CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Design och implementation av ett GPU-programspråk med högre ordningens funktioner

Kasper Fång Wiik
Alex Janum
Hanna Kron
Rachel Samuelsson
Spyridon Siarapis
Wilmer Zetterqvist

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2025

Kandidatarbete 2025

Design och implementation av ett GPU-programspråk med högre ordningens funktioner

Kasper Fång Wiik
Alex Janum
Hanna Kron
Rachel Samuelsson
Spyridon Siarapis
Wilmer Zetterqvist

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
Göteborgs universitet
Göteborg, Sverige 2025

Design och implementation av ett GPU-programspråk med högre ordningens funktioner

Kandidatarbete vid Datateknik, Chalmers

Kasper Fång Wiik, Alex Janum, Hanna Kron, Rachel Samuelsson, Spyridon Siarapis, Wilmer Zetterqvist

© Kasper Fång Wiik, Alex Janum, Hanna Kron, Rachel Samuelsson, Spyridon Siarapis, Wilmer Zetterqvist 2025

Handledare: Ulf Assarsson, Institutionen för data- och informationsteknik
Medrättande lärare: Magnus Myreen, Institutionen för data- och informationsteknik
Examinator: Patrik Jansson, Institutionen för data- och informationsteknik

Kandidatarbete 2025
Institutionen för data- och informationsteknik
Chalmers tekniska högskola and Göteborgs universitet
SE-412 96 Göteborg
Telefon +46 31 772 1000

Typsatt med TYPST
Göteborg, Sverige 2025

Design and Implementation of a GPU Programming Language with Higher Order Functions

Kasper Fång Wiik, Alex Janum, Hanna Kron, Rachel Samuelsson,
Spyridon Siarapis, Wilmer Zetterqvist,

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

A new programming language, `gpulang`, is introduced together with its abstractions that enable automatic parallelisation of programs. Central to the language is the concept of *streams*, an abstract data type that may only be inspected through built-in higher-order functions such as `map`, `reduce`, and `filter`. This restricted access model is intended to facilitate optimisation by the compiler. We describe the design of the language as well as the implementation of its compiler, which includes code generation for NVIDIA and AMD GPUs using the existing HIP framework. Preliminary performance measurements suggest that `gpulang` achieves performance comparable to the functional GPU language Futhark, but trails behind the Thrust C++ library.

Keywords: compiler, GPU, parallelisation, programming language, streams

Design och implementation av ett GPU-programspråk med högre ordningens funktioner

Kasper Fång Wiik, Alex Janum, Hanna Kron, Rachel Samuelsson,
Spyridon Siarapis, Wilmer Zetterqvist,

Institutionen för data- och informationsteknik
Chalmers tekniska högskola och Göteborgs universitet

Sammandrag

Ett nytt programspråk, `gpulang`, introduceras tillsammans med abstraktioner som möjliggör automatisk parallellisering av program. Centralt i programspråket är konceptet *streams*, en abstrakt datatyp som enbart kan inspekteras med hjälp av inbyggda funktioner av högre ordning som `map`, `reduce`, och `filter`. Denna begränsade åtkomstmodell uppmuntrar är avsedd att underlätta vid optimering i kompilatorn. Vi presenterar språkets design samt implementationen av dess kompilator, som inkluderar kodgenerering för GPU-hårdvara från NVIDIA och AMD genom det befintliga ramverket HIP. Preliminära prestandamätningar ger vissa indikationer på att `gpulang` uppnår jämförbar prestanda med det funktionella GPU-språket Futhark, men faller långt bakom C++-biblioteket Thrust.

Nyckelord: kompilator, GPU, parallellisering, programspråk, streams

Förord

Gruppen vill rikta ett stort tack till vår handledare Ulf Assarsson för sitt stöd kring arbetet, och för att han genom hela processen uppmuntrat oss och trott på gruppens förmåga.

Rachel skulle även tacka sina nära och kära för att de tatt hand om henne genom den stress och ångest som pågått senaste halvåret. Utan deras hjälp hade det ej varit möjligt för henne att fullborda arbetet.

Kasper Fång Wiik, Alex Janum, Hanna Kron, Rachel Samuelsson,
Spyridon Siarapis, Wilmer Zetterqvist,
Göteborg, maj 2025

Ordlista

affina typer	(en: *affine types*) ett typsystem där varje datatyp får användas högst en gång; jämför med linjära typer
backend	hos en kompilator: samlingsnamn för de steg som utförs för att omvandla ett abstrakt syntaxträd till exekverbar kod, samt optimering av denna kod
codegen	kodgenerering, efter sammandragning av en: *code generation*
device	i CUDA, HIP, OpenCL: den delen av programmet som körs på GPU:n
fixed function pipeline	ett utdaterat sätt att designa grafikprocessorer där bara fördefinierade transformationer kan utföras
frontend	hos en kompilator: samlingsnamn för lexer, parser, och type-checker
fält	(en: *array*) en samling element som kan identifieras med ett index
heterogena beräkningar	(en: *heterogeneous computing*) beräkningar som utförs av system som använder två eller flera olika sorters processorer
host	i CUDA, HIP, OpenCL: den delen av programmet som körs på CPU:n
imperativt	(en: *imperative*) en stil av programspråk där programmet definieras som en serie av operationer
indata	(en: *input*) data som ges till en funktion eller till ett program för att bearbetas
interoperabilitet	(en: *interoperability*) olika sorters systems förmåga att förstå och kommunicera med varandra
iterator	en datastruktur som innehåller element som går att iterera över, såsom en lista eller en mängd
kernel	(även en: *compute kernel*) i beräkningsteori: en subrutin som körs på en accelerator
latens	tidsintervallet mellan det att en instruktion schemaläggs och att den utförs och returnerar ett resultat
lexikalanalys	processen att bryta ned en text i dess beståndsdelar—så kallade tokens—samt att kategorisera dessa beståndsdelar
lexikalanalysator	(en: *lexer*) ett program som utför lexikalanalys
linjära typer	(en: *linear types*) ett typsystem där varje datatyp måste användas exakt en gång; jämför med affina typer
parserkombinator	en funktion som sätter samma flera syntaxanalyserare och/eller parser-kombinatorer
pass	räkneord för iterationssteg inom kompilering
prestandamätning	(en: *benchmark*) utvärdering av prestandan hos hårdvara eller mjukvara med hjälp av väldefinierade tester

primitiv datatyp	grundläggande datatyper såsom heltal och flyttal, som agerar byggstenar till mer komplexa datatyper
procedurellt	(en: *procedural*) en stil av programspråk där programmet delas upp i funktioner som anropar varandra och kan mutera programtillståndet
semantisk analys	(en: *type-checking*) processen att verifiera att inga datatyper används på fel sätt i ett program
semantisk analysator	(en: *type-checker*) ett program som utför semantisk analys
shader	ett program som bearbetar antingen vertex eller pixlar inom grafiska beräkningar
stream	i programspråket `gpulang`: en ordnad homogen datamängd av godtycklig storlek, som till skillnad från en array enbart går att interagera med genom fördefinierade stream-operationer
syntax-analys	(en: *parsing*) processen att beskriva hur de olika beståndsdelarna i en text samverkar enligt systematiska regler
syntax-analysator	(en: *parser*) ett program som utför syntaxanalys
token	ett par av en sträng och ett kategorinamn som beskriver vilken funktion strängen har i sitt sammanhang
vertex	inom geometri: ett hörn av en polygon, eller en punkt i en kurva där krökningen är maximal eller minimal

Förkortningar

API	(en: *application programming interface*) en specifikation för hur olika program kan kommunicera med varandra
AST	(en: *abstract syntax tree*) en trädstruktur som beskriver hur konstrukt såsom `if-else` och `assign` innesluter varandra i en hierarki
CPU	(en: *central processing unit*) huvudprocessorn i en dator
CUDA	(en: *compute unified device architecture*) NVIDIA:s plattform för att utföra beräkningar på GPU-hårdvara
GPGPU	(en: *general purpose computing on graphics processing units*) användandet av GPU:er för icke-grafiska ändamål som traditionellt har utförts av en CPU
GPU	(en: *graphics processing unit*) en sidoprocessor, även kallad accelerator, specialiserad för grafiska och vektoriserade beräkningar
HIP	(en: *heterogeneous-compute interface for portability*) en dialekt av C++ som utvecklas av AMD och är kompatibel med både CUDA och ROCm
HPC	(en: *high performance computing*) högprestandaberäkningar
IR	(en: *intermediate representation*) intern representation av en källkod som används av en kompilator eller en VM
OpenCL	(en: *open computing language*) en öppen standard för att skriva plattformsoberoende kod för parallell programmering
ROCm	(en: *Radeon open compute platform*) AMD:s plattform för att utföra beräkningar på GPU-hårdvara
SIMD	(en: *single instruction, multiple data*) en och samma tråd kör instruktioner som behandlar flera dataelement parallellt; se Figur 2 för visualisering
SIMT	(en: *single instruction, multiple threads*) flera trådar kör en och samma instruktion parallellt för olika indata; se Figur 2 för visualisering
SPIR-V	(en: *standard portable intermediate representation*) en binärrepresentation för program skrivna i bland annat OpenCL
VM	(en: *virtual machine*) mjukvara som emulerar en fysisk dators beteende

Innehållsförteckning

Bilagor

Figurförteckning

Tabellförteckning

Förteckning över kodstycken

In fact what I would like to see is thousands of computer scientists let loose to do whatever they want. That's what really advances the field.

— ofta tillskrivet Donald Knuth

1
Introduktion

Grafikkortet har blivit en allt viktigare del av högprestandaberäkningslandskapet. I den klassiska halvårliga TOP500-listans statistik över de snabbaste superdatorerna—synliggjord i Figur 1a—syns en tydlig trend, där allt fler system har en grafikaccelerator [1]. Det huvudsakliga skälet till att allt mer specialiserad hårdvara blivit relevant inom högprestandaberäkning (HPC, en: *high performance computing*) kan relativt tydligt härledas till halvledarindustrin. Omkring 2004 kan det sägas att Dennardskalning[1] slutade gälla [3]. Detta kan exemplifieras med en jämförelse över hur klockfrekvens tidigare ökat exponentiellt i kommersiella mikroprocessorer, men att denna tillväxt har stagnerat under de senaste 20 åren, vilket tydligt visas i Figur 1b.

Då enkeltrådad prestanda tidigare ökat exponentiellt över tid var det inte lika självklart att använda mer specialiserade acceleratorer lämpade för specifika beräkningar, särskilt av ekonomiska skäl[2]. När den enkeltrådade prestandan inte längre skalade på samma sätt som tidigare ändrades förutsättningarna, vilket har resulterat i att acceleratorer blivit allt mer viktiga för fortsatt tillgång till högre datorprestanda. Det gäller likväl inom HPC-världen som inom den allmänna marknaden, där grafikacceleratorer, antingen integrerade i CPU:n (en: *central processing unit*) eller som diskreta enheter, finns i nästintill alla moderna smartmobiler, bärbara datorer, och stationära arbetsstationer (en: *desktop workstations*).

1.1 Syfte

Syftet med projektet är att utforska processen att skapa ett programspråk som är ämnat för att accelerera beräkningar på en GPU (en: *graphics processing unit*), och som har ett typiskt utbud av funktioner av högre ordning (en: *higher order functions*) för att manipulera data. Detta innebär att designa ett språk, samt att skriva en kompilator för att generera exekverbar kod för program skrivna i detta språk. De tänkta grundprinciperna för språket kan sammanfattas som:

- Det ska vara ett procedurellt språk, det vill säga att ett program som består av funktioner som kallar på varandra och kan mutera data.
- Det ska ge användaren kontroll över minneshantering.
- Det ska ge kompilatorn möjlighet att resonera kring sammansättningar av dessa högre ordningens funktioner i minnes- och kodoptimeringssyfte.

1.2 Mål

Slutprodukten är en beskrivning av programspråkets design och funktionalitet, samt en kompilator för språket. Utöver detta skall ett antal testprogram skrivas och användas för att utvärdera designval och prestanda.

[1] Dennardskalning är observationen att effektdensiteten i integrerade halvledarkretsar förblir konstant då transistordensiteten ökar. Detta har möjliggjort mer energieffektiva kretsar med högre klockfrekvenser. Konceptet introducerades 1974 i [2].

[2] Exempelvis hårdvaruutvecklingskostnader för acceleratortillverkaren, mjukvaruutvecklingskostnader för de som utvecklar mjukvara att köra på samt trenden att CPU:er inom ett par år kommer matcha acceleratorns prestanda. Det var inte en självklar investering.

1.3 Historia

Ursprungligen var grafikacceleratorn, precis som namnet antyder, designat för grafikberäkningar. Under 90-talet var hårdvaran formgiven för just denna typ av problem, och en så kallad fixed function pipeline användes för att göra grafikberäkningarna. Detta innebar att hårdvaran i princip bara tog in polygoners hörnpunkter—vertex—samt viss övrig information, transformerade dessa vertexars position och egenskaper så som färg, för att sedan rasterisera polygonerna och slutligen köra program kallade shaders för att beräkna pixel-färgvärden [4]. Det listiga här är att beräkningarna inom ett steg är oberoende av varandra, och därför kan köras parallellt på hårdvarunivå.

Dessa steg i processen att generera grafik var från början inte programmerbara, utan hade bara ett begränsat antal inställningar. Först under 2000-talet började programmerbarheten öka, och flera av stegen började köras på samma allt mer generella processorer i grafikkortet. Denna övergång motiverades av flera skäl, däribland att balansera beräkningsbördan över olika delar av GPU:ns olika steg (en: *pipeline*), som belastas till olika grad av olika sorters datorprogram, och ökande kostnader för att designa datorkretsar med allt högre klockfrekvenser [4]. Då GPU:n gått allt närmare en enormt parallell programmerbar vektormaskin, och även börjat stödja datatyper som flyttal, har allt fler beräkningar kunnat flyttas till GPU:n.

Ett tidigt exempel är en implementation av en matrismultiplikationsrutin för stora matriser i grafik-API:et OpenGL från 2001, som demonstrerade att det sannolikt, på framtida GPU:er med flyttalsstöd, skulle vara betydligt snabbare med matris-matris-multiplikationer i 32-bitars flyttalsprecision på GPU:n än CPU:n [5]. Att använda grafikkort för annat än de grafikberäkningar som de ursprungligen designades för kallas för GPGPU (en: *general purpose computing on graphics processing units*), och är idag ett stort användningsområde för dessa acceleratorer.

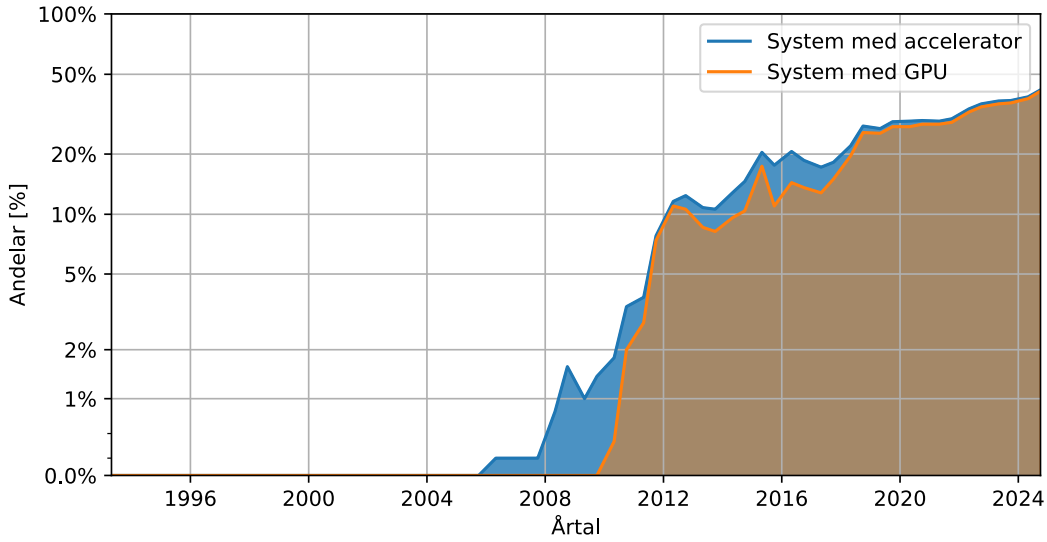
1.4 Befintliga GPU-programspråk

Det är huvudsakligen två andra projekt som har inspirerat detta projekt: programspråket Futhark, och NVIDIA:s C++-bibliotek Thrust.

