Research and Development of Porting SYCL on QNX Operating System for High Parallelism

by

Dengpan Wang

A thesis submitted to the
Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Information Technology: Network Technology

School of Information Technology
Carleton University
Ottawa, Ontario
August, 2021

Abstract

As a standard C++ programming model, SYCL has gained popularity on incorporating various parallel computing frameworks. With the development of hardware technologies, low-level computing devices are becoming increasingly varied and thus result in the great heterogeneity of hardware. Although many computing frameworks, such as OpenCL, OpenMP and CUDA, can benefit to heterogeneous computing, they increase the complexity of cross-platform deployment and reduce productivity due to low portability and miscellaneous features. By comparison, SYCL allows programmers to write high-performance parallel applications in the standard C++ syntax and execute them across vendor-specific hardware without diving into low-level technologies. However, despite the popularity of SYCL on Windows and Linux, there is little research on porting SYCL to QNX, a real-time operating system (RTOS). Therefore, we choose two SYCL implementations and conduct corresponding experiments. In particular, we build a new path of calling OpenCL APIs in SYCL-GTX and significantly reduce the time of compiling SYCL kernels. Although the overall performance of SYCL-GTX on QNX is evaluated on Linux, our experiments demonstrate that there are many possible optimizations that can improve SYCL-GTX on QNX.

Acknowledgments

First and foremost, with my most sincere gratitude, I would like to acknowledge my supervisor Prof. F. Richard Yu for his dedication, patience, encouragement, and support. Without his guidance, knowledge and expertise, this dissertation would have not been possible. Besides, I also want to appreciate Blackberry. Without the technical support of Blackberry, I could not set up the environment of our experiments well.

I would also like to thank my uncle for his encouragement and support for research interests and educational pursuits. Besides, I would also like to appreciate my parents and cousin for support in my daily life.

Table of Contents

List of Tables

List of Figures

Nomenclature

Abbreviation	Description
AOT	Ahead of Time
API	Application Programing Interface
ASIC	Aplication-Specific Intergrated Circuits
CTA	Cooperative Thread Array
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DPC	Data Parallel Computing
DSP	Digital Signal Processors
FPGA	Field Programmable Gate Arrays
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
HDL	Hardware Description Language
ICD	Installable Client Driver
IR	Intermediate Representation
ISA	Instruction Set Architecture
JIT	Just-in-Time
LLVM	Low-Level Virtual Machine

MPI	Message Passing Interface
PCIe	Peripheral Component Interconnect express
PTX	Parallel Thread Execution
RAII	Resource Acquisition Is Initialization
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SMCP	Single-source Multiple Compiler-Passes
SPIR	Standard Portable Intermediate Representation
SPIR-V	Standard Portable Intermediate Representation - V
SSA	Static Single Assignment
SVM	Shared VIrtual Machine
TBB	Thread Building Blocks
TPU	Tensor Processing Unit

Chapter 1

Background

1.1 Heterogeneous Computing

1.1.1 Hardware Heterogeneity

With the rise in general-purpose computing and the improvement of hardware technologies, it is pervasive to build a heterogeneous computing system to meet various computing requirements. General PCs (Personal Computer) and laptops, usually making use of a multi-core CPU (Central Processing Unit), an embedded GPU (Graphic Processing Unit), and a discrete GPU, are the best examples of heterogeneous computing systems. The computing devices in such a heterogeneous system may use different instruction sets and memory layouts, resulting in increasing heterogeneity in hardware.

Traditional CPUs focus on executing instructions sequentially as fast as possible by increasing CPU clock frequency or the number of instructions executed in a single clock cycle. As general-purpose processors, CPUs have been competent for most computation tasks until the explosion of machine learning, but multi-core and simultaneous multi-threading technologies do not make CPUs perform well on such tasks requiring high data parallelism [2].

