feature)(cluster) |>
              reduceSeq(update)(cast(l(0)) :: u32)
```

```
          testF(dist)(tuple)
        }))(
          makePair(cast(l(4294967295L)) :: u32)
            (makePair(cast(l(0)) :: u32)(cast(l(0)) :: u32))
        ) |> select
      ))
    )
  )
)
```

6.2.4 Convolution

Convolution is one of the most common yet non-primitive operations in the domain of signal processing. That being said, together with the existence of the specialized hardware convolution engine, merits for a specialized benchmark that would test the performance and energy efficiency both for the convolution executed on the RISC-V processing cores in the cluster, and for the convolution executed on the specialized hardware convolution engine.

Expression describing convolution with 3×3 filter in RISE is available in listing 6.11.

Listing 6.11: Convolution with 3 X 3 filter in RISE

```
valexpr: ToBeTyped[Rise]={
  depFun((w:Nat, h: Nat)=>
    fun((w‘.‘h‘.‘i16) ->:(3‘.‘3‘.‘i16) ->:((w - 2)‘.‘(h - 2)‘.‘i16))(
      (in, filter)=>
        gap8Run(8)(
          in |>
            slide2D(3, 1) |>
            mapPar(mapPar(fun(sub=>{
              zip(sub |> join)(filter |> join) |>
                map(fun(x=>fst(x) * snd(x))) |>
                reduceSeq(add)(li16(0))
            }))
        )
    )
  )
}
```

[1]Adapted from https://github.com/rise-lang/shine/blob/main/src/main/scala/apps/kmeans.scala

The expression declares two size variables, w and h, which represent the width and height of the input matrix respectively. The type of the expression is:

$$(w \times h) : i16, (3 \times 3) : i16 \rightarrow ((w-2) \times (h-2)) : i16 \tag{6.8}$$

which means that it accepts a matrix of size $w \times h$, a matrix of size 3×3, and produces a matrix of size $((w-2) \times (h-2))$. All of the matrices contain elements of 16-bit signed integers.

The underlying lambda has 2 parameters *in* and $filter$ which correspond to the input matrices. The expression is first wrapped with $gap8Run$ primitive, indicating that the underlying code has to be run on the cluster. In the context of this expression, this is particularly important, because of the transformation that, once applied, will attempt to produce code that utilizes the hardware convolution engine which resides in the cluster. Attempting such a transformation with an expression set to be run on the fabric controller would not yield code utilizing that engine.

Input matrix is first piped through a $slide2D$ pattern with parameters $slide = 3, size = 1$, which creates a matrix of neighbourhoods of size 3×3. The matrix of neighbourhoods is then mapped over a function with a pair of *mapPar* patterns that first join both the input matrix and the filter matrix and then zip them. The zipped array of input and filter matrices is then mapped over a function that multiplies respective elements just before reducing the array with *add* operator with the initial element being 0.

The expression describing convolution in listing 6.11, when translated can be run on the cluster of the GAP8 chip. However, that expression won't utilize hardware convolution engine *per se*, regardless of its existence. To utilize the engine, the series of patterns in the provided expression have to be transformed to a pattern that will indicate the usage of the hardware convolution engine. That pattern will further in compilation be translated to the concrete invocation of the engine. The optimization strategy which attempts to transform a series of patterns constituting convolution into one single pattern which indicates usage of the convolution engine is available in 5.11. The attempt of applying the optimization strategy can be observed on line 4 in listing 6.12.

Listing 6.12: Applying GAP8-specific convolution transformation

```
1 valconv: Strategy[Rise]=
2   (gap8hwConvMerge ‘@‘ everywhere)
3
4 vallowExpr=conv(exprOnAcc).get
5 valmodule=util.gen.gap8.hosted.fromExpr(lowExpr)
6 valcode=GAP8.Module.translateToString(module)
```

After the attempt of applying the optimization strategy, expression is further translated to the native programming model for the GAP8 chip. The expression after application of the optimization strategy somewhat resembles the body of the lambda on line 7 in listing 6.13 since the strategy transforms a series of patterns that constitute convolution operation with the pattern that utilizes hardware convolution engine.

Listing 6.13: Utilizing gap8hwConv3x3 primitive directly

