Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Peter Žužek

Implementacija knjižnice SYCL za heterogeno računanje

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Ljubljana, 2016

University of Ljubljana
Faculty of Computer and Information Science

Peter Žužek

Implementation of the SYCL Heterogeneous Computing Library

masters thesis

the 2nd cycle masters study programme
computer and information science

Supervisor: izr. prof. dr. Patricio Bulić
Co-supervisor: doc. dr. Boštjan Slivnik

Ljubljana, 2016

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Peter Žužek

Implementacija knjižnice SYCL za heterogeno računanje

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Patricio Bulić
Somentor: doc. dr. Boštjan Slivnik

Ljubljana, 2016

Declaration of Masters Thesis authorship

I, the undersigned Peter Žužek am the author of the Master Thesis entitled:

Implementation of the SYCL Heterogeneous Computing Library

With my signature, I declare that:

- the submitted Thesis is my own unaided work under the supervision of izr. prof. dr. Patricio Bulić and co-supervision of doc. dr. Boštjan Slivnik,
- all electronic forms of the Masters Thesis, title (Slovenian, English), abstract (Slovenian, English) and keywords (Slovenian, English) are identical to the printed form of the Masters Thesis,
- I agree with the publication of the electronic form of the Masters Thesis in the collection "Dela FRI".

In Ljubljana, 18. March 2015 Author's signature:

Acknowledgments

I would like to thank both of my mentors, izr. prof. dr. Patricio Bulić and doc. dr. Boštjan Slivnik, people from Codeplay Software, and Klemen Rahne who helped running tests, for helping me with this thesis.

Peter Žužek, 2016

Contents

Povzetek **iii**

Abstract **v**

Razširjeni povzetek **vii**

1 Introduction **1**

2 Heterogeneous Computing **5**

2.1 Overview of a modern CPU 5

2.2 Overview of GPU architectures 8

2.3 Other processing units 14

2.4 The interconnect 17

2.5 Heterogeneous System Architecture 19

3 Programming framework **23**

3.1 OpenCL 23

3.2 SYCL 33

4 Implementation **43**

4.1 Anatomy of a sycl-gtx application 44

4.2 The OpenCL code generator 47

4.3 Limitations 51

4.4 Example code 56

4.5 Porting the OpenCL example to sycl-gtx 63

4.6 Additional remarks 66

5 Tests **67**

5.1 smallpt 67

5.2 Porting smallpt to sycl-gtx 71

5.3 Testing environment 75

5.4 Results 77

6 Conclusion **85**

List of Acronyms

Cg	C for Graphics
CISC	Complex Instruction Set Architecture
CPU	Central Processing Unit
CU	Compute Unit
CUDA	Compute Unified Device Architecture
DSP	Digital Signal Processor
EU	Execution Unit
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FMA	Fused Multiply-Accumulate
FPGA	Field Programmable Gate Array
GPC	Graphic Processing Cluster
GPU	Graphics Procesing Unit
GPGPU	General Purpose Graphics Processing Unit
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HLSL	High Level Shader Language
hQ	Heterogeneous Queuing
HSA	Heterogeneous System Architecture
HSAIL	Heterogeneous System Architecture Intermediate Language
hUMA	Heterogeneous Unified Memory Architecture
ILP	Instruction Level Parallelism
IPC	Instructions Per Clock

ISA	Instruction Set Architecture
JIT	Just-In-Time
MAC	Multiple-Accumulate
OoO	Out-of-Order
OpenCL	Open Computing Language
OpenMP	Open Multi-Procesing
PCIe	Peripheral Component Interconnect Express
PE	Procesing Element
RAII	Resource Acquisition Is Initialization
RISC	Reduced Instruction Set Architecture
RNG	Random Number Generator
RTTI	Run-time Type Information
SIMD	Single Instruction Multiple Data
SMM	Streaming Multiprocessor Maxwell
SMT	Simultaneous Multi Threading
SoC	System-On-Chip
SPIR	Standard Portable Intermediate Representation
SPMD	Single Program Multiple Data
SYCL	Though this may seem like an abbreviation, it is not
TDP	Thermal Design Point
TSMC	Taiwan Semiconductor Manufacturing Company
VLIW	Very Large Instruction Word
VPU	Vector Procesing Unit

Povzetek

Heterogeno računalništvo postaja vedno bolj popularno zaradi zmanjšanega napredka pri hitrosti osredjih procesorjev, izjemne rasti zmogljivosti grafičnih procesorjev in razvoja novih programabilnih čipov, razvitih za specifične naloge. Vendar je programiranje heterogenih sistemov še vedno zapleteno zaradi zelo različne strojne opreme ter potrebe po podvajanju podatkov in sinhronizaciji. Specifikacija SYCL je bila razvita z namenom poenostavitve heterogenega programiranja, kar doseže z naslanjanjem na OpenCL in moderni C++. Odprtokodne implementacije SYCL-a še ni bilo, v čemer smo videli priložnost za razvoj lastne. Odločili smo se, da ne bomo prilagali obstoječih prevajalnikov ali celo razvili novega, temveč nam je uspelo udejaniti velik del specifikacije SYCL tako, da smo razvili generator OpenCL kode, ki prevaja SYCL kodo tik pred izvajanjem, t.j. Just-In-Time. Naše delo je bilo povzeto v članku "An Overview of sycl-gtx", objavljenem na konferenci PPoPP 2016.

Ključne besede

SYCL, OpenCL, heterogeno, vzporedno, JIT

Abstract

Heterogeneous computing is becoming more popular with the lack of CPU performance increases, the exceptional rate of GPU performance growth, and the emergence of other programmable computing elements. However, programming heterogeneous systems is still problematic due to differing hardware, explicit data copying, and synchronization. The SYCL specification aims to simplify heterogeneous programming by building on top of OpenCL and employing modern C++. However, there is no open-source implementation of SYCL available, which presented an opportunity for us to develop one. We restricted ourselves to not modify any existing compilers or write new ones, but we managed to implement a large part of the SYCL specification by developing an OpenCL code generator that compiles SYCL code in a Just-In-Time manner. Our work was summarized in an article called "An Overview of sycl-gtx", which was presented at the PPoPP 2016 conference.

Keywords

SYCL, OpenCL, heterogeneous, parallel, JIT

Razširjeni povzetek

Pri razvoju centralnih procesnih enot (CPE) je vedno težje doseči višjo zmogljivost. Zgodovinsko gledano se je razvoj osredotočal na pohitritev izvajanja enega zaporedja kode, ali prek povečevanje frekvence CPE ali prek večjega števila ukazov, ki jih CPE lahko izvede v enem urinem taktu (Instructions Per Clock oz. IPC). Razvoj proizvodnega procesa je običajno omogočal oboje – slavni Moorov zakon predvideva podvojitev števila tranzistorjev na enoto površine vsaki dve leti. Ti tranzistorji so se porabili za razvoj novih zmogljivosti CPE, kar je vodilo v višanje IPC, obenem pa so manjši tranzistorji omogočili višje frekvence. CPE so se ponašale z vedno naprednejšo notranjo arhitekturno zasnovo, npr. izvajanje strojnih ukazov v drugačnem zaporedju, kakor so prevzeti (Out-of-Order oz. OoO), uporaba cevovodov, prevzem in izvajanje večih ukazov v istem urinem taktu, uporaba pomnilniške hierarhije z namenom skrivanja zakasnitev pri dostopu do glavnega pomnilnika, predvidevanje pogojnih skokov ipd.

Vendar so se v zgodnjih 2000-ih pojavile težave – višanje frekvence CPE ni bilo več smotrno, kajti potreba po električni moči in proizvodnja toplote sta bili nenadoma previsoki. Namesto višanja frekvence je še bolj pomemben postal razvoj notranje arhitekture CPE. Industrija se je posvetila CPE z visokim IPC in nižjimi frekvencami [2]. Sčasoma je bilo možno dvigniti tako IPC kakor frekvenco, vendar je bil ta dvig drastično nižji od zgodovinskega razvoja. Moderna CPE Intel Core i7-6700K (izdana pozno 2015) se ponaša s frekvenco 4 GHz [3], medtem ko je CPE Intel Pentium 4 HT 3.4 [4] dosegla 3.4 GHz že v 2004, kar je manj kot 18-odstotno zvišanje v 11 letih. In čeprav

je bil začetni skok v IPC pri omenjeni spremembi relativno visok, se je kasneje tudi razvoj IPC upočasnil, kajti i7-6700K ima le 22% višji IPC kakor CPE i7-2600K [5], ki je bila izdana leta 2011.

Vseeno se je Moorov zakon obdržal skozi vsa ta leta in še vedno velja v letu 2016 (čeprav je videti, da se upočasnjuje) – čemu se torej posvečajo vsi dodatni tranzistorji na CPE? Večinoma se uporabijo za vgradnjo ostalih komponent na isto vezje poleg CPE. CPE so pridobile več jeder, integrirano grafiko in ostale specializirane enote. Dodajanje jeder je relativno enostavno, vendar z večimi jedri naloga izkoriščanja polne zmogljivosti CPE prenese na programerja, ki se mora naučiti pisati večnitno kodo. Programerji imajo še dandanes težave s tem, kajti tradicionalna programerska orodja niso bila spisana z večnitnostjo v mislih.

Poleg CPE so zanimiv razvoj doživele tudi grafične procesne enote (GPE) [6, 7], ki so se sprva uporabljale izključno za obdelavo računalniške grafike, sčasoma pa so pridobile zmogljivosti bolj splošnega računanja. Grafična opravila lahko bolj učinkovito izrabijo dodatne tranzistorje kakor aplikacije, spisane za CPE, zato je napredek zmogljivosti GPE opazno presegel napredek zmogljivosti CPE. GPE so postale bolj podobne mnogojedrnim CPE (sicer s šibkejšimi jedri). Vendar je programiranje zahtevno že za mnogojedrne CPE, kaj šele za GPE, ki so kljub podobnostim še vedno precej drugačne od CPE, poleg tega pa se mora programer ukvarjati še s komunikacijo med CPE in GPE.

Pojavila se je torej potreba po poenotenju programiranja in komunikacije med CPE, GPE in po možnosti ostalimi računskimi enotami v sistemu, ki jih je možno programirati. Ker se računske enote precej razlikujejo med seboj, izvajajo se pa vzporedno, se tak princip imenuje heterogeno programiranje. Iz te potrebe se je razvil standard OpenCL (Open Computing Language oz. odprt računski jezik) [8], ki skrije nekatere razlike med računskimi enotami (Compute Unit oz. CU) in priskrbi enoten način programiranja teh enot. CPE, GPE, procesorje digitalnega signala (Digital Signal Processor) in katerokoli enoto, ki sledi standardu OpenCL, se lahko programira na isti način.

Poleg poenotenja heterogenih enot je OpenCL zasnovan z vzporednostjo v mislih, kar poenostavi programiranje večjedrnih enot.

Čeprav je OpenCL poenostavil in poenotil programiranje heterogenih enot, je za programiranje še vedno relativno zahteven. Programer mora namreč ročno poskrbeti za nizkonivojske podrobnosti, kot je priprava računskih enot, rezervacija in sproščanje pomnilnika in podatkov, ki si jih enote izmenjajo, opravljanje sinhronizacije itd. Poleg tega je OpenCL osnovan na jeziku C, ki je sicer učinkovit, vendar mu manjkajo naprednejše možnosti višjenivojskih jezikov, ki bi poenostavile programiranje. Poleg OpenCL obstajajo druge rešitve za enostavno vzporedno programiranje (npr. OpenMP [9]) ali za programiranje GPE v jeziku C++ (C++AMP), vendar nobena ne ponuja tako splošnega pristopa k heterogenemu računanju kakor OpenCL. Možna izjema bi bila CUDA [10], ki je sicer zelo podobna OpenCL, le da se naslanja na C++. Največji problem CUDE je vezanost na GPE podjetja Nvidia, kar CUDI bistveno omeji razširjenost.

Zaradi teh razlogov se je skupina Khronos, ki skrbi za razvoj OpenCL, odločila za nov standard, ki bi poenostavil pristop OpenCL s pomočjo C++ v moderni različici. Ta nov standard so poimenovali SYCL [11] in marca 2014 izdali provizorično specifikacijo standarda, ki pa ni imela nobene konkretne implementacije. Lastno implementacijo je najavilo podjetje Codeplay Software, ki je eden izmed glavnih pobudnikov standarda, vendar so se odločili za zaprto, komercialno rešitev [12]. Kmalu se je pojavila odprtokodna implementacija SYCL-a po imenu triSYCL, vendar ni bilo videti, da bi se razvoju le-te posvečalo kaj dosti pozornosti.

Tako smo se odločili, da prispevamo lastno odprtokodno implementacijo SYCL-a, ki smo jo izdali pod permisivno licenco [14]. Glavni namen te naloge je bil razvoj čim večjega dela SYCL specifikacije brez uporabe posebnega prevajalnika, temveč le kot knjižnico, spisano v jeziku C++11, ki med izvajanjem programa (Just-In-Time oz. JIT) prevaja SYCL kodo v OpenCL. Uspelo nam je razviti veliko osnovnih elementov standarda, popisati naše delo tako v javnem repozitoriju kode kakor v tej nalogi in pognati ter ana-

lizirati par poskusov. Na ta način je tudi ta naloga zasnovana: začne se s predstavitvijo heterogenega računanja, sledi pregled programerskih okolij OpenCL in SYCL, nato obrazložitev našega dela pri samem razvoju specifikacije in na koncu so rezultati naših poskusov.

Chapter 1

Introduction

It has become increasingly more difficult to increase performance of the traditional Central Processing Unit (CPU). Historically, increasing CPU performance focused on executing a single stream of code as fast as possible, either by raising the CPU clock frequency, or by increasing the amount of work the CPU can perform in a single clock cycle (Instructions Per Clock – IPC). The manufacturing improvements usually allowed both – the famous Moore's Law predicted a doubling of transistors per area every 18 months and those transistors could be used to implement new functionality, but smaller transistors also allowed for higher frequencies. CPUs gained more and more advanced architectural designs – making the execution stream Out-of-Order (OoO), employing pipelines, fetching and executing multiple instructions at a time, implementing memory caches to hide the memory access latency [1], predicting conditional branches etc.

But problems emerged in the early 2000s – it was no longer feasible to increase the CPU frequency, because the power requirements were suddenly too high. Since major frequency increases were out of the question, it became clear that architectural changes played an even larger role now. The industry focused on high IPC, lower frequency CPUs [2], and was eventually able to raise both the IPC and the frequency, but much lower than what were historical standards. A modern (late 2015) Intel Core i7-6700K has a

frequency of 4 GHz [3], while the Intel Pentium 4 HT 3.4 [4] reached 3.4 GHz already in 2004, less than an 18% increase in 11 years. And even though the initial jump in IPC after this shift was substantial, the i7-6700K only has a 22% higher IPC than the i7-2600K [5], released in 2011.

However, Moore's Law persisted through all these years and still holds true even in 2016 (although it has seemingly slowed down slightly) – so what are all the added transistors used for? Mostly to integrate other components on the same chip alongside the CPU. CPUs started employing more cores, integrated graphics, and other, specialized hardware. Adding more cores is relatively simple, but the burden of extracting performance then falls on the programmer, who has to learn how to write parallel code – programmers still struggle with this, as traditional programming tools weren't written with parallel code in mind.

There has been another interesting development alongside CPUs – graphics processing units (GPU) [6, 7], which were first used only to process graphics, but later gained general purpose computing capabilities. Graphics workloads can use more transistors more efficiently than a CPU, so the increases in graphics performance were more evident. GPUs became similar to CPUs with a lot of (lower performing) cores, but as mentioned, parallel programming is already difficult for the CPU. Additionally, despite the similarities, a GPU is still pretty different from a CPU, while the programmer also needs to take care of the communication between them.

A need emerged to unify programming and inter-communication of the CPU, the GPU, and any other programmable processing elements. Because the computing elements differ from each other, but are programmed to execute alongside each other, this is called heterogeneous computing. A standard was developed, called OpenCL (for Open Computing Language) [8], which abstracted the different computing elements into Compute Units (CUs) and provided a unified way to program them – CPUs, GPUs, Digital Signal Processors, and any other CUs that conform to the OpenCL standard can be programmed in the same manner. Along with unifying heterogeneous CUs,

OpenCL is also designed to be parallel, which simplifies extracting performance from multiple cores.

But even though OpenCL unified heterogeneous programming with an open standard, it still wasn't simple. In OpenCL, the programmer needs to manually take care of many low-level details, like initializing the CUs, allocating and releasing memory that is exchanged between CUs, performing synchronization, etc. Additionally, OpenCL is designed upon the C language, which, while efficient, lacks more advanced features of higher level languages that would simplify programming. There are alternatives for simple parallel programming (e.g. OpenMP [9]), or for GPU-accelerated C++ (C++AMP), but none offer the very general approach to heterogeneous computing. A notable exception may be CUDA [10], which is very similar to OpenCL, but relies on C++. The problem is that CUDA is exclusive to NVIDIA GPUs, which significantly limits its reach.

That's why members of the Khronos Group, who oversights OpenCL development, decided on a standard that would greatly simplify OpenCL programming using modern C++. They called the standard SYCL [11] and released a provisional specification in March 2014. However, this was only a specification, with no available implementation at the time. Codeplay Software, one of the leading contributors to SYCL, announced their own implementation [12], but also that it would be proprietary. A project emerged to provide an open-source implementation, called triSYCL [13], but it wasn't clear whether any active development was going on.

That's when we decided to implement the SYCL specification on our own, releasing it under a permissive open source license [14]. The main goal was to develop as much of the SYCL specification without using a special compiler, but rather as just a C++11 library that performs JIT compilation at runtime. Over the course of this Thesis we managed to implement a lot of essential functionality, document what has been done, and run some experiments. That is also how the Thesis is structured: a bit of explanation of heterogeneous computing, followed by an overview of OpenCL and

SYCL, then an explanation of our implementation efforts, and finally the experimental results.

Chapter 2

Heterogeneous Computing

In this chapter we present the need for heterogeneous computing. We start by explaining traditional computing, using a single Central Processing Unit, continue with the history of Graphics Processing Units, how GPUs became more CPU-like in order to allow general purpose computation, discuss some other approaches to accelerating computing, present a common bottleneck in heterogeneous computing, the interconnect, and finally discuss a modern heterogeneous architecture.

2.1 Overview of a modern CPU

We already discussed some of the historical developments of CPUs in the introduction, so in this section we present the Intel Core i7-6700K [3] as an example of a modern high performance CPU.