By comparison, GPUs are more effective than CPUs when computation-intensive applications are executed [3]. Furthermore, benefiting from SIMD (Single Instruction

Multiple Data) architectures, GPUs are more friendly to machine learning applications that usually compute massive data in parallel, although they suffer from the extra overhead of sending the data back and forth between GPU's and CPU's memory [4]. However, only discrete GPUs require data transfers via PCIe, while integrated GPUs can share data between CPU and GPU without copying through the external data bus. Some graphics cards, mainly NVIDIA GPUS, particularly embed tensor cores and specific function units (SFUs) to accelerate operations used in artificial intelligence applications, especially deep learning.

In addition, there are also many other accelerators available with various strengths to fulfil the acceleration of computation-intensive tasks. For example, FPGA (Field Programmable Gate Array), a hardware circuit, has the performance of the hardware solutions and can be programmed using a hardware description language (HDL). Besides their flexibility and performance, FPGAs can scale by connecting serval FPGA boards together, fastly executing frequent instructions and efficiently handling a vast amount of streaming data. Another example is DSP (Digital Signal Processor). As with any specialized circuits, DSPs are more power-efficient than general-purpose processors for these types of applications. It is a must in many portable devices to process analog signals, such as audio and video. They can also be used in high-performance computing for scientific applications. It is also worth noting that Google has designed TPU (Tensor Processing Unit) chips to accelerate AI (Artificial Intelligence) applications, specifically accelerating neural networks used in machine learning [5]. A TPU can do matrix multiplication more efficiently than a GPU by quantizing and transforming floating points to 8-bit integers.

As we can see, to fulfil the requirements of heavy computation in the AI era, there are a variety of accelerators used to enhance data parallelism. As a result, it is becoming a critical task for different operating systems to effectively manage and incorporate heterogeneous compute resources.

1.1.2 Parallel Framework Heterogeneity

Apart from the heterogeneity of the hardware solutions, a myriad of computing platforms has been proposed to enhance parallel computing or manage heterogeneous

hardware resources on an operating system. Therefore, programmers can use those computing models to attain high parallelism in computer-intensive applications.

- MPI. Message Passing Interface is a message-passing application interface, together with protocol and semantic specification for how its features behave in any implementation [6]. It has become a de facto programming model for distributed memory architectures. Any MPI implementation should support creating processes to execute the same code on different chunks of the data, thereby attaining high data parallelism. Also, although MPI does not have a memory model, it can be used with shared memory models like OpenMP or accelerator languages like CUDA.

- OpenMP. It is designed for shared memory architectures, such as CPUs and assigns different tasks to threads in parallel. However, OpenMP requires programmers to take care of checking data dependencies and thread sharing, making it error-prone. It is worth noting that OpenMP has enabled offload tasks to accelerators since the version of 4.0 [7], making it less different from OpenCL.

- OpenCL. OpenCL is designed to abstract underlying parallelism and thus maximize the portability of programs by unifying all computation resources. As a result, OpenCL programs can accelerate code (called kernels) in parallel on cross-vendor platforms, such as CPUs, GPUs, and FPGAs, and hide the complexities of the low-level parallelism, thereby easing the efforts of memory control and task scheduling.

- OpenACC. It is designed to simplify parallel computing of heterogeneous systems. The parallelized part in OpenACC is in the way of OpenMP and in a philosophy similar to OpenCL. In contrast, it has no support of other types of accelerators.

- CUDA. It is a GPU programming language, exploiting GPUs to solve computational problems efficiently. Following the same philosophy as OpenCL, a CUDA kernel can be offloaded from the host to the device (i.e., GPUs). By grouping processing units into a hierarchy structure, the CUDA programming model provides high data parallelism. Unfortunately, it is exclusive to NVIDIA hardware and not compatible with other vendors' devices.