```
/**
*HWCE_ProcessOneTile3x3_MultiOut(e1,output,NULL,NULL,e2,0,n,m,0x7)
**/
valexpr: ToBeTyped[Rise]={
  fun((w‘.‘h‘.‘i16) ->:(3‘.‘3‘.‘i16) ->:((w - 2)‘.‘(h - 2)‘.‘i16))(
    (in, filter)=>
      gap8Run(8)(gap8hwConv3x3(0)(in)(filter))
  )
}
```

This also implies that the hardware convolution engine can be utilized directly, though that to some extent violates the idea of the encapsulation of the underlying hardware from the programmer, which is the main reason why programmers, i. e. domain scientists should opt-in using RISE or similar domain-specific language. Furthermore, describing computations from high-level patterns opens a relatively large exploration space for future optimizations which could be performed without the programmer's explicit knowledge, yielding even higher performance and energy efficiency.

The aforementioned expressions can be slightly modified to support convolution with filters of other sizes. More concretely, one has to change the type of the expression, parameters of the *slide2D* primitive in case of the manually written RISE expression (listing 6.12), or the concrete primitive in case of the directly-invoked convolution (listing 6.13).

•Filter of size 5 $\times$ 5 Type of the expression should be:

$$(w \times h) : i16, (5 \times 5) : i16 \rightarrow ((w-4) \times (h-4)) : i16 \tag{6.9}$$

Parameters of the *slide2D* should be $size = 5, step = 1$ in case of a manually-written RISE expression or *gap8hwConv5x5* in case of a direct invocation.

•Filter of size 7 $\times$ 7 The type of the expression should be:

$$(w \times h) : i16, (7 \times 7) : i16 \rightarrow ((w-6) \times (h-6)) : i16 \tag{6.10}$$

Parameters of the *slide2D* should be $size = 7, step = 1$ in case of a manually-written RISE

expression or $gap8hwConv7x7$ in case of a direct invocation.

•Filter of size 7 $\times 4$ The type of the expression should be:

$$(w \times h) : i16, (7 \times 4) : i16 \rightarrow ((w-6) \times (h-3)) : i16 \tag{6.11}$$

Parameters of the $slide2D$ should be $size_w = 7, step = 1, size_h = 4, step = 1$ in case of a manually-written RISE expression or $gap8hwConv5x5$ in case of a direct invocation.

Native code that is translated from the expressions is available in the appendix B.2.

6.3 Evaluation

6.3.1 Performance evaluation

As mentioned previously in the text, performance is measured in active cycles of the processing cores, i. e. cycles during which the core was active, excluding sleeping, etc., and in wall-clock time. The results for benchmarks, excluding convolution as a special case, are given in table 6.1. Data in columns *Hand-tuned code* and *Generated code* are given in the raw number of active cycles measured for hand-tuned and generated code respectively, in terms of average and standard deviation of a sample. Column *Diff* provides the difference between hand-tuned and generated code, from the perspective of the generated code. If a result is negative, it means that the generated code is more performant, and vice versa. *Gain* is given in percentages, again from the perspective of the generated code, but with the inverted sign. A positive percentage means that there is a gain in performance compared to the hand-tuned code. The same data is plotted for clarity on figure 6.5. Each bar group represents one benchmark, with green shaded bars representing hand-tuned code, and blue shaded bars representing generated code.

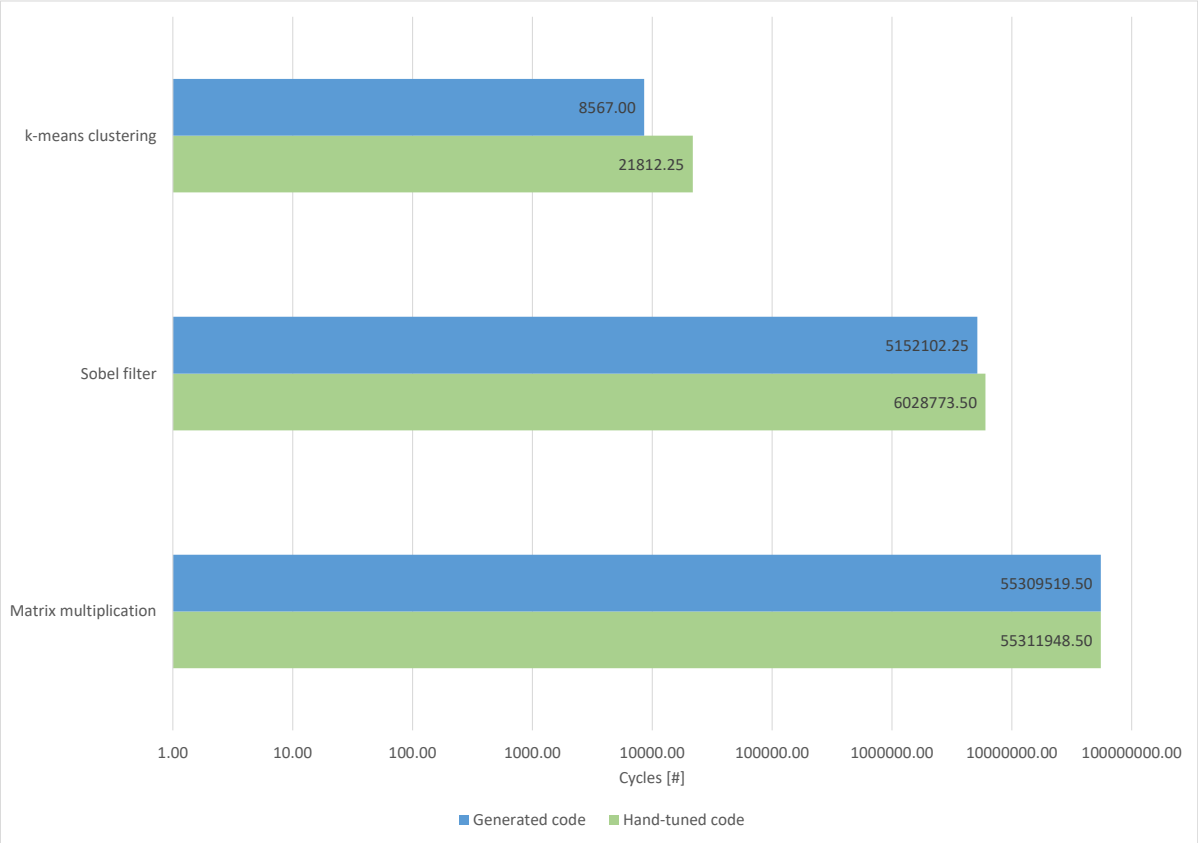
Table 6.1: Clock cycle-wise performance evaluation results

	Hand-tuned code [#]	Generated code [#]	Diff [#]	Gain [%]
Matrix multiplication	55311948.50 ± 72120.86	55309519.50 ± 15632.25	-2429	+0.0044
Sobel filter	6028773.5 ± 1830.44	5152102.25 ± 4594.54	-876671.25	+14.5415
k-means clustering	21812.25 ± 26.02	8567.00 ± 76.00	-13245.25	+60.7240

Clock cycle-wise performance analysis suggests promising results. Gains can be observed regarding all three benchmarks, with matrix multiplication benchmark being on par, and huge

gains regarding Sobel filter and k-means clustering benchmarks, especially the latter. The results here should be taken *cum granulo salis* though, as the increase in performance greater than 60% certainly draws suspicion. Such results, and results in an increase regarding active cycles, could suggest that processing cores were better exploited, i. e. that their duty cycle was higher but that the duration of the computation did not necessarily become shorter.

Figure 6.5: Cycle-wise performance comparison



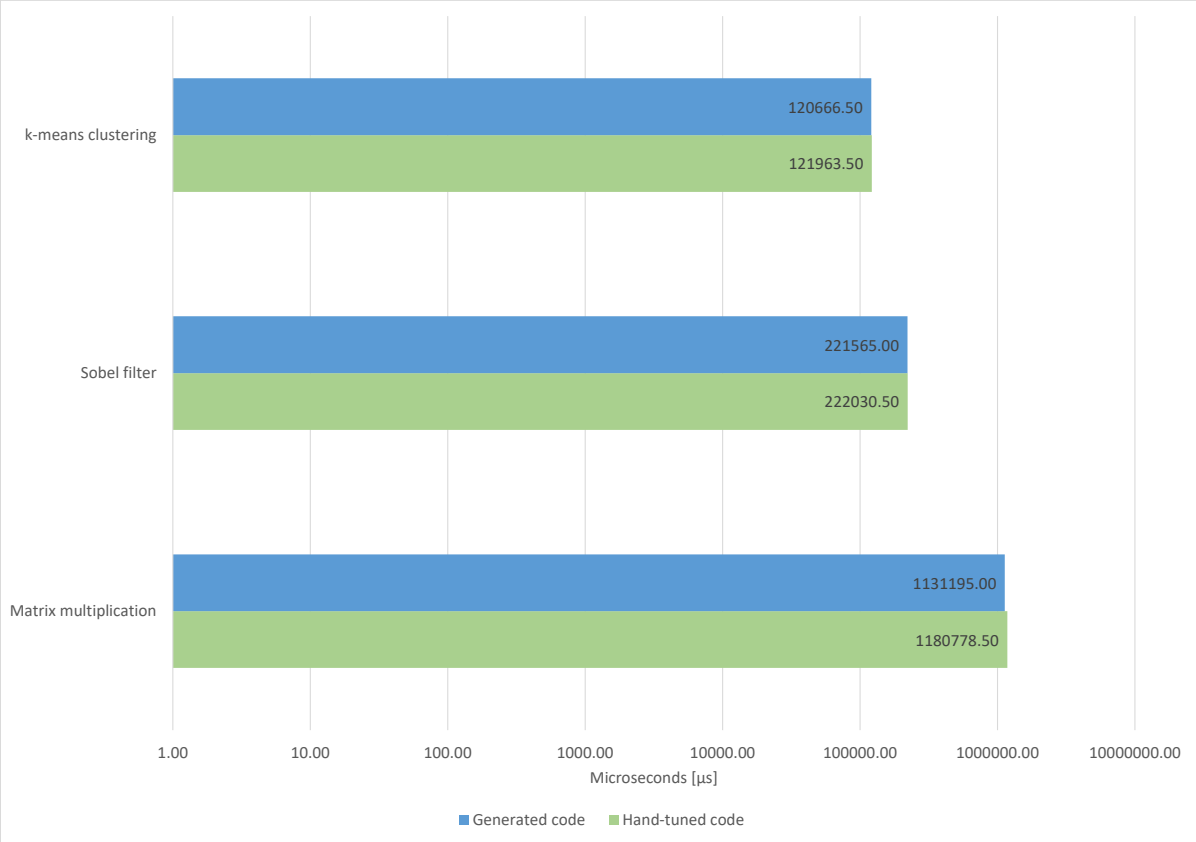
When it comes to measuring wall-clock time, the results are provided in table 6.2. This time columns *Hand-tuned code* and *Generatedcode* provide average and standard deviation of sample for wall-clock time measured in μS, for hand-tuned code and generated code respectively. *Diff* provides the difference in time, again in μS from the perspective of generated code, negative result implying that generated code executed in a shorter amount of time. *Gain* provides a relative percentage increase, but again with an inverted sign, which means that a positive result indicates shorter execution of time of the generated code. Data are plotted for clarity on figure 6.6, with the same representation scheme as on the previous figure.

The results again show the increase in performance, observed through shorter execution time for all of the benchmarks, with matrix multiplication gaining the most, followed by k-means clustering, and at last Sobel filter. Gains achieved for generated code range from 0.2% to 4.2% which cannot be considered a significant gain, but is at least on-par with respect to hand-tuned code.

Table 6.2: Wall-clock time performance evaluation results

	Hand-tuned code [μS]	Generated code [μS]	Diff [μS]	Gain [%]
Matrix multiplication	1180778.50 $\pm$ 57945.11	1131195.00 $\pm$ 1330.48	-49583.50	+4.1992
Sobel filter	222030.50 $\pm$ 1884.54	221565.00 $\pm$ 3773.39	-465.50	+0.2097
k-means clustering	121963.50 $\pm$ 2197.50	120666.50 $\pm$ 2680.05	-1297.00	+1.0634

Figure 6.6: Wall-clock time performance comparison



6.3.2 Energy efficiency evaluation

The energy efficiency evaluation was conducted by continuously measuring the power consumption of the platform while the code was being executed. One can argue that energy consumption can be derived from performance, because code that executes faster, usually stresses the platform for a shorter period of time, thus consuming more energy but in a shorter period of time. Nevertheless, the energy consumption was measured as one of the main topics that this thesis covered is energy-efficient computing. Furthermore, a computation can consume energy less-efficient than the analogous computation while yielding the same result.

Energy consumption and implicitly energy efficiency were measured in two different ways. As previously described in the text, the most accurate way of measuring energy consumption is by calculating the area below the plotted function of power with respect to time. Since the measurement tool, i. e. oscilloscope, provided enough data with a satisfying resolution, by utilizing relation 6.5, numerical integration was performed to calculate energy consumption for each benchmark. The results are provided in table 6.3 and for clarity on figure 6.7. Columns *Hand-tuned code* and *Generated code* present the average and standard deviation of a sample of measured energy consumption in Joules. *Diff* presents the difference between measurements from the perspective of the generated code, negative result meaning that generated code consumed less energy. *Gain* presents a difference in percentages, again from the perspective of the generated code, but with an inverted sign to intuitively hint at the spirit of the result. Positive gain implies less energy consumed and vice versa.

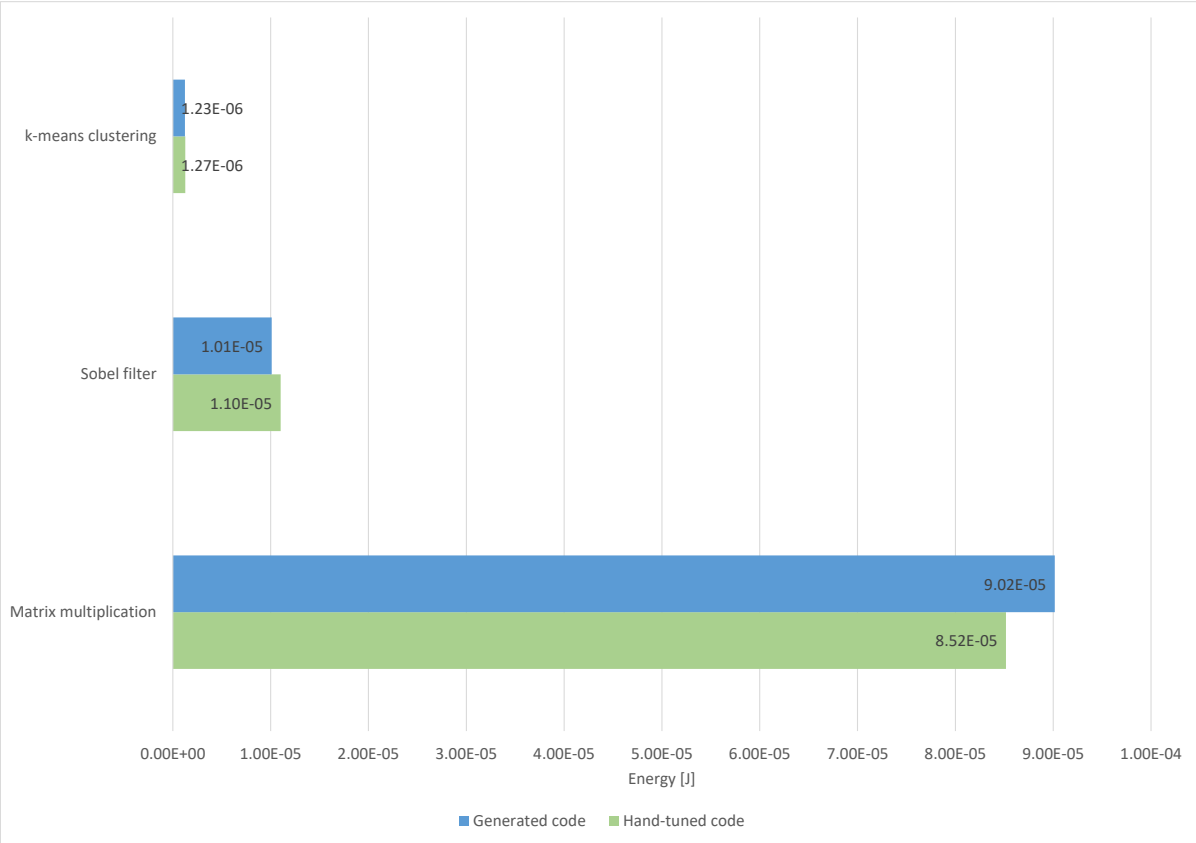
Table 6.3: Energy efficiency evaluation - Numerical integration

	Hand-tuned code [J]	Generated code [J]	Diff [J]	Gain [%]
Matrix multiplication	8.52×10^{-5} $\pm\ 2.77\times10^{-6}$	9.02×10^{-5} $\pm\ 2.24\times10^{-6}$	4.98×10^{-6}	-5.85
Sobel filter	1.10×10^{-5} $\pm\ 2.01\times10^{-7}$	1.01×10^{-5} $\pm\ 4.36\times10^{-8}$	-9.20×10^{-7}	+8.35
k-means clustering	1.27×10^{-6} $\pm\ 2.51\times10^{-8}$	1.23×10^{-6} $\pm\ 2.52\times10^{-8}$	-3.56×10^{-8}	+2.81

Results show gains in energy efficiency for Sobel filter and k-means clustering benchmark and loss for matrix multiplication benchmark. This interestingly depicts how performance is not necessarily tied to energy consumption, as an increase has been observed with respect to performance (section 6.3.1) for all of the benchmarks. While this cannot be clearly explained at this point in time, the probable cause might be due to the worse utilization of processing

cores which yielded higher power consumption for generated code, although it was executed in a shorter period of time.

Figure 6.7: Energy consumption comparison (Numeric integration)



The second method by which the energy consumption was measured is deriving it from the average power in a time period, as described by relation 6.3 in the previous section describing the methodology. Again, samples collected from the probes of the oscilloscope were used to extract a subset of those samples which represented measurements of the voltage of the internal DC/DC regulator during the execution of computation of interest. An average of those samples was calculated, and that average was multiplied by the duration of the timeframe in which the samples were collected. The resulting calculations are available in table 6.4, again as an average with a standard deviation of a sample for *Hand-tuned code* and *Generated code*, with *Diff* showing the difference in energy consumption from the perspective of generated code, and with *Gain* showing a difference in percentage, positive result meaning that there is an increase in energy efficiency and vice versa. Results are for clarity available on figure 6.8 as well.

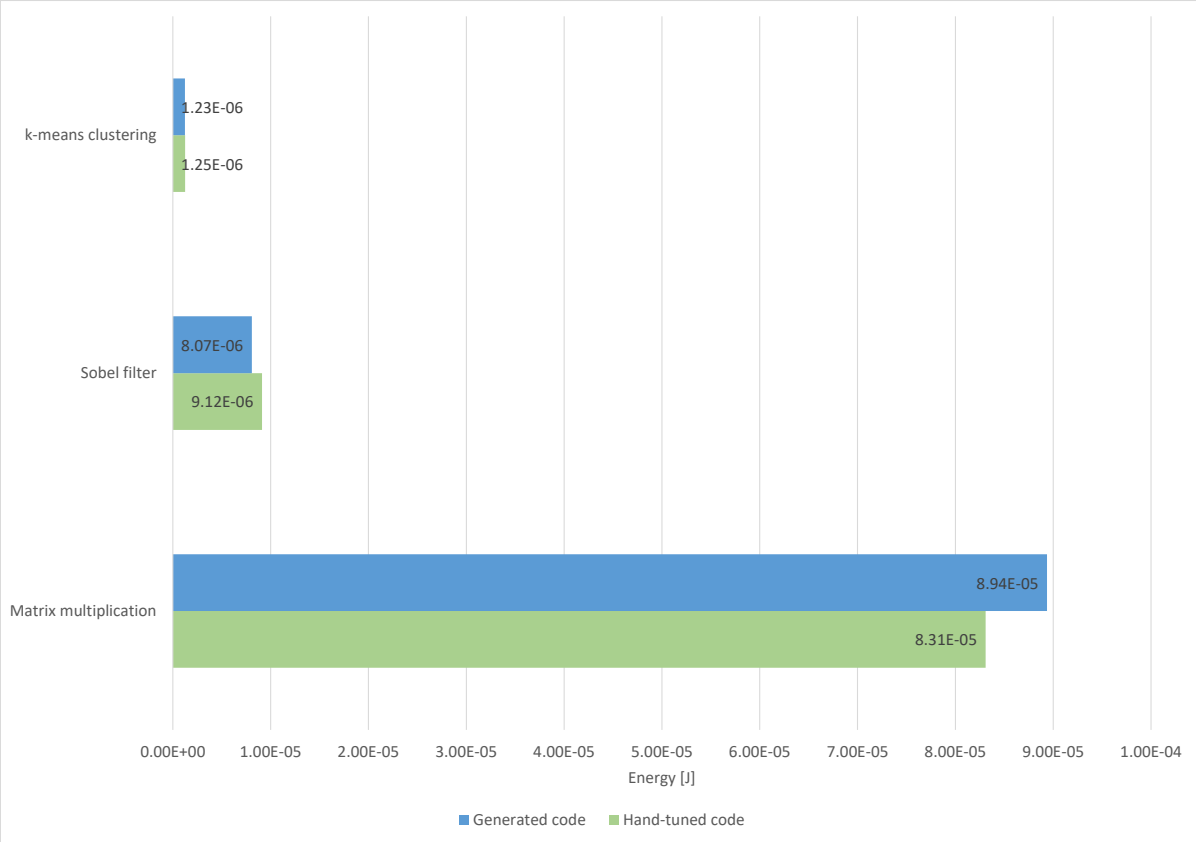
As can be seen in the corresponding table (6.4) and figure (6.8), matrix multiplication benchmarks show some loss in energy efficiency, while the rest two benchmarks show positive results, i. e. increases in energy efficiency. These results correspond to the previously depicted ones obtained by numerically integrating power function with respect to time, but with some deviations. Deviations could be explained by potential variations in the voltage measurement which

Table 6.4: Energy efficiency evaluation – Average voltage and wall-clock time

	Hand-tuned code [J]	Generated code [J]	Diff [J]	Gain [%]
Matrix multiplication	8.31×10^{-5} $\pm\ 2.41\times10^{-6}$	8.94×10^{-5} $\pm\ 2.25\times10^{-6}$	6.28×10^{-6}	-7.5604
Sobel filter	9.12×10^{-6} $\pm\ 1.56\times10^{-7}$	8.07×10^{-6} $\pm\ 2.17\times10^{-8}$	-1.04×10^{-6}	+11.4582
k-means clustering	1.25×10^{-6} $\pm\ 2.21\times10^{-8}$	1.23×10^{-6} $\pm\ 2.60\times10^{-8}$	-2.60×10^{-8}	+2.0739

biased the average of the measurements slightly.

Figure 6.8: Energy consumption comparison (Average with wall-clock time)