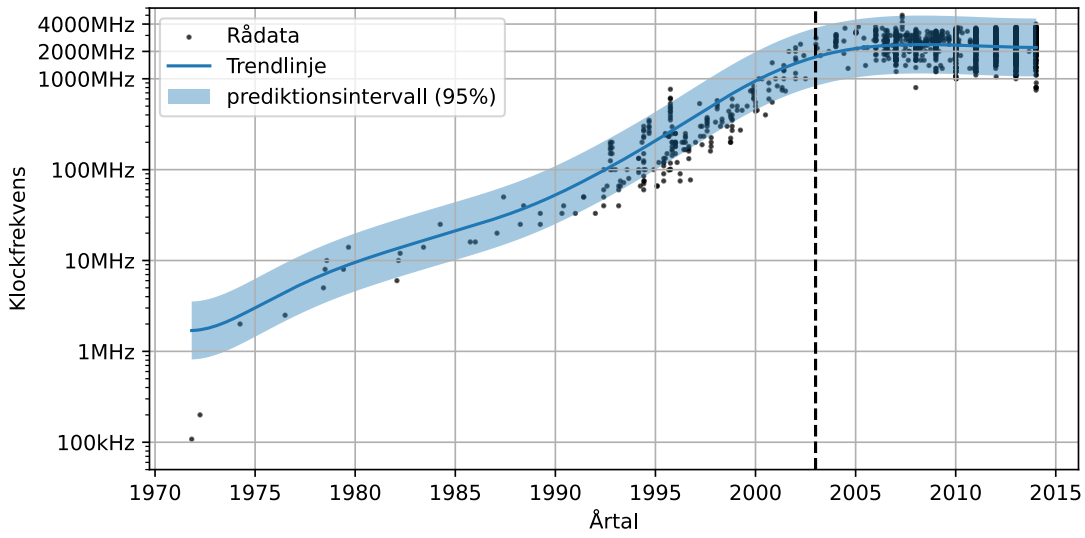
Futhark är en del av ett pågående forskningsprojekt på Köpenhamns universitet. Språket beskrivs som ett "statiskt typat, dataparallellt, och rent funktionellt arrayspråk i ML-familjen" [7]. I skrivande stund kan Futharks kompilator generera kod för AMD:s HIP (en: *heterogeneous-compute interface for portability*), NVIDIA:s CUDA (en: *compute unified device architecture*), KHRONOS:s OpenCL-standard, samt även enkel- och flertrådad CPU-kod. Kod i detta programspråk kan alltså köras på många olika acceleratorer och även på CPU:er. Futhark är designat för GPGPU-beräkningar, och är främst tänkt som ett komplement till andra befintiga språk för små men beräkningsintensiva delar av en applikation, snarare än att kompletta program skrivs direkt i språket [7].

Kodbiblioteket Thrust erbjuder parallelliserade algoritmer och datastrukturer, och exponerar dessa genom en C++-API inspirerad av STL[5] [8]. Liksom Futhark kan även detta projekt köras via olika system för beräkningsacceleration, främst NVIDIA:s CUDA, Intel:s oneTBB, och OpenMP [9]. Eftersom Thrust är ett C++-bibliotek kan det enklare integreras mer direkt i C++-kodbaser än Futhark, som är ett eget programspråk.

[5] STL (en: *standard template library*) är en del av standardbiblioteket i C++. Thrust har till exempel både en variant av `std::vector` som backas av GPU-minne samt exponerar de iteratorer som typiskt medföljer en sådan struktur.



(a) andel datorer på TOP500-listan med accelerator



(b) CPU:ers klockfrekvens över tid

Figur 1: Hårdvarutrender över tid. I delfigur (a) visas statistik över grafikacceleratorer i TOP500-listan med de idag 500 snabbaste superdatorerna[3]. Notera den uppåtgående trenden med avseende på förekomst av accelerator, och speciellt grafikacceleratorer. I delfigur (b) visas CPU-modellers klockfrekvenser över tid, med data från Stanfords `cpudb`-dataset [6]. Trendlinjen är vår egen[4].

[3]Datorerna i TOP500-listan rangordnas utifrån sina respektive resultat på `LINPACK`-benchmarken.

[4]Trendlinjen $\texttt{clock} = f(t)$ uppskattas genom att applicera ett Gaussiskt filter med standardavvikelse 1095 dagar (3 år) efter linjär omsampling av datan för att få ekvidistanta punkter längs tidsaxeln. Många processorer är satta på samma datum och för dessa tas ett medelvärde av deras respektive klockfrekvenser vid omsamplingen. Prediktionsintervallet för varje datapunkt $(\texttt{clock}_i, t_i)$ tas utifrån modellen $\texttt{clock}_i = f(t_i)\varepsilon_i$ för log-normal-fördelad ε_i med modellparameter σ_ε, dvs. $\varepsilon_i = \exp(z_i)$ med $z_i \sim \mathcal{N}(0, \sigma_\varepsilon^2)$.

Både Futhark och Thrust bygger på liknande idéer. Program i Futhark består av sammansatta array-operationer som verkar på array:er. Dessa array-operationer är sammansättningar av högre ordningens funktioner såsom `filter` (för att ta bort element som inte matchar ett predikat), `reduce` (reduktion), och `map` (för att applicera en funktion elementvis på samtliga element). Futhark exponerar praktiskt taget inga minnes-hanteringsdetaljer till programspråkets användare [10].

Thrust bygger också på operationer med liknande form som i Futhark. Däremot får programmeraren även mer ansvar och kontroll över lågnivådetaljer i Thrust. Exempel på operationer i Thrust är `thrust::scan_inclusive` (inklusiv prefixsumma), `thrust::transform` (motsvarande `map` i Futhark), och `thrust::reduce` (generisk parallell reduktion som antar associativitet för den valda reduktionsoperatorn) [8]. Denna metod för att bygga parallelliserbara program genom att sätta samman operationer på array:er har varit en huvudsaklig inspiration för det programspråk som designats i detta kandidatarbete.

2
Teori

I detta avsnitt beskrivs den teori som ligger till grund för arbetet. Avsnittet ger först en beskrivning av det moderna grafikkortet, för att sedan ge en kort introduktion till relevant kompilatorteori.

2.1 GPU-teori

Grafikkortet, eller GPU:n (en: *graphics processing unit*), har en arkitektur som bygger på hög parallellism med många kärnor som var och en kör breda[6] vektorberäkningar parallellt. Beräkningarna hämtas från ett flertal samtida instruktionsströmmar per kärna, istället för ett fåtal snabbare kärnor med enbart 1 eller 2 samtida instruktionsströmmar per kärna. GPU:n är alltså formgiven för dataparallella problem, och kan typiskt leverera prestandaförbättringar på 100x eller mer jämfört med motsvarande lösning på CPU:n [4]. Detta avsnitt ger först en kort introduktion till GPU:ers programmeringsmodell med fokus på SIMT och SIMD, för att sedan ge en kortare överblick över GPU-hårdvara, slutligen följt av de mjukvarugränsnitt som används för att programmera dagens GPU:er.

2.1.1 Programmeringsmodell

Vanligtvis programmeras GPU-hårdvara, sett till den lägsta abstraktionsnivå som tydligt exponeras, med en av två programmeringsmodeller: SIMT eller SIMD. SIMT (en: *single instruction*, *multiple threads*) innebär att trådar i princip enbart kör skalära instruktioner[7] medan SIMD (en: *single instruction*, *multiple data*) istället innebär att trådar utför vektorinstruktioner. En SIMD-maskin kan dock ses som en SIMT-maskin om samtliga SIMD-lanes ses som distinkta SIMT-trådar. Det är så GPU-hårdvara typiskt är utformad, och att exponera den underliggande SIMD-arkitekturen som en SIMT-arkitektur ger ofta mer frihet till GPU-tillverkarens kompilator, både vad gäller kompatibilitet mellan olika GPU-modeller samt optimeringsmöjligheter vid kontrollflödesdivergens[8].

De två stora GPU-tillverkarna, NVIDIA och AMD, har gjort något olika val kring hur SIMT- och SIMD-modeller av hårdvaran exponeras. NVIDIA-hårdvara har valt att (i princip) bara exponera en SIMT-modell, medan AMD:s istället ger möjlighet att programmera antingen med en SIMT-modell (via ROCm/HIP) eller en SIMD-modell (via GCN-ISA[10]). Vanligtvis är dagens SIMT-modell en hybrid mellan SIMT och SIMD, där data inom SIMD-bredden kan delas mellan de trådar som ingår i samma SIMD-grupp med trådar[11] för att accelerera *horisontella operationer*. Vid SIMD-beräkningar (samt vid beräkningar med CUDA/HIP-block) är det viktigt att skilja mellan *vertikala* och *horisontella* operationer.

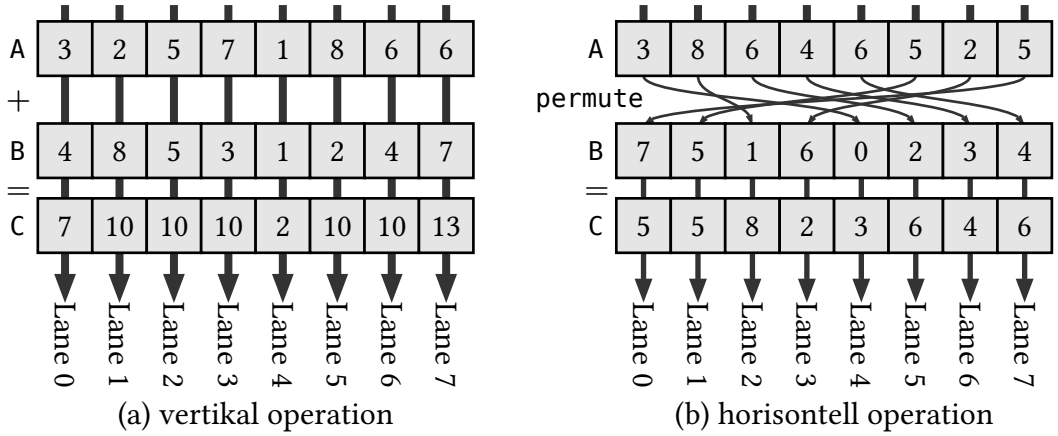
[6] Med bredd menas här antalet *lanes*, eller dataelement, som behandlas med en och samma processorinstruktion.

[7] Skalära instruktioner verkar enbart på skalärer, till skillnad från vektorinstruktioner som istället verkar på vektorregister där operationer appliceras elementvis på samtliga vektorelement i en och samma instruktion.

[8] En frihet som SIMT-modellen erbjuder i CUDA kan ses vid hantering av divergenta grenar (en: *divergent branches*) i en och samma CUDA-*warp*, som sedan NVIDIA Volta-mikroarkitekturen kan köras omlott (en: *interleaved*) med det som NVIDIA kallar *independent thread scheduling*. Divergens hanteras alltså genom att en *warp* delas upp i oberoende delar som senare återförenas. Se NVIDIA:s whitepaper [11] om Volta för detaljer.

[10] GCN-ISA (en: *graphics core next instruction set architecture*) är AMD:s instruktionsuppsättning sedan AMD:s GPU-familj under namnet *Southern Islands*, som släpptes först 2012.

[11] Med detta avses det som i HIP/CUDA kallas för *warp*, och ofta även *wave32/64* i AMD:s grafikkortsarkitekturer.



(a) vertikal operation (b) horisontell operation

Figur 2: Exempel på dataflödet i en vertikal operation i (a) (elementvis addition) och en horisontell operation i (b) (permutation[9]). Pilarna går längs databeroendena i de respektive beräkningarna. Här ordnas respektive lane med första lanen, även kallad lägsta lanen, till vänster med index 0 medan sista (högsta) lanen visas längst till höger. I diagram är även motsatt lane-ordning, med den högsta lanen till vänster, också vanlig.

Vertikala operationer kännetecknas av att operationer sker elementvis; olik-indexerade element har aldrig databeroenden på varandra och påverkar således inte varandra. Horisontella operationer kännetecknas av motsatsen, att databeroenden finns mellan olik-indexerade element. I Figur 2 exemplifieras detta med ett exempel på en vertikal operation och ett exempel på en horisontell operation. Distinktionen mellan vertikala och horisontella operationer är viktig eftersom databeroenden trådar emellan i princip alltid kräver viss koordination, vilket allt som oftast kostar i prestanda.

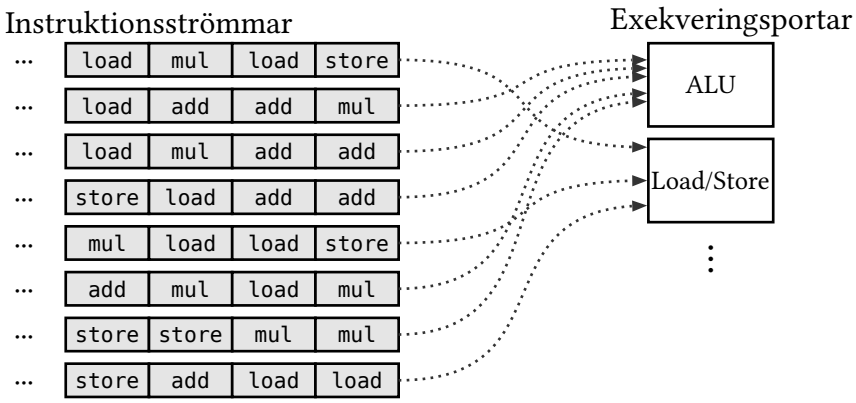
2.1.2 GPU-hårdvara

GPU-hårdvara skalar väl till många kärnor, eftersom den är designad för att exekvera tusentals trådar parallellt för enkla och likformiga beräkningar. Dess arkitektur prioriterar hög genomströmning snarare än låg latens. Just latens är annars huvudproblemet för CPU:ers prestanda. Att inte behöva optimera för latens medför flera fördelar. Till att börja med kan instruktioner att exekvera plockas från många instruktionsströmmar istället för att, som i CPU-fallet, lägga stora resurser på instruktionsparallell exekvering för enbart en eller två instruktionsströmmar, utan att det kostar i hårdvarans utiliseringsgrad. Figur 3 visar samtida exekvering av flera instruktionsströmmar som tilldelas olika exekveringsportar för exekvering. På moderna NVIDIA-GPU:er kan högst två instruktioner plockas (en: *dispatch*) per klockcykel [12]. Vidare kan GPU:er byggas av många enkla, massproducerbara, billiga, och lågklockade kärnor istället för ett fåtal komplicerade, och högklockade (typiskt >3GHz) kärnor. På så sätt kan GPU:er skalas upp till betydligt mer beräkningskapacitet än CPU:er, men på bekostnad av enkeltrådad prestanda.

2.1.3 GPGPU-Plattformar

I industrin används i praktiken huvudsakligen bara ett fåtal plattformar för generella beräkningar på GPU. Absolut vanligast är NVIDIA:as CUDA-plattform, följt av AMD:s ROCm/HIP-plattform, och därefter OpenCL-standarden.

[9]Operationen `permute`$(A, B) := \left(a_{b_0}, a_{b_1}, .., a_{b_{N-1}}\right)$ för godtycklig vektor $A = (a_0, .., a_{N-1})$ och giltiga index $B = (b_0, .., b_{N-1})$ in i A. Här är N *SIMD-bredden*, vilket är antalet lanes i ett vektorregister.



Figur 3: Kärnor (de som NVIDIA kallar SM:er, från en: *streaming multiprocessor*) i GPU:er plockar typiskt instruktioner från flera instruktionsströmmar och har flera instruktioner in flight på olika exekveringsportar. Figuren visar vilka instruktioner som kan gå till vilka portar.

CUDA är NVIDIA:s GPGPU-plattform, och är i princip NVIDIA-specifikt [13]. Instruktionsformatet för CUDA kallas PTX, vilket ofta genereras med NVIDIA:s CUDA-kompilator `nvcc` [4].

AMD:s plattform för heterogena beräkningar mellan CPU- och GPU-hårdvara kallas ROCm [14]. Ekosystemet stödjer flera olika sätt att programmera GPU:n, såsom OpenCL och HIP. HIP är ett API likt CUDA, men med högre portabilitet då det kan använda antingen ROCm eller CUDA-plattformen för att bygga applikationer [15].

I motsats till de ovan nämnda plattformsberoende alternativen CUDA och ROCm är OpenCL en KHRONOS-standard som öppet kan implementeras av olika hårdvarutillverkare på marknaden. Både AMD och NVIDIA implementerar OpenCL-standarden. Standarden är mindre formad efter GPU:er och är därför svårare att använda då maximal prestanda vill uppnås. Detta kan tänkas vara en stor del av förklaringen till att CUDA har en betydligt större marknadsandel.

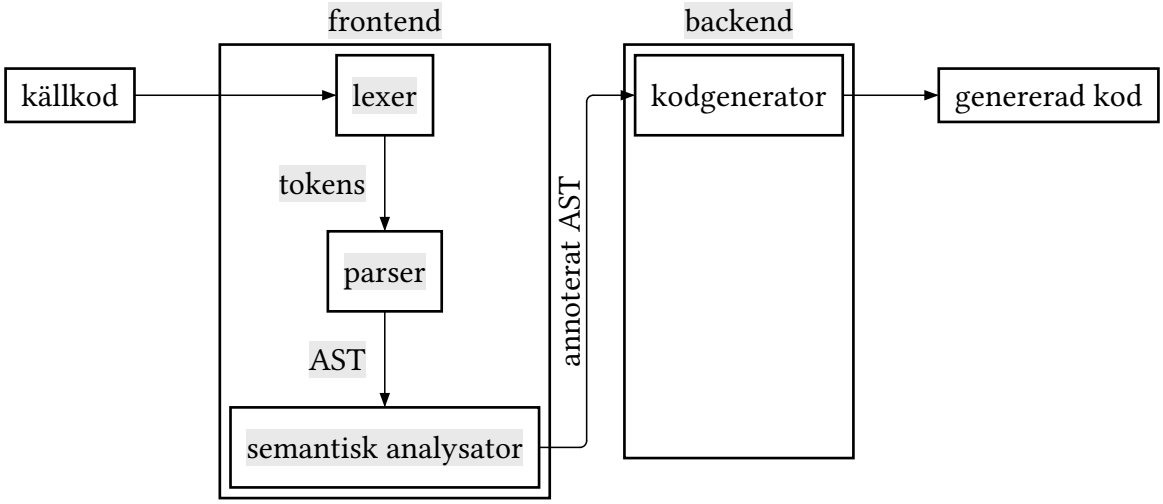
2.2 Principer för kompilatorer

Kompilatorer är program som konverterar källkod till någon annan sorts kod, vanligen maskinkod eller andra körbara representationer. Ibland kan kompilatorer istället för att kompilera ner till ett mer direkt körbart format som maskinkod generera källkod till ett annat programspråk. Ett exempel på detta är TypeScript som kompileras till JavaScript.

En kompilator består av ett antal steg, eller pass[12] (en: *passes*) som de ofta kallas, där utdata från passet innan är indatan till passet efter [16, s. 5]. De typiska passen är

1. En lexer, som styckar upp koden som kompileras i så kallade tokens. Några exempel på tokens är variabelidentifierare, tal, och strängar.
2. En parser som sätter samman dessa tokens till ett *syntaxträd*, ett så kallat AST (en: *abstract syntax tree*).