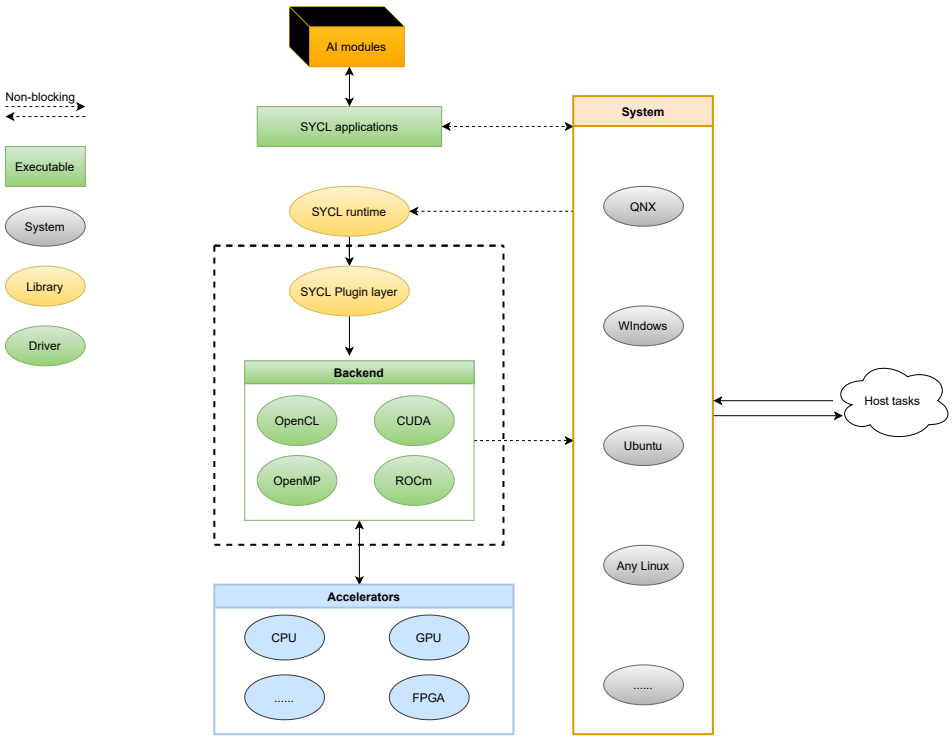


Figure 1.1: The future workflow of AI modules on multiple systems.

The above discussion is not an exhaustive list of parallel programming for heterogeneous systems. There are many other computing models/standards that are built with multi-core processors and not accelerators in general. For example, threading building blocks (TBB), developed by Intel as a C++ template library, enables programmers to break the computations and group them as parallelized tasks with dependencies. Besides, PThreads is also another essential model. It is a language-independent standard and can be implemented in any language to support multi-threading parallel execution. However, although programmers can have fine-grained control of threads in Pthreads, they are at the expense of more coding, thus reducing their productivity.

1.2 Motivation and Problem Statement

Since 2014, SYCL has been proposed as a single-source C++ programming model that can incorporate mainstream computing frameworks, providing more user-friendly programmability with unified programming interfaces. Both Linux and Windows have supported mainstream SYCL implementations to address the heterogeneity of hardware and computing frameworks. However, the development of autonomous driving requires heterogeneous hardware to accelerate autonomous driving [8] [9] [10]. Therefore, QNX needs beyond a traditional real-time operating system in terms of functions and undertake many other tasks to address the increase of computations and computing heterogeneity. Mainly, it requires incorporating different computing frameworks targeting various accelerators to exploit cross-vendor processors to handle various computing tasks.

By porting SYCL to QNX, we may have some benefits when dealing with heavy computations on QNX:

- User-friendly programability. Different frameworks have a quite different abstraction of low-level parallelism and thus make it complicated to port them to QNX one by one. Moreover, it may be exhausting to expose different computing frameworks to developers and thus not benefit the autonomous driving ecosystem on QNX, as developers must dedicate great efforts to master different programming languages to develop secure, reliable QNX applications with high parallelism. Consequently, it requires significant efforts to implement those frameworks on QNX and complicates programmability. From this perspective, SYCL, a unified computing framework designed to incorporate those computing models and improve productivity, is an excellent choice to integrate the features of miscellaneous computing models and empower QNX to perform well in heavy computations. In particular, [11] and [12] have proposed integrating SYCL as a standard C++ library class on parallelism and heterogeneous computing. Therefore, it should be another reason for QNX to support SYCL standards if SYCL could be integrated into C++ standards.