The i7-6700K is based on Intel's Skylake architecture and was introduced to the market in september 2015. It is built using Intel's 14nm manufacturing process, their second generation process to employ FinFET transistors. Intel did not disclose the number of transistors for the chip, although estimates range from 1.4 to 1.7 billion, with a die size of $122mm^2$. It incorporates four Skylake cores, 24 GPU compute units, 8 MB L3 cache (shared between CPU and GPU), and a DDR4/LPDDR3 memory controller. We focus here on just

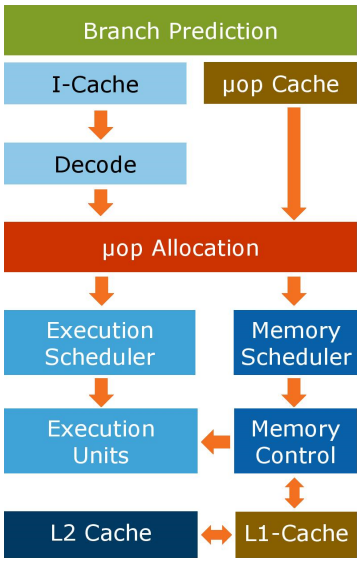

Figure 2.1: Simplified Skylake pipeline (figure source: [3]). Only the execution units perform work on data and the caches help with memory bandwidth, other parts are designed for extracting IPC.

the Skylake core and briefly discuss some other elements in other sections.

The i7-6700K has a base frequency of 4 GHz – the frequency is guaranteed to run within it's 91W Thermal Design Point. It also features a boost frequency of 4.2 GHz, which it can achieve on one core for short periods of time. The idea is that since some common workloads cannot be parallelized, but take a relatively short time to complete, the CPU can ramp up the frequency on that core without exceeding its TDP. The boost here is actually quite low, only 5%, at least compared to mobile processors that are also based on the Skylake architecture – like the Intel Core m7-6Y75, which has a TDP of 4.5W, a base frequency of 1.2 GHz, and can boost up to 3.1 GHz, an increase of 158%. These frequencies are achieved as a combination of the low power manufacturing process and a suspected pipeline length of 14 to 19 stages, depending on the instruction – the exact number wasn't released, but is thought not to have changed much from Intel's Haswell architecture [15].

Because clockspeeds have somewhat stagnated since the failure of Dennard scaling [16], CPUs rely heavily on extracting Instruction Level Parallelism (ILP) from code in order to achieve high performance. Intel processors are externally (visible to the programmer) CISC designs, but internally the CISC instructions are converted into RISC-like μ-ops – ever since Intel's Sandy Bridge architecture, the core also includes a μ-op cache, alongside the regular instruction cache. Before instructions are even fetched from memory, the branch predictor tries to guess the location of the next instruction – a misprediction incurs higher penalties in longer pipelines, but the design of a good branch predictor is something of a secret in the industry. To extract ILP, the Skylake core is Out-of-Order (OoO), storing 224 instructions in the OoO window for potential reordering, trying to utilize its functional units as efficiently as possible. The Skylake core can dispatch six fetched μ-ops at once to the scheduler queue, which then in turn dispatches six μ-ops to the execution units. The execution units consist of integer and floating point ALUs as well as load and store units.

Memory accesses are cached in the L1, L2, and L3 caches. The Level 1 cache is split into 32 KB of instruction cache and 32 KB of data cache. Each core also has 256 KB of L2 cache and 8 MB L3 is shared between all cores.

The Skylake core, implementing a 64-bit x86-compliant architecture, features 8 legacy general-purpose registers, extended from 32-bit to 64-bit, and 8 new 64-bit general-purpose registers. However, as is typical of a CISC instruction set, it also features many special purpose registers [17, 18].

The i7-6700K also features HyperThreading, a commercial name for two-way Simultaneous Multi Threading (SMT). This is done by storing two sets of registers (two contexts [19]), so when some instruction is waiting for an action to complete (for example waiting for data to be retrieved from the main memory), the core can switch to the other context and proceed working on an independent set of instructions.

2.2 Overview of GPU architectures

2.2.1 History of Graphics Processing Units

Computer graphics have evolved through the years – at first, only simple text input to the screen was needed. But as CPUs became faster, 3D rendering became feasible, and 3D scenes got more and more complex. The basic steps needed to be taken to display a 3D scene on a 2D screen go like this [6]:

1. Set up the scene (move objects, camera, ...)
2. Simplify the actual displayed scene – remove invisible objects, reduce the detail level for objects that are far away from the camera etc.
3. Transform the scene to a 2D view
4. Calculate lighting
5. Prepare renderable triangles (Triangle Setup and Clipping)
6. Render the triangles

Steps 3–6 are considered to be the 3D graphics pipeline, but at first it was all computed on the CPU. Step 6 was the first to be moved to a dedicated graphics processor. Through time, step 5, 4, and 3 were also brought to the graphics processor (in that order), which caused Nvidia to coin the term GPU in 1999, since the whole 3D graphics pipeline was now handled by the graphics processor instead of the CPU.

Rendering a scene is not that complex of a task, but rather just very repetitive – performing a simple independent calculation for every vertex, triangle, and pixel, and then perform some rendering operations (ROP stage) to actually output to display. A single core CPU would take a long time to perform all operations (e.g. render about a million pixels 60 times per second), so offloading the graphics pipeline to specialized graphics hardware brought large speed improvements and allowed the CPU to focus more on other aspects of the application. It was also relatively simple to add multiple pipelines to a

GPU, so it was easier to gain performance from adding transistors to a GPU compared to a single core CPU. This allowed the GPU performance growth to significantly outpace CPU performance growth [6].

The idea of a graphics pipeline was introduced by Silicon Graphics Inc. (SGI), who also introduced OpenGL in 1989 [7], an API for 2D and 3D graphics programming. But while SGI focused more on professional graphics, many companies emerged in the mid 1990s that offered graphics hardware to consumers – 3DFX, Nvidia, ATI, and Matrox relied on PC games that started to use 3D graphics acceleration.

When the first "true" GPUs emerged in 1999 (handling the whole 3D pipeline in specialized graphics hardware), they were known to feature a fixed function pipeline. Fixed function meant that after the scene data was sent to the GPU, the GPU took over the whole computation, so it was not possible to modify the scene in the middle of the graphics pipeline. This was inflexible, as when the OpenGL and DirectX APIs gained new functionality, existing GPU were not able to take advantage of it. In 2001, Nvidia released the Geforce 3, which made some of the graphics pipeline stages programmable. Along with data, the programmer could send in a program (called a shader), that influenced the execution of the vertex or the pixel stage. The shader was written in an assembly-like shader language, but soon Nvidia and Microsoft developed Cg, C for Graphics, which simplified shader programming. Microsoft later developed HLSL (High Level Shader Language), which was basically Cg, but only for their proprietary DirectX 9 API.

2.2.2 GPGPU

The combination of a programmable pipeline, the parallel nature of GPUs, and the aforementioned performance growth that outpaced CPUs, GPUs became very interesting for non-graphic, compute applications [7]. But these compute applications on the GPU really started to take off when Nvidia introduced the Geforce 8 series in 2006. The Geforce 8 series moved from having different kinds of shaders (geometry, vertex, pixel) to employing a

single, more general kind of shaders, called unified shaders. Unified shaders are much more similar to a traditional CPU design, as they can also execute general purpose code, coining the term General Purpose GPU.

Still, a GPU core is usually much simpler than a CPU core, in the sense that it dedicates little resources to extracting ILP and rather focuses on ALU units. This does not provide high performance for serial code, but graphic workloads are generally easily parallelizable, so doubling the core count can efficiently provide an almost double speedup. Because the core is simpler, it also takes up less die area, making it possible to put more cores into the same sized chip. Having a high core count also provides an opportunity for many general algorithms that are easily parallelizable [20].

2.2.3 Modern GPUs

In this section we present the Nvidia Geforce GTX 980 [21] as an example of a modern high performance GPU. It was released in September 2014, but due to the lack of a newer high performance manufacturing process at TSMC (Taiwan Semiconductor Manufacturing Company), where Nvidia manufactures their chips, the architecture is still relevant as of this writing.

The GTX 980 is manufactured on TSMC's 28nm process using 5.2 billion transistors on a die $398mm^2$ in size and a TDP of 165W. It features 2048 CUDA cores. CUDA stands for Compute Unified Device Architecture, which has been the base of Nvidia GPUs since their first GPGPU, the Geforce 8800 GTX. The number of cores cannot be directly compared to other architectures, as what the core is capable of varies more significantly in GPUs than in CPUs – the 980 GTX actually features 28,9% less CUDA cores than its direct predecessor, the GTX 780 Ti, while being slightly faster. Since it's primarily a GPU, it also features 128 texture units and 64 Raster Output Units, both of which are not relevant for GPGPU. It also features it's own memory, 4 GB of GDDR5, with an effective frequency of 7 GHz, connected via a 256-bit bus.

The GTX 980 has a base clock frequency of 1126 MHz and can boost

Figure 2.2: Streaming Multiprocessor Maxwell (figure source: [21]). All of the cores are meant for processing data. The figure has been cropped slightly and modified to show only two out of four warps.

to 1216 MHz (8%), similarly to what was described for Skylake, although it works somewhat differently. It is highly unlikely that just a single CUDA core will be occupied – instead, most of the time most of the GPU is fired up, consuming power and generating heat, so the boost clocks cannot be as high. The GTX 980 targets a temperature of 80°C, but allows it to reach 91°C while boosting the clock. Sometimes the temperature may exceed even that, so there is a hard limit set to 95°C. During boost operation, the TDP limit increases from 165W to 206W.

Double floating point execution units were long not present on GPUs, while many lower performance GPUs still do not support it. The GTX 980 does support double precision floating point operations, but only at a rate of $\frac{1}{32}$ of single precision. Double precision is necessary in some scientific computations, so the low rate on GTX 980 slightly reduces its GPGPU appeal.

The GTX 980 is based on the Maxwell 2 architecture. The high core count is organized in a hierarchical way – the whole chip is split into four

Graphic Processing Clusters (GPCs), and each GPC contains four SMMs (Streaming Multiprocessor Maxwell). Each SMM contains four execution units ("warps") and within each warp there are 32 CUDA cores [10]. The warp is a SIMD unit (actually it's what Nvidia calls a "SIMT" unit, which is similar), executing the same instruction on all 32 cores at once This means that the cores are not completely independent, which can have some performance implications when writing general purpose code. Consider the following example:

```
1 int i = this_thread_id();
2 if(i % 2 == 0) {
3   execute0();
4 }
5 else {
6   execute1();
7 }
```

Here only half of the cores need to execute the code within the if-statement. But since the cores execute as a SIMD unit, we get a slowdown: first all cores execute `execute0()`, but half of the results are discarded, and then all cores execute `execute1()`, where the other half of the results are discarded.

Each SMM contains a 4 KB instruction cache and each warp contains an instruction buffer. A warp scheduler assigns workload to cores within a warp – at every instruction issue time, the scheduler issues one instruction. Additionally to 32 cores, a warp also contains 8 load/store units and 8 Special Function Units (trigonometry functions). It is also good to know that the concept of a warp was somewhat different in earlier Nvidia architectures – a warp was always a collection of 32 consecutive threads that execute in parallel like a SIMD unit, but unlike previous architectures, with Maxwell each warp has its own warp scheduler.

Each warp contains 16384 32-bit registers – a 64 KB register file. This is 512 registers per core, although all 16384 are available to the entire warp, with certain limitations [10].

The memory on a SMM consist of [22]:

- a read-only constant memory,
- 24 KB of a unified L1 data/texture cache,
- 96 KB of shared memory.

L1 cache is hidden from the programmer unless using it as a texture cache, while the constant and shared memory need to be explicitly addressed. The whole GPC also includes 2 MB of L2 cache.

2.2.4 Comparison of Skylake and Maxwell

Feature	Skylake CPU	Maxwell GPU
Cores	4 (8 threads)	2048 CUDA (16 SMMs)
ILP	14-stage pipeline, OoO, ...	In-order
Base clock	4 GHz	1.126 GHz
Boost clock	4.2 GHz	1.216 GHz
GP* registers	64-bit, 16 per core	32-bit, 16384 per warp
L1 cache	32 KB + 32 KB per core	24 KB per SMM
L2 cache	256 KB per core	2 MB per SMM
L3 cache	8 MB per chip	None
Other memory	Possible 64/128 MB eDRAM	Constant, 96 KB shared
Main memory	64-bit LDDR3/DDR4	4GB 256-bit GDDR5

Table 2.1: Comparison of Skylake and Maxwell architectures. GP stands for "General Purpose".

We can see from Table 2.1 that even though a GPU supports general purpose code, a CPU and a GPU serve quite different types of workloads. While a CPU is focused on single core performance and extracting ILP – high clockspeed, long pipeline, wide OoO window, branch prediction, lots of instruction and data caching — a GPU offers instead a great environment

for inherently thread-parallel code – high number of cores, wide memory interface, special constant and shared memory for sharing between threads.

2.3 Other processing units

We mostly take a look into heterogeneous computing using popular CPUs and GPUs, but there are some other designs that could potentially be part of a heterogeneous system.

2.3.1 Digital Signal Processors

Digital Signal Processors (DSPs) process a stream of data (a signal) [23]. They are designed to process large amounts of data as quickly as possible – data is often required within a prespecified timeframe, e.g. real-time video processing needs to provide 30 image frames per second (or similar). A general purpose CPU can usually perform the same tasks as a DSP, but DSPs, being designed specifically for processing data, can offer better performance and/or lower power consumption. As such, they are mostly preferred to CPUs in power constrained devices.

DSPs are optimized to execute multiplications, additions, and fused multiply-accumulate (FMA or MAC), which is required for calculating convolution, FIR filters, and Fast Fourier Transform (FFT), to name just a few examples. DSPs also often feature fixed point arithmetic units, which are less flexible, but more efficient that floating point units. The memory architecture of a DSP is optimized for streaming data, fetching multiple data and instructions at the same time – the instructions are then issued either in a superscalar fashion or as a VLIW (Very Large Instruction Word) and the data is processed in SIMD units. Separate program and data memories are typical for a DSP in order to increase bandwidth.

2.3.2 FPGA

A Field Programmable Gate Array is a circuit that is designed to be configured for a specific task after it has already been manufactured [24]. Unlike other processing elements presented here, an FPGA does not have an instruction set – instead, a programmer designs hardware blocks (using a Hardware Description Language, HDL) and sends them as configurations to the FPGA. An FPGA consist of an array of programmable logic blocks and a reconfigurable interconnect. By enabling and disabling specific logic blocks and interconnections, any combinatorial function can be configured, as long as there is a large enough number of basic logic blocks. The very regular structure of FPGAs means that they benefit greatly from newer manufacturing processes – FPGA designer Altera was the first to employ Intel's 14nm process [25], which was the most advanced commercial process at the time.

Eschewing instruction processing (fetch, decode, issue, ...) and designing hardware blocks to perform a specific function can lead to significantly higher performance and lower power versus a general purpose CPU, GPU, or DSP. This does come with a cost, though: writing HDL is more difficult than a regular programming language ("compilation" can take a few hours) and FPGA reconfiguration time limits the general-programmability aspect. FPGAs are more suited for prototyping hardware designs or for applications that are loaded once and ran many times.

FPGAs can be used in many different ways – one interesting result came from Microsoft [26], where their deployment of FPGAs in datacenters increased the thoroughput by 95% while only using 10% more power and at 30% higher costs. Intel also plans to integrate an FPGA on its Xeon line of chips [27].

2.3.3 Intel Xeon Phi

Intel developed the Xeon Phi as a GPGPU competitor – the original idea was to develop a discrete GPU using x86 cores and software rendering (x86 refers

to the backwards-compatible CISC programming model of Intel CPUs). The project, codenamed Larrabee, was canceled due to delays and underwhelming performance [28].

But Intel got rid of Larrabee's GPU-specific functions and introduced a PCIe connected accelerator, named the Xeon Phi [29]. It promised to deliver the same functionality as a GPGPU – high performance for thread-parallel applications – but with a much simpler programming model. We touch a bit on GPGPU programming in section 3.1, but here we can mention that it is quite different than traditional programming models. On the other hand, the x86 model has been known to programmers and compilers for over 30 years – apart from dealing with sending data between the CPU and the Xeon Phi and some other specifics, the programmer sees the Xeon Phi as if it were a normal Intel CPU, just with a large number of cores. Combined with the support for high double precision floating point performance, this lead to adoption in High Performance Computing applications (it's a key component of the No. 1 supercomputer in the world [30]).

Architecturally, one could say the Xeon Phi sits somewhere in between a regular CPU and a GPU. The current generation Xeon Phi, codenamed Knights Corner, has its core based on Intel's Pentium P54C core [31]. The basic P54C core is a dual-issue OoO design with a 5-stage pipeline [32], but the Xeon Phi core is heavily modified. Along with adding new instructions (including 64-bit ones), extending the pipeline, using the 22nm manufacturing process, and significantly increasing/adding caches, the Xeon Phi incorporates four-way SMT and includes a Vector Processing Unit (VPU). The VPU is a an enhanced SIMD unit that can process 16 single precision floating point operations at once, or 8 double precision ones. Xeon Phi includes 60 cores, which with SMT works out to 240 processing threads (at full utilization). So we can see that the performance of a single core sits between a GPU core and a regular Intel CPU core, while the total number of threads it can process in parallel is also somewhere in between.

Xeon Phi is supposed to feature a new version in 2016, codenamed Knights

Landing [33], which will move from enhanced Intel Pentium to enhanced Intel Silvermont cores. The core count will increase only to 72, so most of the claimed 3x performance improvement will come from the improved OoO cores. To help alleviate bandwidth concerns, the chip will also feature 16GB of on-package memory.

2.4 The interconnect

A significant problem in heterogeneous computing is connecting the various computing elements together. A standard model is to assume a GPU or other accelerator communicating with the CPU over a PCI Express (PCIe) bus. However, even PCIe 3.0 can only carry 1 GB/s per lane [34] – high end GPUs are connected by 16 lanes, providing almost 16 GB/s of bandwidth. In some contexts that may sound a lot, but let's consider the bandwidth difference between CPUs and GPUs with the help of the Skylake–Maxwell comparison table in section 2.2.4.

A typical 64-bit DDR3 1600 MHz memory bus provides 12.8 GB/s of bandwidth: $\frac{64b}{8} * 1.6GHz = 12.8GB/s$ – this is actually less than what 16 lanes of PCIe provide, although this is just one channel as opposed to the usual two, while PCIe also exhibits higher latency. On the other hand, typical 256-bit GDDR5 memory running at a 7 GHz effective rate (shipped on the GPU board) provides 224 GB/s of bandwidth – this is required because of the high number of processing elements on a GPU, which can process large amounts of data in parallel. We see that PCIe bandwidth is much lower than what GPUs are able to use, therefore any data copying between a CPU and GPU needs to be limited.