[12]Ibland, om de olika passen kör samtidigt och inte ett i taget, sägs kompilatorn vara enkelpassig (en: *single pass*). Exakt vad som utgör ett pass är inte entydigt, men vi har valt att anamma denna terminologi för de olika stegen i vår *kompilatorpipeline*. En *pipeline* är helt enkelt ett system där data processeras stegvis och utdatan från föregående steg blir indatan till det nästa.



Figur 4: Ett förenklat diagram som illustrerar de olika pass som en typisk kompilator består av.

3. En semantisk analysator som ser att det nyss parsade syntaxträdet beskriver ett giltigt program, och annoterar detta med information som vilka variabler givna variabelidentifierare åsyftar och härleder programmets datatyper.
4. Slutligen, en kodgenerator som genererar kompilatorns slutliga utdata, dvs. någon form av ny kod, utifrån detta annoterade AST.

Pass 1–3 ovan brukar tillsammans kallas för frontend, medan pass 4 kallas för backend. Läsaren ska inte missledas av att backend här beskrivs som ett enda pass; kodgeneration är den överlägset mest svåra och komplicerade delen av en optimerande kompilator. De olika passen visas i Figur 4. I de följande avsnitten beskrivs dessa olika typiska kompilatorpass.

2.2.1 Lexikalanalys

Ett program som utför lexikalanalys kallas för en lexikalanalysator, eller en lexer (från en: *lexing*). Denna process implementeras ofta med hjälp av reguljära uttryck (en: *regular expressions*, "*regex*") och ändliga automater [16, s. 109], vars mål är att identifiera textens minsta meningsfulla beståndsdelar, som kallas för *lexem* (en: *lexeme*), eller mer typiskt tokens[13]. För varje identifierat token lagras då vilken sorts token som identifierats, såsom en sträng, ett heltal, eller en variabelidentifierare; samt avkodad information, såsom strängens textinnehåll, heltalets binära representation, respektive variabelidentifierarens text. I detta steg görs typiskt inte några försök att avkoda hur de olika tokens som finns i källkoden relaterar till varandra, utan detta görs i nästa steg: syntaxanalysen.

2.2.2 Syntaxanalys

En syntaxanalysator, eller parser, är ett program som bygger upp en trädstruktur som beskriver relationerna mellan de tokens som tidigare identifierats, ett så kallat (abstrakt) syntaxträd (AST). Det finns många metoder som utvecklats för syntaxanalys, och ett vanligt sätt att kategorisera dessa är som antingen *top-down* eller *bottom-up*, baserat på vilken del av syntaxträdet som genereras först. Att en metod är *top-down* innebär att det genereras uppifrån och ner, det vill säga att trädet successivt fylls på med underordnade noder till dem som redan finns [16, s. 192], medan en *bottom-up*-parser istället

[13]Inom lexikalanalysen finns det på en teknisk nivå en skillnad mellan lexem och tokens, men denna skillnad är inte meningsfull i detta arbete. Därmed används dessa två begrepp synonymt i denna text.

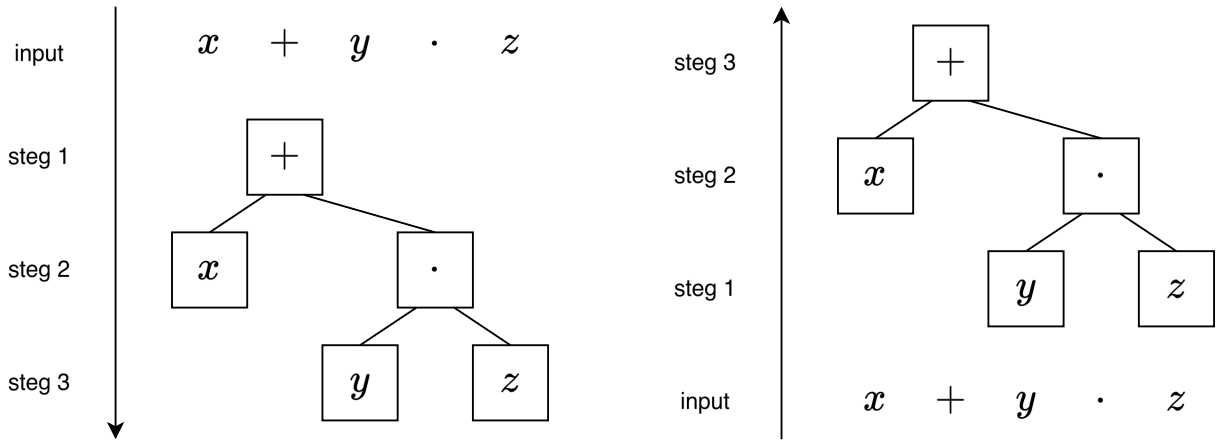
utgår från de understa noderna, och hittar steg för steg hur dessa bör länkas samman. En schematisk jämförelse presenteras i Figur 5.

En vanlig metod för *top-down*-parsing är känd som *recursive descent*. Principen bakom denna metod kan beskrivas som att tokens matchas rekursivt mot alla grammatiskt möjliga strukturer, tills man antingen når en godkänd struktur, eller att alla möjligheter har utarmats, vilket tyder på att programmet som beskrivs inte följer språkets grammatik.

En vanlig metod för *bottom-up*-parsing är den tabelldrivna LR-metoden, där tabellen styr vilken stack-relaterad operation, antingen `shift` eller `reduce`, som bör utföras i varje steg [16, s. 299]. Metodens främsta nackdel är det krävs en hel del arbete att översätta grammatiken för ett typiskt programmeringsspråk till en sådan tabell [16, s. 241], som kan underlättas för språk med så kallad *kontextfri grammatik* med hjälp av så kallade *parser generatorer*.

En annan algoritm för *bottom-up*-parsing är *Floyd's operator-precedence*, där en symboltabell (en: *symbol table*) används för att hierarkiskt tilldela en prioritet till alla operationer i språket. Denna metod följer intuitivt från hur programkod och matematisk notation brukar läsas och beskrivas, där operationer som multiplikation (*) förväntas utföras före operationer som addition (+) i uttryck som `a + b * c`, som då ska läsas `a + (b * c)` och inte som `(a + b) * c`. Denna metod har vidareutvecklats i *Pratt parsing* genom att kombinera Floyds metod med *recursive descent*, vilket istället ger en *top down*-metod [17].

Syntaxanalyseraren identifierar de ingående relationerna mellan tokens som en trädstruktur, men härleder varken (typiskt) vad identifierare refererar till (till exempel vilken specifik variabeldeklaration eller konstant som åsyftas) eller vilka datatyper de olika noderna i syntaxträdet har. Syntaxanalyseraren identifierar inte heller om programmet är semantisk korrekt, vilket vanligtvis handlar om att samtliga identifierare korrekt ska hänvisa till någon annan del av programmet, och att operationer ska göras mellan kompatibla datatyper. Detta görs istället vid semantisk analys.



(a) *top-down* (b) *bottom-up*

Figur 5: Schematisk jämförelse över den huvudsakliga skillnaden mellan *top-down*- och *bottom-up*-metoder inom syntaxanalys, med avseende på vilken del av syntaxträdet som genereras först.

2.2.3 Semantisk analys

Semantisk analys utförs genom att rekursivt traversera syntaxträdet och i varje nod kontrollera att programmet följer de typregler som definierats för språket. Typinferens är processen att generera en korrekt typ för ett uttryck medan det kontrolleras att alla deluttryck är korrekt typade [16, s. 386-887].

Hindley-Milner är ett typsystem som tillåter både generiska uttryck och typinferens [18]. Systemet används som grund för flertalet befintliga programspråk, däribland Haskell [19].

Substrukturella typsystem gör det möjligt att begränsa tillgången till resurser och deras operationer. Detta görs oftast genom att definiera en ny abstrakt datatyp [20]. Ett substrukturellt typsystem av intresse för projektet är linjära typsystem baserade på Jean-Yves Girards linjära logik [21]. En linjär datatyp kännetecknas av att den kan bara användas en gång, den kan inte dupliceras, och den måste användas vid något tillfälle [20].

Detta innebär att varken kopior eller överlappande referenser är tillåtna, vilket bland annat garanterar riskfri frigöring och därav också muterande av data, eftersom det då är känt att datan inte kommer återanvändas [21]. Linjära typer kan inte kastas bort, vilket här innebär att värden med sådana datatyper både måste bli explicit allokerade och konsumerade [20]. En generalisering av linjära typsystem är affina typsystem, där linjära datatyper inte längre med nödvändighet måste konsumeras. Detta motiverades ursprungligen av funktionell programmering, men används nu även i programspråk som Rust för att garantera minnessäkerhet [22].

2.2.4 Kodgenerering

Efter att den första halvan av en kompilator, det som kallas frontend, har producerat ett annoterat AST, tar backenden över. Det är i detta steg som källkoden, nu i form av ett AST, slutligen omvandlas till kompilatorns utdataformat, ofta maskinkod. Denna process genomförs typiskt genom att AST:et först omvandlas till ett intermediärt kodformat, IR (av en: *intermediate representation*). Denna IR-kod kan sedan omvandlas till programkod för kompilatorns målmaskin[14]. Fördelen med IR är tvåfaldig. För det första kan IR som en mellanrepresentation stegvis, upprepade gånger, transformeras för att närma sig ett allt mer *optimerat* program. Med optimerat åsyftas här programkod med kortare körtid och/eller kortare representation (mätt i bytes på målarkitekturen), beroende på vad som önskas. Kompilatorer som på ett eller annat sätt transformerar det ingående programmet för att bli mer optimerat kallas för *optimerande kompilatorer*. I nästkommande delavsnitt ger först en kort beskrivning av IR på kontrollflödesgrafform (*CFG-form*) och på *SSA-form* (detta är inte med i projektets slutprodukt) för att sedan beskriva transpilering (vilket är med i slutprodukten).

2.2.5 Kontrollflödesgrafer (CFG)

Kontrollflödesgrafer, eller CFG:er (av en: *control flow graph*), beskriver programkod inom en och samma funktion. Detta görs genom att funktionen partitioneras in i så kallade *CFG-block* (en: *basic block*). Varje CFG-block är då en maximal sekvens av efterföljande satser eller IR-instruktioner som har ett helt sekventiellt kontrollflöde, det vill säga att varje block exekveras alltid från dess första sats eller instruktion fram till dess sista. När ett CFG-block A körts färdigt fortsätter exekveringen antingen genom att funktionen i fråga returnerar, eller genom att exekveringen fortsätter i något block B, inte nödvändigt distinkt från block A. Att programmet gått från att köra ett block till att köra ett annat kallas för att

[14]Med målmaskin menas här den maskin, eller datorarkitektur, kompilatorns utgående program är tänkt att köra på. Ofta kan en och samma kompilator konfigureras för att producera programkod för en vald maskinarkitektur, till exempel `x64` (även känd som `x86_64` eller AMD64) eller AArch64.

```
int x = 5;
x = x + 5;
int y = 8;
y = x * 7 / y;
if(y == 5){
  x = y;
}else{
  y = y * 5;
}

while(y > 5){
  y = x ^ y;
  x = x * 123;
  x = (x >> 2) ^ x;
}

return x;
```

block A

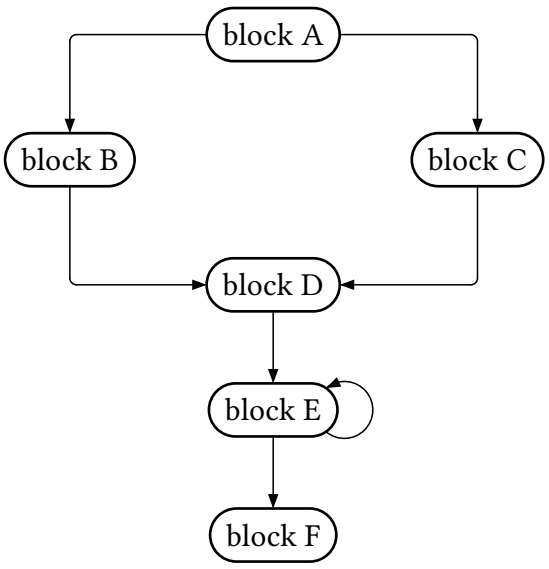
block B

block C

block D (tomt)

block E

block F



Figur 6: Kontrollflödesgraf som beskriver ett enkelt program skrivet i programspråket C. Notera att loopkroppen, block E, antingen hoppar till sig självt eller ut ur loopen till Block F, beroende på villkoret `y > 5`.

programexekveringen *hoppat* (en: *jumped*) från block A till block B. Vilket block B hoppas till bestäms typisk av ett *villkor*. Ett villkor innebär här ett booleskt uttryck, som alltså evaluerar antingen till sant eller till falskt, som sedan entydigt bestämmer nästa programblock. Då villkoret uppfylls (evaluerar till sant) väljs ett förbestämt block B_T (en: *true*), annars väljs ett block B_F (en: *false*). Alla dessa koncept exemplifieras i Figur 6, där kontrollflödesgrafen för en funktionskropp skriven i programspråket C visas. Kontrollflödesgrafer visar sig vara en bra grund för att representera programkod, men bestämmer inte formatet på CFG-blockens ingående instruktioner. Ett av dessa format är SSA.

2.2.6 Single Static Assignment (SSA)

Grundidén bakom SSA (från en: *single static assignment*) är att samtliga programvariabler i IR-representationen, efter att de för första gången tilldelats ett värde, sedan inte längre kan modifieras värdemässigt. Tanken är alltså att ett program, som från början inte har denna egenskap, skrivs om av kompilatorn till en form där varje variabel i den nya representationen enbart tilldelas ett värde en och endast en gång. Dock fungerar denna idé inte i sin direktaste mening. Det räcker att beakta kontrollflödessatser som `if` och `while` för att detta ska inses. Ett förtydligande exempel är en loop, där en yttre variabel tilldelas ett nytt värde beroende på sitt föregående värde vid varje loopiteration. För denna typ av fall introduceras det i SSA-representationen en speciell operation som enbart finns i IR:en: φ-funktionen[15]. Denna funktion, beroende på vilket det föregående CFG-blocket i programexekveringen var, väljer utifrån vilket detta föregående block var vilket SSA-värde som uttrycket ska evaluera till.

För att konkretisera introduceras snabbt syntax för φ-funktionen. Låt $\varphi(x, A; y, B)$ evaluera antingen till x eller y, beroende på om föregående CFG-block var A eller B. På så sätt kan variabler effektivt sett

[15]Det finns andra, ekvivalenta, sätt att se på SSA som inte använder φ-funktioner, men dessa presenteras inte i denna text.

överföra beräknade värden från körningen av ett CFG-block till ett annat. Notera att φ-funktionen även kan hänvisa till det egna CFG-blocket, och funktionens resulterande värde då blir variabelvärdet från den föregående körningen av blocket. Detta kan alltså ses som att varje loopiteration skickar vidare det som beräknats till nästa iteration och slutligen skickar vidare värdena till resten av funktionen då loopiterationen terminerar.

I det här arbetet har varken kontrollflödesgrafer eller SSA-form använts, men dessa är viktiga koncept inom teorin för optimerande kompilatorer. Något som har använts i arbetet är istället *transpilering*, som beskrivs nedan.

2.2.7 Transpilering

Transpilering innebär att en kompilator, hädanefter *transpilator* (en: *transpiler*), istället för att själv emittera maskinkod genererar källkod som en annan kompilator sedan kan behandla. Transpilering kallas ofta för *source-to-source*, eftersom både transpilatorns in- och utdata är källkod. Om båda programspråken—det som transpileras från och det som transpileras till—är någorlunda lika kan detta ofta vara en bra början för en kompilator, då allt från debuginformation till programoptimering nu tillgängliggörs av den existerande miljön. Att från ett AST eller en IR generera C-kod är till exempel sällan svårt, men har en kostnad då kontroll över kompileringsprocessen efter transpileringssteget nu ges till den andra kompilatorn.

3
Teknisk beskrivning

Detta avsnitt ämnar ge läsaren en övergripande förståelse för det nya programspråket `gpulang`, både sett till utformning och implementation. Denna tekniska beskrivning är tänkt att ligga till grund för vidare diskussion kring programspråkets utvecklingsprocess samt de metodval som gjorts under processen.

På en övergripande nivå är `gpulang` ett procedurellt språk, med fokus på GPU-accelererad array-programmering. Denna array-programmering faciliteras dels genom ett nytt koncept, streams, som representerar arrayberäkningar och dels genom funktioner av högre ordning, som `map` och `filter`, som verkar på streams. Detta nya koncept beskrivs närmare nedan. Trots att språket är procedurellt och därmed imperativt, det vill säga att dess programsatser synbart exekverar i den ordning de getts i programkoden, ges stora parallelliseringsmöjligheter i och med att streams inte kan avläses innan de *konsumeras*. Härnäst introduceras streams, som utgör kärnan av detta programspråk, och konsumering av dessa.

3.1 Streams

Ett centralt koncept i språket är en abstrakt datatyp som benämns *stream*. En stream kan liknas vid en lista i den bemärkelse att den är en ordnad homogen datamängd av godtycklig storlek. När ett program kompileras och exekveras är det känt vilken typ av data en stream innehåller, men inte dess storlek eller faktiska innehåll. Till skillnad från en lista är det inte möjligt att inspektera eller modifiera datan som en stream innehåller direkt via index.

Det enda sättet att läsa av eller modifiera dess innehåll är genom fördefinierade *stream-operationer* i språket. Dessa innefattar bland annat `map`, `filter`, och `reduce`, och motsvarar välkända funktioner av högre ordning (en: *higher order functions*) inom bland annat funktionell programmering. Dessa operationer verkar på array:er så väl som streams, och varje operation resulterar då i en ny stream. För att återföra stream-beräkningen på array-form används `collect`-operationen. Tabell 1 visar alla stream-operationer som tillhandahålls av `gpulang`.