- Portability. As QNX is a commercial operating system for its great security and real-time, there are not abundant open-source projects on QNX compared

to Linux and Windows. Nevetheless, SYCL could enable developers to migrate their Linux or Windows applications to QNX easier, instead of rewriting, re-optimzing, re-testing their SYCL applications for QNX. As a result, developers could reuse their source code and experience on QNX and improve their productivity, thereby thriving the QNX developement community.

Moreover, if SYCL would be workable on QNX, as one of the most popular RTOSs on cars, QNX will have the ability to incorporate heterogeneous hardware and do heavy computation directly. As a result, the future workflow of computing on autonomous driving may be shown in Figure 1.1. Since the computations happen on discrete accelerators, the core functions of QNX will not be affected a lot. Therefore, QNX can keep its security and real-time while handling intensive data, thereby maintaining a competitive edge in autonomous driving.

Besides, benefiting from the pluggable backend philosophy in SYCL, QNX could introduce new computing backends selectively without refactoring the high-level programming interfaces. More importantly, although many researchers [13] [14] focus on edge computing for autonomous driving to deliver enough computing power, it is still essential to handle compute-intensive tasks in QNX environments to ease related experiments. By comparison, the simplicity and programmability of SYCL could allow programmers to accelerate their QNX applications without extra mental burdens, attracting more programmers to develop innovative car applications for QNX and thus thriving the autonomous driving community of QNX.

Due to the disparate landscape of parallel and heterogeneous systems, we need to demonstrate if SYCL could indeed empower QNX. Hence, in this thesis, the research questions are summarized as follows:

- **RQ1**: Many existing SYCL implementations are mainly written for Windows and Linux. Which one of the existing SYCL implementations is compatible with QNX?
- **RQ2**: Mainstream SYCL implementations may cover different features of the previous and latest SYCL specifications, so what limitations do they have when migrating into QNX environments.

- **RQ3**: If existing SYCL implementations could work in a QNX environment, how does it perform well compared to Linux environments? If SYCL on QNX would underperform on Linux, what do possible reasons result in the worse performance? And what could potential solutions improve its performance on QNX?

1.3 Major Contribution

It is not true that SYCL implementations could be migrated into QNX without engineer refactoring, so we choose the possible SYCL implementation, SYCL-GTX, that can work on QNX and then avoid rewriting most of SYCL toolchains. Besides, after refactoring and building the SYCL runtime, we test floating-point multiplication in vector and matrix to illustrate and compare the performance of SYCL and OpenCL in both QNX and Ubuntu. Our major contribution can be summarized as follows:

- By following RQ1 and RQ2, we compare existing SYCL implementations and try to refactor them. After refactoring and building SYCL-GTX for QNX, we demonstrate the feasibility of SYCL on QNX.
- We propose our research methods and schemes according to RQ2 and RQ3. In the experiments of floating-point multiplication in vector and matrix, by analyzing the performance difference of SYCL on QNX and Ubuntu compared to pure OpenCL, we point out the possible problems and potential improvements of SYCL on QNX. Meanwhile, we provide a new way to compile SYCL kernels on the basis of SYCL-GTX and turn out the significant improvement of compiling SYCL kernels on QNX.
- by referring to our experiment results and RQ3, we compare the compilation mechanisms of different SYCL implementations and indicate a possible solution for QNX to build a mature, product-ready SYCL environment.

Chapter 2

Related Work

In this chapter, we introduce some techniques used by popular parallel computing frameworks. We focus on explaining the architecture of LLVM and comparing it with traditional GCC architecture to give the reason why many computing frameworks tend to embrace LLVM in cross-compilation environments and heterogeneous systems. Besides, as the two primary intermediate represewntations, SPIRV and PTX are also introduced to show how they can convert to native instruction sets and how hardware drivers consume them. Besides, there are some research that demonstrate the potential applications of SYCL and the performance difference between SYCL, OpenCL and CUDA.