The limited bandwidth can be observed in some Systems-On-Chip (SoCs), where the integrated GPU shares the same global memory as the CPU, which is typically of the DDR3 variety – a dual channel configuration presents 25.6 GB/s for both the CPU and GPU to share. The reason CPU-centric DDR3 is used instead of the GPU-centric GDDR5 is that GDDR5 has much higher

access latencies – for a GPU this is usually no problem, because it needs to read large amount of data at a time, but a CPU is much more sensitive to memory latency.

Let's consider an example. Intel HD 4400 graphics employs 20 Execution Units (EUs), while Intel HD 5000 graphics employs 40 EUs – these integrated GPUs are built using the same architecture with more or less the same frequencies, the only significant difference is the doubling of EUs [35]. Doubling the number of computing units on a GPU usually results in almost double the performance, but Intel HD 5000 is only up to 15% faster than HD 4400 and in some cases it's not faster at all. On the other hand, comparing Intel HD 4600 and Intel Iris Pro 5200 (variants of the HD 4400 and HD 5000, respectively, used in higher power desktop as opposed to mobile chips) reveals that Iris Pro 5200 performance is 50% higher or more – the reason why it isn't even higher is that Intel Core i7-4770K, which houses the HD 4600, is allowed more thermal headroom [36].

What makes the Iris Pro 5200 special is the use of an additional layer in the memory hierarchy – 128MB of external DRAM (eDRAM), which can be viewed as some sort of L4 cache (it doesn't act quite like an L4 cache, as it can be bypassed, but that detail isn't very important in this discussion). This means that the Iris Pro 5200 has normal access to 25.6 GB/s of DDR3 bandwidth and additional 50 GB/s when the data can be cached.

Newer memory standards are being developed in order to raise the available bandwidth. DDR4 promises clockspeeds up to 3.2 GHz in four channel configurations [37], which provides 102.4 GB/s of bandwidth on a 64-bit interface. For GPUs, AMD was the first to employ High Bandwidth Memory (HBM) on its Radeon Fury X GPU [38], which uses a much wider 1024-bit interface at 1 GHz, which, when arranged in four stacks, gives 512 GB/s of bandwidth. Power and/or cost currently prevent wide adoption of these faster solutions, although that will definitely change with newer manufacturing processes.

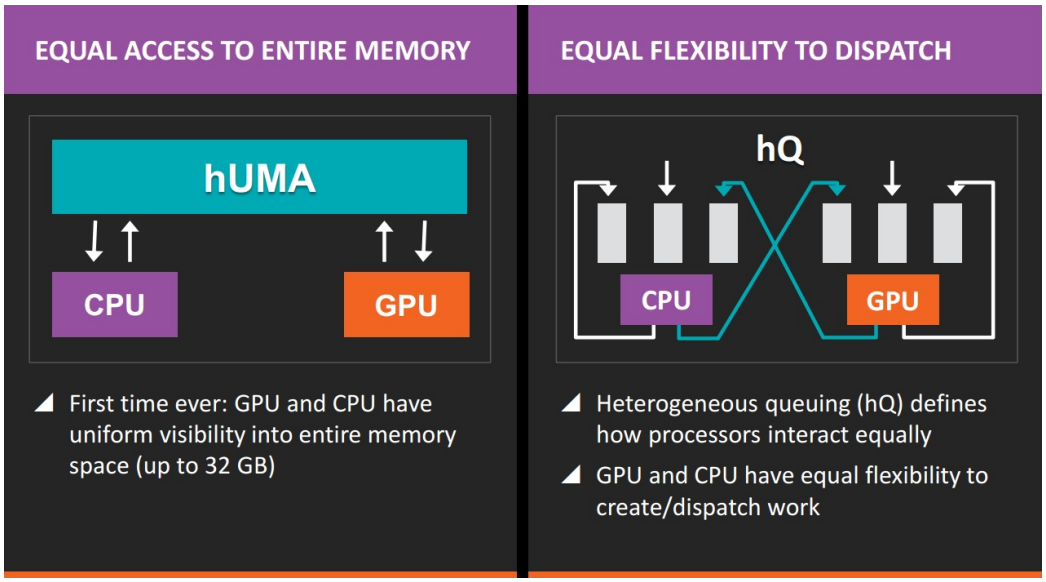

Figure 2.3: Two main features of HSA (figure source: [39]). The figure has been cropped.

2.5 Heterogeneous System Architecture

Continuing the discussion on memory bandwidth, different computing units in a heterogeneous system employ either each their own memory, tailored for their needs, or employ a shared memory, which is usually a disadvantage to at least one of the different units.

But sharing memory provides an optimization opportunity – in some cases, the memory does not need to be copied at all, when just passing a memory address (a pointer) would suffice. In order for that to work, the different computing units need to have a unified memory space, but traditionally the memory spaces were separate. AMD was among the first to embrace this unified memory idea [39], which they call Heterogeneous Unified Memory Architecture (hUMA) – their first product to implement hUMA was the Kaveri architecture. There are many benefits of hUMA:

1. Eliminating CPU-GPU copies.

2. Access to the entire address space. The GPU is no longer limited to its own onboard memory and the memory can be upgraded just like regular main memory.

3. Unified addressing in hardware. Without this, the application had to ask the GPU driver to allocate a GPU page table for a given range of CPU virtual addresses, because the GPU had a separate virtual address space. This only worked for simple data structures (arrays) and the page table initialization introduced some performance overhead. But in hUMA pointers can be freely exchanged between CPU and GPU, with no driver overhead.

4. Demand-driven paging. CPU virtual memory can point to other addresses than those of physical memory, e.g. to the hard drive – this is called demand-driven paging. But GPUs traditionally did not implement this – the application had to know the range of addresses it needed and map them to a GPU buffer object. Having fixed memory is problematic with dynamic data structures (e.g. linked lists), where pointers could point to anywhere in memory.

5. CPU-GPU coherence. In addition to being able to see the same data, the CPU and the GPU should also be able to see write operations to this data in order to ensure data consistency, which is complicated due to the cache hierarchy. This is an optional feature for the programmer to use, because it incurs some overhead. Atomic operations are also provided.

But hUMA is just one part of AMD's Heterogeneous System Architecture (HSA). Another important part is HSAIL (HSA Intermediate Language), which is a portable pseudo-ISA for heterogeneous compute (ISA – Instruction Set Architecture). HSA was first implemented by AMD, but is actually a specification, developed by the HSA foundation, where AMD plays an important role. Because different manufacturers may implement HSA, the HSA foundation wants the same applications be able to run on different

hardware, but this requires a standard software interface. However, HSA targets a wide range of different computing units (CPU, GPU, DSP, ...) so a unified ISA would not be feasible. Instead, the compiler generates HSA Intermediate Language, while the actual binary is produced Just-In-Time by the HSA driver.

Another important feature of HSA is Heterogeneous Queuing (hQ), which is about optimizing task queuing. There are three important improvements on this front:

1. User-mode queuing. Queuing tasks does not need to invoke the GPU driver and system calls anymore. This reduces overhead and makes even small tasks feasible to queue to the GPU.

2. Dynamic Parallelism. Normally it's the CPU that queues work for the GPU, but now it is also possible for the GPU to queue tasks for itself.

3. CPU callbacks. In addition to queuing work for itself, the GPU can also invoke CPU functions – this especially benefits legacy CPU code that is not GPU-aware.

Some of the described HSA functionality had already been available before – e.g. OpenCL SPIR (Standard Portable Intermediate Representation) is similar to HSAIL, Nvidia had already supported dynamic parallelism and unified memory addressing in software (but not in hardware). But AMD was the first to implement the full HSA version 1.0.

Chapter 3

Programming framework

3.1 OpenCL

3.1.1 Overview

OpenCL is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous systems [8]. It is maintained by the Khronos Group, which consists of many hardware and software companies. It is designed to be efficient, to map to the underlying hardware as close as possible, while still providing a powerful programming toolchain. OpenCL is most commonly used with CPU and GPU devices, but there are also DSPs available that support OpenCL [40], and even FPGAs from Altera [41] and Xilinx [42] have an OpenCL SDK.

OpenCL consists of the OpenCL framework and a specially designed language for programming devices, called OpenCL C, which is a subset of ISO C99, but with extensions for parallelism. It also offers interoperability with OpenGL and similar graphics APIs.

The OpenCL framework consists of the following components:

1. Platform layer. Allows the host to discover devices and their capabilities and to create contexts.

2. Runtime. Allows the host to manipulate contexts after creation.

3. Compiler. Creates program executables that contain OpenCL kernels.

OpenCL was initially developed by Apple [43], which holds trademark rights and who submitted the initial proposal to Khronos. The Compute Working Group was formed within Khronos, which released the OpenCL 1.0 specification in November 2008. The first actual implementation was provided by Apple in their Mac OS X Snow Leopard Operating System in August 2009 [44]. AMD opted for OpenCL instead of its own Close to Metal framework [45] and Nvidia decided to support OpenCL alongside its own CUDA.

OpenCL has gone through multiple revisions [8]. Version 1.1 was released in 2010, and Version 1.2 in 2011 – the latter sees very widespread support today in 2016 and is also the focus in this thesis. Version 2.0 brought a lot new features in 2013, e.g. shared virtual memory, nested parallelism, a generic address space etc. The newest specification is 2.1, which was released in 2015 and replaces the OpenCL C language with OpenCL C++.

3.1.2 Architecture

Platform model

An OpenCL platform consists of a host connected to one or more devices. A device is divided into one or more compute units, which are further divided into processing elements (PEs), which perform the actual computation. The PEs execute a single stream of instructions either as SIMD units (Single Instruction Multiple Data) or as SPMD units (Single Program Multiple Data, each PE maintains its own program counter). To support devices with varying capabilities, OpenCL considers multiple version identifiers: the platform, the device, and the OpenCL C language versions.

Execution model

An OpenCL program executes in two parts: a kernel executes on a device and a host program executes on the host. When the host submits a kernel for execution, an index space is defined, and the kernel executes for each index. An instance of kernel execution is called a work-item, and each work-item executes the same code, but it can perform different computation based on its index (global ID). Work-items are organized into work-groups, which also receive their own index (a local ID), and are meant to offer a more coarse grained view into the execution. The OpenCL index space is an NDRange, which is an N-dimensional index space, where N can be 1, 2, or 3.

The host defines a context for kernel execution, which includes devices, kernels, program objects, and memory objects. The execution is controlled by a command queue, which includes commands for kernel execution, memory access, and synchronization. The command queue can be either in-order or out-of-order (OoO), although most OpenCL implementations don't support OoO. Kernel execution and memory commands generate event objects, which are used to control execution between commands and to aid the communication between host and devices. A single context can be associated with multiple queues, which run concurrently and independently – synchronization between them needs to be managed by the programmer with the help of `event` objects.

Memory model

A work-item can access four distinct memory regions;

1. Global memory.
2. Constant memory. Remains constant during execution of a kernel.
3. Local memory. Shared by all work-items within a work-group.
4. Private memory. Only visible to the work-item.

The host has no access to local and private memories, while a device can access any of them. The host can, however, dynamically allocate all except private memory, while a device can allocate all except global memory, but it needs to do it statically. The host and device memory models are generally independent, but there is some necessary interaction, which is managed by either copying data or mapping memory regions. OpenCL uses a relaxed consistency model, which means that the state of work-item memory is not guaranteed to be consistent across all work-items at all times. Consistency needs to be enforced through synchronization points.

In most cases, global memory is the main system memory and private memory is represented by registers on the device. Constant and local memory are usually emulated in main memory on a CPU device, but with GPUs this is often an architectural feature (see section 2.2.3).

Programming model

As explained, in OpenCL code is split into host code and device code. The host code is the main program (usually runs on a CPU), which also prepares the device and sends it commands and data. The host can recognize an OpenCL platform – distinguished by the OpenCL platform version – and each platform contains one or more devices. OpenCL 1.2 recognizes multiple types of devices: CPU, GPU, accelerator, etc. A device reveals its properties to the host, e.g. the number of its compute units (cores).

At the very least, the host needs to initialize the desired device, prepare the execution context and command queue, send data, instruct the device to execute the required code, and retrieve the new data. Code that executes on the device is called a kernel and is written in a specialized version of C, called OpenCL C. OpenCL C code needs to be either in a separate file or passed as a string to the OpenCL compiler. The OpenCL environment and compiler are vendor specific and usually ship as part of an OS driver. The kernel is sent to the device and executed by work items. The number of work items is specified by the host and work items get mapped to the available compute

units by the OpenCL environment and by the hardware itself. Each work item executes the same kernel, but is also assigned a unique ID, which can come useful in the kernel.

A classic example is vector addition. Suppose we have three arrays – A, B, and C – each one of length n. On the host, we would write:

```
1 int A[n], B[n]; // Filled somewhere ...
2 for(int i = 0; i < n; ++i)
3   C[i] = A[i] + B[i]
```

But in OpenCL C, we write:

```
1 __kernel void add(__global int* A, __global int* B, __global int* C) {
2   int i = get_global_id(0);
3   C[i] = A[i] + B[i];
4 }
```

The host needs to copy A and B to the device, invoke the kernel with n work items, and copy C back from device to host. Each work item reads only one element from A, one from B, adds the values, and stores the sum to C. If n is lower than the number of compute units, this is done in one step, in parallel, because it has no interdependencies. Even if n is larger than the number of compute units, a significant speedup can still be achieved – the number of compute units on a GPU is usually significantly higher than on a CPU.

In the example, the kernel accessed global memory. This can have different meanings: on a CPU, global memory is main memory (e.g. DDR4), while for a discrete GPU it's its own memory (typically GDDR5). As discussed in the memory hierarchy section, this memory can be a bottleneck for computation.

3.1.3 Comparison to CUDA

Along with their first GPGPU, Nvidia also released CUDA in 2006 [46], which is very similar to OpenCL. Note that CUDA is both the name of a core on an Nvidia GPU as well as the programming model, which we discuss in this section.

CUDA is meant only for Nvidia GPUs, but OpenCL is designed to be much more general, covering CPUs, GPUs, DSPs, FPGAs, or anything else, from any manufacturer, as long as the device designers opt to include compatibility with the OpenCL standard – and OpenCL is supported by a large number of hardware and software companies.

AMD provides a guide on porting CUDA to OpenCL [47], which highlights the similarities. Instead of work-items and work-groups, CUDA uses threads and thread blocks. Local memory in OpenCL is called shared in CUDA, while private memory in OpenCL is local in CUDA. OpenCL provides global indexes within a kernel as opposed to CUDA (that doesn't have global indexes), and uses functions instead of predefined variables for indexing.

Both provide synchronization of work-items within a work-group and between all work-items, but OpenCL provides more options with regard to read/write synchronization. CUDA does not have a command queue that would provide task parallelism. A big difference is that in addition to offline compilation, OpenCL also supports runtime compilation.

OpenCL requires kernel arguments to be annotated with their memory space, which is not required in CUDA. Additionally, while CUDA encourages scalar code and OpenCL supports it, it is usually more efficient to use vector types.

A study found that CUDA provides up to 30% better performance than OpenCL for Nvidia GPUs, but the performance difference could be almost entirely reduced by manually optimizing the OpenCL code [48]. The difference could also be attributed to the fact that Nvidia provides both OpenCL and CUDA for their GPUs, but prefers optimizing for their own platform.

3.1.4 Simple OpenCL code example

Let us consider a simple example. Suppose we have arrays of N integers `A` and `B` and perform the following operation for each value within the array `A` (index `i`): if value `A[i]` is odd, store to `C[i]` the sum of `A[i]` and `B[i]`, else compute `B[i]` to the power of 5 and store the result to `C[i]`. Standard C++ code would be:

```
1  int A[N], B[N], C[N]; // N known from before, actual values elsewhere
2  for(int i = 0; i < N; ++i) {
3    if(A[i] % 2) {
4      C[i] = A[i] + B[i];
5    }
6    else {
7      int Ci = B[0];
8      for(int j = 1; j < 5; ++j) {
9        Ci *= B[j];
10     }
11     C[i] = Ci;
12 }}
```

Of course, this example is completely artificial and even computing the power operation is not optimal. But it will serve our purpose of demonstrating various approaches to computing it.

This computation can be easily parallelized, because there are no interdependencies. However, writing OpenCL code for even this simple example requires quite a lot of code. The OpenCL kernel is pretty straightforward (stored in file `example.cl`):

```
1  __kernel void example(
2    const __global int* A, const __global int* B, __global int* C
3  ) {
4    int i = get_global_id(0);
5    if(A[i] % 2) {
6      C[i] = A[i] + B[i];
```

```
  }
  else {
    int Ci = B[0];
    for(int j = 1; j < 5; ++j) {
      Ci *= B[j];
    }
    C[i] = Ci;
}}
```

Apart from the call `get_global_id` to obtain the index, the kernel body is the same as the body of the main `for` loop above. However, before we can use the kernel, we need to get the platform, device, and context, initialize the buffers, the program, the kernel, and the queue and at the end manually free all created objects. Here is the host code:

```
int A[N], B[N], C[N]; // N known from before, actual values elsewhere
cl_int error;

// Get platform
cl_uint numPlatforms;
clGetPlatformIDs(0, nullptr, &numPlatforms);
std::vector<cl_platform_id> platforms(numPlatforms);
clGetPlatformIDs(numPlatforms, platforms.data(), nullptr);
auto platform = platforms[0];

// Get device
cl_uint numDevices;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 0, nullptr, &numDevices);
std::vector<cl_device_id> devices(numDevices);
clGetDeviceIDs(
  cpPlatform, CL_DEVICE_TYPE_GPU, numDevices, devices.data(), nullptr
);
auto device = devices[0];

auto context = clCreateContext(
```

```
21   nullptr, 1, &device, nullptr, nullptr, &error);
22
23 // read text file containing the kernel
24 std::string kernelCode = read("example.cl");
25 auto codePtr = kernelCode.c_str();
26 auto codeLength = kernelCode.length();
27
28 // Build kernel program
29 auto program = clCreateProgramWithSource(
30   context, 1, &codePtr, &codeLength, &error);
31 clBuildProgram(program, 1, &device, "", nullptr, nullptr);
32
33 // Initialize buffers on device and copy input data
34 auto bufA = clCreateBuffer(
35   context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
36   sizeof(int) * N, (void*)A, &error
37 );
38 auto bufB = clCreateBuffer(
39   context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
40   sizeof(int) * N, (void*)B, &error
41 );
42 auto bufC = clCreateBuffer(
43   context, CL_MEM_WRITE_ONLY,
44   sizeof(int) * N, nullptr, &error
45 );
46
47 // Prepare queue and kernel arguments
48 auto queue = clCreateCommandQueue(context, device, 0, &error);
49 auto kernel = clCreateKernel(program, "example", &error);
50 clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&bufA);
51 clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&bufB);
52 clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&bufC);
53 size_t globalWorkSize[] = { N };
```

```
54
55 // The actual kernel call
56 clEnqueueNDRangeKernel(
57   queue, kernel, 1, nullptr, globalWorkSize, nullptr,
58   0, nullptr, nullptr
59 );
60
61 // Wait for queue and read data back from device
62 clFinish(queue);
63 clEnqueueReadBuffer(
64   queue, bufC, CL_TRUE, 0, sizeof(int) * N,
65   C, 0, nullptr, nullptr
66 );
67
68 // Release all created objects
69 clReleaseCommandQueue(queue);
70 clReleaseMemObject(bufC);
71 clReleaseMemObject(bufB);
72 clReleaseMemObject(bufA);
73 clReleaseProgram(program);
74 clReleaseContext(context);
```

We have a lot of initialization code that significantly exceeds the line count of the kernel that performs the actual computation. Note that we skipped error handling to make the code more readable. The code is very long, cumbersome, and error prone. We suppose long time OpenCL experts may know all of these function calls and what they do by heart, but we constantly needed to consort the OpenCL specification about the function signatures and the call order, even though we've already written quite a few kernels. A common theme is an abundance of `0` and `nullptr` values passed to the functions – this is partly because the functions are very flexible and offer a lot of calling options, but also because OpenCL was designed for the C language, which doesn't support function overloading.