Att inte kunna läsa eller modifiera datan direkt innebär att användaren har mindre kontroll över programflödet, men i sin tur innebär det att kompilatorn ges nya möjligheter att optimera programkoden. En datamängd där vissa element väljs bort med en `filter`-operation behöver exempelvis inte ordnas på nytt; istället kan dessa element maskeras bort när programmet körs på GPU-hårdvara.

3.1.1 Konsumering

För att uppnå hög prestanda är det önskvärt att stream-operationer kan agera *in-place* när detta är möjligt, det vill säga utan att skapa en kopia av datastrukturen eller att allokera mer minne. Notera däremot att stream-operationer inte bara har möjlighet att modifiera datan i en stream, utan även dess datatyp (t.ex. genom `map`) eller antal element (t.ex. genom `filter`).

Tabell 1: Lista av inkluderade stream-operationer samt deras typ och funktion. Notationsmässigt avser `[T]` en homogen array som innehåller data av typen `T`, medan `{T}` avser en stream som innehåller data av typen `T`.

Namn	Typ	Beskrivning av operation
map	`fn<T, U>({T}, fn(T): U): {U}`	På varje element i en stream appliceras en funktion som mappar typen `T` till `U`.
map2	`fn<T, U, R>({T}, {U}, fn(T, U): R): {R}`	Elementen i två streams av typerna `T` och `U` kombineras med en funktion som mappar till `R`.
scan	`fn<T>({T}, fn(T, T): T, T): {T}`	Element med index i kombinerar elementen i en stream från index 0 till i genom kumulativ applikation av en given funktion, som beskriver en associativ operation med ett neutralt element.
reduce	`fn<T>({T}, fn(T, T): T, T): T`	Kombinerar alla element i en stream med en given funktion, som beskriver en associativ operation med ett neutralt element.
reduce_c	`fn<T>({T}, fn(T, T): T, T): T`	Liknar `reduce`, men operationen antas vara både associativ och kommutativ.
scatter	`fn<T>({T}, {T}, {u32}): {T}`	Flyttar element från en stream till platser med givna index i en annan stream.
filter	`fn<T>({T}, fn(T): bool): {T}`	Returnerar en ny stream som består av de element som predikatet ger `true` för.
rotate	`fn<T>({T}, i32): {T}`	Elementen i en stream förskjuts. Element som förskjuts ur ena ändan roterar till andra sidan.
collect	`fn<T>({T}): [T]`	Omvandlar en stream till en konkret arraytyp.

När en stream-operation appliceras konsumeras därför alla ingående streams som använts som indata i operationen. En stream kan alltså högst användas en gång. I de fall när samma stream behöver användas flera gånger i ett program är lösningen att först spara den som en array, som i sin tur kan kopieras och användas flera gånger för att skapa flera identiska streams. Metaforiskt kan alltså streams ses som *strömmar* i samma bemärkelse som avses inom vektoranalys; de beskriver hur data flyttas från punkt till punkt i ett källfritt vektorfält. Inom samma metafor så utgör array:er *källor* i ett vektorfält, och stream-operationer som `reduce` och `collect` utgör *sänkor*.

Konsumering är en del av språkdesignen som framtagits för att säkra att programmeraren inte använder ogiltiga data. Logik för att säkerställa att så inte sker har inte implementerats i detta arbete, vilket diskuteras mer i detalj i Avsnitt 6.4.

3.1.2 Notation

Streams är ett centralt koncept i `gpulang` och klammerparenteser (`{}`) har tilldelats för att beteckna en streams datatyp. Detta innebär att syntaxen för typdeklarationer hos funktioner och variabler som innehåller streams motsvarar den för array:er. En stream som innehåller 32-bitars heltal av typen `u32` betecknas alltså `{u32}`, medan en array som innehåller data av samma typ betecknas `[u32]`. Detta är enkelt att förena med det att klammerparenteser även kan designera kodblocks början och slut, eftersom typdeklarationer alltid föregås av kolon (`:`) vilket inte är fallet för kodblock.

3.2 Syntax

Språket har en syntax som inspirerats av flera befintliga programspråk, särskilt Rust, Python, och JavaScript. På programspråkets syntaxmässigt högsta nivå, vilket är alla konstruktioner utom funktioner, konstanter, och struktar, finns möjlighet att importera andra källfiler samt definiera just funktioner, konstanter och struktar. Funktionskroppar i språket byggs upp av satser, och de typer av satser som tillåts i programspråket är bland annat uttryck, variabeldeklarationer och tilldelningssatser, samt flödeskontrollmekanismer som `if`, `for`, `while`, `continue`, `break`, och `return`.

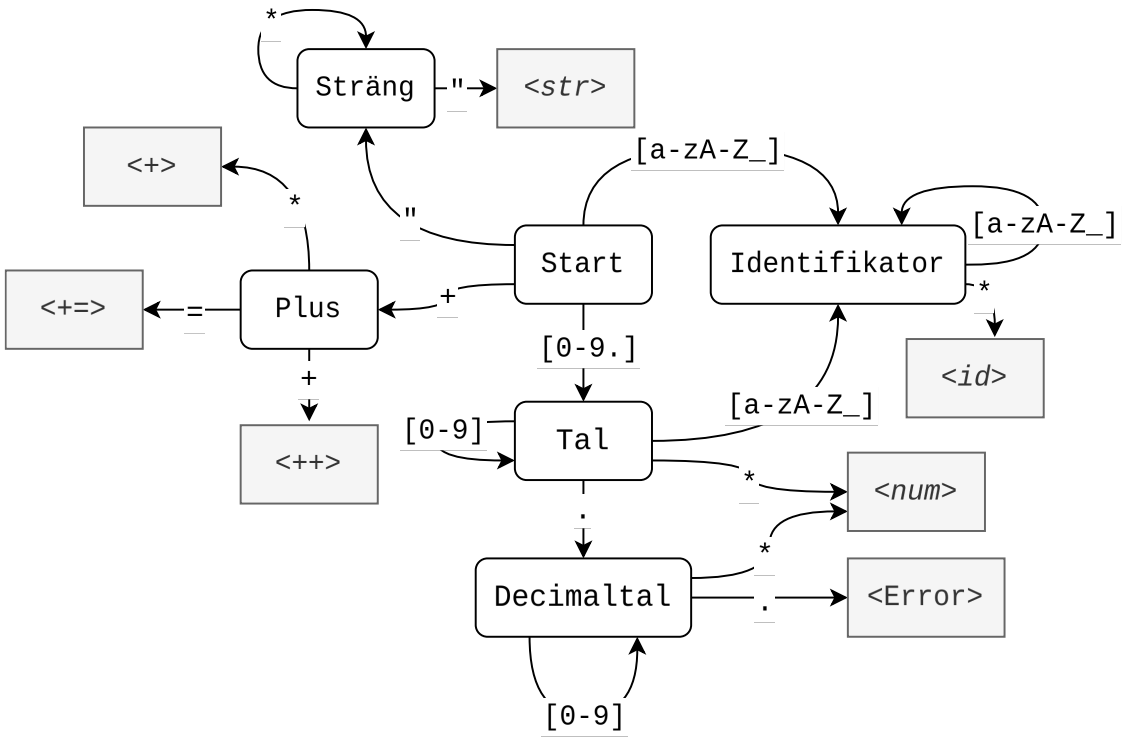
Uttryck innefattar här litteraler för tal, strängar, arrayer, och struktar, alla enkla operatorer (se Tabell 2), array- och struktindexering, funktionsanrop, lambda-uttryck, samt variabler.

Definitioner använder `let`/`const` som i JavaScript, `for`-loopar använder samma syntax som i C, och kod delas in i olika block genom indentering som i Python. Klammerparanteser (`{}`) stödjs också som ett alternativ till indentering. Lambda-uttryck skrivs som i JavaScript med alternativa typannoteringar. Funktionsanrop kan även skrivas med samma syntax som vanligtvis används för metoder, dvs. att `a.f(...)` är synonymt med `f(a, ...)`.

Stream-operationer använder reserverade namn enligt Tabell 1 och har därför ingen särskild syntax.

3.3 Lexikalanalys

Källkoden omvandlas från ett flöde av tecken till en samling av tokens med hjälp av en lexer som agerar som en ändlig automat (en: *finite-state machine*). Varje gång ett tecken konsumeras sparas antingen en token, eller så läggs tecknet i en buffer. Detta beslut beror på analysatorns nuvarande tillstånd (en: *state*), och detta tillstånd ändras dessutom i varje steg. En förenklad schematisk beskrivning av hur detta sker kan ses i Figur 7, men den faktiska implementationen använder många fler tillstånd för att hantera syntax för bland annat kommentarer och tal skrivna på grundpotensform såsom `1e6`.



Figur 7: Diagram över några av de tillstånd som kompilatorn använder vid lexikalanalys. Gråmarkerade tillstånd innebär att en token genereras, och att programmet därefter återgår till tillståndet `Start`.

3.3.1 Indentering

För att möjliggöra för användaren att skriva mer läsliga program i `gpulang` kan kodblock definieras genom en *offside rule*, även känt som *significant leading whitespace*, där antalet mellanslag i början av en rad markerar den nuvarande kodmiljön[16]. Därmed behöver användaren eller den som läser koden inte längre hålla koll på klammerparenteser (`{}`).

Detta hanteras också av parsern i samma pass, genom en implementation av samma algoritm som beskrivs i dokumentationen för Python 3.13.3 [24], där en stack används för att hålla koll på indenteringsnivåerna i stigande ordning, och `Indent`-/`Dedent`-tokens genereras varje gång indenteringsnivån ändras. Eftersom indenteringen alltid sker i början av varje rad och därefter kan betraktas som whitespace, så kan denna process ske parallellt med resten av lexikalanalysen, utan att kräva en extra pass.

Eftersom det finns tillfällen där klammerparenteser och semikolon kan göra koden *mer* läslig[17], så tillåts användaren även dessa utöver denna *offside rule*. Det finns också vissa undantag där indentering och radbrytning behandlas helt och hållet som whitespace i språket när det motiveras av att tillåta användaren att skriva mer läslig kod.

3.4 Syntaxanalys

De tokens som genereras av lexern organiseras hierarkiskt i ett abstrakt syntaxträd med hjälp av en parser. Implementationen av denna process är till stor del inspirerad av tekniken *Pratt parsing*, som

[16]Detta koncept är numera väldigt populärt tack vare programspråk som Python och Haskell, men är betydligt äldre och beskrevs redan 1966 i [23].

[17]Ett vanligt exempel är korta `if`-satser som `if x == 0 { return(y); }`.

beskrivs ovan i Avsnitt 2.2. Likt många andra programspråk består varje rad av ett program skrivet i `gpulang` av antingen en sats (en: *statement*), eller av ett tilldelande (en: *assignment*). Dessa separeras antingen av radbrytning eller av semikolon, som båda har översatts till `EOL`-tokens (av en: *end of line*) av lexern, vilket gör det lätt att isolera dem och översätta dem en i taget till grenar i syntaxträdet.

För varje sats identifieras den token som innehåller operatorn med högst prioritet bland alla tokens i satsen, med undantag för dem som innesluts av parenteser. En lista över alla operatorer som används av `gpulang` och deras motsvarande prioritet visas i Tabell 2, och syntaxen för dessa operatorer såväl som deras inbördes prioritet tar stor inspiration av grammatiken som används av C++ [25, s. 72–94]. En nod skapas i syntaxträdet för denna operator, och listan över tokens i satsen delas i två, som beskriver de två delsatserna kring operatorn[18]. Samma algoritm appliceras rekursivt för varje delsats, och noden i syntaxträdet ges underordnade noder baserat på resultaten från de rekursiva anropen.

Om ingen operator återstår i en delsats tilldelas istället den enda token som återstår direkt som underordnad nod. Detta motsvarar vanligtvis en identifikator (en: *identifier*) för en variabel, konstant, eller funktion; alternativt en litteral (en: *literal*) för ett värde med en primitiv datatyp, såsom `1.0`, `true`, eller `"Hello world"`. Särskild logik används för att identifiera vilken semantisk funktion en rundparentes (`()`) eller hakparentes (`[]`) har i en sats. Notera exempelvis att hakparenteser används både för array-index, som i `x[0] = 1`, och för array-litteraler, som i `x = [0, 1]`.

3.4.1 Plus- och minustecken

Ett specialfall som beskrivs i Tabell 2 är att plus- (`+`) och minustecken (`-`) kan användas både som unära och binära operationer. En intuitiv lösning för att särskilja dessa är att behandla de unära varianterna som *prefix*, det vill säga att `-5` är talet 5 med ett negerande prefix, och att `-x` är identifikatorn `x`, också med ett negerande prefix. Denna tolkning ger stor vikt till mellanslag som särskiljande tecken, vilket är oönskvärt när många programmerare ser uttryck som `x-y` som synonyma med `x - y`.

En lösning för detta problem följer av att notera att unära operatorer alltid förekommer till vänster[19] av deras operander. Genom att ge unära operatorer lägre prioritet än binära operatorer kan vi förvänta oss att en unär operator alltid är längst till vänster av en delsats när ingen operator med högre prioritet återstår. Om vi hittar ett minustecken längst till vänster i vår delsats, eller direkt till höger av en annan operator, bör vi alltså tilldela det prioritet **2** precis som de andra unära operatorerna `!` och `~`; annars tilldelas det prioritet **4** för att väljas framför operatorer såsom multiplikation och division.

3.4.2 Toppnoder

Beskrivningen ovan av syntaxanalys förklarar hur en lista av satser omvandlas till ett syntaxträd, men eftersom `gpulang` är ett procedurellt programspråk så återfinns dessa satser bara inuti funktionsdefinitioner i källkoden. Innan denna process kan påbörjas behöver vi alltså först identifiera dessa funktiondefinitioner och de kodblock som de innehåller. I Kodstycke 1 följer ett exempel på ett `gpulang`-program som summerar alla jämna tal i `xs`. Notera att även satser som är tänkt att exekveras direkt när programmet körs återfinns i en funktion, det vill säga `main()`.

[18] Detta gäller för binära operatorer. Unära operatorer som `!` och `~` leder istället bara till ett enda rekursivt anrop.

[19] Samma princip kan användas för unära operatorer som alltid är till höger av deras operander, det viktiga är att de alltid är på samma sida.

Tabell 2: Prioritetsordning inom syntaxanalys för alla enkla operatorer som används av `gpulang`.

Prioritet	Token	Kommentar
1	`( ) [ ]`	Allting inuti två parenteser som matchar varandra i en delsats tilldelas prioritet **0**.

2	`!`
	`~`
	`+ -`
	`.`
		Inverterar ett booleskt värde (en: *logical NOT*).
		Inverterar varje bit för sig (en: *bitwise NOT*).
		Har prioritet **2** som prefix, annars prioritet **4**.
		Högerassociativ operator som används för metodanrop.

3	`* /`
	`%`
	`\`
		Multiplikation och division.
		Modulo-operator, det vill säga rest vid division.
		Heltalsdivision, det vill säga division utan rest.

4	`+ -`	Addition och subtraktion. Har prioritet **2** som prefix, annars prioritet **4**.

5	`<<`
	`>>`
	`>>>`
		Logiskt/aritmetiskt vänsterskift
		Logiskt högerskift.
		Aritmetiskt högerskift.

6	`< >` `<= >=`	Jämförelser för strikt mindre/större. Jämförelse för mindre/större än eller lika med.
7	`== !=`	Jämförelser för lika med eller skiljt från.
8	`&`	Logiskt AND för varje par av bitar (en: *bitwise AND*).
9	`^`	Logiskt XOR för varje par av bitar (en: *bitwise XOR*).
10	`|`	Logiskt OR för varje par av bitar (en: *bitwise OR*).
11	`&&`	Sant om och endast om två booleska värden är sanna (en: *logical AND*).
12	`||`	Sant om och endast om något av två värden är sanna (en: *logical OR*).
13	`=>`	Lambda-uttryck för anonyma funktioner. Används bland annat tillsammans med stream-operationer såsom `map` och `filter`.
14	`=` `++ --`	Tilldelar ett värde till ett uttryck. Alla binära operationer ovan i tabell stöds som prefix. Exempelvis tolkas `x >>= 1` likadant som `x = x >> 1`. `x++` tolkas som `x = x + 1`, och `x--` tolkas som `x = x - 1`.

```
 1 import add.gpulang
 2
 3 fn is_even<T: Int>(x: T): bool
 4     return x % 2 == 0
 5
 6 fn main()
 7     let xs: [i32] = [1, 2, 3, 4, 5, 6, 7, 8]
 8     let scan_evens: [i32] = xs.filter(is_even).scan(add, 0).collect()
 9
10     print(scan_evens)
```

Kodstycke 1: Ett `gpulang`-program som stegvis beräknar summan av alla jämna tal i en array.

```
1 fn add<T: Num>(x: T, y: T): T
2     return x + y
```

Kodstycke 2: En enkel `gpulang`-källfil som definierar en funktion för att addera två tal.

Varje rad som inte är del av ett indenterat kodblock i `gpulang` måste börja med ett av fyra nyckelord: `fn`, `const`, `import`, eller `struct`. Ordet `fn` följs alltid av en identifikator som innehåller namnet på funktionen, och därefter antingen av tecknet `<` om parametrarna för funktionen använder generiska typer, och annars av tecknet `(`, och så vidare tills vi når slutet av raden. Eftersom alla rader som motsvarar en sådan toppnod kan beskrivas med endast ett fåtal tillstånd, så är denna delen av syntaxanalysatorn implementerad som en ändlig automat som liknar den som används för lexikalanalys som beskrevs ovan i Avsnitt 3.3.