2.1 Introduction to requisite techniques of SYCL

2.1.1 Low Level Virtual Machine

LLVM (Low-Level Virtual Machine) began as a research project at the University of Illinois, intending to provide a modern, SSA (Static Single Assignment [15]) -based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages [16]. Since then, LLVM has grown to be an umbrella project that hosts and develops a set of close-knit low-level toolchain components. A

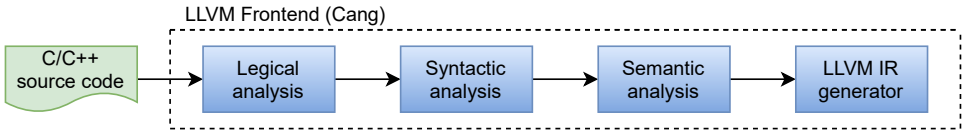


Figure 2.1: A typical compiler pipeline.

number of commercial and open-source projects[1] and academic research[2] use widely LLVM.

2.1.2 The Architecture of LLVM

LLVM was designed as reusable libraries with well-defined interfaces working as plug-ins in LLVM-based projects. Apart from the modularized libraries, the essential parts of the LLVM infrastructure include frontends, IR and backends.

LLVM Frontend

The frontend translates source code written in arbitrary programming languages into the LLVM IR. Any LLVM frontend should conform to a typical compiler pipeline shown in Figure 2.1 (taking the Clang compiler as an example). The modular design of LLVM also allows developers to write a custom LLVM frontend for a particular language through LLVM official libraries. Therefore, many projects are using their customed LLVM frontends, such as LLILC [17] and rustc [18].

LLVM IR

The LLVM IR plays the middle point between the frontends and the backends and does not depend on any particular source language or specific target architecture. Therefore, there is no high-level language feature or machine-dependent feature in LLVM IR [19]. It uses an abstract RISC (Reduced Instruction Set Computer) -like

[1]The list of open-source projects based on LLVM. See https://llvm.org/Users.html
[2]The list of academic projects on LLVM. See https://llvm.org/pubs/

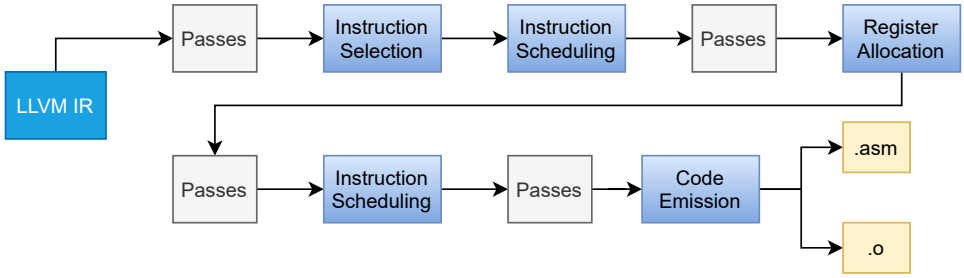


Figure 2.2: The sequential passes of LLVM backends.

instruction set [20] to represent arbitrary programs while keeping high-level information of programs to support sophisticated optimizations and transformations.

To avoid machine-specific constraints, LLVM provides the typed virtual registers in the form of SSA [15] and does not limit the number of virtual registers. Code in LLVM IR is organized in three address code [21] [22] that is very close in spirit to machine code and easily compressed for high-density LLVM outputs. Besides, there are two representations of the LLVM IR – assembly (`.ll`) and binary-encoded (`.bc`) representations. Developers can use a wide range of open-source tools to convert them each other or into other language representations, such as `emcc` [23] for WebAssembly [24] and `llvm-spirv` [25] for SPIR-V.