We look again at this particular example – exploring alternatives to parallelization – in sections 3.2.4 and 4.5.

3.2 SYCL

3.2.1 Specification

SYCL is a C++ programming model for OpenCL [11]. It aims for single source compilation of host and device code using standard C++11. In its version 1.2 (first released version) it targets OpenCL 1.2 compatibility, although OpenCL isn't necessary – OpenCL interoperability is specified in the SYCL API, but the underlying system could be something else. For example, the open source triSYCL [13] implementation is based on OpenMP and suggests the base could be swapped for CUDA.

SYCL has three main goals:

1. Simplicity. With OpenCL, programmers need to learn to write separate device and host code, in two separate languages, and how to connect the two together in one system. SYCL allows for single source compilation, reusing the language experience and the compiler. Additionally, SYCL simplifies programming flow by relying on higher-level C++ paradigms as opposed to OpenCL C.

2. Reuse. The C++ type system allows for complex interactions between different code units, abstract interface design, and reuse of library code.

3. Efficiency. Tight integration with the type system and library code reuse enables the compiler to perform inlining and other optimizations.

SYCL is designed to allow a compilation flow with multiple compilers, seamlessly integrated to provide the final program. By allowing the compilation to be split across multiple compilers, it offers the advantage of allowing integration with existing toolchains and choosing the optimal compiler for the target device. SYCL recognizes at least two compilers: the host compiler

(used for writing the application code) and the device compiler. The device compiler compiles the code that should execute on the target device. The host and device compiler could be just one compiler or separate compilers.

SYCL is as close to standard C++ as possible, although there are a few limitations due to the fact that it is supposed to support many different target devices, which are usually not tightly coupled with the host, and that the underlying OpenCL standard is not as flexible. The restrictions include function pointers and virtual functions, exceptions, Runtime Type Information (RTTI), or any libraries that rely on these features. At the same time, the remaining C++ features (templates, inheritance, ...) allow for new kinds of heterogeneous computing libraries, which can be both simple and efficient. SYCL extends OpenCL in two important ways:

1. Hierachical Parallelism. This offers a simple syntax for expressing the data-parallel OpenCL execution model, with code layers serving to avoid fragmentation of code and to more efficiently map to CPU-style architectures.

2. Data access is separated from data storage. By heavily relying on the C++ idiom Resource Acquisition Is Initialization (RAII), where data is acquired in an object constructor and automatically released in the destructor when that object goes out of scope, SYCL removes a lot of dependencies that usually complicate parallel programming.

3.2.2 Architecture

SYCL builds upon OpenCL, so most of the terminology and structure is reused. The basics are still the same: the host prepares code (a kernel) that can be executed on the devices. But SYCL provides an important addition: host fallback (the SYCL host device). This means that if no OpenCL device is available, code is executed on the host. Note that the host CPU can also act as an OpenCL CPU device, but host fallback is a separate target, guaranteeing that code is executed even if OpenCL fails.

The target users of SYCL are C++ programmers who want the performance and portability of OpenCL with the higher-level language flexibility of C++ across the host/device code boundary. That's why SYCL provides fully compatible interoperation with OpenCL. We already mentioned some limitations of C++ code inside kernels, but outside kernels host code supports anything the compiler of choice is able to provide.

In this section we go quickly over the architecture of SYCL, since it is similar to that of OpenCL, and will mostly just list extensions over OpenCL. All SYCL classes are part of the `cl::sycl` namespace.

Platform model

A SYCL application runs on a host according to the standard C++ CPU execution model. The SYCL application submits command group functors to queues, which execute either on an OpenCL device or on the SYCL host device. SYCL executes kernels on a device by enqueuing OpenCL commands. SYCL can use any parallel execution facility available to execute the kernels as long as it executes within the semantics of the OpenCL kernel execution model.

SYCL presents the user with a set of devices grouped into platforms. The device version indicates the device's capabilities and corresponds to the highest version of the OpenCL specification for which the device is conformant, but it is limited by the platform version.

Execution model

SYCL executes kernels either on the SYCL host device or on an OpenCL device from a host CPU program, which defines the context and manages execution. OpenCL commands (data transfer, synchronization, kernels, ...) are grouped in SYCL into a functor called command group. Each command group has a handler which associates sets of data movement operations and enqueued kernels on the underlying OpenCL queue with the command group.

At kernel submission an index space is created, just as in OpenCL,

only that in SYCL the NDRange index space is accessed through templated classes, e.g. `nd_range<N>`, where `N` is the dimensionality of the index space (1, 2, or 3). An `nd_range<N>` consists of a global and a local range, each represented by an object of type `range<N>` and an offset of type `id<N>`. Each work-item is identified by an `nd_item<N>` object, which encapsulates a global, a local, and a work-group ID, all of type `id<N>`. SYCL allows the work-group size to be undefined, which hands the decision over to the SYCL framework.

SYCL manages the following resources (only SYCL-specific changes noted here):

1. Platforms.

2. Contexts. All OpenCL resources are attached to a context and a context can only wrap devices owned by a single platform. Data movement between devices within a context may be efficient and hidden by the runtime, but data movement between contexts involves the host.

3. Devices. SYCL additionally provides a `device_selector` class, which is used to determine device selection – SYCL provides a few selectors, but the programmer can supply their own. As mentioned, SYCL also provides a host device (host fallback).

4. Command groups. Submitted to a SYCL queue.

5. Kernels. Defined as C++ functors or lambda functions. All kernels must have a name – it is either taken from the name of the functor or needs to be supplied in case of a lambda. Names are necessary to enable linking with different compilers.

6. Program objects. These are OpenCL objects that store implementaion data for the SYCL kernels. Required for advanced use.

7. Command queues. A command queue is associated with a context, a platform, and a device, which can be either automatically chosen by SYCL or specified by the programmer.

The command queue schedules commands for execution. The commands are executed asynchronously with respect to the host thread and can be scheduled in any order the SYCL framework sees fit as long as the order preserves the semantics. This means that SYCL must provide proper data movement, kernel execution, and synchronization between different queues, devices, and the host. The underlying OpenCL queues may operate in-order or OoO, with SYCL providing automatic synchronization commands.

Memory model

There are four distinct memory regions just like in OpenCL. SYCL uses templated `buffer` and `image` classes for exchanging data between host and device. An important difference to OpenCL is that while in OpenCL a memory object is attached to a specific context, in SYCL a `buffer` or an `image` object can encapsulate multiple underlying OpenCL memory objects and host memory allocations to enable the same `buffer` or `image` to be shared between different devices, contexts, and platforms. SYCL then provides the necessary synchronization and data movement to maintain semantic integrity.

`buffer` and `image` data is accessed using `accessor` objects. An `accessor` is specified with the `target` attribute, which defined how the data is accessed (global memory, constant memory, image samplers), and with the `mode` attribute, which specifies read or write access (or both). The `mode` also specifies whether previous data should be discarded, or the programmer may even request atomic access.

It is not possible to directly pass a pointer to host memory as a kernel parameter because the device does not necessarily support the host's address space. But `buffer` and `image` objects can be constructed using host pointers, using explicit pointer classes depending on the accessed memory region.

Just as OpenCL, SYCL uses a relaxed memory consistency model. Consistency can be enforced through synchronization using barriers. But SYCL enforces consistency for `buffer` and `image` objects at certain synchronization points, derived from completion of enqueued commands. As mentioned,

SYCL also provides atomic operations, using the `atomic` class, but the extent is limited by device capabilities.

Programming model

A SYCL program is written in standard C++ and it allows the host and device code to be written in the same C++ source file – in OpenCL, either a separate file is required for OpenCL C code or the code is submitted as a string. SYCL relies on C++03 (apart from RTTI), the `function`, `string`, and `vector` classes from the standard library, and some of the modern C++11 features (like lambdas and rvalue references). SYCL programs are explicitly parallel and expose the full capabilities of the underlying OpenCL model, but SYCL additionally provides an abstraction layer to hide the complexity of the OpenCL model.

SYCL provides multiple ways to launch (invoke) kernels:

1. Single task. Only a single work-item executes the kernel. Enqueuing a single task on multiple queues supports task-parallelism.

2. Basic data parallel kernel. Multiple work-items, each one executes its own instance of the kernel. The local work-group size is chosen by the SYCL runtime.

3. Work-group data parallel kernel. As above, but the local work-group size needs to be specified. Along with global memory, work-items can also access local memory – all work-items within a work-group share the same local memory. Synchronization between work-groups is achieved using local barriers.

4. Hierarchical data parallel kernels. The programmer can use special syntax, provided by SYCL, to highlight the hierarchical nature of the parallelism. This is purely a compiler feature and does not change the execution model of the kernel.

Synchronization is enforced by SYCL at the following points: buffer, queue, or context destruction, accessor construction, and command group enqueue. The programmer can also enforce synchronization using the OpenCL event system or SYCL `event` objects. SYCL also supports synchronous and asynchronous error handling.

SYCL recognizes three different kinds of scopes: application, command group, and kernel scope. Kernel scope is the code sent to kernel invocation, a single kernel function, represented by a functor or a lambda. The kernel function is compiled by the device compiler and executed on the device. The command group scope specifies a unit of work submitted to the queue for execution, and it consists of accessors and a single kernel function. All other code belongs to the application scope.

SYCL automatically manages the lifetime of most internal OpenCL objects, except when the programmer requests access to those internal objects. Internal OpenCL objects are reference counted.

More about the programming model can be observed in chapter 4.

3.2.3 ComputeCpp suite

Khronos provided only the open SYCL specification [49], while an actual implementation was developed by Codeplay Software. Codeplay are part of the SYCL Working Group in Khronos and one of the main contributors to the standard. They provide SYCL as part of their ComputeCpp suite [12], which is, as opposed to the SYCL specification, proprietary. Codeplay does, however, provide evaluation licenses for developers to get acquainted with SYCL – we contacted them and successfully obtained an evaluation license for the purposes of this thesis.

ComputeCpp is currently available for Ubuntu 14.04 and Windows 7. This may have been the reason for some of our problems, because our own platform was Microsoft Visual Studio 2013 on Windows 10, which wasn't officially supported – even though we had the latest Intel drivers installed, some of the more complex code samples would not run. So because Windows

10 wasn't officially supported, we downloaded the Windows 7 version and installed it to the Program Files folder.

The installation includes SYCL include files, the device compiler, and library files (`.lib` and `.dll`), along with some documentation, tools, and sample code. Because the downloaded release was targeted at Visual Studio, the tools included Visual Studio build customizations and a project template to simplify ComputeCpp integration. Another included tool is the SPIR verifier – SPIR is an intermediate representation of OpenCL code and ComputeCpp lists SPIR support as a requirement. Code examples are provided in the directory `sample_code`, and a CMake build system is used to set up the Visual Studio solution. We managed to compile and run some of the examples, but not all of them. An additional issue was that while the original code would work, changing it and recompiling very often failed.

The way ComputeCpp works is that before the host compiler (Visual Studio) is invoked, the device compiler (provided by Codeplay) first goes through the code and compiles kernels into a `.sycl` file using the SPIR format. Then the host compiler compiles the code as usual and the `.sycl` file is linked.

Supporting only certain host compilers definitely hurts SYCL portability – our implementation (called sycl-gtx, chapter 4), by contrast, can be used anywhere OpenCL 1.2 and C++11 can be compiled. Of course, sycl-gtx does not properly implement the specification (see section 4.3), but it tries to get as close as possible. We did, in fact, find this aspect of ComputeCpp to be a huge help when developing sycl-gtx: ComputeCpp's SYCL conformance was much better than that of sycl-gtx. We often tried writing new tests and implementing the required functionality, only to discover that our interpretation of the specification lacked something, because it wouldn't even compile in ComputeCpp. After adjusting the test so that it compiled and ran in ComputeCpp we were also able to fix sycl-gtx.

We assumed the ComputeCpp implementation properly implements the specification and did not check for any potential errors. Since Codeplay were one of the main proposers of the SYCL standard, we believe it's reasonable

to assume their implementation is the most complete.

3.2.4 Porting the OpenCL example to SYCL

In section 3.1.4 we presented an example of a simple C++ loop without interdependencies and how to parallelize it using OpenCL. We show here the same parallelization effort, but using SYCL, which allows the kernel to be part of the normal application code:

```
1  int A[N], B[N], C[N]; // N known from before, actual values elsewhere
2  using namespace cl::sycl;
3
4  {
5    // Create a queue for a GPU device
6    // Automatically selects platform, device, and context
7    gpu_selector gpu;
8    queue q(gpu);
9
10   // Create buffers on device
11   auto rN = range<1>(N);
12   auto bufA = buffer<int>(A, rN);
13   auto bufB = buffer<int>(B, rN);
14   auto bufC = buffer<int>(C, rN);
15
16   q.submit([&](handler& cgh) {
17     // Get access to buffers
18     auto a = bufA.get_access<
19       access::mode::read, access::target::global_buffer>(cgh);
20     auto b = bufB.get_access<
21       access::mode::read, access::target::global_buffer>(cgh);
22     auto c = bufC.get_access<
23       access::mode::write, access::target::global_buffer>(cgh);
24
25     // The actual kernel
```

```
26    cgh.parallel_for<class example>(rN, [=](id<1> i) {
27      // Note the lowercase letters
28      // - we access the buffer accessors
29      // not the arrays or buffers
30      if(a[i] % 2) {
31        c[i] = a[i] + b[i];
32      }
33      else {
34        int Ci = b[0];
35        for(int j = 1; j < 5; ++j) {
36          Ci *= b[j];
37        }
38        c[i] = Ci;
39  } }); });
40 } // All data is automatically synchronized
41   // when queue goes out of scope
```

Compared to the OpenCL version, this is a considerably smaller amount of code, with the bonus that both the kernel and the host code are in a single source file and the kernel is written in C++ (although this particular kernel doesn't use any C++ specific features). The code is also arguably much easier to understand.

The kernel body is more or less the same as it was in the sequential C++ version or the OpenCL version. The most important exception is that we need to use accessor objects, retrieved from the buffers, instead of arrays.

All data movement is completely automatic. There are a few synchronization points to retain data integrity, e.g. in buffer and queue destructors and in accessor constructors. The SYCL runtime tries to minimize and optimize data movement by trying to smartly schedule buffer copying.

SYCL is explained more in-depth in chapter 4, particularly this example may be better understood by referring to sections 4.1 and 4.5.

Chapter 4

Implementation

The Khronos Group prepared only a specification, but no implementation of SYCL. The already mentioned triSYCL is hosted on GitHub [13] under the University of Illinois/NCSA Open Source License, but it doesn't seem to have much functionality implemented. On the other hand, SYCL is offered as part of the commercial ComputeCpp suite [12], developed by Codeplay Software, who are also among the main contributors to the SYCL specification. While ComputeCpp seems to be mostly implemented, it is offered under a commercial license, which does not suit the needs of many interested developers (we managed to obtain an evaluation license for inspecting ComputeCpp for the purposes of this thesis).

We decided to base our implementation on OpenCL 1.2 and C++11, without the need for any special compilers – this way, the transition from OpenCL to SYCL would be greatly simplified. The SYCL specification calls for a special compiler to enable same source compilation of host and device code, but we decided against it – while something like LLVM would have eased the development efforts, SYCL itself is already an extensive specifications, in our opinion too large for one person to develop fully in a year. So the real goal of this thesis was to develop as much of the SYCL specification as possible within a year and without using a special compiler, but rather as just a C++11 library that calls a self-developed JIT compiler at runtime.

The library approach has some drawbacks, though, which are discussed in a later section. Our implementation is called sycl-gtx and the whole project is open sourced under the MIT license, available on GitHub [14].

4.1 Anatomy of a sycl-gtx application

The first question when implementing a large specification is where to begin. The specification provides class interfaces for all of the publicly accessible classes, so that was a good starting point, to just copy the classes verbatim into their own header files, where class methods could be implemented as needed.

We then set ourselves a milestone: try to get the first SYCL code example from the specification to work. We are referring to the example from the section "Anatomy of a SYCL application", which we post here in full in order to study it a bit:

```
1  #include <CL/sycl.hpp>
2  int main() {
3    using namespace cl::sycl;
4
5    int data[1024]; // initialize data to be worked on
6
7    // by sticking all the SYCL work in a {} block, we ensure
8    // all SYCL tasks must complete before exiting the block
9    {
10     // create a queue to enqueue work to
11     queue myQueue;
12
13     // wrap our result variable in a buffer
14     buffer<int> resultBuf(&result, range<1>(1024));
15
16     // create some commands for our queue
17     myQueue.submit([&](handler& cgh) {
```

```
18      // request access to our buffer
19      auto writeResult=resultBuf.get_access<access::mode::write>(cgh);
20
21      // enqueue a parallel_for task
22      cgh.parallel_for<class simple_test>(
23        range<1>(1024), [=](id<1> idx
24      ) {
25        writeResult[idx] = idx[0];
26      });
27    }); // end of our commands for this queue
28
29  } // end scope, so we wait for the queue to complete
30
31  // print result
32  for(int i = 0; i < 1024; i++) {
33    std::cout << "data[" << i << "] = " << data[i] << std::endl;
34  }
35
36  return 0;
37 }
```

This piece of code was crucial to get the basics working. We should note, however, that this piece of code is from the final SYCL 1.2 specification (Revision Date 2015-05-08). We had some problems due to SYCL starting of as a provisional specification (Revision Date 2014-03-09) and gradually evolving – sometimes only certain conventions were changed (e.g. line 17 used to be `command_group(myQueue, [&]() {`), while other changes lead to significant refactoring. "Anatomy of a SYCL application" also used to be simpler due to the use of `single_task` instead of `parallel_for` on line 22, but that didn't change the milestone much.