3.4.3 Importera filer

Nyckelordet `import` används för att kopiera in all källkod från en annan fil, huvudsakligen för att göra det lätt att återanvända samma funktion i flera olika program. I Kodstycke 1 ovan så innehåller filen alltså `add.gpulang` källkod i stil med Kodstycke 2.

Importerade filer hanteras av parser:n genom att syntaxträd genereras för både den kompilerade och den importerade filen, som därefter kombineras till ett enda träd. Om de två träden innehåller toppnoder som motsvarar funktioner, konstanter, eller struktar med samma namn, så inkluderas endast noden i den filen som kompilerades direkt. Prioriteringen är en följd av den nuvarande implementationen av `gpulang` som saknar koncept som *namespaces*.

`import`-satser avgörs iterativt, vilket innebär att en importerad fil kan i sin tur importera ännu fler filer. Detta ger upphov till ett grafteori-relaterat problem eftersom en cykel av `import`-satser kan leda till att parser:n fastnar i en loop där ett antal filer kompileras om och om igen i samma sekvens. Detta problem går att lösa med samma tidskomplexitet $\mathcal{O}(n)$ som krävs för att importera alla filer, där n är antalet `import`-satser bland filer som läses in, eller ekvivalent antal kanter i en graf.

För att åstadkomma detta så innehåller varje anrop att importera en fil en lista på alla filer som redan besökts, och om anropet försöker importera en fil som redan finns i denna lista[20] så har en cykel

[20]Detta innebär att antalet jämförelser som görs växer för varje successivt anrop som är del av samma iterativa importkedja, vilket i värsta fall kan leda till $\mathcal{O}(n^2)$ jämförelser för n filer som importeras i ordning. Mer tidseffektiva algoritmer finns, såsom *Floyd's tortoise and hare* som har tillskrivits Robert W. Floyd[21] [26, s. 7] som antyder att det räcker att jämföra

upptäckts, och kompilatorn bör stoppa och ge ett felmeddelande till användaren. Notera att det är viktigt att filimportering behandlas som en *riktad* graf, eftersom det är helt rimligt att kod från två olika filer använder samma bibliotek från en tredje fil.

3.5 Semantisk analys

Type-checkern i `gpulang` implementerar en variant av Hindley-Milners typsystem. Till skillnad från Hindley-Milners typsystem, som tillåter kvantifiering över typer på ett godtyckligt sätt, så implementeras endast detta för funktionstyper i `gpulang`. Dessutom har typsystemet i `gpulang` några fall av sub-typing, samt möjlighet för *ad-hoc* polymorfism likt den som beskrivs i [27] för vissa inbyggda binära operationer såsom addition och multiplikation.

Implementationen av detta typsystem följer i stora drag Algoritm W som beskrivs i [18]. Syntaxträdet traverseras rekursivt, och så kallade *constraints* ackumuleras. När man når ett löv där typen är okänd skapas en ny typ-variabel, om detta exempelvis är en heltalslitteral skapas även ett constraint som säger att denna typ-variabel måste motsvara ett heltal. När en viss typ förväntas så skapas även ett constraint som säger att den typ som infererats måste vara en subtyp till den förväntade typen. Efter att varje nod har infererats så görs ett försök att lösa alla samlade constraints, och om det går så resulterar det i en tilldelning från typvariabler till nya typer, och dessa substitueras i hela trädet innan funktionen returnerar för noden. Efter att hela trädet traverserats på detta vis så delas *default*-typer ut baserat på alla constraints som återstår.

Implementeringen har alltså tre huvudsakliga delar: inferens, constraint-lösning, och substitution. Utöver dessa delar—som är standard i de funktionella språk som algoritmen designades för—krävs det även att ny kod skrivs för modifiering av värden, och för att identifiera vilka sorts uttryck som är värden som går att modifiera.

Utöver själva type-checkern så kontrolleras det även i detta steg att inga cirkulära definitioner finns i programmet. Dessutom omordnas definitionerna i programmet så att allt definieras innan det används. Detta görs genom att traversera alla definitioner och bygga en graf över vilka andra definitioner de använder. Sedan sorteras listan av toppnivå-definitioner med topologisk sortering på grafen.

3.6 HIP-kodgenerering

Resultatet i det här skedet av kompilationen är ett AST annoterat med typer. Detta AST översätts därefter till kod i HIP, som är en dialekt av C++ som utvecklats av grafikkortstillverkaren AMD. Varje nod i syntaxträdet översätts sekventiellt till motsvarande uttryck i HIP, och vad som ingår i detta uttryck bestäms utifrån rekursiv analys av nodens undernoder. Många enklare uttryck i `gpulang` kan översättas direkt till ett motsvarande uttryck i HIP, men översättningen av stream-operationer är mer av en stegvis översättning till funktionsanrop som ej utnyttjar full parallellisering av funktionerna i en stream. I Bilaga A visas HIP-kod som kompilatorn kan generera.

Omvandlingen till HIP medför vissa regler som annars inte utgör något problem för vad som är ett giltigt program i `gpulang`. Till exempel måste en användardefinierad funktion som använder argument av typen array eller stream alltid inkludera storleken av denna typ som efterföljande argument.

---

en nod $2n$ steg in med den n steg in. Eftersom dessa jämförelser tar avsevärt mindre tid än att läsa in och kompilera filerna på vägen, så kan denna naiva approach ändå sägas vara att föredra, eftersom den upptäcker cykler tidigare vid många grenar. Möjlighet till förbättring finns däremot för programmeraren som rutinmässigt importerar miljardtals filer.

[21] Samma Floyd som den Floyd bakom *Floyd's operator-precedence* som beskrevs i Avsnitt 2.2

3.6.1 Hantering av streams

Inledningsvis när ett uttryck som innehåller en stream översätts till HIP så fastställs det först att denna stream existerar. Om streamen inte existerar så avbryts exekveringen av programmet och ett felmeddelande returneras. Existens av streams hanteras genom en datastruktur som kallas för `HashMap`. När en identifikator av typen array eller stream deklareras, antingen som en variabel eller som en funktionsparameter, så skapas också en motsvarande nyckel i denna `HashMap` med information om streamen som ett tillhörande värde.

I det motsatta fallet kontrollerar översättningsfunktionen att streamen är en device, dvs. att den har data allokerad på GPU:n. Om den inte har det, så allokeras data på GPU:n utifrån vad som är förknippat med streamen på host-sidan. Enligt designprinciperna som beskrivs ovan i Avsnitt 3.1 så motsvarar detta att en array kopieras till en stream.

3.6.2 Konsumering av streams

När en stream använder funktionsanropet `collect` så markeras den som konsumerad, eftersom den därefter befinner sig på host-sidan, och det kan inte längre garanteras att datan på CPU- och GPU-sidan överensstämmer. Även funktionsanropet `filter` beaktas särskilt med avseende på konsumering och `collect`. Vid filtrering allokeras en GPU-array som håller reda på vilka element som uppfyller predikatet i `filter`-anropet, och den används sedan i *stream compaction*[22] när streamen konsumeras genom `collect`.

Stream compaction utförs även för stream-operationer som inte är `map`. Detta eftersom element i övriga operationer paras ihop med annan data, och ihopparningen kan bli inkorrekt om filtrering ej har slutförts genom *stream compaction*.

3.6.3 Stream-operationer

När programmet exekveras utgörs en stream av en kedja av funktionsanrop som var och en manipulerar den ingående datan. I översättning till HIP utgörs dessa funktionsanrop av handskrivna GPU-kernels som motsvarar de olika inbyggda stream-operationer språket exponerar. Majoriteten av dessa operationer listas i Tabell 1. Dessa handskrivna kernels är skrivna i HIP som template-funktioner (från en: *template function*), vilket tillåter både användardefinierade typer såväl som användargivna funktioner att användas inom kernel-koden. Operationer som `map` kan på så vis implementeras med en och samma kernel-kod, trots att `map` kan operera på olika ingående datatyper och applicera godtyckliga användargivna transformer på den ingående datan. I Kodstycke 3 visas hur kerneln för `map` implementeras.

```
1 template <typename T, typename U, typename Func>
2 __global__ void mapKernel(T* input, U* output, uint32_t n, Func lambda) {
3   int global_id = blockIdx.x * blockDim.x + threadIdx.x;
4
5   if (global_id < n) {
6     output[global_id] = lambda(input[global_id]);
7   }
8 }
```

Kodstycke 3: Template-funktion i HIP för en GPU-kernel som utför en map-operation.

[22] *Stream compaction* innefattar att ta bort ogiltiga element från en datamängd vilket vanligen resulterar i en kortare och *kompaktare* sekvens av data.

```
1 threadsPerBlock = NUM_THREADS >> ((2 * NUM_THREADS * sizeof(T)) / (SM_MEM));
```

Kodstycke 4: Kod i HIP för att beräkna blockstorlek givet maximala antalet trådar `NUM_THREADS` och en maximal storlek `SM_MEM` på ett blocks delade minnesbank för en datatyp `T`.

I den ovan nämnda listan med stream-operationer, Tabell 1, utmärker sig både `scan` och `reduce` (samt varianter därav) som särskilt avancerade att implementera som GPU-kernels. Dessa beskrivs därför i närmare detalj i de två nästkommande delavsnitten.

3.6.4 Stream-operationen `scan`

Stream-operationen `scan` beräknar som bekant prefix-summan över den ingående streamen. I `gpulang` utgår `scan`-implementation från en algoritm som beskrivs i GPU Gems 3 [28]. Denna algoritm har anpassats för att implementeras med template-funktioner, och är en *inklusiv scan-operation* (en: *inclusive scan*), dvs. en summering av samtliga tidigare element, inklusive det egna elementet[23].

Implementationen tar även hänsyn till storleken på datatypen av operanderna för att dynamiskt specificera blockstorleken[24] av kernelanrop.

Genom att anta maximala antalet trådar `NUM_THREADS` och en maximal storlek `SM_MEM` på den delade minnesbanken för ett block kan blockstorleken beräknas redan vid kompilering. Låt t vara datatypens storlek, i antal bytes. Om `NUM_THREADS` trådar används kommer programmet att ta upp $2 \cdot$ `NUM_THREADS` $\cdot$ t bytes, vilket utnyttjas för att halvera antalet trådar ett lämpligt antal gånger, se Kodstycke 4. Tvåan kommer utav att algoritmen behandlar två element per tråd.

3.6.5 De reducerande stream-operationenerna `reduce` och `reduce_c`

Stream-operationen `reduce` är konstruerad genom att endast implementera *up-sweep*-fasen i *scan*-algoritmen som beskrivs i [28]. Stream-operationen `reduce_c`, som utöver associativitet även antar kommutativitet[25] i den ingående operatorn, är till viss del implementationsmässigt inspirerad av algoritmen presenterad av Harris i [29].

Genom att anta kommutativitet kan operationen snabbas upp, eftersom att reduktioner kan utföras på *warp*-nivå och kräver ej heller att trådar synkas flera gånger. Implementation har dock nackdelen att den endast kan appliceras på datatyper som har en storlek på 32 eller 64 bitar. Detta är en följd av begränsningar i funktionen `__shfl_down_sync` [30]. Teoretiskt kan dock den kommutativa varianten av `reduce` ha bättre körtidsprestanda, då *warp*-nivån medför att beräkningarna har bättre minneslokalitet.

3.6.6 Funktioner och struktar

En annan typ av uttryck utöver streams som också hanteras på ett särskilt vis är användardefinierade funktioner och struktar. För att funktioner ska kompilera till GPU-kod genom HIP används *funktorer*, som kan beskrivas som funktionsobjekt och som beter sig nästan identiskt med normala funktioner. Funktorer tillåter att funktioner ges som argument i stream-operationer, och möjliggör även att HIP kan generera funktionerna som kod utan funktionsanrop, vilket ofta kallas *inline*.

[23] En scan-opereration där det egna elementet inte medtas i summeringen ut i utdataelementet på samma plats kallas å andra sidan för en *exklusiv scan*.

[24] Blockstorleken är antalet trådar som exekveras parallellt i en GPU-kärna med möjlighet till synkronisering och gemensamt minne.

[25] Namnet `reduce_c` kommer av initialbokstaven i engelskans *commutative*.

Funktorer förekommer i större utsträckning i STL (en: *standard template library*), som är en del av standardbiblioteket i C++, vid funktionsanrop i samband med template-funktioner (en: *template functions*) [31]. De handskrivna kernels som används vid kodgenerering i `gpulang` är template-funktioner och lämpar sig därför väl för funktorer.

Struktar behandlas särskilt på så vis att konstruerare (en: *constructors*) automatiskt genereras för att GPU-koden som genereras av HIP ska känna till hur datan i struktar kopieras vid tilldelande (en: *assignment*).

Både funktioner och struktar får även macro-tilläggen `__host__` och `__device__` för att dessa ska vara synliga och gå att kompilera på båda målplattformarna, vilket är nödvändigt om en funktion eller strukt används på både CPU:n och GPU:n.

3.6.7 Lambda-uttryck

Översättningen till HIP inkluderar även enkla lambda-uttryck. Funktorer utnyttjas för lambda-uttryck på samma sätt som beskrivs ovan i Avsnitt 3.6.6. Detta tillåter även att icke-lokala variabler används i lambda-uttryck, genom att de tas med som argument i konstrueraren av funktorens strukt. Se Bilaga A för ett kodexempel i `gpulang` som använder icke-lokala variabler i ett lambda-uttryck.

4
Metod

Projektet genomfördes iterativt i tre faser: design, implementation, och utvärdering av programspråket. Arbetet i dessa faser skedde parallellt och överlappade till stor del, eftersom återkoppling från en fas kontinuerligt ledde till justeringar i de andra faserna. Nedan följer en beskrivning av processen såväl som tester för prestandamätning.

4.1 Designprocess

Tidigt i designprocessen identifierades vilka primitiva datatyper som `gpulang` borde innehålla, däribland heltal med och utan tecken, flyttal, och booleska värden. Det bestämdes även att funktioner borde ha möjlighet att ta generiskt typade parametrar som indata, för att på så vis låta användaren skriva återanvändbara abstraktioner med högre ordningens funktioner. Stöd för mer generella C++-liknande template-funktioner samt generiska parametrar utanför funktioner, som exempelvis till generiska struktar, övervägdes, men valdes bort för att förenkla utvecklingsprocessen.

Eftersom GPU:er inte lämpar sig särskilt väl för exekvering av rekursiva funktioner, bestämdes det att varken dessa eller cykliska definitioner inte skulle tillåtas. Detta hindrar även användaren från att skriva program utan mening, som två globala konstanter vars definitioner beror på varandra. Det tilläts däremot att definiera funktioner och konstanter i vilken ordning som helst, då detta ansågs vara ergonomiskt.

4.1.1 Syntax

Syntaxen utformades för att vara bekant för programmerare med erfarenhet av populära språk som Rust, JavaScript, och Python. Samtidigt valdes det att undvika vissa komplexiteter, såsom de som medförs av C/C++:s funktionspekare. Att använda `let`-satser tillåter användaren att utelämna typdeklarationer när dessa är överflödiga.

För att möjliggöra metodliknande-anrop trots att språket saknar möjligheten att definiera metoder bestämdes det att syntax som `a.f(...)` bör tolkas som synonymt med `f(a, ...)`. Detta ansågs vara särskilt viktigt med avseende på läslighet av den kod som förväntas skrivas i `gpulang`, eftersom streams är tänkta att modifieras genom kedjade anrop av fördefinierade stream-operationer. Ett exempel på skillnaden mellan dessa två varianter för kedjade anrop illustreras i Kodstycke 5. Genom att stödja båda sorternas syntax låter vi användaren välja stil efter egen erfarenhet.

4.1.2 Streams

Flera idéer övervägdes med avseende på hur data bör representeras när den används av högre ordningens funktioner för att modifieras på GPU-hårdvara, däribland att implementera separata typer när element flyttats eller maskerats bort. Det diskuterades även till vilken grad dessa element skulle gå att inspektera, exempelvis genom att index:a en mask och antingen få ett element som returvärde, eller ett speciellt värde som indikerar att värdet är maskerat. Det var inte heller klart om omvandlingen mellan denna datarepresentation och array:er borde vara implicit eller explicit.

```
// metod-baserad syntax
fn reduce_test()
    let xs: [i32] = [1, 2, 3, 4, 5, 6, 7, 8]
    let sum: i32 = xs.filter(is_even)
                     .scan(add, 0)
                     .reduce_c(add, 0)

// funktions-baserad syntax
fn reduce_test_alt()
    let xs: [i32] = [1, 2, 3, 4, 5, 6, 7, 8]
    let sum: i32 = reduce_c(scan(filter(xs, is_even), add, 0), add, 0)
```

Kodstycke 5: Jämförelse mellan läslighet för metod-baserad och funktions-baserad syntax illustrerad med ett kodexempel i `gpulang`. Dessa två funktioner översätts till identiska syntaxträd av kompilatorn skapad i detta projekt.

I slutändan valdes det att ha en enda konkret typ som benämndes `Stream<T>`, vilket senare förenklades till `{T}` i språkets syntax. Det valdes att tillåta implicit omvandling från arrays när dessa används som indata för operationer som verkar på streams, men att kräva att streams behöver explicit konverteras till arrays via en `collect`-operation. Detta beslutades dels med motiveringen att flera typer gör språket mer förvirrande för användaren, men en annan fördel med att inte låta användaren indexera en mask var att möjligheten öppnades för att representera streams hur som helst i kompilatorns backend, med implikationer för kodoptimering.

4.1.3 Typsystem

Omvandlingen till streams sker implicit med subtyper, och medför att program kan vara mer ergonomiska att skriva. För att ha kvar minneskontrol krävs det som nämnt ovan att streams sparas explicit till en array, eftersom arrays måste ha konkreta minnesrepresentationer. För att möjliggöra modifiering *in-place* noterades det att det hade varit idealt att använda affina typer för att låta streams konsumeras efter användning, så att ingen modifiering som invaliderar datatypen kan ske. Detta hade medfört att användaren inte kan råka skriva program som modifierar eller läser ut ogiltiga data av misstag.