LLVM Backend

The backends consume LLVM IR and generate assembly code or object code binaries through sequential passes (shown in Figure 2.2) for multiple targets. Hence, LLVM can work on multiple host platforms[3]. At the phase of instruction selection, the code generator converts IR into the target-machine nodes, a data class representing machine instructions. The set of virtual registers in LLVM IR will be transformed into target-specific registers at the point of register allocation. Finally, the backend pipeline will finalize the target-machine nodes and target-specific register allocations and output native assembly code or object code.

Such a pipeline allows developers to write retargetable LLVM code generator for

[3]The list of supported target architectures by LLVM. See https://llvm.org/docs/GettingStarted.html#hardware.

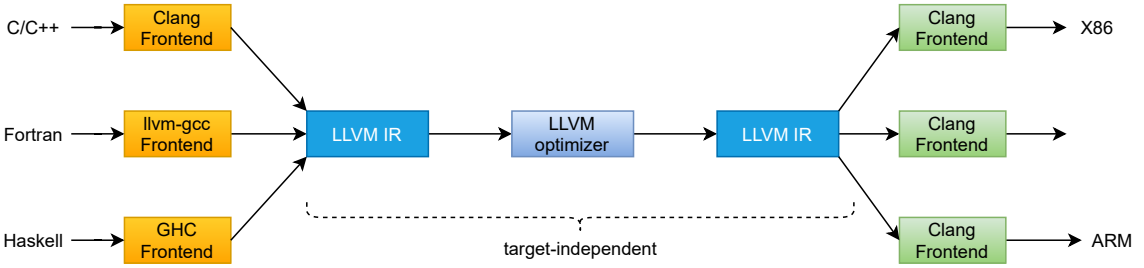


Figure 2.3: The three phase design (based on three address code) of LLVM.

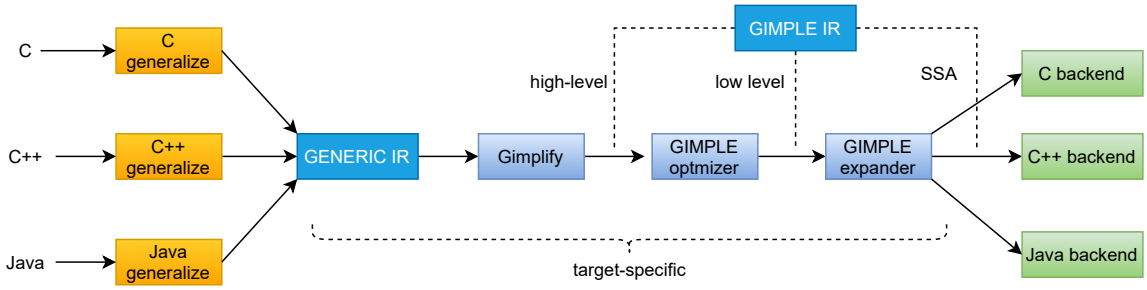


Figure 2.4: The three phase design (based on tree structure) of GCC.

their new targets through modifying parts of those passes without rewritting an entire code generator from scratch [26].

The Comparison of GCC and LLVM

Like GCC[4], the LLVM infrastructure also implements the three-phase design (illustrated in Figure 2.3) to support multiple source languages and target architectures. However, since LLVM IR is self-contained[5], LLVM gives a more explicit division of frontends and backends in terms of functions and thus ease the efforts to port different source languages and targets. The LLVM frontends and backends only need to operate on the standalone LLVM IR, whereas GCC only supports distinct variants of tree-structured representations in the three-phase design (illustrated in Figure 2.4) since GCC 4.0 [27].

[4]GNU Compiler Collection. The GCC compiler is one of GCC frontends. See wikipedia https://en.wikipedia.org/wiki/GNU_Compiler_Collection.

[5]LLVM IR is language- and target-independent, and supports to covnert each other between its two representations.

GCC frontends output GENERIC IR [28] converted into the three kinds of GIMPLE IR[6] by Gimplify, GIMPLE optimizer and GIMPLE expander (See Figure 2.4). Finally, GCC backends will hand SSA GIMPLE IR [29] over different backends to output native assembly code. Although GCC also modularizes the conversion of representations and enables the intermediate representations to work independently from source languages and target architectures, the intermediate outputs of the entire process are invisible to developers and cannot convert conversely. Consequently, developers can not reuse the intermediate outputs and need to compile their programs from source code and walk through the sequential process strictly.