The main goal of SYCL is to simplify heterogeneous programming, so this relatively simple code may not properly convey the amount of work that went into getting in to function properly. We present an approximate list of

steps taken by sycl-gtx:

1. Initialize the OpenCL platform, context, device, and command queue on line 11.

   - It selects the first available OpenCL platform (further work is required to make it more intelligent).
   - It selects the default device using the default device selector. Currently the default is just to select the first device on the platform.
   - The context also contains a default handler for asynchronous events (not implemented yet).
   - The queue keeps a record of the buffers used within it (not part of the first milestone, but currently working).

2. Create a buffer object on the device from existing data, on line 14. `range<1>(1024)` tells SYCL that the data is one-dimensional and features 1024 elements.

3. Submit commands to the queue on lines 17–27. Commands are stored as functions with metadata in sycl-gtx. `handler& cgh` is the command group handler – the `myQueue.submit` function accepts a whole group of commands, and the handler provides a link between the queue and the commands. The submitted function containing commands represents command group scope.

4. Obtain write access to the device buffer on line 19. `get_access` returns an `accessor` object, which allows the kernel code to manipulate data on the device.

5. Submit a command to enqueue a kernel on lines 22–24. Use 1024 work-items and a one-dimensional kernel. Additionally, provide an index for the current work-item – the dimensionality of the index needs to agree with that of the kernel. The kernel is provided as a C++11 lambda function, but is also given a name (`simple_test`) – the name is defined

as a class, but a class with this name does not need to exist. The kernel function opens up (and closes) kernel scope.

6. Compile kernel code on line 25. This is by far the most complicated step, which deserves a special section: 4.2. In this particular case, each work-item accesses its own element in the device buffer and assigns it its own ID. An `id` object can be used as an index into a buffer, but it needs to be explicitly converted into a number for calculations.

7. The queue starts executing the commands it received, plus a few hidden ones.

   (a) Initialize all buffers that were accessed in this set of commands and weren't initialized yet.

   (b) For each submitted kernel, copy the data for the buffers it uses from host to device, execute the kernel, and copy the results back. Based on the access modes, some copies can be avoided, and there are additional optimizations for inter-kernel dependencies.

8. Synchronization on line 29. All commands need to finish, all data needs to reside where the programmer expects it to. This wasn't such an issue for the first milestone, since OpenCL queues (used by SYCL queues) are by default in-order, and data copying was a blocking operation. However, for conformance with the specification, a lot of checks and potential waiting need to be employed.

4.2 The OpenCL code generator

The core and the main differentiator of the sycl-gtx implementation is the OpenCL code generator. SYCL code is compiled with the host compiler just like any other code, but special classes capture the code in the program executable. When the executable is run, it executes host code as normal, but when it reaches calls to these special classes that hold information about

the SYCL code, the code generator is invoked, which produces OpenCL C code, line by line. The generated code is then fed to the OpenCL C compiler, which is part of the OpenCL specification and is provided with device drivers. Because the SYCL code is captured at compile time, but actually compiled for the device at host runtime, this is a form of Just-In-Time compilation (JIT).

4.2.1 The `source` class

The code generator itself is coupled with the kernel source handler, which is the class `cl::sycl::detail::kernel_::source` (from now on referred to as the `source` class). An instance of this class stores the kernel name, lines of generated OpenCL C code, and a list of accessors that were used within the kernel. Before code generation can occur, the kernel scope needs to be entered, using the static method `source::enter(source& src)`, which is called inside a kernel invocation call (e.g. `cgh.parallel_for`) – it also has a corresponding `exit` method. The scope is basically just a static pointer to a `source` object. Because the pointer is static, there may be some problems with multithreaded code, although it is also defined to be thread local (different pointer for each thread) – this should help, although it wasn't tested extensively.

The `source` class contains a static `register_resource` method (for keeping track of used accessors within the kernel) and a static `add` method (for adding lines of OpenCL C code to the kernel). After kernel scope is exited, an argument list string is created from the used accessors – needs to take into account the underlying type, the access mode (read, write, ...), and the access target (global buffer, constant buffer, ...). The kernel name, the argument list, and the lines of code are joined into a single string that is later passed to the available OpenCL compiler.

We say that the code generator *emits* a line of code when the static `source` class method `add` is called, which is done using the `scope` pointer, so something like this:

```
1 cl::sycl::detail::kernel_::source::scope->add(line_of_code);
```

4.2.2 The `data_ref` class

Now we have the `source` class, but it doesn't perform much code generation – it mostly manages accessors and lines of code that were sent to it. The second piece of the code generating puzzle is the `cl::sycl::detail::data_ref` class (from now on the `data_ref` class). As mentioned in the memory model section, the host does not have access to the private memory region of the device – but `data_ref` offers a way to emulate it. All objects within kernel scope have to be derived from `data_ref` or at least should be able to interact with `data_ref`. Each `data_ref` object contains a string, which is that object's representation of an expression in OpenCL C. By overloading operators of `data_ref`, new OpenCL C expressions can be formed. All expressions remain internal and hidden from the programmer, disguised as regular data types. When a statement is encountered, the statement (which consists of expressions) is passed as a line of code to the `source` class.

To better explain it, consider the following kernel code:

```
1 int2 a(1, 10);
2 int2 b(1, 2);
3 b *= 2;
4 int2 c = a + b;
```

`int2` is a vector type, which holds two integer values, as per the SYCL specification. But in sycl-gtx `int2` is derived from `data_ref` (as are all vector types). Line 1 represents creation of a variable `a` of type `int2`. What actually happens is that the constructor of `int2` is called, which generates a name for the variable (this code is executed at runtime, so the original variable names `a, b, c` don't exist anymore). Specifically, it generates the string `"_int2_0"`, which is stored in the `data_ref` part of the `int2` type to represent this variable as a variable in device private memory. Line 2 is similar, and together they emit the following lines of code to the `source` class (comments

are not generated, just added here for clarity):

```
1 int2 _int2_0 = (int2)(1, 10); // a
2 int2 _int2_1 = (int2)(1, 2); // b
```

On line 3, the object `b`, which is an instance of `data_ref`, executes a call to its `operator*=(int)` function. This function converts the integer `2` into a string and emits the following line of code to the `source` class:

```
1 _int2_1 *= 2;
```

On line 4, the first thing to happen is that the object `a` (instance of `data_ref`), gets a call to its `operator+(const data_ref&)` function. This function returns a new `data_ref` object, where the expression string is `"(_int2_0 + int2_1)"`. Then, the `data_ref(const data_ref&)` constructor is called, to construct the object `c`. The object `c` gets the generated name `"_int2_2"` as its expression string, but the constructor also takes the temporary `data_ref` object and emits a single line of code to the `source` class:

```
1 int2 _int2_2 = (_int2_0 + _int2_1);
```

Similarly, `data_ref` overloads all of the other operators to interact with other `data_ref` objects and with all the basic numeric types. In the case of vector data types, there are additional overloads to ensure type safety (e.g. an `int2` cannot be directly assigned to `int3`).

4.2.3 Line 25 explained

With the code generator cleared up, we can finally discuss line 25 from the code in section 4.1, "Anatomy of a sycl-gtx application":

```
1 writeResult[idx] = idx[0];
```

`idx` is of type `id<1>`, which also derives from `data_ref`, but it receives special treatment. As soon as kernel scope is entered, the code generator emits:

```
1 const int _sycl_gid0 = get_global_id(0);
2 const int _sycl_gid = _sycl_gid0;
```

`writeResult` is an accessor, which is not derived from `data_ref`, but has an overloaded `operator[](const data_ref&)` method, which takes the expression name as the index (here `_sycl_gid`). The accessor also has a temporary name generated by the `source` class: `_sycl_buf1`. The call `idx[0]` should return an integer as per the specification – the index of the work-item executing the kernel instance – but sycl-gtx returns a `data_ref` object with an expression name `_sycl_gid0`. Thus the assignment looks like this in the generated OpenCL C code:

```
1 _sycl_buf_1[_sycl_gid] = _sycl_gid0;
```

For more examples on how sycl-gtx and the code generator work, please refer to section 4.4.

4.3 Limitations

As one can gather from the section on code generation, sycl-gtx employs the device compiler as a two-part design: an OpenCL code generator, which is compiled by the host compiler and invoked at runtime, and the OpenCL device compiler, which is also invoked at runtime to compile the generated OpenCL C code. Here we can see the main disadvantage of sycl-gtx: the device compiler does not have the same kernel information as the host compiler, because it is run at runtime: the host compiler actually eliminated most type information and all variable names, which leaves the device compiler in a tough spot. The `data_ref` class tries to capture as much information as it can – there are some other attributes captured besides the expression string. But the fact remains that some information is lost and cannot be retrieved.

4.3.1 Scalar numeric types are not directly available in kernels.

`data_ref` is able to interact with numeric scalars, but creating a scalar variable within a kernel does not necessarily work. Consider the following code,

where `i` is of type `id<1>`:

```
1 int k = 2 * i[0];
```

This code seems reasonable: `i[0]` is the current work-item index and we want to multiply it by two and store the result. This should work as per the specification, but it does not in sycl-gtx – `i[0]` return a `data_ref` instead of an integer and a `data_ref` cannot be converted to an integer. sycl-gtx solves this by providing an `int1` vector class, which represent an integer – `float1`, `long1`, etc. are also available. `int1` derives from `data_ref`, so it can be used in kernel scope. Since we want the code, written for sycl-gtx, to be valid on any SYCL implementation, we also provide a compatibility header, which contains typedefs for these types and should be included in every sycl-gtx file:

```
1 #ifndef SYCL_GTX
2 #include "sycl_gtx_compatibility.h"
3 #endif
```

Now we can write the line

```
1 int1 k = 2 * i[0];
```

and expect correct results in sycl-gtx and any other SYCL implementation.

Note that in some cases directly using scalars would still yield correct results in sycl-gtx, for example (`a` is an accessor):

```
1 int k = 5;
2 a[0] = k;
3 k *= 3;
4 a[1] = k;
```

This code is completely valid, because the values get inlined:

```
1 _sycl_buf1[0] = 5;
2 _sycl_buf1[1] = 15;
```

However, we do not recommend relying on these cases, as wrong behavior could easily be overlooked.

4.3.2 Control flow is not directly available

Just as variable names, a C++ program also cannot recognize control flow structures during runtime. Consider the following kernel code (`i` is again of type `id<1>`):

```
1 if(i[0] > 100) { ... }
2 else { ... }
```

Similarly to the previous case, this does not work, because `i[0]` is a `data_ref`, which can be compared to 100, but the result is another `data_ref` – this will not return the required results, because `i[0]` is not the actual work-item index, just a representation of it. Additionally, even if `data_ref` could be evaluated in a boolean context, the host program would then only execute one of the `if-else` branches, which means that the code generator would not have full kernel coverage.

Instead, we decided to use macros. Macros are somewhat unwanted in modern C++ code, but in sycl-gtx control flow macros aid the generator in evaluating each expression and statement within the whole kernel exactly once. The above code would thus be written as:

```
1 SYCL_IF(i[0] > 100) { ... }
2 SYCL_ELSE { ... }
3 SYCL_END
```

Note that control flow macros implicitly open a new scope, but that scope needs to explicitly closed with `SYCL_END`. What these macros actually do is that they emit lines of code to the generator. The macros are also included in the compatibility header, since their translation to regular C++ is pretty straightforward.

The above sycl-gtx code would translate into OpenCL C like this:

```
1 if(_sycl_gid0 > 100)
2 {
3   ...
4 }
```

```
5 else
6 {
7   ...
8 }
```

The call to `SYCL_IF` translates to lines 1 and 2 – line 1 is the `if`, line 2 is a new scope. Similarly `SYCL_ELSE` translates to lines 4, 5, and 6 – it recognizes that a new scope had to be have been opened for the program to be valid, so it closes it on line 4, emits the `else` statement on line 5 and opens up a new scope on line 6. Finally, line 8 contains the explicitly closed scope, `SYCL_END`.

`for` loops have similar macros – the idea is to have the code generator inspect the initialization, condition, increment, and the whole body of the loop, which means it needs to execute everything exactly once. Consider the following code:

```
1 SYCL_FOR(int1 j = 0, j < 100, ++j) { ... }
2 SYCL_END
```

This gets translated into OpenCL C:

```
1 int1 _int1_0 = 0;
2 for(; _int1_0 < 100; ++i)
3 {
4   ...
5 }
```

There are a few considerations here. We notice that variable initialization is on a separate line, line 1. This is because that line of code is emitted by the `int1` constructor (which we used because we cannot directly use scalars). Next we notice one of the limitations of macros: we need to use commas to properly pass arguments, and other programmer-supplied commas are not allowed as part of macro arguments. Another important consideration is the increment `++j`: this can be either a standalone statement or an expression within a statement. The problem is that it is not really possible to determine, when it is which. We decided to make it an expression – the standalone increment statement (`++j;`) should instead be replaced with a compound

assignment sum (`j += 1;`). By making it an expression, we can guarantee proper behavior in expressions – in fact, the `SYCL_FOR` macro would not work correctly if increment was treated as a statement, because it would generate the following code:

```
1 int1 _int1_0 = 0;
2 ++_int1_0;
3 for(; _int1_0 < 100; _int1_0)
4 {
5   ...
6 }
```

It may be interesting to investigate what happens if we use a regular `for` loop instead of the special sycl-gtx macros and the `int1` type, just `for(int j = 0; j < 100; ++j)`. There are no `data_ref` instances here, so the code generator knows nothing about the for loop. What happens is that the loop body gets executed 100 times – which means that calls to the code generator inside the loop also get executed 100 times, generating too much code. This could actually be used as an extension to SYCL, to knowingly generate code, but we do not recommend it because it does not conform to the specification.

At this point it may be important to note that SYCL allows function calls within kernels, but only if those function adhere to the restricted C++ kernel code syntax. In sycl-gtx, this is also allowed, but all function calls get inlined.

4.3.3 No host fallback

Because of all the above mentioned limitations, we decided not to implement host fallback. It definitely is possible – maybe ship a custom gcc compiler along with the SYCL library to act as the SYCL host device compiler, or keep track of detailed type information within `data_ref`. But since the specification is quite extensive, we decided against it.

4.3.4 Many unimplemented features

Along with host fallback, many features are missing in sycl-gtx, e.g. images, atomics, constant buffers, etc. We will try to provide further effort on the implementation, but it is our hope that because the whole project is open source other developers may implement at least some missing features.

4.4 Example code

Once the first milestone was reached, we started implementing additional tests. For example, we implemented passing functors as kernels (a functor is an object that acts as a function) and invoking kernels asynchronously. Some of the more interesting tests are described below in order to provide a better overview of SYCL and sycl-gtx.

4.4.1 Vector addition

We wanted to compile the test presented on the Codeplay Developer Blogs [50] (slightly simplified here). This is one of the best showcases for parallelization, because adding two vectors is a locally simple operation (sum two adjacent elements) without interdependencies, so it is very simple to parallelize.

Assuming `count` is the size of a vector, and `a, b, r` are the vectors, the following is the usual approach:

```
1 for(int i = 0; i < count; ++i) {
2   r[i] = a[i] + b[i];
3 }
```

The SYCL implementation requires specifying a queue, creating buffers for vectors, getting access to device data using data accessors, and passing the kernel to the device. So, assuming now that `cgh` is the handler for the command group that is sent to the queue and `a, b, r` are the accessors to device buffers, the following is the SYCL kernel:

```
cgh.parallel_for<class addition>(range<1>(count), [=](id<1> i) {
  r[i] = a[i] + b[i];
});
```

We see that the actual kernel doesn't differ much from the serial code implementation. Of course, there is some initialization required, but as a result we get code that is completely parallelized and able to run on any OpenCL device.

We can also observe the generated OpenCL C code:

```
__kernel void _sycl_kernel_0(
  __global int* _sycl_buf3,
  __global const int* _sycl_buf1,
  __global const int* _sycl_buf2
) {
  const int _sycl_gid0 = get_global_id(0);
  const int _sycl_gid = _sycl_gid0;
  _sycl_buf3[_sycl_gid]
    = (_sycl_buf2[_sycl_gid] + _sycl_buf1[_sycl_gid]);
}
```

In OpenCL C, a kernel function starts with `__kernel`, followed by `void` – it doesn't have a return type, because it doesn't return in the classic sense. `_sycl_kernel_0` is a generated kernel name; instead of `0` there could be any number, as it's meant only to prevent name clashes. As we've shown in section 4.1, the programmer needs to supply a kernel name, if the kernel is a lambda function, or the name is taken from the supplied functor. However, this name is not visible at runtime, so the OpenCL code generator is unable to see it, although it tries to provide a translation between the compile-time and the runtime names.

Because the code asked for access to global buffers (not shown here, but similar to 4.1), the data is passed as a `__global` pointer of the base element type (here `int`). Buffer names are also generated, but their order isn't always obvious. Since we only read from buffers `a` and `b`, we asked for write access,

which also provides the `const` specifier.

Lines 6 and 7 are temporaries to hold the index. The `i` that was passed to the kernel can be resolved as an index to `_sycl_gid`, or as a number `i[0]` (which can also be used as an index) to `_sycl_gid0`. This is somewhat redundant in a one-dimensional kernel, but is included for consistency.

The actual calculation is done on lines 8 and 9 (in two lines because of formatting). Every arithmetic operation is wrapped into parentheses in order to preserve operator precedence.