Regarding energy efficiency evaluation, there is one perspective point for future work. If a computation is run on a general-purpose controller or on one core in the cluster, it might take longer to compute, but would definitely consume less energy. On the other hand, if a computation is run on the whole system, it will definitely be computed in less time, but consume more energy. This poses a solid ground for optimization of the energy consumption regarding the application being run on the system, and the context the system is placed in.

6.3.3 Programability evaluation

Programmability evaluation poses an ungrateful task, as evaluating the programmability of the programming model, often implies a relatively high amount of subjectivity. Some metrics exist but are rarely used in practice. One of the most naive metrics is lines of code. Although that metric by itself cannot efficiently grade a programming model, as it is heavily influenced by the programmer's style, including bracketing and spacing rules, it can show what the model is capable of with respect to program brevity and conciseness. Table 6.5 provides lines of code for the each benchmark used for evaluation purposes. Each column contains lines of code counted for the respective benchmark.

Table 6.5: Naive comparison of programming models with respect to lines of code

	Hand-tuned code	Generated code	RISE expression
Matrix multiplication	114	166	13
Sobel filter	214	233	43
k-means clustering	176	220	34

Again, for clarity, data is presented on figure 6.9 as well. Each bar group represents one benchmark. Green shaded bars represent lines of code of hand-tuned code, blue bars represent the same for generated code, and at last, grey-shaded bars represent lines of code of analogous RISE expressions.

The results are expected, benchmarking applications expressed in RISE are up to an order of magnitude shorter than the hand-tuned code. Generated code is somewhat longer than the hand-tuned code, but this observation can be disregarded as the generated code itself in real applications can be observed just as another step in the compilation process.

When it comes to benchmarking utilization of HWCE but in terms of programmability, some of the scenarios were analyzed. Table 6.6 provides lines of code for hand-tuned code that utilizes HWCE for convolution (*Hand-tuned code with HWCE*), for the generated code that performs convolution but without utilizing HWCE (*Generated code without HWCE*), for RISE program expressing convolution by direct utilization of the HWCE through the appropriate primitives (*With direct usage of HWCE*), and for RISE program expressing convolution by a series of patterns that could later be transformed to convolution-utilizing primitives (*Witout direct usage of HWCE*).

The data presented is clear. Either direct or indirect utilization of HWCE in RISE yields an order of magnitude fewer lines of code than manual utilization. Generated code again yields a larger program, which can objectively be disregarded. Indirect usage of HWCE from RISE clearly contains more lines of code which are expected, but not that much important as the

Figure 6.9: Total lines of code comparison

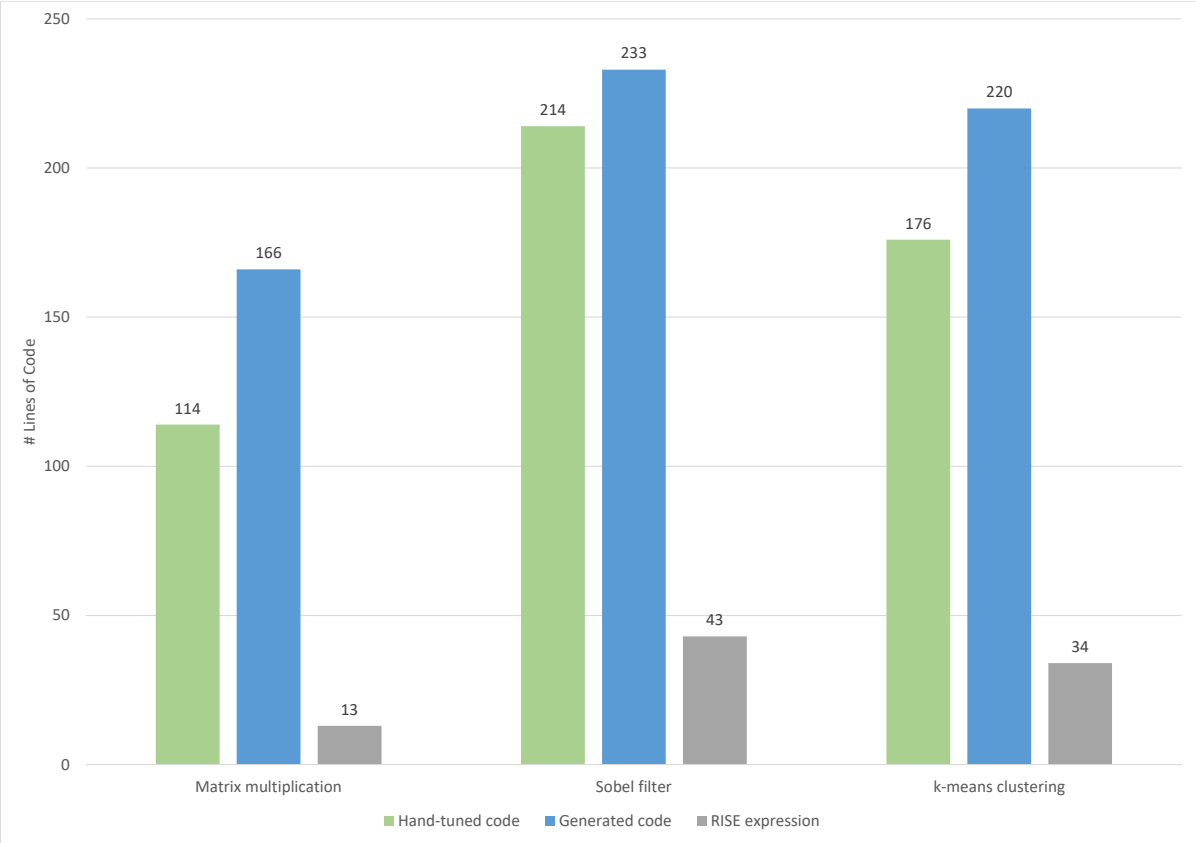


Table 6.6: Naive comparison - Convolution benchmark

	Native programming model		RISE	
	Hand-tuned code with HWCE	Generated code without HWCE	With direct usage of HWCE	Without direct usage of HWCE
Convolution	120	186	9	17

absolute difference between the number of lines of code between direct usage of HWCE and without direct usage of HWCE is relatively small.

Chapter 7

Conclusion

The constant increase of the need for computing power can be easily perceived in the current temporal moment. That need is driven by the tremendous increase of the aggregated data regarding physical simulations, climate models, and other similar application domains, further followed by big data processing and brain-inspired computing. Everything mentioned drives the need to enter the so-called exascale domain. The exascale domain is a domain of computing processing power in which there would exist supercomputers with processing power on the scale of exaFLOPS. The importance of reaching that domain is stressed out by the existence of multiple projects in well technologically-developed countries with the ultimate goal to develop an exascale machine in near future. There are some allegations that ceratin projects already achieved that goal. Regarding reaching the exascale domain, one of the key points is heterogeneity. Heterogeneity, when considered in the context of computing systems implies that such system or processing node consists of dissimilar or diverse components, i. e. processing units of different purposes. Heterogeneous systems in most cases imply systems containing a general-purpose processing core, coupled with a non-general purpose accelerator. An accelerator can be a general-purpose graphics processing unit that can achieve high throughput in intensive data-parallel applications, or an accelerator for a specific domain, e. g. cryptographic processing or matrix calculations. When considering the latter type of accelerators, such accelerators usually achieve the best results regarding performance and energy efficiency, but with a cost of being unusable in other domains. Given everything stated, there exists a large gap for customizable accelerators which would balance between achieving high performance and energy efficiency in the application domain they are accelerating, and providing an interface for customization of themselves to some extent.

However, heterogeneity itself does not guarantee an increase in performance or energy efficiency. Heterogeneity has to be appropriately and efficiently exploited, together with parallelism on every level, which is inherently hard. This thesis tackled the problem of the programmability of complex heterogeneous systems with customizable accelerators. The exist-

ing models usually rely on imperative programming paradigm which approaches algorithm decomposition with respect to *how* calculations are performed. On the other hand, the proposed model approaches problem decomposition with respect to *what* is being calculated, thus offering domain scientists who usually exploit massively available computing power embodied in heterogeneous computing systems, a cleaner, simpler, and more concise programming model. Furthermore, the conventional programming approach usually requires a deep understanding of the underlying hardware's architecture, explicit memory manipulation, memory synchronization, etc. Another benefit of the proposed model, mirroring the original idea of the approach in this thesis, is an abstraction of the underlying hardware and peculiarities implied by the highly parallel and heterogeneous systems.

This thesis aimed to fulfill the expected scientific contribution consisting of two parts:

1. Programming model for heterogeneous systems with customizable accelerators based on a domain-specific language
2. Algorithms for accelerator customization based on program features for performance and energy efficiency optimization

The first part of the thesis is completely fulfilled through extensions of the infrastructure based on the RISE language for supporting heterogeneous systems with customizable accelerators. However, the second part of the expected scientific contribution was heavily influenced by the inexistence of the anticipated customizable accelerators, which would expose tunable parameters. Accelerators and general-purpose controllers usually can tune frequency and voltage, but those parameters cannot be fully considered customizations. Instead of algorithms for accelerator customizations, support for the specialized part of the chip, i. e. hardware convolution engine was added, serving partly as a showcase of the principle of supporting specialized hardware by providing transformations for a series of patterns that constitute operations that that hardware supports. Infrastructure around RISE language then applies provided transformations that offloads computation to the underlying hardware without explicit knowledge or invocation of the programmer. Furthermore, extensive benchmarks and evaluations of the proposed solution were provided.

The proposed solution was compared to the native programming model for the chosen hardware platform in terms of performance, energy efficiency, and programmability. Carefully hand-tuned code was compared with code generated by the Shine compiler from the expressions describing the same benchmarks in RISE. It is experimentally shown that it is possible to achieve performance and energy efficiency of the generated code that is on par or better than hand-tuned code, while at the same time providing a programming model that is cleaner, simpler, and more concise than the conventional or native model.

Regarding future work in the field of this thesis, multiple prospective opportunities are available. First, there is an objective need to investigate into memory systems of heterogeneous

systems and their hierarchy, minding latencies, and speeds of particular parts of the memory subsystem. The current approach relies on the fact that there exists a memory that is accessible both by the general-purpose controller and the accelerator. While this is true for the platform targeted in this thesis, that might not be the case in general. Furthermore, if a system contains multiple types of memories available only to disjunct parts of the system, there is a high probability that memory available only to one part of the system will be faster for that respective part of the system. Therefore, the proposed model could be extended by either explicit or implicit directives for data splitting and chunked data transferring to faster memories accessible by the processing element which operates on data. Such a mechanism should include writing back that data and generating API calls that would invoke the needed DMA transfers.

Second, an optimization strategy that translates a series of patterns constituting convolution operation needs to be generalized and applicable to actual computing problems. Begin too-specifically defined, renders it unable to be applied correctly to applications utilizing convolution. Regarding optimization strategies for accelerators for specific domains, the current system requires manual writing optimization strategies for each piece of the specialized hardware contained by the underlying platform. Descriptors of the underlying hardware could be introduced which could be used to generate optimization strategies, thus lowering the need for compiler engineers to write those strategies manually. Hardware descriptors could be used to improve other aspects of the compiler as well.

Third and last, while the approach provided in this thesis ultimately provides a model that is simpler, cleaner, and more concise than the imperative-based native programming model, it can arguably be made even easier or cleaner. Functional programming paradigm and domain-specific languages have a relatively steep learning curve that could repel programmers and domain-scientist from exploiting it for good. It is therefore a responsibility of researchers to make the provided models as approachable as possible to drive their further usage in domains of industry, academia, and research in general.

Appendix A

Hand-tuned code

This appendix provides an example of a hand-tuned code for the Sobel filter benchmark available in subsection 6.2.2, with code necessary to perform performance and energy efficiency measurements described in section 6.