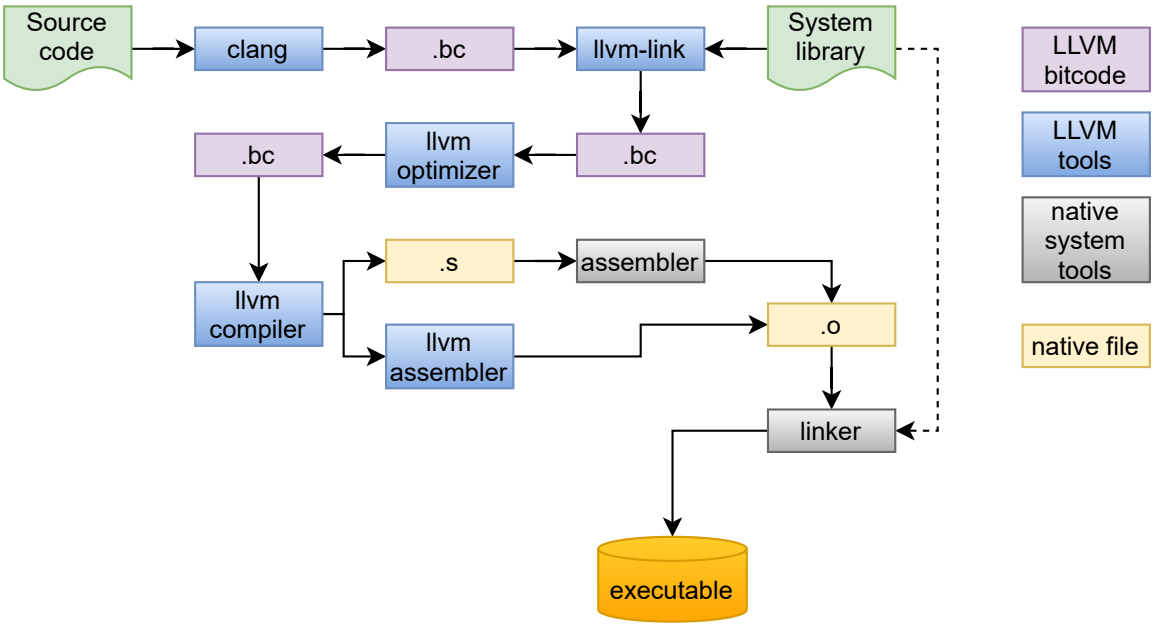
By comparison, the three phases of the LLVM infrastructure only need to interface with the LLVM IR and break passes of each phase into modularized tools, thereby enabling developers to customize their programs by modifying those tools. Remarkably, LLVM IR can be (de)serialized efficiently to/from LLVM bitcode using tools `llvm-as`/`llvm-dis`[7] since the mapping of LLVM assembly [20] and bitcode [30] representations is strict and lossless [26]. For example, in Figure 2.5 (a), the LLVM compiler frontends can emit LLVM bitcode/assembly representations before generating native object code. Other LLVM tools can re-link, re-optimize or retarget the LLVM IR to other supported architectures. In contrast, the gcc compiler hides the outputs of intermediate representations and only emits native assembly code and object code in Figure 2.5 (b).

To summarize, LLVM makes everything is pluggable and customizable and can port to different target architectures and languages conveniently, whereas the entire process of GCC compilation is opaque, exclusive and not allow developers to modify in demand.