4.4.2 Matrix rotation

SYCL is also designed to deal with 2D and 3D data. A simple example is rotating a $N \times N$ matrix A and storing the result into B:

```
1 for(int x = 0; x < N; ++x) {
2   for(int y = 0; y < N; ++y) {
3     B[N - y - 1][x] = A[x][y];
4 }}
```

For SYCL, we first need to serialize the data to store it into a buffer (assuming we're working with matrices of floats), e.g.:

```
1 buffer<float, 2> a_buf(reinterpret_cast<float*>(A), range<2>(N, N));
```

The accessors are not affected by the dimensionality of the data or the kernel, so they are basically the same as in section 4.1. And now the kernel:

```
1 cgh.parallel_for<class rotation>(range<2>(N, N), [=](id<2> i) {
2   b[N - i[1] - 1][i[0]] = a[i];
3 });
```

We immediately recognize that `i[0]` stands for `x` and `i[1]` for `y`, while as an added convenience we can just use `i` as a 2D index and SYCL resolves it based on the work-item index and the work-group size. This can be observed from the generated OpenCL C code:

```
1 __kernel void _sycl_kernel_0(
```

```
2   __global float* _sycl_buf2, __global const float* _sycl_buf1
3 ) {
4   const int _sycl_gid0 = get_global_id(0);
5   const int _sycl_gid1 = get_global_id(1);
6   const int _sycl_gid =
7     _sycl_gid1 * get_global_size(0) + _sycl_gid0;
8   _sycl_buf2[((1024 - _sycl_gid1) - 1) + _sycl_gid0 * 1024]
9     = _sycl_buf1[_sycl_gid];
10 }
```

4.4.3 Parallel reduction sum

Reduction is the process where a single binary operation is applied to a series (array) of values, where the left operand is the accumulator. An example is summation: the operation is +, the initial value of the accumulator is 0, and every value in an array is added to the accumulator. This is a slightly more difficult problem than vector addition or matrix rotation, but it can be parallelized quite efficiently. Serial code for a vector `a` of size `N` would look something like this:

```
1 int sum = 0;
2 for(int i = 0; i < N; ++i) {
3   sum += a[i];
4 }
```

For a parallel implementation, we can operate on the array itself, but we need to split the process into multiple steps [51]: first sum in parallel each two neighboring elements and store the result in the place of the first of those two elements (`a[0]+=a[1]`, `a[2]+=a[3]`, `a[4]+=a[5]`), then sum up those sums in the same manner (`a[0]+=a[2]`, `a[4]+=a[6]`, `a[8]+=a[10]`), and so forth until there is only one sum left.

This distance between the elements that need to be summed is called a stride, and it increases exponentially: $1, 2, 4, 8, ...$ On the other hand, since on each step we have half as many operations, there are a total of $log_2(N)$

steps. The SYCL implementation is as follows:

```
auto s = stride_.get_access<access::mode::read_write>(cgh);
for(size_t stride = 1; stride < N; stride *= 2) {
  cgh.parallel_for<class reduction_sum>(
    range<1>(N / 2 / stride),
    [=](id<1> index) {
      auto i = 2 * s[0] * index;
      a[i] += a[i + s[0]];
      s[0] *= 2;
});}
```

We use a `for` loop to issue $log_2(N)$ kernels with different strides – the kernel should only be compiled once, but in the current sycl-gtx implementation it is always compiled, which leads to some slowdown. These kernels would normally be launched asynchronously, but since there are interdependencies, the SYCL runtime makes sure they execute in the right order. `a` is here a read-write accessor to the array `a`.

One problem is the stride: we need to pass it to the kernel invocation (host code) – and also to the device. In the kernel invocation, we specify the size of the work-group (the number of work-items) by passing `range<1>(N / 2 / stride)`. The reason for this particular size is that at every step the number of operations halves (controlled by `stride`), and we always use one work-item to sum up two values (the division by 2). Besides the host-side `stride` we also have an accessor `s`, which offers access to a one-dimensional device buffer of size 1, `stride_`.

```
__kernel void _sycl_kernel_0(
  __global int* _sycl_buf1,
  __global int* _sycl_buf2
) {
  const int _sycl_gid0 = get_global_id(0);
  const int _sycl_gid = _sycl_gid0;
  _sycl_buf2[((2 * _sycl_buf1[0]) * _sycl_gid)]
    += _sycl_buf2[(((2 * _sycl_buf1[0]) * _sycl_gid) + _sycl_buf1[0])];
```

```
9   _sycl_buf1[0] *= 2;
10 }
```

The OpenCL C code above is pretty straightforward: `a` is represented by `_sycl_buf2` and `s` is represented by `_sycl_buf1`. There is one issue, however: `i` is not stored as a value, but is instead expanded into `((2 * _sycl_buf1[0]) * _sycl_gid)`.

4.4.4 Vector data types

SYCL supports device vector types and also provides special vector types for OpenCL interoperability. In the following code excerpt we define a test vector of three elements and assign it to 10 other vectors in parallel:

```
1  const int size = 10;
2  auto vectors = buffer<float3>(range<1>(size));
3  auto testV = buffer<float3>(range<1>(1));
4  {
5    auto testVector_ = testV.get_access<
6      access::mode::discard_write, access::target::host_buffer>();
7    auto& testVector = testVector_[0];
8    testVector.x() = 1;
9    testVector.y() = 2;
10   testVector.z() = 3;
11 }
12 myQueue.submit([&](handler& cgh) {
13   auto v = vectors.get_access<access::mode::discard_write>(cgh);
14   auto testVector_ = testV.get_access<access::mode::read>(cgh);
15   cgh.parallel_for<class vectors>(range<1>(size), [=](id<1> i) {
16     auto testVector = testVector_[0];
17     v[i] = float3(testVector.x(), testVector.y(), 0);
18     v[i].z() = testVector.z();
19   });
20 });
```

We define `vectors`, a buffer of vectors of three floating point numbers on line 2, the buffer contains 10 vectors. On the next line a test vector `testV` is defined with the same type as `vectors`, but with only one element. Since we want to define an initial value for `testV`, but `testV` is a buffer, we need to obtain a host accessor on lines 5 and 6. `discard_write` means that we only want to write to this buffer and do not care about any previous values. Note that host access to this buffer is wrapped in a new scope (lines 4 to 11). This is done because obtaining a host accessor means giving control over the buffer over to the programmer and SYCL isn't allowed to access the buffer while the programmer has control over it. Also worth noting is that we obtained an accessor to an array of one element, so line 7 serves only to simplify access to this element. Lines 8, 9 and 10 then showcase the initialization of the test vector element to $(1, 2, 3)$.

For the kernel, first the buffer accessors are obtained on lines 13 and 14 (line 14 would not have worked if a new scope hadn't been employed for the host accessor on lines 4 to 11). The one-dimensional kernel is enqueued on line 15 with 10 work-items. Line 16 also serves just to simplify access to the single array element. In line 17 a vector is assigned using a vector constructor, while line 18 demonstrates assigning a single vector element. Again we can observe the generated OpenCL C kernel:

```
1 __kernel void _sycl_kernel_0(
2   __global const float3* _sycl_buf1, __global float3* _sycl_buf2
3 ) {
4   const int _sycl_gid0 = get_global_id(0);
5   const int _sycl_gid = _sycl_gid0;
6   float3 _float3_2 = (float3)(_sycl_buf1[0].s0, _sycl_buf1[0].s1, 0);
7   _sycl_buf2[_sycl_gid] = _float3_2;
8   _sycl_buf2[_sycl_gid].s2 = _sycl_buf1[0].s2;
9 }
```

We notice that a temporary was created on line 6, even though it is used only once (line 7) and could easily have been inlined. This is one of the quirks

of sycl-gtx, as the proper copying and movement of data within the kernel has proven to be a slightly elusive goal. In the current implementation sycl-gtx tries to be more conservative with inlining, instead preferring to create temporaries, which has proven to be more accurate during testing, although there are cases where it may still fail. For example, using `auto&` instead of `auto` on line 16 of the above C++ code yields an error in sycl-gtx as of February 2016, although it would be correct SYCL – indeed, ComputeCpp works properly with `auto&`.

Otherwise, the vector constructor on line 6 is pretty straightforward in OpenCL C, as well as the single element access on line 8 (apart from the unnecessary temporary on line 7). One thing to note is that instead of `.x`, `.y`, and `.z`, access to elements is provided as `.s` and the sequential number of the element (as per the OpenCL specification [52]) – although this was a pretty arbitrary choice.

4.5 Porting the OpenCL example to sycl-gtx

We've seen a simple parallel kernel written in OpenCL and SYCL (in sections 3.1.4 and 3.2.4, respectively). Based on all the presented sycl-gtx code examples we now have a good idea of how the example can be ported to sycl-gtx:

```
1  int A[N], B[N], C[N]; // N known from before, actual values elsewhere
2  using namespace cl::sycl;
3
4  {
5    gpu_selector gpu;
6    queue q(gpu);
7
8    auto rN = range<1>(N);
9    auto bufA = buffer<int>(A, rN);
10   auto bufB = buffer<int>(B, rN);
11   auto bufC = buffer<int>(C, rN);
```

```
12
13   q.submit([&](handler& cgh) {
14     auto a = bufA.get_access<
15       access::mode::read, access::target::global_buffer>(cgh);
16     auto b = bufB.get_access<
17       access::mode::read, access::target::global_buffer>(cgh);
18     auto c = bufC.get_access<
19       access::mode::write, access::target::global_buffer>(cgh);
20
21     cgh.parallel_for<class example>(rN, [=](id<1> i) {
22       SYCL_IF(a[i] % 2) {
23         c[i] = a[i] + b[i];
24       }
25       SYCL_ELSE {
26         int1 Ci = b[0];
27         SYCL_FOR(int1 j = 1, j < 5, ++j) {
28           Ci *= b[j];
29         }
30         SYCL_END
31         c[i] = Ci;
32       }
33       SYCL_END
34 }); }); }
```

There are only a few differences to the pure SYCL example, all in the kernel: the use of macros for control flow and replacing `int` with `int1`. Here's what the kernel gets translated to:

```
1  __kernel void _sycl_kernel_0(
2    __global int* _sycl_buf3,
3    __global const int* _sycl_buf1,
4    __global const int* _sycl_buf2
5  ) {
6    const int _sycl_gid0 = get_global_id(0);
7    const int _sycl_gid = _sycl_gid0;
```

```
8   if(_sycl_buf1[_sycl_gid] % 2) {
9     _sycl_buf3[_sycl_gid] =
10      _sycl_buf1[_sycl_gid] + _sycl_buf2[_sycl_gid];
11  }
12  else {
13    int int_0 = _sycl_buf2[0];
14    int_1 = 1;
15    for(; int_1 < 5; ++int_1) {
16      int_0 *= _sycl_buf2[int_1];
17    }
18    _sycl_buf3[_sycl_gid] = int_0;
19  }}
```

Interestingly, replacing the `for` loop with a macro is not necessary in this case, because the loop would just be unrolled. So the loop:

```
1  for(int j = 1; j < 5; ++j) {
2    Ci *= b[j];
3  }
```

would become:

```
1  int_0 *= _sycl_buf2[1];
2  int_0 *= _sycl_buf2[2];
3  int_0 *= _sycl_buf2[3];
4  int_0 *= _sycl_buf2[4];
```

But this kind of behavior may seem unpredictable to the programmer, so the supplied macros are preferred.

Regarding the `if` statement, not using the `SYCL_IF` macro leads to a compile-time error: the expression `a[i] % 2` returns a `data_ref` object in sycl-gtx, which is not convertible to `bool`.

4.6 Additional remarks

During the process of implementing the SYCL specification we found some minor errors in the specification itself. Our work started while the specification hadn't yet been finalized, so some errors were expected. We noted most of them as comments to the specification and made the annotated version available alongside main code in the public repository [14]. Unexpectedly, the release of the final SYCL 1.2 specification did not get rid of all the errors. We contacted Codeplay (one of the proposers and the main implementers of the specification) about the errors and they have acknowledged a corrected specification was going to be released.

We also wrote an article summarizing this thesis, called "An Overview of sycl-gtx", which was presented at the SYCL'16 workshop of the PPoPP 2016 conference [53].

Chapter 5

Tests

Tests were mostly developed to ensure SYCL conformance – our approach to the implementation was in a way test–driven, where we would write a test an then implement the necessary functionality. These tests were already discussed in section 4.4 – here we discuss another test with a focus on performance observations.

5.1 smallpt

We stumbled upon an implementation of the smallpt ray tracer in SYCL, presented on the Codeplay developer blog [54]. This would have been a great opportunity to test sycl-gtx, to see whether it would be able to compile a ray tracer – although very small by ray tracer standards, it was much more comprehensive than any of the proof-of-concept tests we'd written.

smallpt is a global illumination renderer, written in 99 lines of C++ [55], which are open sourced. Ray tracing is a conceptually simple technique for rendering a scene: send rays of light from the camera, follow (trace) their movement through the scene using a physical simulation, and add color to the ray based on the light sources it finally hits (if any) and the materials encountered on the path. Computationally, it's very expensive: for each camera pixel at least one ray (sample) needs to be used, while multiple

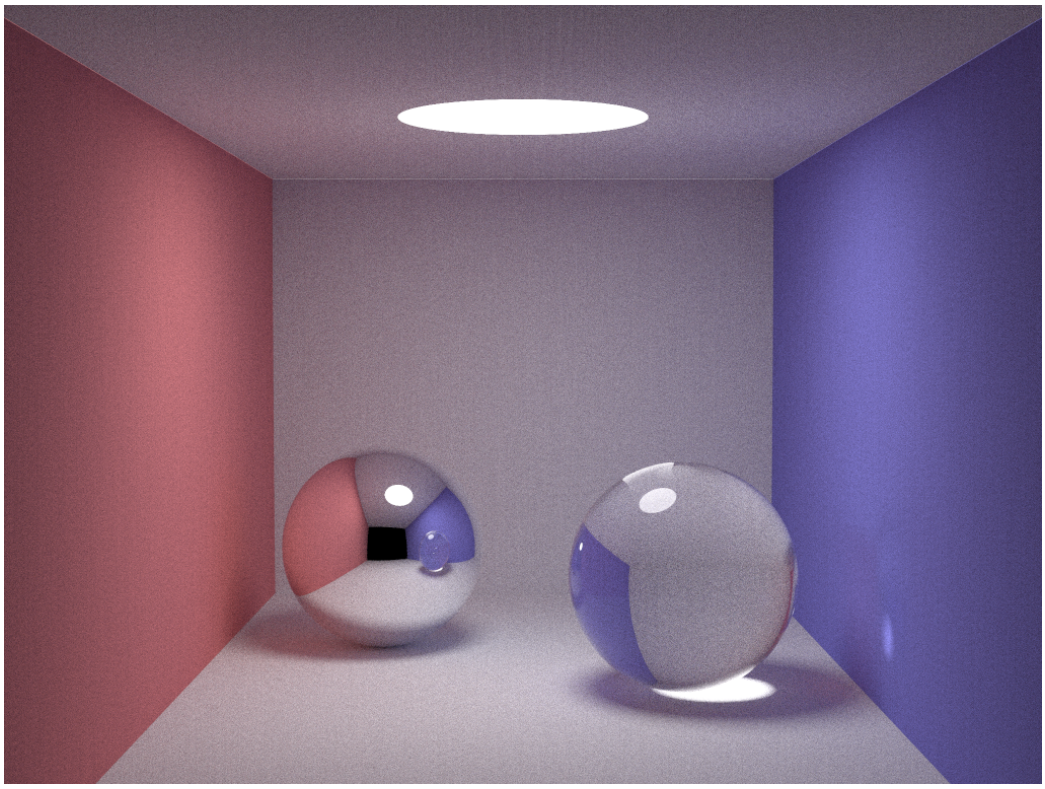

Figure 5.1: Example output of running the smallpt tester with 512 samples per pixel using Intel HD Graphics 4600.

samples per pixel provide better accuracy.

However, pixels are computed independently from one another, which means that ray tracing is inherently parallel – a common modern resolution of 1920 times 1080 means 2073600 pixels, offering lots of opportunities for multi-core hardware. Using multiple samples per pixel exposes even more parallelism, although the samples need to be averaged for the end result.

To understand smallpt slightly better, we list here some of the more important features:

- Global illumination via unbiased Monte Carlo path tracing.
- Antialiasing through 2x2 super-sampling.

- Specular, diffuse, and glass reflection.
- Russian roulette for path termination.

As mentioned, we were inspired by the Codeplay blog post and they provide the full code for the SYCL-ported smallpt program. The first thing they did was move from the default double precision floating point numbers to single precision. The reason was that many GPUs still don't support double precision operations. Replacing double with single precision would normally imply shorter runtimes, but in this case it actually slightly slows down the computation. The reason for the slowdown in smallpt is that lower precision leads to artifacts, which represent unnecessary computation. To combat this, the scene was adjusted (smaller spheres) and the margin of error when calculating intersections was increased.

The second problem was the random number generator (RNG). smallpt relies heavily on randomization, but OpenCL does not have a standard random number generator. Instead, Codeplay wrote their own based on a Xorshift RNG [56].

The third problem was replacing recursion, which smallpt relies on to compute reflectance, with iteration, because OpenCL does not yet support recursion. In general, any recursion can be replaced with iteration with the help of an additional stack, but in this case this was slightly simpler to achieve – on recursion, the traced ray is modified and the pixel colors and reflectance are accumulated.

Then, the code could be ported to SYCL – create a queue, data buffers, submit the kernel, and wait for the computation to end. We modified their code slightly to fit into the tester we've written (passing the device to execute the queue on and controlling extra parameters) and renamed the variables to be more self-descriptive. Here is the function with full SYCL code sans the kernel:

```
1 void compute_sycl(
2   void* device_,
```

```
3   int width, int height, int samplesPerPixel,
4   Ray camera, Vec cx, Vec cy, Vec initialRadiance, Vec* colors) {
5   queue q(*(device*)device_);
6   {
7     // data is wrapped in SYCL buffers.
8     buffer<Vec, 1> color_buffer(colors, range<1>(width * height));
9     buffer<Sphere, 1> spheres_buffer(&spheres_glob[0], range<1>(9));
10    auto cg = [&](handler& cgh) {
11      kernel_r smallpt = {
12        // enabling access of the data on the device for SYCL.
13        color_buffer.get_access<access::mode::write>(cgh),
14        spheres_buffer.get_access<
15          access::mode::read,
16          access::target::constant_buffer>(cgh),
17        width, height, samplesPerPixel,
18        camera, cx, cy, initialRadiance
19      };
20      nd_range<2> ndr(range<2>(width, height), range<2>(8, 8));
21      ch.parallel_for(ndr, smallpt);
22    };
23    // submitting the command group to the SYCL command queue
24    // for execution.
25    q.submit(cg);
26  }
27 }
```

We can observe the main mission of SYCL in this function: to simplify heterogeneous programming. Apart from the three problems we discussed above, writing SYCL code is rather straightforward, because it doesn't require a lot of code and follows a template. Instead, the focus is shifted to writing the kernel, which is in this case the most important part of the application. Note that the classes `Vec`, `Ray`, and `Sphere` were already part of the original smallpt code.