```
#include<stdio.h>

/* PMSISincludes.    */
#include"pmsis.h"

/* Gap_libincludes.    */
#include"gaplib/ImgIO.h"

/* Imagedimensions    */
#defineIMG_LINES 240
#defineIMG_COLS 320

#defineSTACK_SIZE 2048

/* Forconvenience    */
typedefunsignedcharbyte;

/* Pointersinmemoryforinputandoutputimages              */
byte* ImageIn_L2;
byte* ImageOut_L2;

intG_X[] = {
    -1, 0, 1,
    -2, 0, 2,
    -1, 0, 1
};

intG_Y[] = {
```

```
    -1, -2, -1,
    0, 0, 0,
    1, 2, 1
};

/**
* GAP8doesnotimplementmathlibrary
*       (oritcertainlydoesn'tdosoforatleastpartofit)
* This(reused)implementationisappropriateforembeddeddevices
* See:https://stackoverflow.com/questions/1100090/
*   looking-for-an-efficient-integer-square-root-algorithm-for-arm-thumb2
*/
uint32_t SquareRoot(uint32_t a_nInput)
{
    uint32_t op  = a_nInput;
    uint32_t res = 0;
    //1u<<14foruint16_ttype;
    uint32_t one = 1uL << 30;

    while(one > op)
    {
        one >>= 2;
    }

    while(one != 0)
    {
        if(op >= res + one)
        {
            op = op - (res + one);
            res = res +  2 * one;
        }
        res >>= 1;
        one >>= 2;
    }
    returnres;
}

voidsubpicture(byte  * in_picture, byte* out_subpicture,
    intpicture_size,intwidth,intcurr_element)
{
    byte is_top_row_pixel = curr_element - width < 0;
    byte is_left_column_pixel = curr_element % width == 0;
    byte is_bottom_row_pixel = curr_element + width >= picture_size;
    byte is_right_column_pixel = (curr_element + 1) % width == 0;

```

```
74      out_subpicture[0] = !is_top_row_pixel && !is_left_column_pixel      ?
75          in_picture[curr_element - width - 1]  : 0;
76      out_subpicture[1] = !is_top_row_pixel                               ?
77          in_picture[curr_element - width]      : 0;
78      out_subpicture[2] = !is_top_row_pixel && !is_right_column_pixel     ?
79          in_picture[curr_element - width + 1]  : 0;
80
81      out_subpicture[3] = !is_left_column_pixel                           ?
82          in_picture[curr_element - 1]          : 0;
83      out_subpicture[4] =
84          in_picture[curr_element];
85      out_subpicture[5] = !is_right_column_pixel                          ?
86          in_picture[curr_element + 1]          : 0;
87
88      out_subpicture[6] = !is_bottom_row_pixel && !is_left_column_pixel   ?
89          in_picture[curr_element + width - 1]  : 0;
90      out_subpicture[7] = !is_bottom_row_pixel                            ?
91          in_picture[curr_element + width]      : 0;
92      out_subpicture[8] = !is_bottom_row_pixel && !is_right_column_pixel  ?
93          in_picture[curr_element + width + 1]  : 0;
94
95  }
96
97  /* Mainclusterentrypoint,executedoncore0              */
98  voidcluster_entry_point(void  * args)
99  {
100     /* Benchmarking.Countactivecycles        */
101     pi_perf_conf(1 << PI_PERF_ACTIVE_CYCLES);
102     pi_perf_reset();
103     pi_perf_start();
104     inttime1 = pi_perf_read(PI_PERF_ACTIVE_CYCLES);
105
106     int i,j;
107
108     intaccu_x;
109     intaccu_y;
110
111     byte tmp_subpicture[9];
112     memset(tmp_subpicture, 0, 9);
113
114     /* Thisdirectivewillparallelizeonallofthecluster cores                  */
115     #pragma omp parallelforfirstprivate(tmp_subpicture)
116     for(i = 0; i < IMG_LINES * IMG_COLS; ++i){
117         subpicture(ImageIn_L2, tmp_subpicture,
118             IMG_LINES * IMG_COLS, IMG_COLS, i);
```

```
119
120         accu_x = 0;
121         accu_y = 0;
122
123         for(j = 0; j < 9; ++j){
124             accu_x = accu_x + (tmp_subpicture[j] * G_X[9 - j - 1]);
125             accu_y = accu_y + (tmp_subpicture[j] * G_Y[9 - j - 1]);
126         }
127
128         ImageOut_L2[i] =
129             (byte) SquareRoot(accu_x * accu_x + accu_y * accu_y);
130     }
131
132
133     /* Stopthecounterandprint#activecycles              */
134     pi_perf_stop();
135     inttime2 = pi_perf_read(PI_PERF_ACTIVE_CYCLES);
136     printf("Totalcycles:%d\n", time2 - time1);
137 }
138
139 structpi_device gpio_a1;
140 structpi_gpio_conf gpio_conf;
141
142 #defineFREQ_FC (250 *1000000)
143 #defineFREQ_CL (175 *1000000)
144
145 /* Entrypoint-ExecutesonFC          */
146 voidsobel_filter_main()
147 {
148     printf("MainFCentrypoint\n");
149
150     pi_pad_set_function(
151         PI_PAD_12_A3_RF_PACTRL0,
152         PI_PAD_12_A3_GPIO_A0_FUNC1
153     );
154     pi_gpio_e gpio_out_a1 = PI_GPIO_A0_PAD_12_A3;
155     pi_gpio_flags_e cfg_flags = PI_GPIO_OUTPUT;
156     pi_gpio_pin_configure(&gpio_a1, gpio_out_a1, cfg_flags);
157     pi_gpio_pin_write(&gpio_a1, gpio_out_a1, 0);
158
159     char *in_image_file_name ="valve.pgm";
160     charpath_to_in_image[64];
161     sprintf(path_to_in_image,"../../../%s", in_image_file_name);
162
163     intimage_size_bytes = IMG_COLS * IMG_LINES * sizeof(byte);
```

```
164
165     /* Allocatememoryforbothinputandoutputimagesin L2memory                    */
166     ImageIn_L2 = (byte *) pi_l2_malloc(image_size_bytes);
167     ImageOut_L2 = (byte *) pi_l2_malloc(image_size_bytes);
168
169     if(ReadImageFromFile(
170         path_to_in_image, IMG_COLS, IMG_LINES, 1, ImageIn_L2,
171         image_size_bytes, IMGIO_OUTPUT_CHAR, 0
172     ))
173     {
174         printf("Failedtoloadimage%s\n", path_to_in_image);
175         pmsis_exit(-1);
176     }
177
178     /* Preparecl usterdescriptionstructureandopencluster              */
179     structpi_device cl_device;
180     structpi_cluster_conf cl_configuration;
181     pi_cluster_conf_init(&cl_configuration);
182     cl_configuration.id = 0;
183     pi_open_from_conf(&cl_device, &cl_configuration);
184     if(pi_cluster_open(&cl_device))
185     {
186         printf("Clusteropenfailed\n");
187         pmsis_exit(-1);
188     }
189
190     printf("FCFREQ:%d\n", rt_freq_get(RT_FREQ_DOMAIN_FC));
191     printf("CLFREQ:%d\n", rt_freq_get(RT_FREQ_DOMAIN_CL));
192
193     /* Preparetaskdescriptionstructure        */
194     structpi_cluster_task * cl_task =
195         pmsis_l2_malloc(sizeof(structpi_cluster_task));
196     memset(cl_task, 0,sizeof(structpi_cluster_task));
197     cl_task->entry = cluster_entry_point;
198     cl_task->arg = NULL;
199     cl_task->stack_size = (uint32_t) STACK_SIZE;
200
201     printf("Start\n");
202     /* Benchmarking.Countactivecycles        */
203     pi_gpio_pin_write(&gpio_a1, gpio_out_a1, 1);
204     longtime_usec1 = rt_time_get_us();
205
206     /* Sendtasktocluster,blockuntilcompletion              */
207     pi_cluster_send_task_to_cl(&cl_device, cl_task);
208
```

```
209
210     pi_gpio_pin_write(&gpio_a1, gpio_out_a1, 0);
211     longtime_usec2 = rt_time_get_us();
212     printf("Wallclocktime:%ldusec\n", time_usec2 - time_usec1);
213     printf("End\n");
214
215     /* Writeimagetofile        */
216     char *out_image_file_name ="img_out.ppm";
217     charpath_to_out_image[50];
218     sprintf(path_to_out_image,"../../../%s", out_image_file_name);
219     printf("Pathtooutputimage:%s\n", path_to_out_image);
220     WriteImageToFile(
221         path_to_out_image, IMG_COLS, IMG_LINES, 1,
222         ImageOut_L2, GRAY_SCALE_IO
223     );
224
225     pi_cluster_close(&cl_device);
226
227     printf("Clusterclosed,overandout\n");
228
229     pmsis_exit(0);
230 }
231
232 /* PMSISmainfunction      */
233 intmain(intargc,char        *argv[])
234 {
235     printf("\n\n\t      *** SobelFilter(OMP)      ***\n\n");
236     returnpmsis_kickoff((void     *) sobel_filter_main);
237 }
```

Listing A.1: Sobel filter benchmark – Hand-tuned code

Appendix B

Generated code

This appendix provides examples of the generated programming codes for the GAP8 chip. The first example in section B.1 is generated code for Sobel filter benchmark available in subsection 6.2.2, while the second example in section B.2 provides the generated code which utilizes the hardware convolution engine.

B.1 Sobel filter benchmark

```
//Acceleratorfunctions

#include<stdint.h>
//movedhere
#include"gap8/gap8.h"
uint32_t gap_sqrt(uint32_t a_nInput)
    {
    uint32_t op  = a_nInput;
    uint32_t res = 0;

    uint32_t one = 1uL << 30;
    while(one > op){
        one >>= 2;
    }
    while(one != 0) {
        if(op >= res + one){
        op = op - (res + one);
        res = res +  2 * one;
        }
        res >>= 1;
        one >>= 2;
    }
    returnres;
```

```
24      }
25
26  structcluster_params {
27      uint8_t* output;
28      intn2;
29      intn1;
30      int* e5;
31      int* e4;
32      uint8_t* e3;
33  };
34
35  voidcluster_core_task(void  * args){
36      /* Benchmarking.Countactivecycles        */
37      pi_perf_conf(1 << PI_PERF_ACTIVE_CYCLES);
38      pi_perf_reset();
39      pi_perf_start();
40      inttime1 = pi_perf_read(PI_PERF_ACTIVE_CYCLES);
41
42      structcluster_params  * cl_params = (structcluster_params  *)args;
43      uint8_t* output = (*cl_params).output;
44      intn2 = ( *cl_params).n2;
45      intn1 = ( *cl_params).n1;
46      int* e5 = (*cl_params).e5;
47      int* e4 = (*cl_params).e4;
48      uint8_t* e3 = (*cl_params).e3;
49      {
50      //collapse(2)
51      #pragma omp parallelforcollapse(2)
52      for(inti_793 = 0; i_793 < n1; i_793 = 1 + i_793) {
53          //#pragmaomppparallelfor
54          for(inti_794 = 0; i_794 < n2; i_794 = 1 + i_794) {
55          /* reduceSeq */
56          {
57              uint32_t x742;
58              x742 = (uint32_t)0;
59              /* unrollingloopof9       */
60              x742 = x742 + (((uint32_t)((i_794 < 1) ? ((uint8_t)0) :
61                  ((i_793 < 1) ? ((uint8_t)0) : e3[((-1 + i_794) +
62                      (-1 * n2)) + (i_793 * n2)]))) * ((uint32_t)e4[0]));
63              x742 = x742 + (((uint32_t)((i_793 < 1) ? ((uint8_t)0) :
64                  e3[(i_794 + (-1 * n2)) +
65                      (i_793 * n2)])) * ((uint32_t)e4[1]));
66              x742 = x742 + (((uint32_t)(((2 + i_794) < (1 + n2)) ?
67                  ((i_793 < 1) ? ((uint8_t)0) : e3[((1 + i_794) +
68                      (-1 * n2)) + (i_793 * n2)]) :
```

```
                    ((uint8_t)0))) * ((uint32_t)e4[2]));
            x742 = x742 + (((uint32_t)((i_794 < 1) ? ((uint8_t)0) :
                e3[(-1 + i_794) + (i_793 * n2)])) * ((uint32_t)e4[3]));
            x742 = x742 + (((uint32_t)e3[i_794 +
                (i_793 * n2)]) * ((uint32_t)e4[4]));
            x742 = x742 + (((uint32_t)(((2 + i_794) < (1 + n2)) ?
                e3[(1 + i_794) + (i_793 * n2)] :
                    ((uint8_t)0))) * ((uint32_t)e4[5]));
            x742 = x742 + (((uint32_t)((i_794 < 1) ? ((uint8_t)0) :
                (((2 + i_793) < (1 + n1)) ? e3[((-1 + i_794) + n2) +
                    (i_793 * n2)] : ((uint8_t)0)))) * ((uint32_t)e4[6]));
            x742 = x742 + (((uint32_t)(((2 + i_793) < (1 + n1)) ?
                e3[(i_794 + n2) + (i_793 * n2)] :
                    ((uint8_t)0))) * ((uint32_t)e4[7]));
            x742 = x742 + (((uint32_t)(((2 + i_794) < (1 + n2)) ?
                (((2 + i_793) < (1 + n1)) ? e3[((1 + i_794) + n2) +
                    (i_793 * n2)] : ((uint8_t)0)) :
                        ((uint8_t)0))) * ((uint32_t)e4[8]));
            /* reduceSeq */
            {
            uint32_t x716;
            x716 = (uint32_t)0;
            /* unrollingloopof9      */
            x716 = x716 + (((uint32_t)((i_794 < 1) ? ((uint8_t)0) :
                ((i_793 < 1) ? ((uint8_t)0) : e3[((-1 + i_794) +
                    (-1 * n2)) + (i_793 * n2)]))) * ((uint32_t)e5[0]));
            x716 = x716 + (((uint32_t)((i_793 < 1) ? ((uint8_t)0) :
                e3[(i_794 + (-1 * n2)) +
                    (i_793 * n2)])) * ((uint32_t)e5[1]));
            x716 = x716 + (((uint32_t)(((2 + i_794) < (1 + n2)) ?
                ((i_793 < 1) ? ((uint8_t)0) : e3[((1 + i_794) +
                    (-1 * n2)) + (i_793 * n2)]) :
                        ((uint8_t)0))) * ((uint32_t)e5[2]));
            x716 = x716 + (((uint32_t)((i_794 < 1) ? ((uint8_t)0) :
                e3[(-1 + i_794) + (i_793 * n2)])) * ((uint32_t)e5[3]));
            x716 = x716 + (((uint32_t)e3[i_794 +
                (i_793 * n2)]) * ((uint32_t)e5[4]));
            x716 = x716 + (((uint32_t)(((2 + i_794) < (1 + n2)) ?
                e3[(1 + i_794) + (i_793 * n2)] :
                    ((uint8_t)0))) * ((uint32_t)e5[5]));
            x716 = x716 + (((uint32_t)((i_794 < 1) ? ((uint8_t)0) :
                (((2 + i_793) < (1 + n1)) ? e3[((-1 + i_794) + n2) +
                    (i_793 * n2)] : ((uint8_t)0)))) * ((uint32_t)e5[6]));
            x716 = x716 + (((uint32_t)(((2 + i_793) < (1 + n1)) ?
                e3[(i_794 + n2) + (i_793 * n2)] :
```