4.2 Implementationsprocess

Själva kompilatorn implementerades i programspråket Rust som ansågs möjliggöra hög prestanda och tillhandahöll statiska typer samt minnessäkerhet. Rust har också stöd för *enum*-typer som kan innehålla data och dynamiska *match*-uttryck som förenklade processen att definiera och traversera syntaxträd.

4.2.1 Frontend

Tidigt i projektets gång diskuterades användningen att ett externt bibliotek för att generera lexer och parser till kompilatorn. Efter att ha undersökt lämpliga bibliotek för detta beslutades det att skriva dessa för hand istället för att minimera beroendet till kod skriven av andra. Detta innebar också att kompilatorn består av färre steg som är lättare att förstå, eftersom vi har kontroll över formatet på datan i varje mellansteg. Valet av metod för syntaxanalys motiverades därav huvudsakligen av att den skulle vara lätt att skriva från grunden, snarare än att den behövde vara så effektiv som möjligt.

Type-checkern implementerades med hjälp av Hindley-Milners typsystem och befintliga algoritmer för detta, eftersom detta möjliggjorde både typinferens och generiska funktioner, som under design-

processen utmärktes som önskvärda egenskaper för språket. I eftersträvan av högre prestanda så valdes det att byta ut flera av de icke-muterbara elementen av algoritmen, som återfinns i funktionella implementationer, mot muterbara element. Exempelvis muterades kontext snarare än att skapa kopior när nya variabler definierades. Detta ledde till att variabler explicit behövde tas ur kontext, snarare än att förlita sig på flödeskontroll. Denna kompromiss ledde till ett antal buggar, men efter att dessa åtgärdades så förväntas implementationen prestera bättre än om den hade använt en icke-muterbar variant.

Eftersom Hindley-Milner-implementationer redan är väl studerade inom funktionell programmering så låg fokuset under arbetat på att anpassa dessa väl till ett `gpulang`, som designades som ett imperativt språk. Inom denna implementation valdes det att skjuta upp typinferens för att möjliggöra olika *constraints* för olika numeriska typer. Modifierbarhet behövde också utgöra en *constraint* eftersom språket stödjer konstanter med attributet `const`.

En topologisk sortering av alla definitioner på den översta nivån av syntaxträdet implementerades för att stödja att dessa definieras i godtycklig ordning i källfilen, samt för att upptäcka cykliska definitioner. Detta steg var nödvändigt för att kunna implementera en backend som använder HIP, eftersom källkod för HIP behöver struktureras så att funktioner definieras först innan de används.

4.2.2 Backend

Ursprungligen var det planerat att en *Sea of Nodes*-baserad[26] IR skulle användas som en mellanrepresentation, för att därefter kompilera instruktioner direkt till PTX, SPIR-V, eller annan liknande lågnivåmiljö. Parallellt med detta arbete påbörjades utvecklandet av en transpilerande backend—vad som ofta kallas *source-to-source*—för att ge möjlighet tidigare i projektet att testa olika aspekter av `gpulang` genom befintliga optimerande kompilatorer. Det beslutades att denna transpilator skulle rikta sig mot HIP-ramverket, då detta hade stöd både för grafikkort tillverkade av AMD och av NVIDIA. OpenCL var ett annat alternativ som övervägdes, då det också hade bra stöd för flertalet plattformar. I slutändan beslutades det att enbart rikta in transpilatorn på HIP, då det gav mer kontroll över den resulterande GPU-koden och därmed möjliggjorde mer specialiserad kod med potential för högre prestanda.

En virtuell maskin (VM, en: *virtual machine*) utvecklades för att stödja utvecklingen av den tilltänkta IR:en. Denna VM var huvudsakligen registerbaserad, och exekverade IR-instruktioner för att göra det möjligt att undersöka var logiska fel uppstod. Detta innebär att den även behövde ha stöd för VM-specifika exekveringssätt, däribland att en instruktion utfördes i taget (en: *single stepping*), eller att samtliga instruktioner i ett CFG-block utfördes tillsammans. Det gick att skriva ut information om maskinens tillstånd efter varje steg, för att på så vis observera exekveringsordning och registervärden. Eftersom IR-implementationen aldrig fullbordades så färdigställdes inte heller VM-implementationen.

4.3 Mätvärden

Prestandamätning av språket är intressant eftersom det kan påvisa om `gpulang` uppnår de förväntade förbättringarna av parallelliserade beräkningar samt hur språket förhåller sig i jämförelse med liknande

[26] *Sea of Nodes* introducerades 1995 av Click och Paleczny [32]. Detta. Det är en SSA-baserad teknik där IR-instruktioner inte har en given ordning och korrekt programbeteende enbart garanteras genom väl valda data- och kontroll-beroenden instruktioner sinsemellan. Detta ger en gemensam representation för både kontrollflöde och dataflöde, vilket underlättar vid kodoptimering. Exempelvis förenklas *code motion*, vilket är transformationer där instruktioner flyttas från sin ursprungliga plats till en annan mer prestandamässigt lämplig, genom att information om den ursprungliga platsen inte längre ingår i IR-representationen. När IR:en sedan omvandlas till maskinkod återförs den först på en form där instruktioner åter har en given plats, ofta en typisk CFG-representation.

projekt som Futhark och optimerade bibliotek som Thrust. För att evaluera prestandan av kompilatorn valdes att köra samma sorteringsalgoritm, radix-sortering, i `gpulang`, Futhark och Thrust.

Valet av radix-sortering till prestandamätningar motiveras av att det är en algoritm som nyttjar parallellism väl och är enkel att implementera. Dessutom nyttjar implementationen många av stream-operationerna i `gpulang`, nämligen: `map`, `reduce`, `scan`, `map2`, och `scatter`. Att täcka alla dessa operationer innebär att större del av språket testas.

4.3.1 Radix-sortering

Algoritmen för radix-sortering har sitt ursprung så tidigt som 1880-talet men appliceringen av algoritmen i datorer visades vara rimlig först 1954 av Seward [33]. Genom att dela in n element med nycklar av längd k lexografiskt utan jämförelser uppnås den teoretiska tidskomplexiteten $\mathcal{O}(nk)$. En typisk version av algoritmen som sorterar i stigande ordning baseras på den minsta signifikanta siffran (LSD, av en: *least significant digit*) av nyckeln. Implementationerna av radix-sortering i `gpulang` och Futhark använder enkel LSD radix-sortering[27] medan Thrust använder en variation av radix-sortering presenterad i [36].

4.3.2 Evaluering

Tester för de olika implementationerna utfördes genom att sortera en lista av n slumpade 32-bitars positiva heltal, $n \in \{10^i \mid i = 1, ..., 8\}$. Mätningen av körtiden för respektive sorteringsalgoritm noterades i mikrosekunder och upprepades minst 10 gånger för varje n. Resultatet sammanställdes därefter som ett medelvärde av körtiderna för respektive n.

Tiden som krävs för att starta programmet eller överföra data mellan GPU och CPU inkluderades ej i mätningarna. Detta motiveras av att dataöverföring tar upp en signifikant del av programmets exekveringstid, samt skiljer sig starkt mellan olika grafikkort med olika hastighet på minnesbuss. Därav är mätningarna mer representativa för algoritmen då denna tid ej inkluderas.

I Futhark användes den inbyggda Futhark-bench funktionen för att mäta exekveringstid. I `gpulang` samt Thrust användes `chrono`, ett standardbiliotek i C++, för att beräkna exekveringstiden som differensen mellan tiderna före och efter exekvering av algoritmen.

Evalueringen av körtidsprestanda utfördes i samma miljö för respektive program med hårdvara enligt Tabell 3. Versioner för mjukvara återfinns i tabell Tabell 4. Futharkkod kompilerades till både C och CUDA, och exekverades genom WSL (en: *Windows subsystem for Linux*). Eftersom WSL nyttjades för Futhark kördes `gpulang` både direkt på systemet och via WSL för att upptäcka en potentiell felkälla. Koden som användes för mätningarna återfinns i Bilaga A.

[27] Algoritmen i `gpulang` är baserad på en representation i Futharks dokumentation [34]. För Futhark nyttjas det publikt tillgängliga Futhark-biblioteket `sorts` implementation av radix-sortering [35].

Tabell 3: Hårdvaruspecifikationer för testmiljön som användes för prestandamätningar.

CPU	RAM	GPU	VRAM
12th Gen Intel(R) Core(TM) i7-12700H	16 GB	NVIDIA GeForce RTX 3070Ti Laptop GPU	8 GB

Tabell 4: Versioner för mjukvara i testmiljön som användes för prestandamätningar.

OS	WSL	ROCm	CUDA	Futhark
Windows 11 v26100.3775	v2.3.26	v6.2.41512	v12.8	v0.26.0

5
Resultat

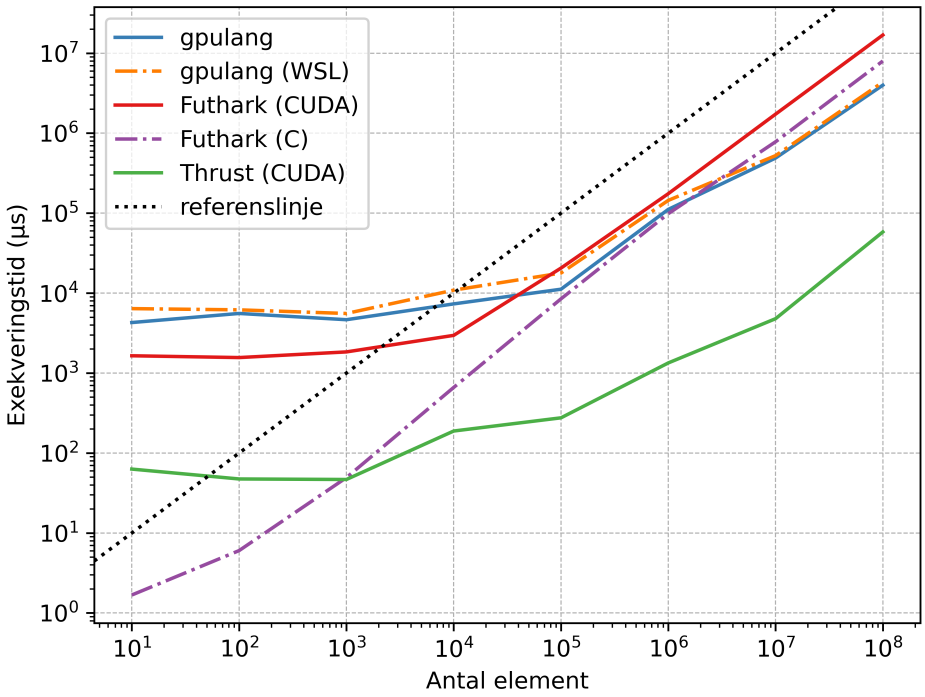
Resultatet av prestandamätningarna för radix-sortering summeras i Tabell 5 och Figur 8. Körtidstesterna utfördes för `gpulang`, Futhark, och Thrust.

Tabell 5 visar att den genomsnittliga skillnaden mellan att köra `gpulang` via WSL istället för direkt på systemet är marginell och varierar från 368 ms till 608 µs långsammare.

Futharks C-implementation är utan undantag snabbare än CUDA-versionen. Implementationerna i `gpulang` har kortare exekveringstid än Futharks C-implementation först vid $n \geq 10^7$, men redan vid 10^5 element är båda versionerna i `gpulang` snabbare än Futharks CUDA-kompilerade testprogram. I Tabell 5 framgår det även att `gpulang` är mer än 4 gånger så snabbt som Futhark (CUDA), och `gpulang` i WSL är mer än 3.8 gånger så snabbt som Futhark (CUDA) för $n = 10^8$.

Thrust har kortast exekveringstid för $n \geq 10^3$, vilket synliggörs i Figur 8. För mindre n är Futharks C-implementation snabbare.

Vid jämförelse med den streckade diagonalen, som visar linjär tidskomplexitet, kan samtliga algoritmer anses ha den väntade linjära tidskomplexiteten för stora n. För $n < 10^3$ tycks endast Futharks C-implementation, som kör på CPU:n, ha linjär tidskomplexitet, medan GPU-algoritmerna har någorlunda konstant körtid.



Figur 8: Genomsnittliga exekveringstider för radix-sortering i olika språk och bibliotek för olika antal element. Båda axlarna använder logaritmisk skala. Den svarta streckade linjen är en referenslinje för linjär tidskomplexitet med lutning 1 µs/element.

Tabell 5: Genomsnittlig exekveringstid för radix-sortering för listor av storlek n. För små värden av n ges tiden i mikrosekunder (µs), och för större n ges den i sekunder (s).

n (antal element)	`gpulang`	`gpulang` (WSL)	Futhark (CUDA)	Futhark (C)	Thrust
10^1	4284 µs	6395 µs	1646 µs	1.7 µs	63 µs
10^2	5567 µs	6175 µs	1563 µs	6.0 µs	47 µs
10^3	4648 µs	5562 µs	1839 µs	49 µs	47 µs
10^4	7328 µs	10889 µs	2962 µs	661 µs	189 µs
10^5	11184 µs	17811 µs	20542 µs	8471 µs	276 µs
10^6	0.1114 s	0.1444 s	0.1750 s	0.09855 s	0.001339 s
10^7	0.4884 s	0.5237 s	1.729 s	0.7802 s	0.004786 s
10^8	3.984 s	4.352 s	16.89 s	7.987 s	0.05819 s

6
Diskussion

Resultaten från prestandamätningen visar på goda förhoppningar till användbarheten av `gpulang` givet vidare utveckling, eftersom det utpresterar Futhark för större indata, men visar på att det är långt ifrån industri-standarden Thrust. Vi ser stora förbättringsmöjligheter till `gpulang` och belyser dessa i slutet av detta avsnitt.

6.1 Prestanda

Resultaten i Avsnitt 5 som visar att Thrust har kortast körtid är föga förvånande, eftersom det är ett bibliotek som är specialiserat på paralleliserade algoritmer och som används i industrisammanhang. Att Futharks C-implementation, som kör på CPU:n, har kortare körtid än Thrust för mindre antal element är också väntat, eftersom en CPU-baserad implementation av radix-sortering har en helt linjär tidskomplexitet, medan GPU-algoritmerna har nära konstant körningstid, så länge antalet element inte överstiger vad GPU:n kan behandla parallellt. Därefter övergår algoritmerna till en linjär tidskomplexitet. GPU-hårdvara har dock en del overhead för att starta program, vilket medför en längre körningstid även för kortare arrayer.

Mer uppseendeväckande är resultatet att `gpulang` har kortare körtid än motsvarande program i Futhark. Eftersom `gpulang` inte genomför några direkta kodoptimeringar, och att Futhark är ett projekt som pågått i flera år, är det rimligt att förvänta att det mer väletablerade språket har bättre körtidsprestanda än `gpulang`. Detta var dock inte fallet, och potentiella faktorer som medför att `gpulang` visade på bättre körtidsprestanda än Futhark diskuteras nedan.

6.1.1 Körtid i Futhark

Futhark beskrivs ha en starkt optimerande kompilator [7]. Något Futhark bland annat gör är att "smälta samman" funktionsanrop, vilket teoretiskt kan leda till bättre körtidsprestanda. Det kan dock vara så att i fallet för radix-sortering så medför denna sammansmältning att funktionsanropen sker på ett mindre optimalt sätt, exempelvis genom indirekta funktionsanrop istället för *inline*.

Närmare analys av koden som genererades av Futhark tycks även peka på att PTX för CUDA kompileras vid körtid, snarare än att kompileras tillsammans med den exekverbara filen. Även detta kan medföra sämre körtid för programmet i CUDA, då kompilering av moduler potentiellt tar upp en del av körtiden.

Att Futhark har genomgående kortare körtid för C-implementationen, jämfört med Futhark kompilerat för CUDA, kan möjligen tillskrivas samma faktorer som nämndes ovan. I övrigt visar den CPU-bundna varianten, jämfört med Thrust, hur parallella beräkningar ger större nytta vid större datamängder och ett större antal operationer.

6.1.2 Körtid i Thrust

Som tidigare beskrivet har GPU-implementationerna en slags platå för små n, eftersom GPU:n kan behandla alla element parallellt i dessa fall. Thrust tycks ha ytterligare en platå med något längre körtid, vilket kan innebära att biblioteket byter till en algoritm som kan behandla fler element parallellt, istället

för att övergå till den förväntade linjära tidskomplexiteten som de andra implementationerna uppvisar. För större n syns dock en linjär tidskomplexitet även för Thrust i Figur 8.

6.1.3 Tidskomplexitet i `gpulang`

Att mäta prestandan av programspråket `gpulang` genom radix-sortering innebar bland annat att testa flera olika stream-operationer. Vid närmare bedömning av Figur 8 kan samtliga grafer indikera att algoritmerna har linjär tidskomplexitet för stora n, även för implementationerna i `gpulang`. Detta indikerar att de kernels som används vid källkodskompilering i `gpulang`:s kompilator har korrekt tidskomplexitet, och därför även är korrekt implementerade.

6.1.4 Felkällor för prestandamätningar

Testerna i `gpulang` gjordes både direkt på systemet samt via WSL för att undersöka om WSL medförde latens och därför hade signifikant påverkan på de uppmätta resultaten. Tabell 5 visar att skillnaden mellan att köra testerna i `gpulang` direkt på systemet eller via WSL är liten; mätvärdena är åtminstone inom samma storleksordning. Det är därför inte troligt att WSL har en inverkan på Futharks körtid, vilket antyder att `gpulang` har mer än 4 gånger så bra körtidsprestanda. Även direkt jämförelse med resultatet från `gpulang` i WSL stödjer detta, eftersom programmet uppvisar 3.8 gånger bättre körtidsprestanda jämfört med Futhark i CUDA.