By porting the code to SYCL, Codeplay reported a speedup factor of

almost 40 when using an AMD Radeon HD R9 295X, achieving only slightly lower precision than the original program.

5.2 Porting smallpt to sycl-gtx

The ported smallpt code is available for download [54] – we've managed to get it fit into our tester and compile it using ComputeCpp (with quite some difficulties). But it doesn't work with sycl-gtx at all. Replacing control flow with macros and scalars with one-dimensional vectors in kernel code was trivial. However, we had significant problems with custom data structures.

Instead of trying to get the Codeplay port work in sycl-gtx, we decided to start from the original smallpt code. That way, we could document all the changes as commits [14] and would gain a better understanding of smallpt. Another important factor was that at the time that we decided to port smallpt to sycl-gtx, sycl-gtx was not nearly implemented enough to allow for a ray tracer to work, so we worked on the implementation along with porting smallpt.

We took the basic same steps as Codeplay, but with some modifications. We moved from double to single precision, but also implemented an abstraction layer to support the original data. We used a different, simpler RNG. Replacing recursion with iteration was done in pretty much the same manner, though.

Of course, within kernel code all control flow was written using custom macros and scalars were replaced with custom vectors. An additional problem was that if a helper function had more than one return point, the return value needed to be stored as an extra variable and returned once at the end, or passed as an extra input reference. So instead of:

```
1 inline float clamp(float x) {
2   return x < 0 ? 0 : x > 1 ? 1 : x;
3 }
```

we would need to write the more cumbersome (but arguably clearer)

```
1 inline void clamp(float1& x) {
2   SYCL_IF(x < 0)
3     x = 0;
4   SYCL_ELSE_IF(x > 1)
5     x = 1;
6   SYCL_END
7 }
```

Note that in this case we avoided using an extra return variable by modifying the existing input, which was justified by the way this function was called in the kernel.

As mentioned, the biggest problem were custom data structures. Some of this was discussed in the section 4.2 – all data references within a sycl-gtx kernel need to derive or be able to interact with objects of type `data_ref`, otherwise the JIT code generator cannot observe it at runtime. So while passing a `Vec` object (a smallpt vector of three floating point numbers) to a SYCL buffer would properly copy the object to and from the device, the kernel would not be able to interact with it properly.

To solve this, we modified the original smallpt classes to have a templated type, so instead of `Vec` being a collection of three `double` values, we would have `using Vec = Vec_<double>` for the same class and also `Vec_<float1>` for a collection of three `float1` values. We then had to inherit from `Vec_<float1>` and add a few simple constructor to allow for conversions between different types. But this still wouldn't have quite worked: sycl-gtx vector types are only supposed to be used inside kernels, not stored in buffers. Indeed, using `buffer<Vec_<float1>>` would have resulted in a runtime error. Instead, we used `buffer<float3>` and provided a simple constructor to the class that derived from `Vec_<float1>` to deal with the conversion. Here is the full `Vector` class:

```
1 struct Vector : public Vec_<float1> {
2 private:
3   using Base = Vec_<float1>;
```

```
4  public:
5    Vector(float x_ = 0, float y_ = 0, float z_ = 0)
6    : Base(x_, y_, z_) {}
7    Vector(const ::Vec_<float>& base)
8    : Base(base.x, base.y, base.z) {}
9    Vector(const Base& base)
10   : Base(base) {}
11   Vector(float3 data)
12   : Base(data.x(), data.y(), data.z()) {}
13 };
```

But if `float3` is also a sycl-gtx vector class, how come then that `buffer<float3>` works correctly? This is actually a hidden feature of sycl-gtx: the `buffer` class recognizes sycl-gtx vector `float3` and instead stores the data as a regular OpenCL vector `cl_float3`. The accessors in the kernel then return the `float3` values. But this only works if the data type passed to the buffer *is* a sycl-gtx vector, not if it contains one (or more).

Porting the `Sphere` class works similarly, but also requires much more thought. A `Sphere` class is pretty heterogeneous compared to the homogeneous `Vec`: it contains one `float`, three `Vecs`, and one custom `enum` value. We decided to treat every value as a `float1` – which means a `Sphere` would contain 11 values – and pack everything into a `buffer<float16>`. The code:

```
1  struct SphereSycl : public Sphere_<float1> {
2    float1 reflectance;
3    SphereSycl(const float16& data)
4    : Sphere_<float1>(
5        data.lo().lo().w(), // Radius
6        Vector(data.lo().lo().xyz()), // Position
7        Vector(data.lo().hi().xyz()), // Emission
8        Vector(data.hi().lo().xyz()), // Color
9        Refl_t::DIFF // Original enum value, not important
10     ),
11     reflectance(data.hi().lo().w()) // Reference to the enum value
```

```
12    {}
13    float1 intersect(const Ray_<float1>& r) const {
14      float1 return_;
15      ...
16      return return_;
17    }
18  };
```

Note that the `intersect` method needed to be overridden in order to conform to the rules of sycl-gtx kernel code.

This approach has a major drawback: excess data. A `Sphere` has 11 numeric values, but `buffer<float16>` stores 16 for each sphere, which means more data needs to be copied when buffers are copied to and from the device. Luckily, the tested scene contains only 9 spheres. Additionally, `SphereSycl` inherits from `Sphere_<float1>`, but adds a `float1` to act as a reference to the `enum` value, while the original `enum` value is ignored – though this could be avoided by further templating the original `Sphere` class.

Using this approach, passing data finally worked. We just needed to provide some extra code: assign the original spheres to the buffers, prepare a buffer of seeds to be passed to the RNG in the kernel, and copy the buffer of colors to the original array at the end of computation.

We observed that running long computations on the GPU resulted in the computer becoming unresponsive for the duration of the computation, even freezing the display image. This is because the integrated GPU under test was also used to drive the display, so dedicating it to computation would make the screen unresponsive. We decided to solve this by splitting the computation into multiple parts – the split is done vertically with regard to the 2D image buffer and it depends on the number of samples per pixel and the height of the image.

5.3 Testing environment

A tester program was written that evaluates the smallpt computation for an image of size 1024x768 pixels and the default test scene, which consists of a room with colored walls, one glass and one mirror-like ball, and a circular light on the ceiling. The tests consisted of the original smallpt code and the floating point version – both of these also had an OpenMP version.

Briefly, OpenMP is a simple way to parallelize code across CPU cores using preprocessor directives [9]. Example for smallpt:

```
1 #pragma omp parallel for schedule(dynamic, 1) private(initialRadiance)
2 for(int y = 0; y < height; y++) { // Loop over image rows
3   org::compute_inner(
4     y, width, height, samplesPerPixel,
5     camera, cx, cy, initialRadiance, colors);
6 }
```

Here the `#pragma` acts a a signal to the compiler to schedule the body of the `for:y` loop across multiple cores.

In addition to these four tests, we added the sycl-gtx version and the SYCL version from the Codeplay blog [54]. Unfortunately, the SYCL ports were mutually exclusive: The Codeplay version does not work with sycl-gtx and while we managed to successfully compile our version of smallpt using ComputeCpp, it did not work correctly. Since ComputeCpp is not finished yet, it is difficult to speculate, what the problem may be, but our guess is that sycl-gtx misses something that we are unaware of with regard to SYCL specification conformance. What we ended up doing was compiling two tester programs: one that handled the original smallpt and the sycl-gtx version, compiled using Visual Studio 2013 with the OpenMP switch, and another one that handled the Codeplay version and was compiled using ComputeCpp and Visual Studio 2013. Both shared as much code as possible and were compiled using the "Release" target and the `/O2` optimization switch.

The tester tries to find all OpenCL 1.2+ compatible devices in the system and creates a new test for the SYCL test code for each device. For each device

information about it is displayed. The tests are stored as a `struct` containing the device name, the pointer to the device, and the function pointer to the test. The original smallpt tests are stored the same way, only that the "device name" is hard coded and the device pointer is set to `nullptr`.

The tester contains multiple time checks. It has a global time limit, which can be passed as a parameter to the program (in minutes) and is the maximum amount of time available to the test suite – though this isn't strictly enforced, the check only occurs after each test run, not in between. It also has a per test time limit, which is set to 40 seconds and checks the last runtime of the test. The per test limit check is only enforced before the test runs and there are additional exceptions when the test is allowed to run, based on whether there are any OpenCL-accelerated tests available and whether the total test suite running time hasn't exceeded half of the global limit yet. The idea is to prevent the tester from terminating too quickly (more tests means more results) or too late (being an inconvenience to the user) and to prefer OpenCL and stronger hardware (not to waste too much time on slow code, as we can see in section 5.4).

The tests had a ramp-up, starting at 4 samples per pixel and each iteration the number of samples doubled. Note that we do not consider the antialiasing feature of smallpt in our results, which means that the actual number of samples per pixel is 4 times higher than what is listed in our results (the tests on the Codeplay blog did consider this feature, so their 5000 samples per pixel would correspond to our 1250).

The sycl-gtx GitHub repository [14] is constantly changing, but we created a tag that was used for these tests: `smallpt-sycl-gtx-v1.4`.

5.4 Results

5.4.1 Modern quad core CPU with integrated GPU

Below is the full table (Table 5.1) of results obtained when running the tester on an Intel Core i5-4570, which has four cores and an integrated GPU, the Intel HD Graphics 4600 with 20 Execution Units. The CPU has a base frequency of 3.2 GHz and can boost up to 3.6 GHz. The tester was executed on Windows 10 with the latest (February 2016) Intel OpenCL drivers.

	4	8	16	32	64	128	256	512
Original	23.02	45.93	91.82	184.76	369.24	736.02		
OpenMP	7.51	15.03	29.91	60.99	119.87	239.21	479.7	
Float	21.82	43.55	86.83	173.09	342.32	689.48		
OMP Float	7.36	15.31	29.06	58.22	116.59	233.60	466.08	
i5-4570 1.2	2.95	5.27	10.10	19.58	39.03	77.24	154.64	309.89
i5-4570 2.0	3.27	5.41	10.64	20.64	41.78	82.83	165.18	331.34
HD4600 1.2	1.04	1.61	2.75	5.11	10.03	19.90	45.79	93.17
i5-4570 1.2	5.48	10.52	20.67	40.77	80.78	161.40	323.37	
i5-4570 2.0	5.26	10.04	19.90	39.41	78.57	156.69	314.12	
Fallback	51.86	63.81	89.29	151.81	267.52	501.24		

Table 5.1: Results of the smallpt tester running on an Intel Core i5-4570. All times are in seconds. The columns represent test runs when using different numbers of samples per pixel.

The results are grouped and the names are shortened. All values are in seconds. The first row is the number of samples per pixel used – not considering antialiasing. There are multiple samples per pixel because a pixel is treated as a small square surface instead of a point, so it is possible to send multiple rays per one pixel through different coordinates.

The next four rows correspond to the original smallpt code, also with OpenMP ("OMP"), and the single precision variants ("Float"). The three

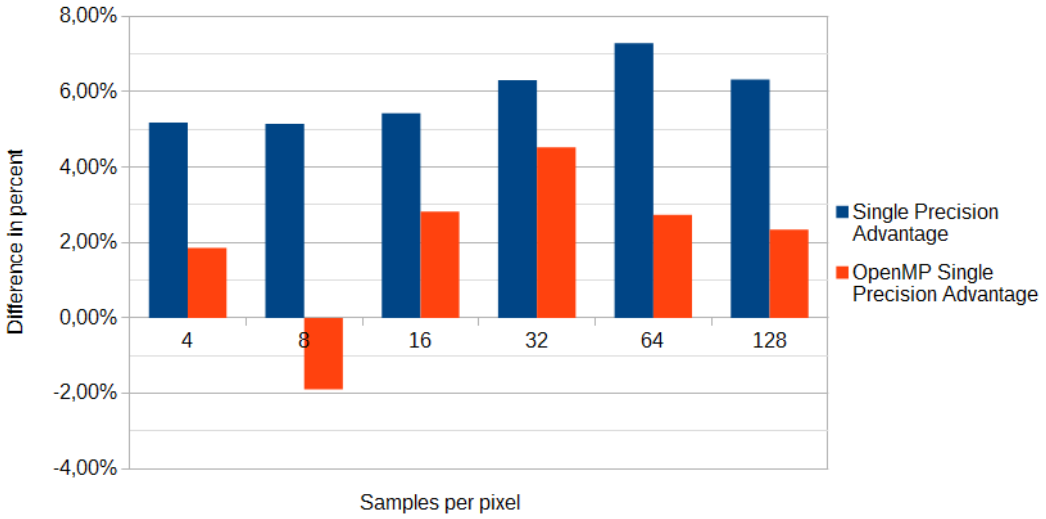

Figure 5.2: The difference between single and double precision variants of the original code.

rows after that are results from sycl-gtx, using the Core i5-4570 CPU with OpenCL 1.2 and 2.0 and the HD Graphics 4600 with OpenCL 1.2.

The last three rows correspond to ComputeCpp, which is similar to sycl-gtx, except that compiling for the GPU failed every time, so there are no GPU results for ComputeCpp – instead the last result is the SYCL host fallback, which was programmatically forced.

Let us look first how much did the original code benefit from moving from double to single precision in the original code, in Figure 5.2. It can be observed that moving to single precision did not bring a big performance improvement. This has already been mentioned, the reason lies in the extra artifacts produced with lower precision. The difference is noticeable and consistent at around 5% for the single threaded version and about 2 to 3% for the OpenMP version. This only confirms that making the SYCL variants use single precision does not present an unfair advantage.

Staying away from OpenCL a bit longer, Figure 5.3 showcases the running times of the smallpt variants that do not use OpenCL. In this figure we ignore single precision, as we've already established the difference isn't large. Even

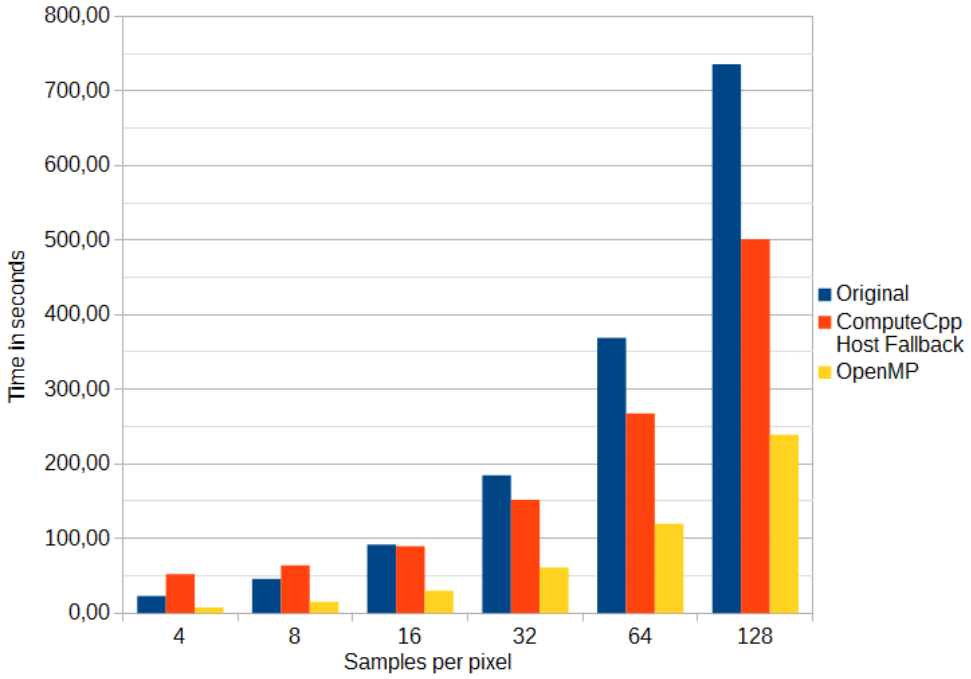

Figure 5.3: Execution time of non-OpenCL variants.

though the Core i5-4570 is a 4 core CPU, OpenMP isn't four times faster than the single-threaded original code, but rather closer to three times faster. Some of the difference may be attributed to the single core boost of the i5-4570, some maybe to OpenMP.

What is more interesting is the ComputeCpp host fallback. This starts up much slower than the single-threaded code and becoming faster later, consistently increasing the gap. Our guess is that there is some overhead at lower sample rates – host fallback basically needs to simulate the OpenCL execution model, which may not be efficient if the workload isn't large enough.

We observed something very strange in the ComputeCpp OpenCL results: the values are almost exactly twice the sycl-gtx values. In fact, if we shift the ComputeCpp values one column to the right, we can observe a very small difference in results, shown in Figure 5.4. Moreover, this difference is very consistent: when using OpenCL 1.2 sycl-gtx is about 4% faster, while when using OpenCL 2.0 ComputeCpp is about 5% faster. We do not know the

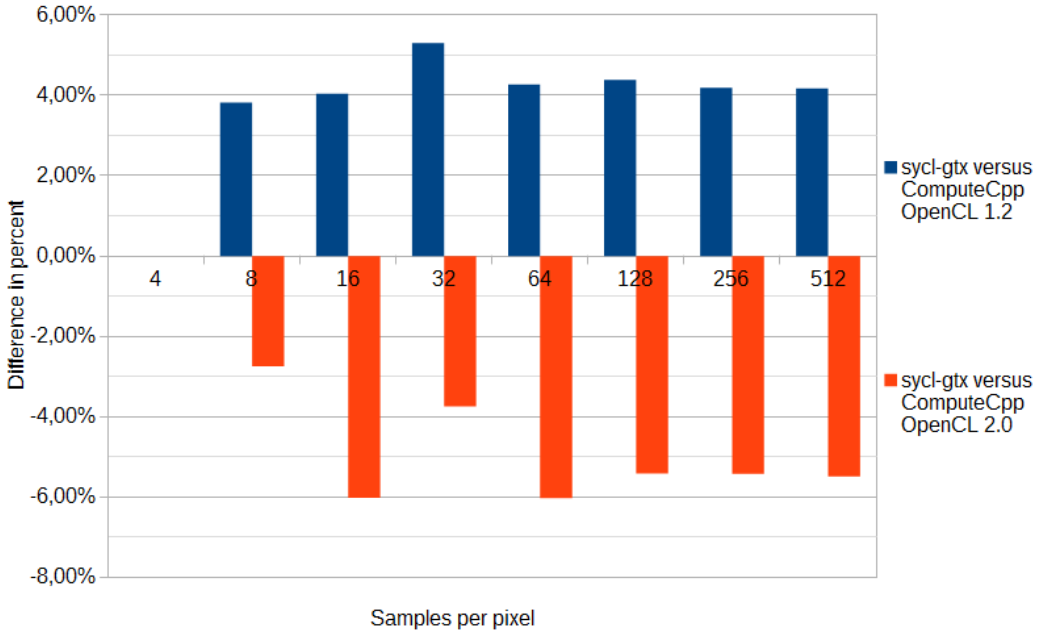

Figure 5.4: Comparison of sycl-gtx and adjusted ComputeCpp results.

reason for this discrepancy. While we did modify the code from the Codeplay blog to fit the tester and use as much common code as possible, there is no known reason for this almost-exactly-factor-of-two difference. After all, our implementation of SYCL is definitely less mature. We presume we've made a mistake somewhere, though careful examination of the code did not reveal it. We compared the image outputs, but they were comparable at the same sample rate.