```
                    ((uint8_t)0))) * ((uint32_t)e5[7]));
            x716 = x716 + (((uint32_t)(((2 + i_794) < (1 + n2)) ?
                (((2 + i_793) < (1 + n1)) ? e3[((1 + i_794) + n2) +
                    (i_793 * n2)] : ((uint8_t)0)) :
                        ((uint8_t)0))) * ((uint32_t)e5[8]));
            output[i_794 + (i_793 * n2)] =
                (uint8_t)gap_sqrt((x742 * x742) + (x716 * x716));
            }
        }
        }
    }
    }
    /* Stopthecounterandprint#activecycles              */
    pi_perf_stop();
    inttime2 = pi_perf_read(PI_PERF_ACTIVE_CYCLES);
    printf("Totalcycles:%d\n", time2 - time1);
}


//Hostcode

//#include"gap8/gap8.h"
structfoo_t {
    Kernel cluster_core_task;
};

typedefstructfoo_t foo_t;

voidfoo_init(foo_t * self){
    (*self).cluster_core_task = loadKernel(cluster_core_task, 2048);
}

voidfoo_destroy(Context ctx, foo_t * self){
    destroyKernel(ctx, (*self).cluster_core_task);
}

voidfoo_run(Context ctx, foo_t * self, Buffer moutput,intn1,intn2,
    Buffer me3, Buffer me4, Buffer me5){
        {
            DeviceBuffer b0 =
                deviceBufferSync(ctx, moutput,
                n1 * (n2 * sizeof(uint8_t)), 0);
            intb1 = n2;
            intb2 = n1;
            DeviceBuffer b3 =
```

```
159                 deviceBufferSync(ctx, me5, 3 * (3 * sizeof(int)), 0);
160             DeviceBuffer b4 =
161                 deviceBufferSync(ctx, me4, 3 * (3 * sizeof(int)), 0);
162             DeviceBuffer b5 =
163                 deviceBufferSync(ctx, me3, n1 * (n2 * sizeof(uint8_t)), 0);
164             structcluster_params * cl_params = (structcluster_params *)
165                 pmsis_l2_malloc(sizeof(structcluster_params));
166             (*cl_params).output = b0;
167             (*cl_params).n2 = b1;
168             (*cl_params).n1 = b2;
169             (*cl_params).e5 = b3;
170             (*cl_params).e4 = b4;
171             (*cl_params).e3 = b5;
172             launchKernel(ctx, (*self).cluster_core_task, 8, cl_params);
173         }
174 }
175
176 voidfoo_init_run(Context ctx, Buffer moutput,intn1,intn2, Buffer me3,
        Buffer me4, Buffer me5){
177     foo_t foo;
178     foo_init(&foo);
179     foo_run(ctx, &foo, moutput, n1, n2, me3, me4, me5);
180     foo_destroy(ctx, &foo);
181 }
182
183
184 Buffer ImageInBuffer;
185 Buffer ImageOutBuffer;
186
187 Buffer gxBuffer;
188 Buffer gyBuffer;
189
190 intG_X[] = {
191     -1, 0, 1,
192     -2, 0, 2,
193     -1, 0, 1
194 };
195
196 intG_Y[] = {
197     -1, -2, -1,
198     0, 0, 0,
199     1, 2, 1
200 };
201
202 #defineIMG_LINES 240
```

```
203 #defineIMG_COLS 320
204
205 structpi_device gpio_a1;
206 structpi_gpio_conf gpio_conf;
207
208 #defineFREQ_FC (250 *1000000)
209 #defineFREQ_CL (175 *1000000)
210
211 void__main(intargc,char **argv)
212 {
213     printf("MainFCentrypoint(Manuallywritten)\n");
214
215     pi_pad_set_function(PI_PAD_12_A3_RF_PACTRL0,
216         PI_PAD_12_A3_GPIO_A0_FUNC1);
217     pi_gpio_e gpio_out_a1 = PI_GPIO_A0_PAD_12_A3;
218     pi_gpio_flags_e cfg_flags = PI_GPIO_OUTPUT;
219     pi_gpio_pin_configure(&gpio_a1, gpio_out_a1, cfg_flags);
220     pi_gpio_pin_write(&gpio_a1, gpio_out_a1, 0);
221
222     char *in_image_file_name ="valve.pgm";
223     charpath_to_in_image[64];
224     sprintf(path_to_in_image,"../../../%s", in_image_file_name);
225
226     Context ctx = createDefaultContext();
227
228     ImageInBuffer = createBuffer(ctx,
229         IMG_COLS * IMG_LINES * sizeof(unsignedchar), 0);
230     ImageOutBuffer = createBuffer(ctx,
231         IMG_COLS * IMG_LINES * sizeof(unsignedchar), 0);
232     gxBuffer = createBuffer(ctx, 9, HOST_READ);
233     gyBuffer = createBuffer(ctx, 9, HOST_READ);
234
235     gxBuffer->inner = G_X;
236     gyBuffer->inner = G_Y;
237
238     if(ReadImageFromFile(path_to_in_image, IMG_COLS, IMG_LINES, 1,
239         ImageInBuffer->inner, IMG_COLS * IMG_LINES * sizeof(unsignedchar),
240         IMGIO_OUTPUT_CHAR, 0))
241     {
242         printf("Failedtoloadimage%s\n", path_to_in_image);
243         pmsis_exit(-1);
244     }
245
246     printf("FCFREQ:%d\n", rt_freq_get(RT_FREQ_DOMAIN_FC));
247     printf("CLFREQ:%d\n", rt_freq_get(RT_FREQ_DOMAIN_CL));
```

```
248
249     printf("Start\n");
250     pi_gpio_pin_write(&gpio_a1, gpio_out_a1, 1);
251     longtime_usec1 = rt_time_get_us();
252
253     foo_init_run(ctx, ImageOutBuffer, IMG_LINES, IMG_COLS,
254         ImageInBuffer, gxBuffer, gyBuffer);
255
256     pi_gpio_pin_write(&gpio_a1, gpio_out_a1, 0);
257     longtime_usec2 = rt_time_get_us();
258     printf("Wallclocktime:%ldusec\n", time_usec2 - time_usec1);
259     printf("End\n");
260
261     /* Writeimagetofile        */
262     char *out_image_file_name ="img_out.ppm";
263     charpath_to_out_image[50];
264     sprintf(path_to_out_image,"../../../%s", out_image_file_name);
265     printf("Pathtooutputimage:%s\n", path_to_out_image);
266     WriteImageToFile(path_to_out_image, IMG_COLS, IMG_LINES, 1,
267         ImageOutBuffer->inner, GRAY_SCALE_IO);
268
269     destroyBuffer(ctx, ImageInBuffer);
270     destroyBuffer(ctx, ImageOutBuffer);
271     destroyBuffer(ctx, gxBuffer);
272     destroyBuffer(ctx, gyBuffer);
273     destroyContext(ctx);
274
275     pmsis_exit(0);
276 }
277
278 intmain(intargc,char      ** argv){
279     printf("\n\n\t    *** SobelFilter(RISE)      ***\n\n");
280     returnpmsis_kickoff((void   *)__main);
281 }
```

Listing B.1: Sobel filter benchmark – Generated code

B.2 HWCE utilization

```
//Acceleratorfunctions
#include<stdint.h>
structcluster_params {
    int16_t* output;
    int16_t* e18;
    int16_t* e19;
};

voidcluster_core_task(void * args){
    structcluster_params * cl_params = (structcluster_params *)args;
    int16_t* output = (*cl_params).output;
    int16_t* e18 = (*cl_params).e18;
    int16_t* e19 = (*cl_params).e19;
    {
        {
            HWCE_Enable();
            HWCE_GenericInit((uint32_t)HWCE_CONV3x3, (uint32_t)0,
                (uint32_t)0);
            HwCE_SetYinMode((uint32_t)1);
            HWCE_ProcessOneTile3x3_MultiOut(
                e18, output, NULL, NULL,
                e19, 0, 6, 6, 0x7
            );
            HWCE_Disable();
        }
    }
}

//Hostcode
#include"gap8/gap8.h"
structfoo_t {
    Kernel cluster_core_task;
};

typedefstructfoo_t foo_t;

voidfoo_init(foo_t * self){
    (*self).cluster_core_task = loadKernel(cluster_core_task, 2048);
}

voidfoo_destroy(Context ctx, foo_t * self){
    destroyKernel(ctx, (*self).cluster_core_task);
}
```