Körtidsmätningarna för `gpulang` och Thrust gjordes genom `chrono`, som är en del av standardbiblioteket i C++. Även om `chrono` ger en relativt hög tidsupplösning kan det fortfarande introduceras mätosäkerhet på grund av externa faktorer såsom annan samtidig processaktivitet i operativsystemet eller GPU-belastning från bakgrundsprogram. För att minimera inverkan från dessa felkällor upprepades mätningarna minst 10 gånger, som beskrivet ovan i Avsnitt 4.3.2. Resultatet sammanställdes sedan som medelvärden, men viss variation i resultatet kan ändå kvarstå.

6.2 Design

Språkets design bedömdes vara välutförd. Konceptet av streams erbjuder potential för omfattande optimeringar och representationer av data, vilket bedömdes vara lovande, även om denna potential inte utforskades fullständigt i detta projekt.

Stöd för generiska funktioner erbjuder fördelar genom möjligheten att typ-säkra inbyggda operationer och samtidigt tillåta användaren att definiera nya generiska operationer. Som exempel skulle en `sum`-funktion kunna implementeras direkt av användaren enligt Kodstycke 6.

```
1 fn sum<T: Num>(xs: {T}): T
2   return xs.reduce_c((x, y) => x + y, 0)
```

Kodstycke 6: Implementation av en generisk summeringsfunktion för streams i `gpulang`.

6.3 Källkodskompilering

Kompilatorn av `gpulang` kan översätta källkod i `gpulang` till HIP-kod. Backend-delen av källkodskompilatorn är dock, som beskrivet i Avsnitt 3.6, något begränsad i omfånget av de `gpulang`-program som går att översätta. Vid hantering av vissa typer av uttryck och satser riskerar backend även att introducera minnesläckor i den genererade HIP-koden. Förbättringar av kodgenereringen genom säkerställande av minnessäkerhet i HIP och med bättre hantering av AST-noder, bör därför övervägas för framtida arbete.

6.4 Avgränsningar

Utelämnandet av affina typer och en optimerande kodgenerator till följd av tidsbrist resulterade i ett mindre typsäkert språk med lägre prestanda än ursprungligen planerat.

6.4.1 Affina typer

I designfasen beslutades det att språket skulle använda sig av affina typer för att tillåta mutering av den data som representeras av streams. Under projektets gång visade det sig att det tog längre tid än förväntat att implementera detta typsystem. I och med detta bedömdes det att typinferens och annan semantisk analys var viktigare och att affina typer fick uteslutas. Därmed är det i nuläget inte möjligt att mutera datan i en stream *in-place* utan att användaren kan observera detta. Detta ledde till ett mindre typsäkert programspråk som kan orsaka beteende som tycks vara oförutsägbart för användare som inte är insatta i detaljerna gällande kompilatorns implementation. Detta hade kunnat åtgärdas genom att implementera affina typer för streams, givet ytterligare utvecklingstid.

6.4.2 Optimerande kodgenerering

Under större del av arbetet var planen att implementera en IR för optimering i backend-delen av kompilatorn, samt att kompilera direkt till GPU-kod (så som PTX för NVIDIA, eller SPIR-V för Vulkan). Detta hade gett mer kontroll till kompilatorn att nyttja stream-abstraktionen för att utföra optimeringspass, samt gett användaren möjligheten att direkt kontrollera GPU-koden som produceras av kompilatorn. I senare stadier av projektet skiftades fokus till att fokusera på den källkodsbaserade backenden för HIP-plattformen, eftersom det bedömdes vara viktigast att kunna generera fungerande kod vid projektets slut, även om optimering gick förlorad. Detta kan ses som en förklaring till språkets låga prestanda jämfört med Thrust.

6.4.3 Prestandamätningar

För att utvärdera prestandan av kompilatorn och programspråket `gpulang` hade det varit önskvärt att utföra flera olika och varierade tester. Till följd av tidsbrist genomfördes endast en prestandamätning i form av körtidsmätningar av radix-sortering. Resultatet av detta är att slutsatser om språkets prestanda är svåra att generalisera.

7
Slutsatser

Projektet resulterade i utvecklingen av ett programspråk med användbara abstraktioner, men framtida arbete krävs för att kunna nyttja dess fulla potential för optimering.

7.1 Språkdesign

Abstraktionen av streams har potential. En abstrakt representation av data möjliggör många optimeringar och linjäritet kan möjliggöra mutering *in-place*. Generiska funktioner är mycket användbara för att låta användaren definiera återanvändbara operatorer för olika former av data.

7.2 Implementation

Den nuvarande implementationen med en källkodsbaserad backend visar på lovande prestanda som ger vissa indikationer till att vara jämförbar med Futhark, men den är långt ifrån lika optimerad som Thrust. Kompilatorn hanterar lexikal- och syntaxanalys korrekt med stöd för feldiagnostik vid felaktiga program. Vid framtida arbete bör affina typer implementeras för ökad typsäkerhet. Att undersöka en backend som avviker från transpilering har också potential för framtida arbete.

7.3 Samhälleliga och etiska aspekter

Vi anser att risken är försumbar att denna forskning kommer leda till direkt skada för någon individ eller grupp av individer. Däremot finns det vissa aspekter som bör tas i beaktning för alla teknologier under snabb utveckling. Vi har identifierat ett par områden där detta är fallet för GPU-hårdvara och har reflekterat kring vår roll inom dessa.

7.3.1 Klimatpåverkan

Det finns en enorm efterfrågan av GPU-teknologi i nuläget, som drivs till stor del av tillämpningar inom maskininlärning. Denna efterfrågan driver inte bara en ökad produktion i nuläget, utan också en kraftigt stigande energiförbrukning för att använda hårdvaran till projekt på stor skala. Ökad utveckling av mjukvara för att kunna köra ännu mer beräkningar på GPU-hårdvara är en bidragande faktor som driver denna efterfrågan. Däremot är det också möjligt att om fler sorters beräkningar kan utföras mer effektivt på denna hårdvara än med nuvarande CPU-implementationer, så är det möjligt att minska energiförbrukningen hos datorer.

En annan faktor som bidrar till koldioxidutsläpp är elavfall. En modell som grundar sig på EU-direktiv för att behandla avfall är *avfallstrappan*, som består av fem steg, där återanvändning bör övervägas framför återvinning [37]. Vi anser att genom att använda och bygga vidare på mjukvara med öppen källkod så bidrar vi till att hårdvara kan användas under längre tid.

7.3.2 Militära tillämpningar och anknytning

Utvecklingen av GPU-teknologi drivs i dagsläget nästan uteslutande av två företag: NVIDIA och AMD [38]. NVIDIA har ett nära samarbete med USA:s försvarsdepartement för att utveckla teknik som används för bland annat drönarfotografi, flygelektronik (en: *avionics*), och generativa modeller för att hantera militära dokument [39]. NVIDIA planerar även att använda sin videostreamningsteknik för utbildning av nya militära rekryter [40]. AMD bidrar också med teknologi för flygelektronik, satellitkommunikation, och militär underrättelseverksamhet [41].

Även om mjukvaran i detta projekt har använt öppen källkod i den mån det går, så går det inte att ignorera att den kommer uteslutande att köras på arkitektur och hårdvara som bidrar finansiellt till dessa företag. Därmed känner vi att vi har ett ansvar att åtminstone vara pålästa nog om teknikens användning inom nuvarande konflikter.

Källförteckning

[1] TOP500.org, ”November 2024 | TOP500”. Åtkomstdatum: 11 februari 2025. [Online]. Tillgänglig vid: https://www.top500.org/lists/top500/2024/11/

[2] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, och A. R. LeBlanc, ”Design of ion-implanted MOSFET's with very small physical dimensions”, *IEEE Journal of solid-state circuits*, vol. 9, nr 5, s. 256–268, 1974.

[3] J. Hennessy och D. Patterson, *Computer Architecture: A Quantitative Approach*. i The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2017.

[4] D. Kirk och W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. i Applications of GPU Computing Series. Morgan Kaufmann Publishers, 2010.

[5] E. S. Larsen och D. McAllister, ”Fast Matrix Multiplies using Graphics Hardware”, i *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, 2001, s. 55.

[6] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, och M. Horowitz, ”CPU DB: Recording Microprocessor History: With this open database, you can mine microprocessor trends over the past 40 years.”, *Queue*, vol. 10, nr 4, s. 10–27, 2012.

[7] Köpenhamns universitets institution för datavetenskap (DIKU), ”Why Futhark?”. Åtkomstdatum: 22 april 2025. [Online]. Tillgänglig vid: https://futhark-lang.org/index.html

[8] NVIDIA Corporation, ”Thrust”. Åtkomstdatum: 22 april 2025. [Online]. Tillgänglig vid: https://developer.nvidia.com/thrust

[9] NVIDIA Corporation, ”Thrust: The C++ Parallel Algorithms Library”. Åtkomstdatum: 22 april 2025. [Online]. Tillgänglig vid: https://nvidia.github.io/cccl/thrust/

[10] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea, ”2. The Futhark Language”. Åtkomstdatum: 22 april 2025. [Online]. Tillgänglig vid: https://futhark-book.readthedocs.io/en/latest/language.html

[11] NVIDIA Corporation, ”NVIDIA Tesla V100 GPU Architecture: The world's most advanced data center GPU. v1.1”, aug. 2017. [Online]. Tillgänglig vid: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[12] NVIDIA Corporation, ”NVIDIA’s Next Generation CUDA™ Compute Architecture: Fermi™. v1.1.”, 2009. [Online]. Tillgänglig vid: https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[13] NVIDIA Corporation, ”NVIDIA CUDA Unleashes Power of GPU Computing”. Åtkomstdatum: 27 januari 2025. [Online]. Tillgänglig vid: https://web.archive.org/web/20070329144655/http://www.nvidia.com/object/IO_39918.html

[14] Advanced Micro Devices, Inc, ”What is ROCm?”. Åtkomstdatum: 29 januari 2025. [Online]. Tillgänglig vid: https://web.archive.org/web/20250124112321/https://rocm.docs.amd.com/en/latest/what-is-rocm.html

[15] Advanced Micro Devices, Inc, ”What is HIP?”. Åtkomstdatum: 13 februari 2025. [Online]. Tillgänglig vid: https://rocm.docs.amd.com/projects/HIP/en/docs-develop/what_is_hip.html

[16] A. V. Aho, M. S. Lam, R. Sethi, och J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2:a utgåvan. Pearson Education, Inc, 2006.

[17] V. R. Pratt, ”Top Down Operator Precedence”, i *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Boston, Massachusetts, USA: ACM, 1973, s. 41–51. doi: 10.1145/512927.512931.

[18] R. Milner, ”A theory of type polymorphism in programming”, *Journal of Computer and System Sciences*, vol. 17, nr 3, s. 348–375, 1978, doi: https://doi.org/10.1016/0022-0000(78)90014-4.

[19] P. Hudak, J. Hughes, S. Peyton Jones, och P. Wadler, ”A history of Haskell: being lazy with class”, i *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, i HOPL III. San Diego, California: Association for Computing Machinery, 2007, s. 12–11. doi: 10.1145/1238844.1238856.

[20] D. Walker, ”Substructural Type Systems”, *Advanced Topics in Types and Programming Languages*. The MIT Press, s. 3–43, 2004. doi: 10.7551/mitpress/1104.003.0003.

[21] P. Wadler, ”Linear types can change the world!”, *Programming concepts and methods*, vol. 3, nr 4, s. 1–2, 1990.

[22] S. Kan, Z. Chen, D. Sanán, och Y. Liu, ”Formally understanding Rust’s ownership and borrowing system at the memory level”, *Formal Methods in System Design*, vol. 64, s. 200–236, 2024, doi: 10.1007/s10703-024-00460-3.

[23] P. J. Landin, ”The Next 700 Programming Languages”, *Communications of the ACM*, vol. 9, nr 3, s. 157–166, 1966.

[24] Python Software Foundation, ”Python 3.13.3 Documentation | Indentation”. Åtkomstdatum: 28 april 2025. [Online]. Tillgänglig vid: https://docs.python.org/3/reference/lexical_analysis.html#indentation

[25] International Organization for Standardization, ”ISO/IEC 9899:2024 — Information technology — Programming languages — C”. 2024. Åtkomstdatum: 19 maj 2025. [Online]. Tillgänglig vid: https://www.iso.org/standard/82075.html

[26] D. Knuth, *The Art of Computer Programming, vol. II: Seminumerical Algorithms*. Addison-Wesley, 1969.

[27] P. Wadler och S. Blott, ”How to make ad-hoc polymorphism less ad hoc”, i *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, i POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, s. 60–76. doi: 10.1145/75277.75283.

[28] M. Harris, S. Sengupta, och J. D. Owens, *Parallel Prefix Sum (Scan) with CUDA*. Boston, MA: Addison-Wesley, 2007, s. 851–876. [Online]. Tillgänglig vid: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

[29] M. Harris, ”Optimizing Parallel Reduction in CUDA”. Åtkomstdatum: 01 maj 2025. [Online]. Tillgänglig vid: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[30] Advanced Micro Devices, Inc, ”HIP C++ Language Extensions: Warp Match Functions”. Åtkomstdatum: 01 maj 2025. [Online]. Tillgänglig vid: https://rocm.docs.amd.com/projects/HIP/en/latest/how-to/hip_cpp_language_extensions.html#warp-match-functions

[31] B. Stroustrup, *The C++ Programming Language*, 4:e uppl. Boston: Addison-Wesley, 2013.

[32] C. Click och M. Paleczny, ”A simple graph-based intermediate representation”, *ACM Sigplan Notices*, vol. 30, nr 3, s. 35–49, 1995.

[33] H. H. Seward, ”Information Sorting in the Application of Electronic Digital Computers to Business Operations”, Cambridge, MA, USA, 1954. Åtkomstdatum: 13 maj 2025. [Online]. Tillgänglig vid: https://bitsavers.trailing-edge.com/pdf/mit/whirlwind/R-series/R-232_Information_Sorting_in_the_Application_of_Electronic_Digital_Computers_to_Business_Operations_May54.pdf

[34] Köpenhamns universitets institution för datavetenskap (DIKU), ”Radix Sort”. Åtkomstdatum: 13 maj 2025. [Online]. Tillgänglig vid: https://futhark-lang.org/examples/radix-sort.html

[35] Köpenhamns universitets institution för datavetenskap (DIKU), ”List of Futhark Packages”. Åtkomstdatum: 13 maj 2025. [Online]. Tillgänglig vid: https://futhark-lang.org/pkgs/

[36] N. Satish, M. Harris, och M. Garland, ”Designing Efficient Sorting Algorithms for Manycore GPUs”, i *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, IEEE, 2009. Åtkomstdatum: 13 maj 2025. [Online]. Tillgänglig vid: https://mgarland.org/files/papers/gpusort-ipdps09.pdf

[37] Council of European Union, ”Directive 2008/98/EC of the European Parliament and of the Council of 19 November 2008 on waste and repealing certain Directives”. Åtkomstdatum: 19 maj 2025. [Online]. Tillgänglig vid: http://data.europa.eu/eli/dir/2008/98/2024-02-18

[38] A. Shilov, ”GPU Market 'Healthy and vibrant' in Q2 2023: Report”. Åtkomstdatum: 02 februari 2025. [Online]. Tillgänglig vid: https://www.tomshardware.com/news/gpu-market-healthy-and-vibrant-in-q2-2023-report

[39] NVIDIA Corporation, ”Generative AI in the Public Sector: The DoD's Strategic Plan”. Åtkomstdatum: 23 januari 2025. [Online]. Tillgänglig vid: https://www.nvidia.com/en-us/on-demand/session/gtc24-s63325/

[40] NVIDIA Corporation, ”Improving US Army Training with Cloud Game Streaming”. Åtkomstdatum: 23 januari 2025. [Online]. Tillgänglig vid: https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s51804/

[41] Advanced Micro Devices, Inc, ”Solutions for Aerospace and Defense”. Åtkomstdatum: 23 januari 2025. [Online]. Tillgänglig vid: https://www.amd.com/en/solutions/aerospace-and-defense.html

A
Kod för prestandamätningar

I följande kodstycken presenteras den kod som användes för prestandamätningar av radix-sortering i `gpulang`, Futhark, och Thrust. Även den genererade HIP-koden som `gpulangs` kompilator producerar visas i ett kodstycke.

Implementation av radix-sortering i `gpulang`

```
// Radix algorithm inspired by https://futhark-book.readthedocs.io/en/latest/fusion.html#radix-sort-revisited

fn check_bit_n(x: u32, b: u32): u32 {
    return (x >> b) & 1
}

// Struct for checking valid sorting
struct Flag {
    val: u32,
    f: i32,
}

fn less_than(a: Flag, b: Flag): Flag {
    if a.f == 1 || b.f == 1 {
        return Flag(val: 0, f: 1);
    } else {
        if a.val <= b.val {
            return b;
        } else {
            return Flag(val: 0, f: 1)
        }
    }
}

const u32_max = 4294967295;

fn main() {
    for (let n = 10, n <= 1e9, n *= 10) {
        print("")
        print(n)
        for (let i = 0, i < 10, i += 1) {
            let xs: {u32} = rand_array(0, u32_max, n);
            let _ = xs.map(x => x); // Forces xs to device

            start_timer();
            for (let b = 0, b < 32, b += 1) {
                let bits1 = xs.map(x => check_bit_n(x, b));
```

```
38                  let bits0 = bits1.map(x => 1-x);
39
40                  let offs = bits0.reduce((x, y) => x + y, 0);
41
42                  // Separating the scans outside avoids memory leak
43                  let idxs_tmp0 = bits0.scan((x, y) => x + y, 0);
44                  let idxs_tmp1 = bits1.scan((x, y) => x + y, 0);
45
46                  let idxs0 = bits0.map2(idxs_tmp0, (x, y) => x * y);
47                  let idxs1 = bits1.map2(idxs_tmp1, (x, y) => x * (y + offs));
48                  let idxs = idxs0.map2(idxs1, (x, y) => (x + y) - 1);
49
50                  let tmp = xs; // Copy xs
51                  xs = xs.scatter(tmp, idxs);
52              }
53              end_timer();
54
55              // Check that it is a valid sorting
56              let flags = xs.map(x => Flag(val: x, f: 0));
57              let is_sorted = flags.reduce(less_than, Flag(val: u32_max, f: 0)).f;
58              if is_sorted == 1 {
59                  let fs = xs.collect();
60                  print(fs)
61              }
62          }
63      }
64  }
```

Kodstycke 7: Implementation av radix-sortering i `gpulang` för körtidsmätningar.