The same problem may be present in the host fallback version as well – returning to Figure 5.3, the host fallback result at 64 samples per pixel is very close to OpenMP at 128 samples per pixel. Due to long execution times we did not provide more data points, but the graph trend certainly suggests similar behavior to what was observed when comparing sycl-gtx and ComputeCpp.

The last chart is probably the most interesting. Figure 5.5 shows the speedup factor of different results compared to the original (single precision)

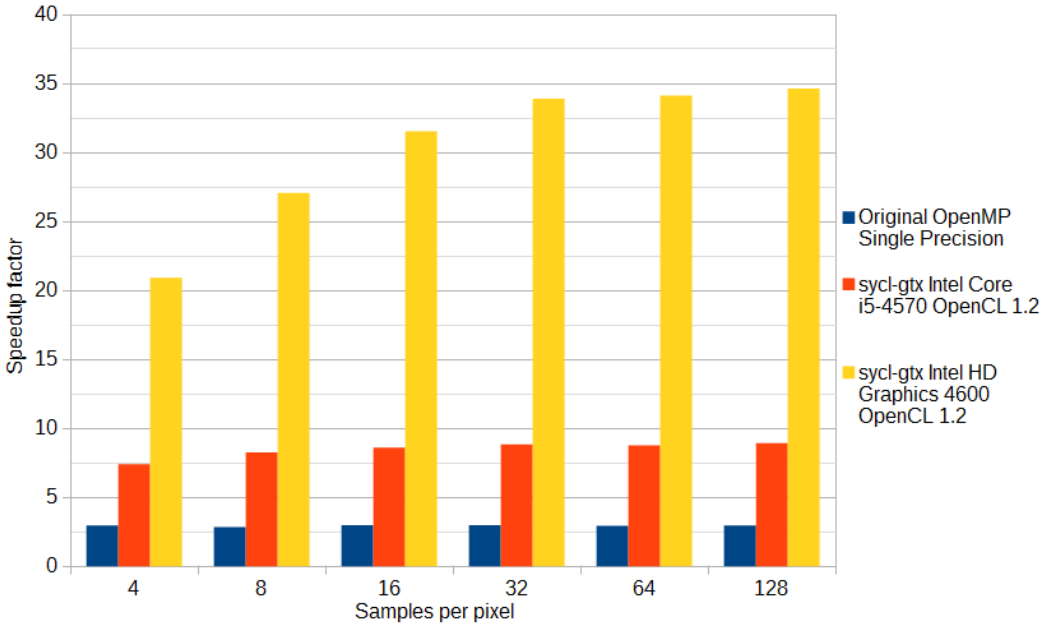

Figure 5.5: Speedup over original smallpt (using single precision).

code. Using OpenMP provides a constant speedup factor of 3, but using sycl-gtx on the same four CPU cores boosts the results by almost another factor of 3, though the advantage is somewhat lower when using lower sample rates. Amazingly, this result can be much improved when switching to the GPU. Even though the Intel HD Graphics 4600 has pretty low performance among modern GPUs, it can still best the four core CPU by another factor of 4, resulting in the final results be almost 35 times faster than the original code. It suffers even more from the initial setup than the CPU version, though, which can be partly attributed to the low memory bandwidth when copying data – when the sample rate increases, the copy times aren't as important, leading to best performance.

If we take the sycl-gtx GPU results at 512 samples per pixel and do a fast calculation: we have 1024 times 768 pixel, 512 samples per pixel, take antialiasing into account (computing 4 samples for each requested one and averaging the results), and consider the total running time of 93.17 seconds, we get $\frac{1024*768*512*4}{93.17} = 17286816.96$ samples per second, or about 864340.85

samples per EU per second. A similar calculation for the sycl-gtx CPU version (OpenCL 1.2) returns 1299342.30 samples per core per second, which is clearly higher per computing element, but not by what the higher IPC and much higher clock frequency would suggest. This is where heterogeneous computing shows its strengths – the GPU is just much more suited for these kinds of tasks due to a much stronger architectural focus on data instead of on instructions.

5.4.2 Older laptop with discrete graphics

Something very interesting occurred on additional testing. The tester was designed to check the platform version is at least OpenCL 1.2 compatible and add to the test all devices, belonging to the platform. However, when running on a laptop with an Intel Core i5-2520M and an Nvidia Quadro 2000M discrete GPU (Windows 10 OS), the Nvidia platform presented itself as "OpenCL 1.2 CUDA 7.5.15", but the actual device version was "OpenCL 1.1 CUDA", meaning it wasn't compatible with the test. On the first run the whole computer froze, and on the second one a Blue Screen of Death was encountered. The results could still be obtained, although much fewer ones, and they are listed in Table 5.2. Additionally, the time limit was much stricter here.

	4	8	16	32	64	128	256
Original	31.61	63.63					
Original OpenMP	14.00	28.06	55.46				
Quadro 2000M	1.90	3.09	5.95	11.31	23.02	45.77	95.16

Table 5.2: Results of the smallpt tester running on a laptop with an Intel Core i5-2520M CPU and an Nvidia Quadro 2000M GPU. All times are in seconds. The columns represent test runs when using different numbers of samples per pixel.

Since the Intel Core i5-2520M is an older CPU, with only two cores

(but with HyperThreading), and a power-restricted mobile one as well, the single core performance is almost 50% slower, while the multi core results (OpenMP) took almost twice longer. But more telling are the GPU results: besides the fact that integrated GPUs have made some interesting progress with regard to performance (the HD 4600 is about twice faster than Quadro 2000M), the heterogeneous ecosystem needs to be properly maintained.

5.4.3 Summary

Since the main focus was to get the implementation working, we didn't run more performance tests. But being able to implement a ray tracer in sycl-gtx with comparable performance to what the leading, proprietary ComputeCpp implementation provides (presuming we've made an error somewhere and using the adjusted results) speaks favorably to the development status of the implementation. There is always room for improvement – even though our ported smallpt compiles with ComputeCpp, it cannot be run there. But sycl-gtx continues to evolve and will hopefully someday be able not only to properly support more of the SYCL specification but also include additional general and device-specific performance optimizations.

Chapter 6

Conclusion

We managed to provide an implementation of SYCL, even though it is not complete. We even managed to write an article about the implementation and got it published on the PPoPP 2016 conference [53]. Some parts of the implementation are missing because there was no time to implement everything, while others are missing because of a fundamental design decision. We decided not to modify any existing compilers – all code in our implementation should be compatible with any compiler that supports C++11 and OpenCL 1.2. The way we implemented SYCL was by writing a code generator which would capture information about the compiled C++ code and use that information to generate OpenCL C, which was then fed to the OpenCL C compiler – basically Just-In-Time compilation of SYCL code. This, however, prevented us from implementing even some basic features – control flow, for example (`if`, `for`, ...), could not be captured this way. We provided workarounds, but ultimately it does not quite follow the SYCL specification.

Nevertheless, many simple SYCL programs, completely conformant to the SYCL specification, were able to compile and run using our implementation. We provided a compatibility header, the inclusion of which ensures that code written for our implementation of SYCL also works on any other implementation. That way, we can still observe what SYCL promises: simple, modern C++ code that can easily be parallelized and executed in heteroge-

neous systems, providing almost the same level of performance as OpenCL with much less programming effort.

Tests were mostly written to prove the correctness of the implementation and to observe how the code generator works, although some testing also focused on performance.

Of course, heterogeneous computing is a large area of study, so we also went over various aspects: how different computing units work, what are their strengths and weaknesses, how heterogeneous programming is approached today.

There is a lot of potential for future work. First, our implementation is far from fully implemented. While we have proven it is possible to implement a ray tracer using sycl-gtx, a lot of the more advanced features are still missing. Second, more performance evaluations are needed. We've demonstrated that there's likely some overhead with small kernels – it would be interesting to compare sycl-gtx performance with "pure" OpenCL and to perform more comparisons with ComputeCpp. Third, in order for the implementation to be useful, it should actually be used, meaning applications should be ported to sycl-gtx. This includes both existing OpenCL applications, applications where OpenCL may have proven not to be worth the implementation effort, and possibly any other applications where parallelization hasn't even been considered yet. We provided a compatibility header that makes any valid sycl-gtx application also valid in other SYCL implementations.

The main implementation of SYCL, ComputeCpp, provides the best insight into SYCL, because it's developed by the same people that also contribute to the SYCL specification. But ComputeCpp is proprietary, while our solution is open-sourced. We hope that despite the flaws in our solution, the open source community will embrace it by writing SYCL code and maybe even develop the implementation further. As we've seen, the appeal of heterogeneous computing is spreading, and SYCL is an important step on the road to the future of computing – and sycl-gtx is a part of that.

Bibliography

[1] J. L. Hennessy, D. A. Patterson, Memory Hierarchy Design - Part 1. Basics of Memory Hierarchies), 2012.
URL http://www.edn.com/design/systems-design/4397051/1

[2] P. Lilly, A Brief History of CPUs: 31 Awesome Years of x86, 2009.
URL http://www.maximumpc.com/a-brief-history-of-cpus-31-awesome-years-of-x86

[3] I. Cutress, The Intel Skylake Mobile and Desktop Launch, with Architecture Analysis, 2015.
URL http://www.anandtech.com/show/9582/

[4] Intel Corporation, Intel® Pentium® 4 Processor supporting HT Technology 3.40 GHz, 2016.
URL http://ark.intel.com/products/27504/

[5] I. Cutress, Comparing IPC on Skylake: Memory Latency and CPU Benchmarks, in: The Intel 6th Gen Skylake Review: Core i7-6700K and i5-6600K Tested, 2015.
URL http://www.anandtech.com/show/9483/

[6] T. S. Crow, Evolution of the Graphical Processing Unit, 2004. doi:10.1.1.142.368.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.368

[7] C. McClanahan, History and Evolution of GPU Architecture, 2010.
URL http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf

[8] Khronos OpenCL Working Group, The open standard for parallel programming of heterogeneous systems, 2016.
URL https://www.khronos.org/opencl/

[9] The OpenMP® API specification for parallel programming, 2016.
URL http://openmp.org/wp/

[10] NVIDIA, CUDA C PROGRAMMING GUIDE, 2015.
URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[11] Khronos OpenCL Working Group, SYCL™ Specification, 2015.
URL https://www.khronos.org/registry/sycl/specs/sycl-1.2.pdf

[12] Codeplay Software, ComputeCpp, 2016.
URL https://www.codeplay.com/products/computecpp

[13] triSYCL, 2016.
URL https://github.com/amd/triSYCL

[14] sycl-gtx, 2016.
URL https://github.com/ProGTX/sycl-gtx

[15] A. Lal Shimpi, Intel's Haswell Architecture Analyzed: Building a New PC and a New Intel, 2012.
URL http://www.anandtech.com/show/6355/

[16] C. Märtin, Post-Dennard Scaling and the final Years of Moore's Law, 2014.
URL https://www.hs-augsburg.de/medium/download/fki/person/maertin_christian/PostDennard.pdf

[17] x64 Architecture, 2015.
URL https://msdn.microsoft.com/en-us/library/windows/hardware/ff561499(v=vs.85).aspx

[18] x86 Registers, 2015.
URL http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html

[19] D. M. Tullsen, S. J. Eggers, H. M. Levy, Simultaneous Multithreading: Maximizing On-Chip Parallelism, in: Proceedings of the 22nd Annual International Symposium on Computer Architecture, 1995, pp. 392–403.

[20] D. Butnariu, S. Reich, Y. Censor, Inherently Parallel Algorithms in Feasibility and Optimization and their Applications, Studies in Computational Mathematics, Elsevier Science, 2001.
URL https://books.google.si/books?id=GyOEo12F970C

[21] R. Smith, The NVIDIA Geforce GTX 980 Review: Maxwell Mark 2, 2014.
URL http://www.anandtech.com/show/8526/

[22] NVIDIA, TUNING CUDA APPLICATIONS FOR MAXWELL, 2015.
URL http://docs.nvidia.com/cuda/maxwell-tuning-guide/

[23] S. A. Dyer, B. K. Harms, Digital signal processing, no. 37 in Advances in Computers, Elsevier Science, 1993.
URL https://books.google.com.sg/books?id=vL-bB7GALAwC

[24] I. Kuon, R. Tessier, J. Rose, FPGA Architecture: Survey and Challenges, Vol. 2 of Foundations and Trends in Electronic Design Automation, 2008. doi:10.1561/1000000005.
URL http://www.doc.ic.ac.uk/~wl/papers/08/kuon08survey.pdf

[25] R. Marrit, Intel to make 14-nm FPGAs for Altera, 2013.
URL http://eetimes.com/document.asp?doc_id=1263080

[26] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, D. Burger, A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services, in: 41st Annual International Symposium on Computer Architecture (ISCA), 2014.
URL http://research.microsoft.com/apps/pubs/default.aspx?id=212001

[27] S. Anthony, Intel unveils new Xeon chip with integrated FPGA, touts 20x performance boost, 2014.
URL http://www.extremetech.com/extreme/184828-intel-unveils-new-xeon-chip-with-integrated-fpga-touts-20x-performance-boost

[28] R. Smith, Intel Kills Larrabee GPU, Will Not Bring a Discrete Graphics Product to Market, 2010.
URL http://www.anandtech.com/show/3738

[29] G. Chrysos, Intel® Xeon Phi™ X100 Family Coprocessor - the Architecture, 2012.
URL https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

[30] TOP500 Lists November 2015, 2015.
URL http://www.top500.org/lists/2015/11/

[31] R. Rahman, Intel® Xeon Phi™ Core Micro-architecture, 2013.
URL https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture

[32] C. M. Kozierok, Intel Pentium ("P5" / "P54C"), 2001.
URL http://www.pcguide.com/ref/cpu/fam/g5P54-c.html

[33] R. Smith, Intel's "Knights Landing" Xeon Phi Coprocessor Detailed, 2014.
URL `http://www.anandtech.com/show/8217`

[34] R. Smith, PCI Express 3.0: More Bandwidth For Compute, in: AMD Radeon HD 7970 Review: 28nm And Graphics Core Next, Together As One, 2011.
URL `http://www.anandtech.com/show/5261`

[35] A. Lal Shimpi, The GPU: Intel HD 5000 (Haswell GT3), in: The 2013 MacBook Air Review (13-inch), 2013.
URL `http://www.anandtech.com/show/7085`

[36] A. Lal Shimpi, Addressing the Memory Bandwidth Problem, in: Intel Iris Pro 5200 Graphics Review: Core i7-4950HQ Tested, 2013.
URL `http://www.anandtech.com/show/6993`

[37] I. Cutress, Comparing DDR3 to DDR4, in: DDR4 Haswell-E Scaling Review: 2133 to 3200 with G.Skill, Corsair, ADATA and Crucial, 2015.
URL `http://www.anandtech.com/show/8959`

[38] R. Smith, High Bandwidth Memory: Wide & Slow Makes It Fast, in: The AMD Radeon R9 Fury X Review: Aiming For the Top, 2015.
URL `http://www.anandtech.com/show/9390`

[39] I. Cutress, R. Garg, A Deep Dive on HSA, in: AMD Kaveri Review: A8-7600 and A10-7850K Tested, 2014.
URL `http://www.anandtech.com/show/7677`

[40] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, J.-D. Choi, An OpenCL Framework for Heterogeneous Multicores with Local Memory, in: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, ACM, New York, NY,

USA, 2010, pp. 193–204. doi:10.1145/1854273.1854301.
URL http://doi.acm.org/10.1145/1854273.1854301

[41] R. Garg, A Look at Altera's OpenCL SDK for FPGAs, 2013.
URL http://www.anandtech.com/show/7334/

[42] SDAccel Development Environment, 2016.
URL http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[43] B. Crothers, OpenCL goes beyond Apple, 2008.
URL http://www.cnet.com/news/opencl-goes-beyond-apple/

[44] NVIDIA, NVIDIA Delivers Comprehensive OpenCL Support under Snow Leopard, 2009.
URL http://www.nvidia.com/object/io_1252000156987.html

[45] T. Valich, AMD Ditches Close-To-Metal, Focuses On DX11 And OpenCL, 2008.
URL http://www.tomshardware.com/news/AMD-stream-processor-GPGPU,6072.html

[46] A. Lal Shimpi, D. Wilson, in: NVIDIA's GeForce 8800 (G80): GPUs Re-architected for DirectX 10.

[47] AMD, OpenCL™ and the AMD APP SDK v2.4, 2011.
URL http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-and-the-amd-app-sdk-v2-4/

[48] F. Jianbin, A. L. Varbanescu, H. Sips, A Comprehensive Performance Comparison of CUDA and OpenCL.

[49] G. Brown, SYCL 1.2 Provisional Specification Announced, 2014.
URL https://www.codeplay.com/portal/sycl-12-provisional-specification-announced

[50] R. Reyes, SYCL Tutorial 1: The Vector Addition, 2014.
URL `http://www.codeplay.com/portal/sycl-tutorial-1-the-vector-addition`

[51] B. Catanzaro, OpenCL™ Optimization Case Study: Simple Reductions, 2010.
URL `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/`

[52] Khronos OpenCL Working Group, The OpenCL Specification, 2012.
URL `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`

[53] P. Žužek, An Overview of sycl-gtx, 2016.
URL `http://conf.researchr.org/event/PPoPP-2016/sycl-2016-papers-an-overview-of-sycl-gtx`

[54] L. Iwanski, SYCL-ing the 'smallpt' Raytracer, 2015.
URL `https://www.codeplay.com/portal/sycl-ing-the-smallpt-raytracer`

[55] K. Beason, smallpt: Global Illumination in 99 lines of C++, 2014.
URL `http://www.kevinbeason.com/smallpt/`

[56] G. Marsaglia, Xorshift RNGs, Journal of Statistical Software 8 (1) (2003) 1–6.