```
44
45 voidfoo_run(Context ctx, foo_t * self,
46     Buffer moutput, Buffer me18, Buffer me19){
47     {
48         DeviceBuffer b0 = deviceBufferSync(
49             ctx, moutput, 4 * (4 * sizeof(int16_t)), 0
50         );
51         DeviceBuffer b1 = deviceBufferSync(
52             ctx, me18, 6 * (6 * sizeof(int16_t)), 0
53         );
54         DeviceBuffer b2 = deviceBufferSync(
55             ctx, me19, 3 * (3 * sizeof(int16_t)), 0
56         );
57         structcluster_params * cl_params =
58             (structcluster_params *)pmsis_l2_malloc(sizeof(
59                 structcluster_params
60             ));
61         (*cl_params).output = b0;
62         (*cl_params).e18 = b1;
63         (*cl_params).e19 = b2;
64         launchKernel(ctx, (*self).cluster_core_task, 8, cl_params);
65     }
66 }
67
68 voidfoo_init_run(Context ctx, Buffer moutput, Buffer me18, Buffer me19){
69     foo_t foo;
70     foo_init(&foo);
71     foo_run(ctx, &foo, moutput, me18, me19);
72     foo_destroy(ctx, &foo);
73 }
74
75 /** Here,void__main()functionismissingwhichshould
76  *     instantiatebuffers,fillthemwithdata,triggersynccalls,
77  *     andlaunchcomputationoncluster.
78  *     Thatfunctionperformsonlyadministrativeoperations,
79  *     andthusisnotconsiderednecessarilyimportant
80  *     forevaluationwhatsoever.
81  */
82
83 intmain(intargc,char ** argv){
84     returnpmsis_kickoff((void *)__main);
85 }
```

Listing B.2: HWCE benchmark – Generated code

Bibliography

[1]TOP500.org, “November 2021 | Top500”, [Online]. Available: https://www.top500.org/lists/top500/2021/11/ (December 9 2021.).

[2]TechTarget, “Japan named HPC leader as world races to exascale ”, [Online]. Available: https://searchdatacenter.techtarget.com/news/252492169/Japan-named-HPC-leader-as-world-races-to-exascale (December 4 2021.).

[3]Messina, P., “The Exascale Computing Project”, Computing in Science and Engineering, vol. 19, no. 3, 2017, pp. 63–67.

[4]RIKEN, “To exascale and beyond”, [Online]. Available: https://www.riken.jp/en/news_pubs/research_news/rr/2019spring/ (November 23 2021.).

[5]The Next Platform, “China Has Already Reached Exascale - On Two Separate Systems”, [Online]. Available: https://www.nextplatform.com/2021/10/26/china-has-already-reached-exascale-on-two-separate-systems/ (November 15 2021.).

[6]Gagliardi, F., Moreto, M., Olivieri, M., and Valero, M., “The international race towards Exascale in Europe”, CCF Transactions on High Performance Computing, vol. 1, no. 1, 2019, pp. 3–13, [Online]. Available: https://doi.org/10.1007/s42514-019-00002-y

[7]Skordas, T., “Toward a European exascale ecosystem”, Communications of the ACM, vol. 62, no. 4, mar 2019, pp. 70–70, [Online]. Available: https://dl.acm.org/doi/10.1145/3312567

[8]Ayris, P., Berthou, J.-Y., Bruce, R., Lindstaedt, S., Monreale, A., Mons, B., Murayama, Y., Södergård, C., Tochtermann, K., and Wilkinson, R., “Realising the European open science cloud”, Tech. Rep., dec 2016, [Online]. Available: https://cris.vtt.fi/en/publications/realising-the-european-open-science-cloud

[9]Rigo, A., Pinto, C., Pouget, K., Raho, D., Dutoit, D., Martinez, P. Y., Doran, C., Benini, L., Mavroidis, I., Marazakis, M., Bartsch, V., Lonsdale, G., Pop, A., Goodacre, J., Colliot, A., Carpenter, P., Radojković, P., Pleiter, D., Drouin, D., and De Dinechin, B. D.,

"Paving the Way Towards a Highly Energy-Efficient and Highly Integrated Compute Node for the Exascale Revolution: The ExaNoDe Approach", Proceedings - 20th Euromicro Conference on Digital System Design, DSD 2017, 2017, pp. 486–493.

[10]Kova č, M., Reinhardt, D., Jesorsky, O., Traub, M., Denis, J.-m., and Notton, P., "European Processor Initiative (EPI)—An Approach for a Future Automotive eHPC Semiconductor Platform", ser. Lecture Notes in Mobility, Langheim, J., Ed. Cham: Springer International Publishing, 2019, pp. 185–195, [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-14156-1_15

[11]European Processor Initiative, "EPI - European Processor Initiative", [Online]. Available: https://www.european-processor-initiative.eu/project/epi/ (December 6 2021.).

[12]Schulte, M. J., Ignatowski, M., Loh, G. H., Beckmann, B. M., Brantley, W. C., Gurumurthi, S., Jayasena, N., Paul, I., Reinhardt, S. K., and Rodgers, G., "Achieving Exascale Capabilities through Heterogeneous Computing", IEEE Micro, vol. 35, no. 4, 2015, pp. 26–36.

[13]NVIDIA Developer, "CUDA Toolkit", [Online]. Available: https://developer.nvidia.com/cuda-toolkit (January 14 2022.).

[14]The Khronos Group Inc, "OpenCL Overview", [Online]. Available: https://www.khronos.org/opencl/ (January 14 2022.).

[15]Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K., "Chisel: Constructing Hardware in a Scala Embedded Language", in Proceedings of the 49th Annual Design Automation Conference on - DAC '12. New York, New York, USA: ACM Press, 2012, p. 1216, [Online]. Available: http://dl.acm.org/citation.cfm?doid=2228360.2228584

[16]Amazon, "Amazon EC2 F1 Instances", [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/ (January 29 2022.).

[17]Chiou, D., "The Microsoft Catapult Project", in 2017 IEEE International Symposium on Workload Characterization (IISWC), vol. 1. IEEE, oct 2017, pp. 124–124, [Online]. Available: http://ieeexplore.ieee.org/document/8167769/

[18]Putnam, A., "FPGAs in the Datacenter", in Proceedings of the on Great Lakes Symposium on VLSI 2017. New York, NY, USA: ACM, may 2017, pp. 5–5, [Online]. Available: https://dl.acm.org/doi/10.1145/3060403.3066860

[19]TOP500.org, "TOP500 Expands Exaflops Capacity Amidst Low Turnover", [Online]. Available: https://www.top500.org/news/top500-expands-exaflops-capacity-amidst-low-turnover/ (December 9 2021.).

[20]TOP500.org, "HPC in 2016: Hits and Misses", [Online]. Available: https://top500.org/news/hpc-in-2016-hits-and-misses/ (December 9 2021.).

[21]Silver, A., "Rethinking CS101 [Resources_Education]", IEEE Spectrum, vol. 54, no. 4, apr 2017, pp. 23–23, [Online]. Available: http://ieeexplore.ieee.org/document/7880452/

[22]Sahami, M., and Roach, S., Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science, ACM Computing Curricula Task Force, Ed. ACM, Inc, jan 2013, vol. 45, no. 2, [Online]. Available: http://dl.acm.org/citation.cfm?id=2534860

[23]Flamand, E., Rossi, D., Conti, F., Loi, I., Pullini, A., Rotenberg, F., and Benini, L., "GAP-8: A RISC-V SoC for AI at the Edge of the IoT", in 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), vol. 2018-July. IEEE, jul 2018, pp. 1–4, [Online]. Available: https://ieeexplore.ieee.org/document/8445101/

[24]Conti, F., Rossi, D., Pullini, A., Loi, I., and Benini, L., "Energy-efficient vision on the PULP platform for ultra-low power parallel computing", IEEE Workshop on Signal Processing Systems, SiPS: Design and Implementation, 2014.

[25]Pullini, A., Conti, F., Rossi, D., Loi, I., Gautschi, M., and Benini, L., "A Heterogeneous Multicore System on Chip for Energy Efficient Brain Inspired Computing", IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 65, no. 8, aug 2018, pp. 1094–1098, [Online]. Available: https://ieeexplore.ieee.org/document/7817777/

[26]Kurth, A., Vogel, P., Capotondi, A., Marongiu, A., and Benini, L., "HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA", 2017, [Online]. Available: http://arxiv.org/abs/1712.06497

[27]IEEE Spectrum, "Gordon Moore: The Man Whose Name Means Progress", [Online]. Available: https://spectrum.ieee.org/gordon-moore-the-man-whose-name-means-progress (January 18 2022.).

[28]Trusted Reviews, "Moore's Law: What is it and why is it dying out?", [Online]. Available: https://www.trustedreviews.com/opinion/what-is-moore-s-law-2946125 (January 18 2022.).

[29]CNET, “CES 2019: Moore’s Law is dead, says Nvidia’s CEO”, [Online]. Available: https://www.cnet.com/tech/computing/moores-law-is-dead-nvidias-ceo-jensen-huang-says-at-ces-2019/ (January 18 2022.).

[30]Alfio Lazzaro, “Programming in Multi-cores Era”, Geneva, Switzerland, [Online]. Available: https://indico.cern.ch/event/59397/contributions/2050044/attachments/996317/1416877/SS_lazzaro.pdf 2010.

[31]Manferdelli, J., Govindaraju, N., and Crall, C., “Challenges and Opportunities in Many-Core Computing”, Proceedings of the IEEE, vol. 96, no. 5, may 2008, pp. 808–815, [Online]. Available: http://ieeexplore.ieee.org/document/4484943/

[32]Sutter, H., and Larus, J., “Software and the Concurrency Revolution”, Queue, vol. 3, no. 7, sep 2005, pp. 54–62, [Online]. Available: https://dl.acm.org/doi/10.1145/1095408.1095421

[33]The Wall Street Journal, “Huang’s Law Is the New Moore’s Law, and Explains Why Nvidia Wants Arm”, [Online]. Available: https://www.wsj.com/articles/huangs-law-is-the-new-moores-law-and-explains-why-nvidia-wants-arm-11600488001 (January 18 2022.).

[34]Bacon, D. F., Rabbah, R., and Shukla, S., “FPGA Programming for the Masses”, Communications of the ACM, vol. 56, no. 4, apr 2013, pp. 56–63, [Online]. Available: https://dl.acm.org/doi/10.1145/2436256.2436271

[35]Flynn, M. J., “Some Computer Organizations and Their Effectiveness”, IEEE Transactions on Computers, vol. C-21, no. 9, sep 1972, pp. 948–960, [Online]. Available: http://ieeexplore.ieee.org/document/5009071/

[36]McCool, M., Robison, A., and Reinders, J., Structured Parallel Programming: Patterns for Efficient Computation, 1st ed. Elsevier, 2012.

[37]Hennessy, J. L., and Patterson, D. A., Computer Architecture: A Quantitative Approach, 5th ed. Elsevier, 2011.

[38]Petersen, W., and Arbenz, P., Introduction to Parallel Computing: A Practical Guide with Examples in C. Oxford University Press, 2004.

[39]Nicolau, A., “Loop Quantization: Unwinding for Fine-Grain Parallelism Exploitation”, Cornell University, Tech. Rep., 1985.

[40]Yeh, T.-Y., and Patt, Y. N., “Two-Level Adaptive Training Branch Prediction”, in Proceedings of the 24th annual international symposium on Microarchitecture - MICRO

24. New York, New York, USA: ACM Press, 1991, pp. 51–61, [Online]. Available: http://portal.acm.org/citation.cfm?doid=123465.123475

[41]Intel, "Intel©64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture", [Online]. Available: file:///home/bpervan/Desktop/Papers/64-ia-32-architectures-software-developer-vol-1-manual.pdf (January 14 2022.).

[42]Lamport, L., "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", IEEE Transactions on Computers, vol. C-28, no. 9, sep 1979, pp. 690–691, [Online]. Available: http://ieeexplore.ieee.org/document/1675439/

[43]Savage, J. E., Models of Computation. Addison-Wesley Reading, MA, 1998, vol. 136.

[44]Diaz, J., Munoz-Caro, C., and Nino, A., "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era", IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 8, aug 2012, pp. 1369–1386, [Online]. Available: http://ieeexplore.ieee.org/document/6122018/

[45]ISO, "ISO/IEC/IEEE 9945:2009 Information technology — Portable Operating System Interface (POSIX®) Base Specifications, Issue 7", [Online]. Available: https://www.iso.org/standard/50516.html (January 14 2022.).

[46]OpenMP, "OpenMP Application Programming Interface", Tech. Rep., nov 2021, [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[47]Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 4.0", University of Tennessee, Knoxville, Tennessee, USA, Tech. Rep., jun 2021, [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[48]Open MPI, "Open MPI:Open Source High Performance Computing", (January 15 2022.).

[49]TOP500.org, "TOP500 - The List.", [Online]. Available: https://www.top500.org/ (November 15 2021.).

[50]Olofsson, A., "Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip", oct 2016, pp. 1–15, [Online]. Available: http://arxiv.org/abs/1610.01832

[51]Guo, K., Zeng, S., Yu, J., Wang, Y., and Yang, H., "A Survey of FPGA-Based Neural Network Accelerator", vol. 9, no. 4, dec 2017, pp. 1–26, [Online]. Available: http://arxiv.org/abs/1712.08934

[52]Wang, C., Gong, L., Yu, Q., Li, X., Xie, Y., and Zhou, X., "DLAU: A Scalable Deep Learning Accelerator Unit on FPGA", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 36, no. 3, 2016, pp. 1–1, [Online]. Available: http://ieeexplore.ieee.org/document/7505926/

[53]Bai, L., Zhao, Y., and Huang, X., "A CNN Accelerator on FPGA Using Depthwise Separable Convolution", IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 65, no. 10, oct 2018, pp. 1415–1419, [Online]. Available: https://ieeexplore.ieee.org/document/8438987/

[54]Li, Y., Liu, Z., Xu, K., Yu, H., and Ren, F., "A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks", ACM Journal on Emerging Technologies in Computing Systems, vol. 14, no. 2, jul 2018, pp. 1–16, [Online]. Available: https://dl.acm.org/doi/10.1145/3154839

[55]Liu, Z., Dou, Y., Jiang, J., Xu, J., Li, S., Zhou, Y., and Xu, Y., "Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks", ACM Transactions on Reconfigurable Technology and Systems, vol. 10, no. 3, 2017.

[56]Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., and Cong, J., "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks", in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. New York, NY, USA: ACM, feb 2015, pp. 161–170, [Online]. Available: https://dl.acm.org/doi/10.1145/2684746.2689060

[57]Strizic, L., Pervan, B., and Knezovic, J., "Deep Learning Accelerator on Programmable Heterogeneous System with RISC-V Processor", in 2019 proceedings of the 42nd international convention MIPRO. IEEE, 2019, pp. 1126–1131, [Online]. Available: https://www.bib.irb.hr/1021661/download/1021661.16_cts_5493.pdf

[58]Karandikar, S., Mao, H., Kim, D., Biancolin, D., Amid, A., Lee, D., Pemberton, N., Amaro, E., Schmidt, C., Chopra, A., Huang, Q., Kovacs, K., Nikolic, B., Katz, R., Bachrach, J., and Asanovic, K., "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud", in 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). IEEE, jun 2018, pp. 29–42, [Online]. Available: https://ieeexplore.ieee.org/document/8416816/