Implementation av radix-sortering i Futhark

```
    1 -- Radix Sort and Blocked Radix Sort copied from
    2   --   https://github.com/diku-dk/sorts/blob/master/lib/github.com/diku-dk/sorts/
radix_sort.fut
    3
    4
    5 -- ISC License
    6 --
    7 -- Copyright (c) 2018. DIKU, University of Copenhagen
    8 --
    9 -- Permission to use, copy, modify, and/or distribute this software for
   10 -- any purpose with or without fee is hereby granted, provided that the
   11 -- above copyright notice and this permission notice appear in all
   12 -- copies.
   13 --
   14 -- THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL
   15 -- WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED
   16 -- WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE
   17 -- AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL
   18 -- DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
   19 -- PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
   20 -- TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
   21 -- PERFORMANCE OF THIS SOFTWARE.
   22
   23 -- Radix Sort
   24 local
   25 def radix_sort_step [n] 't
   26                     (xs: [n]t)
   27                     (get_bit: i32 -> t -> i32)
   28                     (digit_n: i32) : [n]t =
   29   let num x = get_bit (digit_n + 1) x * 2 + get_bit digit_n x
   30   let pairwise op (a1, b1, c1, d1) (a2, b2, c2, d2) = (a1 `op` a2, b1 `op` b2, c1
`op` c2, d1 `op` d2)
   31   let bins = xs |> map num
   32   let flags =
   33     bins
   34     |> map (\x ->
   35               ( i64.bool (x == 0)
   36               , i64.bool (x == 1)
   37               , i64.bool (x == 2)
   38               , i64.bool (x == 3)
   39               ))
   40   let offsets = scan (pairwise (+)) (0, 0, 0, 0) flags
   41   let (na, nb, nc, _nd) = last offsets
   42   let f bin (a, b, c, d) =
   43     (-1)
   44     + a * (i64.bool (bin == 0))
   45     + na * (i64.bool (bin > 0))
   46     + b * (i64.bool (bin == 1))
   47     + nb * (i64.bool (bin > 1))
   48     + c * (i64.bool (bin == 2))
```

```
   49     + nc * (i64.bool (bin > 2))
   50     + d * (i64.bool (bin == 3))
   51   let is = map2 f bins offsets
   52   in scatter (copy xs) is xs
   53
   54 local
   55 def exscan op ne xs =
   56   let s =
   57     scan op ne xs
   58     |> rotate (-1)
   59   let s[0] = ne
   60   in s
   61
   62 local
   63 def get_bin 't
   64             (k: i32)
   65             (get_bit: i32 -> t -> i32)
   66             (digit_n: i32)
   67             (x: t) : i64 =
   68   i64.i32
   69   <| loop acc = 0
   70      for i < k do
   71        acc + (get_bit (digit_n + i) x << i)
   72
   73 local
   74 def radix_sort_step_i16 [n] 't
   75                         (get_bit: i32 -> t -> i32)
   76                         (digit_n: i32)
   77                         (xs: [n]t) : ([n]t, [4]i64, [4]i16) =
   78   let num x = i16.i32 (get_bit (digit_n + 1) x * 2 + get_bit digit_n x)
   79   let pairwise op (a1, b1, c1, d1) (a2, b2, c2, d2) = (a1 `op` a2, b1 `op` b2, c1
`op` c2, d1 `op` d2)
   80   let bins = xs |> map num
   81   let flags =
   82     bins
   83     |> map (\x ->
   84               ( i16.bool (x == 0)
   85               , i16.bool (x == 1)
   86               , i16.bool (x == 2)
   87               , i16.bool (x == 3)
   88               ))
   89   let offsets = scan (pairwise (+)) (0, 0, 0, 0) flags
   90   let (na, nb, nc, nd) = if n == 0 then (0, 0, 0, 0) else last offsets
   91   let f bin (a, b, c, d) =
   92     i64.i16 ((-1)
   93              + a * (i16.bool (bin == 0))
   94              + na * (i16.bool (bin > 0))
   95              + b * (i16.bool (bin == 1))
   96              + nb * (i16.bool (bin > 1))
   97              + c * (i16.bool (bin == 2))
   98              + nc * (i16.bool (bin > 2))
   99              + d * (i16.bool (bin == 3)))
```

```
  let is = map2 f bins offsets
  in ( scatter (copy xs) is xs
     , map i64.i16 [na, nb, nc, nd]
     , [0, na, na + nb, na + nb + nc]
     )

def radix_sort [n] 't
               (num_bits: i32)
               (get_bit: i32 -> t -> i32)
               (xs: [n]t) : [n]t =
  let iters = if n == 0 then 0 else (num_bits + 2 - 1) / 2
  in loop xs for i < iters do radix_sort_step xs get_bit (i * 2)

-- Blocked Radix Sort
local
def blocked_radix_sort_step [n] [m] [r] 't
                            (get_bit: i32 -> t -> i32)
                            (digit_n: i32)
                            (xs: *[n * m + r]t) =
  let (blocks, rest) = split xs
  let (sorted_rest, hist_rest, offsets_rest) =
    radix_sort_step_i16 get_bit digit_n rest
  let (sorted_blocks, hist_blocks, offsets_blocks) =
    unflatten blocks
    |> map (radix_sort_step_i16 get_bit digit_n)
    |> unzip3
  let histograms = hist_blocks ++ [hist_rest]
  let sorted = sized (n * m + r) (flatten sorted_blocks ++ sorted_rest)
  let old_offsets = offsets_blocks ++ [offsets_rest]
  let new_offsets =
    histograms
    |> transpose
    |> flatten
    |> exscan (+) 0
    |> unflatten
    |> transpose
  let is =
    tabulate (n * m + r) (\i ->
                            let elem = sorted[i]
                            let bin = get_bin 2 get_bit digit_n elem
                            let block_idx = i / m
                            let new_offset = new_offsets[block_idx][bin]
                            let old_block_offset = i64.i16 old_offsets[block_idx][bin]
                            let old_offset = m * block_idx + old_block_offset
                            let idx = (i - old_offset) + new_offset
                            in idx)
  in scatter xs is sorted

def blocked_radix_sort [n] 't
                       (block: i16)
                       (num_bits: i32)
                       (get_bit: i32 -> t -> i32)
```

```
152                       (xs: [n]t) : [n]t =
153   let iters = if n == 0 then 0 else (num_bits + 2 - 1) / 2
154   let block = i64.i16 block
155   let n_blocks = n / block
156   let rest = n % block
157   let xs = sized (n_blocks * block + rest) xs
158   in sized n
159      <| loop xs = copy xs
160         for i < iters do
161           blocked_radix_sort_step get_bit (i * 2) xs
162
```

Kodstycke 8: Implementation av radix-sortering i Futhark för körtidsmätningar.

Implementation av radix-sortering i Thrust

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/execution_policy.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <thrust/random.h>
#include <iostream>
#include <chrono>

int main() {
    thrust::default_random_engine rng(0);
    thrust::uniform_int_distribution<uint32_t> dist;

    // Warm up
    // Generate random numbers on host.
    thrust::host_vector<uint32_t> h_vec(10000);
    thrust::generate(h_vec.begin(), h_vec.end(), [&] { return dist(rng); });

    // Transfer to device
    thrust::device_vector<uint32_t> d_vec = h_vec;

    // Sort on device
    thrust::stable_sort(thrust::device, d_vec.begin(), d_vec.end());

    for (size_t n = 10; n <= 100000000; n *= 10) {
        std::cout << "\n" << n << std::endl;
        for (int i = 0; i < 10; ++i) {
            // Generate random numbers on host.
            thrust::host_vector<uint32_t> h_vec(n);
            thrust::generate(h_vec.begin(), h_vec.end(), [&] { return dist(rng); });

            // Transfer to device
            thrust::device_vector<uint32_t> d_vec = h_vec;

            cudaDeviceSynchronize();
            auto start = std::chrono::high_resolution_clock::now();

            // Sort on device
            thrust::stable_sort(thrust::device, d_vec.begin(), d_vec.end());

            cudaDeviceSynchronize();
            auto end = std::chrono::high_resolution_clock::now();
          auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end
- start);

            // Output time in microseconds
            std::cout << elapsed.count() << std::endl;
        }
    }
```

```
50
51     return 0;
52 }
53
```

Kodstycke 9: Implementation av radix-sortering i Thrust för körtidsmätningar.

HIP-kod genererad av kompilatorn för `gpulang`

```
#define __HIP_PLATFORM_NVIDIA__
#include <cstdint>
#include <ctime>
#include <math.h>
#include <string>
#include <hip/hip_runtime.h>
#include "./kernels/msvc_defines.h"
#include "./kernels/helper.h"
#include "./kernels/map_kernel.h"
#include "./kernels/map2_kernel.h"
#include "./kernels/scan_kernel.h"
#include "./kernels/reduce_kernel.h"
#include "./kernels/scatter_kernel.h"
#include "./kernels/collect_kernel.h"
struct check_bit_n {
__host__ __device__
uint32_t operator()(uint32_t x,uint32_t b) const {
return ((x >> b) & 1);
}
};

struct Flag {
uint32_t val;
int32_t f;

__host__ __device__ Flag(uint32_t val_, int32_t f_) : val(val_), f(f_) {}

__host__ __device__ Flag(const Flag& other) : val(other.val), f(other.f) {}
};
struct less_than {
__host__ __device__
Flag operator()(Flag a,Flag b) const {
if (((a.f == 1) || (b.f == 1))) {
return Flag {
0,
1,
};
} else {
if ((a.val <= b.val)) {
return b;
} else {
return Flag {
0,
1,
};
};
};
}
};

```

```
__device__ const uint32_t u32_max= 4294967295;
struct __gpulang_lambda_uint32_t_1 {
__host__ __device__ __gpulang_lambda_uint32_t_1(){}
__host__ __device__
uint32_t operator()(uint32_t x) const {
return x;
}
};

struct __gpulang_lambda_uint32_t_2 {
uint32_t b;
__host__ __device__ __gpulang_lambda_uint32_t_2(uint32_t b_) : b(b_){}
__host__ __device__
uint32_t operator()(uint32_t x) const {
return check_bit_n{}(x,b);
}
};

struct __gpulang_lambda_uint32_t_3 {
__host__ __device__ __gpulang_lambda_uint32_t_3(){}
__host__ __device__
uint32_t operator()(uint32_t x) const {
return (1 - x);
}
};

struct __gpulang_lambda_uint32_t_4 {
__host__ __device__ __gpulang_lambda_uint32_t_4(){}
__host__ __device__
uint32_t operator()(uint32_t x,uint32_t y) const {
return (x + y);
}
};

struct __gpulang_lambda_uint32_t_5 {
__host__ __device__ __gpulang_lambda_uint32_t_5(){}
__host__ __device__
uint32_t operator()(uint32_t x,uint32_t y) const {
return (x + y);
}
};

struct __gpulang_lambda_uint32_t_6 {
__host__ __device__ __gpulang_lambda_uint32_t_6(){}
__host__ __device__
uint32_t operator()(uint32_t x,uint32_t y) const {
return (x + y);
}
};

struct __gpulang_lambda_uint32_t_7 {
__host__ __device__ __gpulang_lambda_uint32_t_7(){}
```

```
103 __host__ __device__
104 uint32_t operator()(uint32_t x,uint32_t y) const {
105 return (x * y);
106 }
107 };
108
109 struct __gpulang_lambda_uint32_t_8 {
110 uint32_t offs;
111 __host__ __device__ __gpulang_lambda_uint32_t_8(uint32_t offs_) : offs(offs_){}
112 __host__ __device__
113 uint32_t operator()(uint32_t x,uint32_t y) const {
114 return (x * (y + offs));
115 }
116 };
117
118 struct __gpulang_lambda_uint32_t_9 {
119 __host__ __device__ __gpulang_lambda_uint32_t_9(){}
120 __host__ __device__
121 uint32_t operator()(uint32_t x,uint32_t y) const {
122 return ((x + y) - 1);
123 }
124 };
125
126 struct __gpulang_lambda_Flag_10 {
127 __host__ __device__ __gpulang_lambda_Flag_10(){}
128 __host__ __device__
129 Flag operator()(uint32_t x) const {
130 return Flag {
131 x,
132 0,
133 };
134 }
135 };
136
137 int32_t main () {
138 srand(time(NULL));
139 for (uint32_t n = 10; (n <= 1000000000); n = (n * 10)) {
140 print_gpulang("");
141 print_gpulang(n);
142 for (int32_t i = 0; (i < 10); i = (i + 1)) {
143 uint32_t* rand_arr_0 = new uint32_t[n];
144 for (uint32_t i = 0; i < n; ++i) {
145 rand_arr_0[i] = static_cast<uint32_t>((rand() % (u32_max - 0)) + 0);
146 }
147 uint32_t* xs = rand_arr_0;
148 uint32_t* d_xs;
149 hipMalloc(&d_xs, n * sizeof(uint32_t));
150 hipMemcpy(d_xs, xs, n * sizeof(uint32_t), hipMemcpyHostToDevice);
151    uint32_t*    d_xs_map_0    =    mapDevice<uint32_t,    uint32_t>(d_xs,    n,
__gpulang_lambda_uint32_t_1());
152 uint32_t* _ = nullptr;
153 uint32_t* d__ = d_xs_map_0;
```

```
timer_gpulang::start();
for (uint32_t b = 0; (b < 32); b = (b + 1)) {
    uint32_t*    d_xs_map_1    =    mapDevice<uint32_t,    uint32_t>(d_xs,    n, __gpulang_lambda_uint32_t_2(b));
uint32_t* bits1 = nullptr;
uint32_t* d_bits1 = d_xs_map_1;
   uint32_t*   d_bits1_map_0   =   mapDevice<uint32_t,   uint32_t>(d_bits1,   n, __gpulang_lambda_uint32_t_3());
uint32_t* bits0 = nullptr;
uint32_t* d_bits0 = d_bits1_map_0;
uint32_t offs = reduceDevice<uint32_t>(d_bits0, n, __gpulang_lambda_uint32_t_4(), 0);
     uint32_t*     d_bits0_scan_0     =     scanDevice<uint32_t>(d_bits0,     n, __gpulang_lambda_uint32_t_5(), 0);
uint32_t* idxs_tmp0 = nullptr;
uint32_t* d_idxs_tmp0 = d_bits0_scan_0;
     uint32_t*     d_bits1_scan_1     =     scanDevice<uint32_t>(d_bits1,     n, __gpulang_lambda_uint32_t_6(), 0);
uint32_t* idxs_tmp1 = nullptr;
uint32_t* d_idxs_tmp1 = d_bits1_scan_1;
 uint32_t*  d_bits0_map2_1  =  map2Device<uint32_t,  uint32_t,  uint32_t>(d_bits0, d_idxs_tmp0, n, __gpulang_lambda_uint32_t_7());
uint32_t* idxs0 = nullptr;
uint32_t* d_idxs0 = d_bits0_map2_1;
 uint32_t*  d_bits1_map2_2  =  map2Device<uint32_t,  uint32_t,  uint32_t>(d_bits1, d_idxs_tmp1, n, __gpulang_lambda_uint32_t_8(offs));
uint32_t* idxs1 = nullptr;
uint32_t* d_idxs1 = d_bits1_map2_2;
 uint32_t*  d_idxs0_map2_0  =  map2Device<uint32_t,  uint32_t,  uint32_t>(d_idxs0, d_idxs1, n, __gpulang_lambda_uint32_t_9());
uint32_t* idxs = nullptr;
uint32_t* d_idxs = d_idxs0_map2_0;
uint32_t* tmp = nullptr;
uint32_t* d_tmp = d_xs;
uint32_t* d_xs_scatter_2 = scatterDevice<uint32_t>(d_tmp, d_xs, n, n, d_idxs);
d_xs = d_xs_scatter_2;
hipFree(d_tmp);
hipFree(d_idxs_tmp1);
hipFree(d_bits1);
hipFree(d_idxs_tmp0);
hipFree(d_idxs0);
hipFree(d_bits0);
hipFree(d_idxs);
hipFree(d_idxs1);
};
timer_gpulang::end();
Flag* d_xs_map_3 = mapDevice<uint32_t, Flag>(d_xs, n, __gpulang_lambda_Flag_10());
Flag* flags = nullptr;
Flag* d_flags = d_xs_map_3;
int32_t is_sorted = reduceDevice<Flag>(d_flags, n, less_than{}, Flag {
u32_max,
0,
```

```
198 }).f;
199 if ((is_sorted == 1)) {
200 uint32_t* fs = collectDeviceToHost(d_xs, n * sizeof(uint32_t));
201 print_array_gpulang<uint32_t>(fs, n);
202 };
203 hipFree(d_xs);
204 hipFree(d__);
205 hipFree(d_flags);
206 };
207 };
208 hipDeviceSynchronize();
209 hipDeviceReset();
210 return 0;
211 }
212
213
```

Kodstycke 10: HIP-kod genererad av `gpulangs` kompilator för radix-sortering på en NVIDIA GPU.