[59]Ovtcharov, K., Ruwase, O., Kim, J.-y., Fowers, J., Strauss, K., and Chung, E. S., "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware", Microsoft Research Whitepaper, 2015, pp. 3–6, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1050.9891&rep=rep1&type=pdf

[60]Tarafdar, N., Eskandari, N., Lin, T., and Chow, P., "Designing for FPGAs in the Cloud", IEEE Design & Test, vol. 35, no. 1, feb 2018, pp. 23–29, [Online]. Available: http://ieeexplore.ieee.org/document/8030335/

[61]Kachris, C., and Soudris, D., "A Survey on Reconfigurable Accelerators for Cloud Computing", in 2016 26th International Conference on Field Programmable Logic and Applications (FPL). IEEE, aug 2016, pp. 1–10, [Online]. Available: http://ieeexplore.ieee.org/document/7577381/

[62]Abel, F., Weerasinghe, J., Hagleitner, C., Weiss, B., and Paredes, S., "An FPGA Platform for Hyperscalers", in 2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI). IEEE, aug 2017, pp. 29–32, [Online]. Available: http://ieeexplore.ieee.org/document/8071053/

[63]Hernández, M., Guerrero, G. D., Cecilia, J. M., García, J. M., Inuggi, A., Jbabdi, S., Behrens, T. E., and Sotiropoulos, S. N., "Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs", PLoS ONE, vol. 8, no. 4, 2013.

[64]Chen, C.-C., Yang, C.-L., and Cheng, H.-Y., "Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform", sep 2018, [Online]. Available: http://arxiv.org/abs/1809.02839

[65]Manavski, S. A., "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography", in 2007 IEEE International Conference on Signal Processing and Communications, no. November. IEEE, 2007, pp. 65–68, [Online]. Available: http://ieeexplore.ieee.org/document/4728256/

[66]Harrison, O., and Waldron, J., "Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware", in International conference on cryptology in Africa. Springer, 2009, pp. 350–367, [Online]. Available: https://nslab.kaist.ac.kr/courses/2015/cs710/paperlist/2-2.pdf

[67]Burtscher, M., and Pingali, K., "An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm", in GPU Computing Gems Emerald Edition. Elsevier, 2011, pp. 75–92, [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9780123849885000061

[68]NVIDIA, "NVIDIA Jetson: The AI platform for autonomous machines.", [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/ (January 23 2022.).

[69]Pervan, B., Guberovic, E., and Turcinovic, F., "Hazelnut - An Energy Efficient Base IoT Module for Wide Variety of Sensing Applications", in Proceedings of the 6th Conference on the Engineering of Computer Based Systems. New York, NY, USA: ACM, sep 2019, pp. 1–4, [Online]. Available: https://dl.acm.org/doi/10.1145/3352700.3352702

[70]Pervan, B., Knezovic, J., and Pericin, K., "Distributed Password Hash Computation on Commodity Heterogeneous Programmable Platforms", in 13th USENIX Workshop on Offensive Technologies, WOOT 2019, co-located with USENIX Security 2019, 2019, [Online]. Available: https://www.usenix.org/conference/woot19/presentation/pervan

[71]Pervan, B., Knezovi ć, J., and Guberović, E., "Energy-efficient distributed password hash computation on heterogeneous embedded system", Automatika, vol. 63, no. 3, 2022, pp. 399–417, [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/00051144.2022.2042115

[72]Deursen, A. V., Klint, P., and Visser, J., "Domain-specific languages: an annotated bibliography", ACM Sigplan Notices, vol. 35, no. 6, 2000, pp. 26–36.

[73]Ghosh, D., DSLs in Action, 1st ed. USA: Manning Publications Co., 2010.

[74]Artho, C., Havelund, K., Kumar, R., and Yamagata, Y., "Domain-specific languages with scala", Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9407, 2015, pp. 1–16.

[75]Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S., "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines", Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2013, pp. 519–530.

[76]Pu, J., Bell, S., Yang, X., Setter, J., Richardson, S., Ragan-Kelley, J., and Horowitz, M., "Programming Heterogeneous Systems from an Image Processing DSL", vol. 14, no. 3, 2016, pp. 1–25, [Online]. Available: http://arxiv.org/abs/1610.09405

[77]Koehler, T., and Steuwer, M., "Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs", in 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, feb 2021, pp. 27–38, [Online]. Available: https://ieeexplore.ieee.org/document/9370337/

[78]Chafi, H., Sujeeth, A. K., Brown, K. J., Lee, H., Atreya, A. R., and Olukotun, K., "A Domain-Specific Approach To Heterogeneous Parallelism", ACM SIGPLAN Notices, vol. 46, no. 8, sep 2011, pp. 35–46, [Online]. Available: https://dl.acm.org/doi/10.1145/2038037.1941561

[79]Brown, K. J., Sujeeth, A. K., Lee, H. J., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K., "A Heterogeneous Parallel Framework for Domain-Specific Languages", in 2011 International Conference on Parallel Architectures and Compilation Techniques. IEEE, oct 2011, pp. 89–100, [Online]. Available: http://ieeexplore.ieee.org/document/6113791/

[80]Sujeeth, A. K., Brown, K. J., Lee, H., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K., "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages", ACM Transactions on Embedded Computing Systems, vol. 13, no. 4s, jul 2014, pp. 1–25, [Online]. Available: https://dl.acm.org/doi/10.1145/2584665

[81]Sujeeth, A. K., Lee, H. J., Brown, K. J., Chafi, H., Wu, M., Atreya, A. R., Olukotun, K., Rompf, T., and Odersky, M., "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning", Proceedings of the 28th International Conference on Machine Learning, ICML 2011, no. Ml, 2011, pp. 609–616.

[82]Brown, K. J., Lee, H., Rompf, T., Sujeeth, A. K., De Sa, C., Aberger, C., and Olukotun, K., "Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns", in Proceedings of the 2016 International Symposium on Code Generation and Optimization. New York, NY, USA: ACM, feb 2016, pp. 194–205, [Online]. Available: https://dl.acm.org/doi/10.1145/2854038.2854042

[83]Steuwer, M., Remmelg, T., and Dubach, C., "LIFT: A functional data-parallel IR for high-performance GPU code generation", in 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, feb 2017, pp. 74–85, [Online]. Available: http://ieeexplore.ieee.org/document/7863730/

[84]Hagedorn, B., Lenfers, J., Kœhler, T., Qin, X., Gorlatch, S., and Steuwer, M., "Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies", Proceedings of the ACM on Programming Languages, vol. 4, no. ICFP, aug 2020, pp. 1–29, [Online]. Available: https://dl.acm.org/doi/10.1145/3408974

[85]Hagedorn, B., Lenfers, J., Koehler, T., Gorlatch, S., and Steuwer, M., "A Language for Describing Optimization Strategies", 2020, [Online]. Available: http://arxiv.org/abs/2002.02268

[86]Leißa, R., Boesche, K., Hack, S., Pérard-Gayot, A., Membarth, R., Slusallek, P., Müller, A., and Schmidt, B., "AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries", Proceedings of the ACM on Programming Languages, vol. 2, no. OOPSLA, oct 2018, pp. 1–30, [Online]. Available: https://dl.acm.org/doi/10.1145/3276489

[87]Koeplinger, D., Feldman, M., Prabhakar, R., Zhang, Y., Hadjis, S., Fiszel, R., Zhao, T., Nardi, L., Pedram, A., Kozyrakis, C., and Olukotun, K., "Spatial: A Language and Compiler for Application Accelerators", ACM SIGPLAN Notices, vol. 53, no. 4, dec 2018, pp. 296–311, [Online]. Available: https://dl.acm.org/doi/10.1145/3296979.3192379

[88]Dubach, C., Cheng, P., Rabbah, R., Bacon, D. F., and Fink, S. J., "Compiling a high-level language for GPUs", ACM SIGPLAN Notices, vol. 47, no. 6, aug 2012, pp. 1–12, [Online]. Available: http://dl.acm.org/citation.cfm?doid=2345156.2254066

[89]Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., and Amarasinghe, S., "GraphIt: A High-Performance Graph DSL", Proceedings of the ACM on Programming Languages, vol. 2, no. OOPSLA, oct 2018, pp. 1–30, [Online]. Available: https://dl.acm.org/doi/10.1145/3276491

[90]Silvano, C., Agosta, G., Bartolini, A., Beccari, A., Benini, L., Besnard, L., Bispo, J., Cmar, R., Cardoso, J. M. P., Cavazzoni, C., Cherubin, S., Gadioli, D., Golasowski, M., Lasri, I., Martinovič, J., Palermo, G., Pinto, P., Rohou, E., Sanna, N., Slaninova, K., and Vitali, E., "ANTAREX: A DSL-based Approach to Adaptively Optimizing and Enforcing Extra-Functional Properties in High Performance Computing", in Euromicro DSD/SEEA 2018, Prague, Czech Republic, 2018, pp. 1–8, [Online]. Available: https://hal.inria.fr/hal-01890152

[91]Cardoso, J. M., Carvalho, T., Coutinho, J. G., Luk, W., Nobre, R., Diniz, P., and Petrov, Z., "LARA: An Aspect-Oriented Programming Language for Embedded Systems", in Proceedings of the 11th annual international conference on Aspect-oriented Software Development - AOSD '12. New York, New York, USA: ACM Press, 2012, p. 179, [Online]. Available: http://dl.acm.org/citation.cfm?doid=2162049.2162071

[92]Diamos, G. F., and Yalamanchili, S., "Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems", in Proceedings of the 17th international symposium on High performance distributed computing - HPDC '08. New York, New York, USA: ACM Press, 2008, p. 197, [Online]. Available: http://portal.acm.org/citation.cfm?doid=1383422.1383447

[93]Rossbach, C. J., Yu, Y., Currey, J., Martin, J.-p., and Fetterly, D., "Dandelion: a Compiler and Runtime for Heterogeneous Systems", in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. New York, NY, USA: ACM, nov 2013, pp. 49–68, [Online]. Available: https://dl.acm.org/doi/10.1145/2517349.2522715

[94]Kjolstad, F., Kamil, S., Chou, S., Lugato, D., and Amarasinghe, S., "The tensor algebra compiler", Proceedings of the ACM on Programming Languages, vol. 1, no. OOPSLA, oct 2017, pp. 1–29, [Online]. Available: http://dl.acm.org/citation.cfm?doid=3152284.3133901

[95]Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., and Fey, D., "HPX – A Task Based Programming Model in a Global Address Space", in Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models - PGAS '14, vol. 2014-Octob. New York, New York, USA: ACM Press, 2014, pp. 1–11, [Online]. Available: http://dl.acm.org/citation.cfm?doid=2676870.2676883

[96]Ofenbeck, G., Rompf, T., Stojanov, A., Odersky, M., and Püschel, M., "Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries", in Proceedings of the 12th international conference on Generative programming: concepts & experiences - GPCE '13. New York, New York, USA: ACM Press, 2013, pp. 125–134, [Online]. Available: http://dl.acm.org/citation.cfm?doid=2517208.2517228

[97]Rompf, T., and Odersky, M., "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs", Communications of the ACM, vol. 55, no. 6, jun 2012, pp. 121–130, [Online]. Available: https://dl.acm.org/doi/10.1145/2184319.2184345

[98]Lab, T. S. P. P., "Argon", [Online]. Available: https://github.com/stanford-ppl/argon (February 5 2022.).

[99]Asanovi ć, K., and Patterson, D. A., "Instruction Sets Should Be Free: The Case For RISC-V", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug 2014, [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[100]RISC-V International, "About RISC-V", [Online]. Available: https://riscv.org/about/ (January 19 2022.).

[101]Waterman, A., and Asanovic, K., "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA", Tech report, vol. I, 2019, [Online]. Available: https://riscv.org/technical/specifications/

[102]Waterman, A., Asanovi ć, K., and Hauser, J., "The RISC-V Instruction Set Manual Volume II: Privileged Architecture", Tech report, vol. II, 2021, [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[103]Gautschi, M., Schiavone, P. D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gurkaynak, F. K., and Benini, L., "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 10, oct 2017, pp. 2700–2713, [Online]. Available: https://ieeexplore.ieee.org/document/7864441/

[104]Perotti, M., Schiavone, P. D., Tagliavini, G., Rossi, D., Kurd, T., Hill, M., Yingying, L., and Benini, L., "HW / SW approaches for RISC-V code size reduction", in Workshop on Computer Architecture Research with RISC-V (CARRV 2020), Online, 2020, [Online]. Available: https://www.research-collection.ethz.ch/handle/20.500.11850/461404

[105]European Processor Initiative, "EPI EPAC1.0 RISC-V Test Chip Samples Delivered", [Online]. Available: https://www.european-processor-initiative.eu/epi-epac1-0-risc-v-test-chip-samples-delivered/ (January 19 2022.).

[106]SiFive, "About", [Online]. Available: https://www.sifive.com/ (January 19 2022.).

[107]Celio, C., Patterson, D. A., and Asanovi ć, K., "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, Jun 2015, [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[108]Celio, C., Chiu, P.-F., Nikolic, B., Patterson, D. A., and Asanovi ć, K., "BOOM v2: an open-source out-of-order RISC-V core", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, Sep 2017, [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html

[109]Asanovi ć, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H., and Waterman, A., "The Rocket Chip Generator", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016, [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[110]Olof Kindgren, "SERV - The SErial RISC-V CPU", [Online]. Available: https://github.com/olofk/serv (January 20 2022.).

[111]Olof Kindgren, "CoreScore", [Online]. Available: https://github.com/olofk/corescore (January 20 2022.).

[112]PULP Platform, “PULP Project Information”, [Online]. Available: https://pulp-platform.org/projectinfo.html (January 22 2022.).

[113]Davide Schiavone, P., Conti, F., Rossi, D., Gautschi, M., Pullini, A., Flamand, E., and Benini, L., “Slow and Steady Wins the Race? A Comparison of Ultra-Low-Power RISC-V Cores for Internet-of-Things Applications”, in 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), vol. 2017-Janua. IEEE, sep 2017, pp. 1–8, [Online]. Available: http://ieeexplore.ieee.org/document/8106976/

[114]lowRISC, “Ibex: An embedded 32 bit RISC-V CPU core”, [Online]. Available: https://ibex-core.readthedocs.io/en/latest/ (January 23 2022.).

[115]Zaruba, F., Schuiki, F., Hoefler, T., and Benini, L., “Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads”, IEEE Transactions on Computers, vol. 70, no. 11, nov 2021, pp. 1845–1860, [Online]. Available: https://ieeexplore.ieee.org/document/9216552/

[116]Zaruba, F., and Benini, L., “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 11, nov 2019, pp. 2629–2640, [Online]. Available: https://ieeexplore.ieee.org/document/8777130/

[117]Traber, A., and Gautschi, M., “PULPino: Datasheet”, Tech. Rep., 2017, [Online]. Available: https://pulp-platform.org/docs/pulpino_datasheet.pdf

[118]Traber, A., Zaruba, F., Stucki, S., Pullini, A., Haugou, G., Flamand, E., Gürkaynak, F. K., and Benini, L., “PULPino : A small single-core RISC-V SoC”, [Online]. Available: https://riscv.org/wp-content/uploads/2016/01/Wed1315-PULP-riscv3_noanim.pdf 2016.

[119]Schiavone, P. D., Rossi, D., Pullini, A., Di Mauro, A., Conti, F., and Benini, L., “Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX”, in 2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S). IEEE, oct 2018, pp. 1–3, [Online]. Available: https://ieeexplore.ieee.org/document/8640145/

[120]Pullini, A., Rossi, D., Loi, I., Tagliavini, G., and Benini, L., “Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing”, IEEE Journal of Solid-State Circuits, vol. 54, no. 7, jul 2019, pp. 1970–1981, [Online]. Available: https://ieeexplore.ieee.org/document/8715500/

[121]Pullini, A., Rossi, D., Loi, I., Di Mauro, A., and Benini, L., “Mr. Wolf: A 1 GFLOP/s Energy-Proportional Parallel Ultra Low Power SoC for IOT Edge Processing”, in ESSCIRC 2018 - IEEE 44th European Solid State Circuits Conference (ESSCIRC). IEEE, sep 2018, pp. 274–277, [Online]. Available: https://ieeexplore.ieee.org/document/8494247/

[122]OpenRISC, “OpenRISC 1000 Architecture Manual”, [Online]. Available: https://openrisc.io/or1k.html (January 23 2022.).

[123]Conti, F., Schilling, R., Schiavone, P. D., Pullini, A., Rossi, D., Gurkaynak, F. K., Muehlberghuber, M., Gautschi, M., Loi, I., Haugou, G., Mangard, S., and Benini, L., “An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics”, IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 64, no. 9, sep 2017, pp. 2481–2494, [Online]. Available: http://ieeexplore.ieee.org/document/7927716/

[124]Rossi, D., Conti, F., Eggiman, M., Mauro, A. D., Tagliavini, G., Mach, S., Guermandi, M., Pullini, A., Loi, I., Chen, J., Flamand, E., and Benini, L., “Vega: A Ten-Core SoC for IoT Endnodes With DNN Acceleration and Cognitive Wake-Up From MRAM-Based State-Retentive Sleep Mode”, IEEE Journal of Solid-State Circuits, vol. 57, no. 1, jan 2022, pp. 127–139, [Online]. Available: https://ieeexplore.ieee.org/document/9560136/

[125]Kurth, A., Forsberg, B., and Benini, L., “HEROv2: Full-Stack Open-Source Research Platform for Heterogeneous Computing”, 2022, pp. 1–14, [Online]. Available: http://arxiv.org/abs/2201.03861

[126]Balkind, J., McKeown, M., Fu, Y., Nguyen, T., Zhou, Y., Lavrov, A., Shahrad, M., Fuchs, A., Payne, S., Liang, X., Matl, M., and Wentzlaff, D., “OpenPiton: An Open Source Manycore Research Framework”, ACM SIGPLAN Notices, vol. 51, no. 4, jun 2016, pp. 217–232, [Online]. Available: https://dl.acm.org/doi/10.1145/2954679.2872414

[127]Balkind, J., Lim, K., Gao, F., Tu, J., Wentzlaff, D., Schaffner, M., Zaruba, F., and Benini, L., “OpenPiton+Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores”, Third Workshop on Computer Architecture Research with RISC-V, CARRV, 2019, [Online]. Available: https://parallel.princeton.edu/papers/balkind_carrv2019.pdf

[128]Gürkaynak, F. K., Schilling, R., Muehlberghuber, M., Conti, F., Mangard, S., and Benini, L., “Multi-Core Data Analytics SoC with a Flexible 1.76 Gbit/s AES-XTS Cryptographic Accelerator in 65 nm CMOS”, in Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems. New York, NY, USA: ACM, jan 2017, pp. 19–24, [Online]. Available: https://dl.acm.org/doi/10.1145/3031836.3031840

[129]Azarkhish, E., Rossi, D., Loi, I., and Benini, L., "Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes", IEEE Transactions on Parallel and Distributed Systems, vol. 29, no. 2, feb 2018, pp. 420–434, [Online]. Available: http://ieeexplore.ieee.org/document/8038819/

[130]Palossi, D., Loquercio, A., Conti, F., Flamand, E., Scaramuzza, D., and Benini, L., "A 64-mW DNN-Based Visual Navigation Engine for Autonomous Nano-Drones", IEEE Internet of Things Journal, vol. 6, no. 5, 2019, pp. 8357–8371.

[131]Greenwaves Technolgies, "GAPuino development board", [Online]. Available: https://greenwaves-technologies.com/product/gapuino/ (December 27 2021.).

[132]GreenWaves Technologies, "PMSIS API documentation", [Online]. Available: https://pmsis.readthedocs.io/en/latest/ (January 21 2022.).

[133]Greenwaves Technolgies, "GAP8 Software Development Kit", [Online]. Available: https://greenwaves-technologies.com/manuals/BUILD/HOME/html/index.html (December 29 2021.).

[134]GreenWaves Technologies, "GAP8 Auto-tiler Manual", [Online]. Available: https://greenwaves-technologies.com/manuals/BUILD/PMSIS_API/html/index.html (January 22 2022.).

[135]Stoltzfus, L., Hagedorn, B., Steuwer, M., Gorlatch, S., and Dubach, C., "Tiling Optimizations for Stencil Computations Using Rewrite Rules in Lift", ACM Transactions on Architecture and Code Optimization, vol. 16, no. 4, dec 2019, pp. 1–25, [Online]. Available: https://dl.acm.org/doi/10.1145/3368858

[136]Atkey, R., Steuwer, M., Lindley, S., and Dubach, C., "Data Parallel Idealised Algol", vol. 0, no. 0, 2018, [Online]. Available: https://homepages.inf.ed.ac.uk/slindley/papers/dpia-draft-july2018.pdf

[137]Steuwer, M., Koehler, T., Köpcke, B., and Pizzuti, F., "RISE & Shine: Language-Oriented Compiler Design", jan 2022, pp. 1–12, [Online]. Available: http://arxiv.org/abs/2201.03611

[138]Stoltzfus, L., Hamilton, B., Steuwer, M., Li, L., and Dubach, C., "Code Generation for Room Acoustics Simulations with Complex Boundary Conditions", Proceedings - 2021 IEEE 35th International Parallel and Distributed Processing Symposium, IPDPS 2021, 2021, pp. 485–496.

[139]Köpcke, B., Steuwer, M., and Gorlatch, S., “Generating Efficient FFT GPU Code with LIFT”, FHPNC 2019 - Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing, co-located with ICFP 2019, 2019, pp. 1–13.

[140]Kristien, M., Bodin, B., Steuwer, M., and Dubach, C., “High-Level Synthesis of Functional Patterns with LIFT”, in Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming. New York, NY, USA: ACM, jun 2019, pp. 35–45, [Online]. Available: https://dl.acm.org/doi/10.1145/3315454.3329957

[141]Mogers, N., Steuwer, M., Smith, A., Dubach, C., Vytiniotis, D., and Tomioka, R., “Towards Mapping Lift to Deep Neural Network Accelerators”, in 1st HiPEAC Workshop on Emerging Deep Learning Accelerators (EDLA), HiPEAC EDLA 2019 ; Conference date: 21-01-2019 Through 21-01-2019", 2019, [Online]. Available: https://www.research.ed.ac.uk/en/publications/towards-mapping-lift-to-deep-neural-network-accelerators

[142]Lücke, M., Steuwer, M., and Smith, A., “Integrating a Functional Pattern-Based IR into MLIR”, CC 2021 - Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction, 2021, pp. 12–22.

[143]RISE, “RISE language Git repository”, [Online]. Available: https://github.com/rise-lang/shine (December 4 2021.).

[144]Budiu, M., Galenson, J., and Plotkin, G. D., “The Compiler Forest”, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2013, vol. 7792 LNCS, pp. 21–40, [Online]. Available: http://link.springer.com/10.1007/978-3-642-37036-6_2

[145]Bruschi, N., Haugou, G., Tagliavini, G., Conti, F., Benini, L., and Rossi, D., “GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors”, in 2021 IEEE 39th International Conference on Computer Design (ICCD), no. Iccd. IEEE, oct 2021, pp. 409–416, [Online]. Available: https://ieeexplore.ieee.org/document/9643828/

[146]Greenwaves Technolgies, “GAPuino V1.1 User’s Manual”, Tech. Rep., 2019, [Online]. Available: https://gwt-website-files.s3.eu-central-1.amazonaws.com/gapuino_v1.1_um.pdf

[147]Sobel, I., and Feldman, G. M., “An Isotropic 3x3 Image Gradient Operator”, 1990, [Online]. Available: https://www.researchgate.net/publication/281104656_An_Isotropic_3x3_Image_Gradient_Operator

[148]Hartigan, J. A., Clustering Algorithms. John Wiley & Sons, Inc., 1975.

[149]Hartigan, J. A., and Wong, M. A., “Algorithm AS 136: A K-Means Clustering Algorithm”, Journal of the Royal Statistical Society, vol. 28, no. 1, 1979, pp. 100–108, [Online]. Available: https://www.stat.cmu.edu/~rnugent/PCMI2016/papers/HartiganKMeans.pdf

Nomenclature

APIApplication Programming Interface

ASTAbstract Syntax Tree

CPUCentral Processing Unit

DPIAData Parallel Idealised Algol

DSLDomain-Specific Language

DSPDigital Signal Processing

FCFabric Controller

GPGPUGeneral-Purpose Graphics Processing Unit

GPUGraphics Processing Unit

HPCHigh-Performance Computing

HWCEHardware Convolution Engine

SoCSystem-on-Chip

TCDMTightly-Coupled Data Memory

List of Figures

List of Tables

Listings

Biography

Branimir Pervan was born in Zagreb where he finished elementary and high school. He graduated in 2015. from the University of Zagreb Faculty of Electrical Engineering and Computing. Throughout his studies and briefly after graduation he worked as a software engineer in various companies. In 2016. he started his postgraduate studies in the field of heterogeneous computing. His scientific and professional interests span across parallelism, heterogeneous computing systems, domain-specific languages, functional programming, and embedded systems. While working on his PhD he undertook multiple visits to research groups of similar interests and attended two summer schools in high-performance computing. From March 2021. to November 2021. he was closely collaborating with a research group at the University of Edinburgh regarding the development of practical outcomes of his doctoral thesis. He is affiliated with Green Light Technologies Ltd. where he works part-time as a software engineer and consultant. Throughout his studies and up to now, he held various governing positions in volunteering organizations. He is a member of professional organizations IEEE, IEEE Computer Society, and HiPEAC. He authored or co-authored several scientific papers published in international journals and conferences.

Bibliography

Journal papers

1. **B. Pervan**, J. Knezović, and E. Guberović, "Energy-Efficient Distributed Password Hash Computation on Heterogeneous Embedded System" Automatika, vol. 63, no. 3, pp. 399–417, 2022, doi: `https://doi.org/10.1080/00051144.2022.2042115`.

Conference papers

1. **B. Pervan** and J. Knezovic, "A Survey on Parallel Architectures and Programming Models," in 2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO), Sep. 2020, pp. 999–1005, doi: `https://doi.org/10.23919/MIPRO48935.2020.9245341`.

2. **B. Pervan**, J. Knezovic, and K. Pericin, “Distributed Password Hash Computation on Commodity Heterogeneous Programmable Platforms” 2019, [Online]. Available: https://www.usenix.org/conference/woot19/presentation/pervan.
3. **B. Pervan**, E. Guberovic, and F. Turcinovic, “Hazelnut - An Energy Efficient Base IoT Module for Wide Variety of Sensing Applications,” in Proceedings of the 6th Conference on the Engineering of Computer Based Systems, Sep. 2019, pp. 1–4, doi: https://doi.org/10.1145/3352700.3352702.
4. L. Strizic, **B. Pervan**, and J. Knezovic, “Deep Learning Accelerator on Programmable Heterogeneous System with RISC-V Processor,” in 2019 proceedings of the 42nd international convention MIPRO, 2019, pp. 1126–1131, [Online]. Available: https://www.bib.irb.hr/1021661/download/1021661.16_cts_5493.pdf.
5. J. Knezovi ć, **B. Pervan**, Z. Relja, and J. Knezović, “Project Houseleek - A Case Study of Applied Object Recognition Models in Internet of Things,” in 2019 proceedings of the 42nd international convention MIPRO, 2019, no. May, pp. 1051–1055, [Online]. Available: https://www.bib.irb.hr/1021658/download/1021658.04_cts_5327.pdf.

Posters

1. **B. Pervan**, J. Knezovic, and M. Steuwer, "Heterogeneous Hardware Programming Made Easy - An Extensible Compiler Targeting Heterogeneous Hardware", 17th International Summer School on Advanced Computer Architecture and Compilation for High-performance Embedded Systems, Sep. 2021

Životopis

Branimir Pervan je rođen u Zagrebu gdje je završio osnovnu i srednju školu. Diplomski studij završio je 2015. g. na Sveučilištu u Zagrebu, Fakultet elektrotehnike i računarstva. Tijekom te kratko nakon završetka studija radio je kao programski inženjer u više različitih tvrtki. 2016. godine započeo je posljediplomski studij u području raznorodnog računarstva. Njegovi znanstveni i profesionalni interesi sežu od paralelizma, raznorodnih računalnih sustava, jezika za specifične domene, funkcijskog programiranja, pa sve do ugradbenih sustava. Za vrijeme rada na doktorskom studiju, poduzeo je više posjeta istraživačkim skupinama sličnoga interesa, te je pohađao dvije ljetne škole u području računarstva visokih performansi. Od ožujka 2021. do studenog 2021. blisko je surađivao s istraživačkom skupinom sa Sveučilišta u Edinburghu radi razvoja praktičnih ishoda doktorskog rada. Suradnik je tvrtke Green Light Technologies gdje povremeno radi kao programski inženjer i konzultant. Kroz studij pa sve do sada, bio je u upravljačkim tijelima različitih volonterskih organizacija. Član je strukovnih udruženja IEEE, IEEE Computer Society i HiPEAC. Autor je ili koautor više znanstvenih radova objavljenih u međunarodnim časopisima ili na konferencijama.