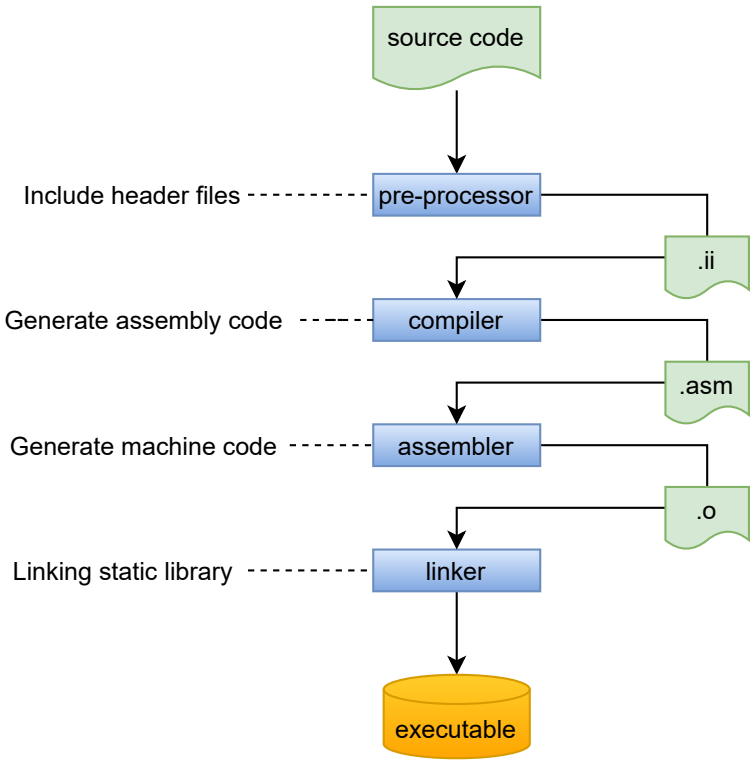
2.1.3 Standard Portable Intermediate Representation

The previous SPIR-V version is SPIR [31] based on LLVM techniques to construct and distribute platform-dependent binaries within the OpenCL stack. SPIRV was

[6]The classification of GIMPLE IR. See https://gcc.gnu.org/wiki/GIMPLE.
[7]The description of the two LLVM tools. See https://llvm.org/docs/CommandGuide/.



(a) The modularized compilation of LLVM.



(b) The monolithic compilation of GCC.

Figure 2.5: The compilation of GCC and LLVM.

announced in March 2015 and fully defined by Khronos Groups without the dependency on LLVM. (still able to convert each other by the tool `llvm-spirv` [25] since both use the SSA [15] form.) It is designed as a cross-vendor intermediate language in the form of self-contained binaries for graphics and parallel computation.

The Mapping of Client Drivers in SPIRV

SPIR-V provides a series of instructions to express a client driver, such as Vulkan [32], OpenCL and OpenGL [33]. The high-level abstraction of low-level architectures enables SPIR-V to port to various platforms and then launch on heterogeneous devices.

- Programming Model. Each SPIRV file is called a SPIRV module that can specify the features[8] of the language frontends[9] of client drivers through the `OpCapability` instruction. SPIRV provides a unified style of instructions mapped into the programming interfaces of supported platforms through `Decoration` instructions. Besides, `Extended instruction sets` are used to support the built-in functions (trigonometric functions, exponentiation, etc.) from high-level languages in SPIRV.

- Execution Model. A SPIRV module contains multiple entry points[10] to potentially share functions between the entry point's call trees [34]. The `OpEntryPoint` instruction allows each entry point to specify the execution model that will be mapped into the execution model of the specified platform.

- Memory Model. SPIRV does not define a typical memory model. Instead, it provides the `OpMemoryModel` instruction to specify which memory model of supported platforms will be chosen and then handle all memory objects decorated by `Decoration` instructions over the client drivers. Besides, the `OpMemoryModel` instruction also supports the two kinds of addressing modes – logical and non-logical addressing modes to express the virtual pointers and physical pointers used in client drivers.

[8] The list of features from `Page 101-104` in [34].
[9] GLSL(OpenGL), ESSL(OpenGL ES), OpenCL C/C++, etc. The full list at `Page 43` in [34].
[10] A function in a SPIRV module where execution begins.

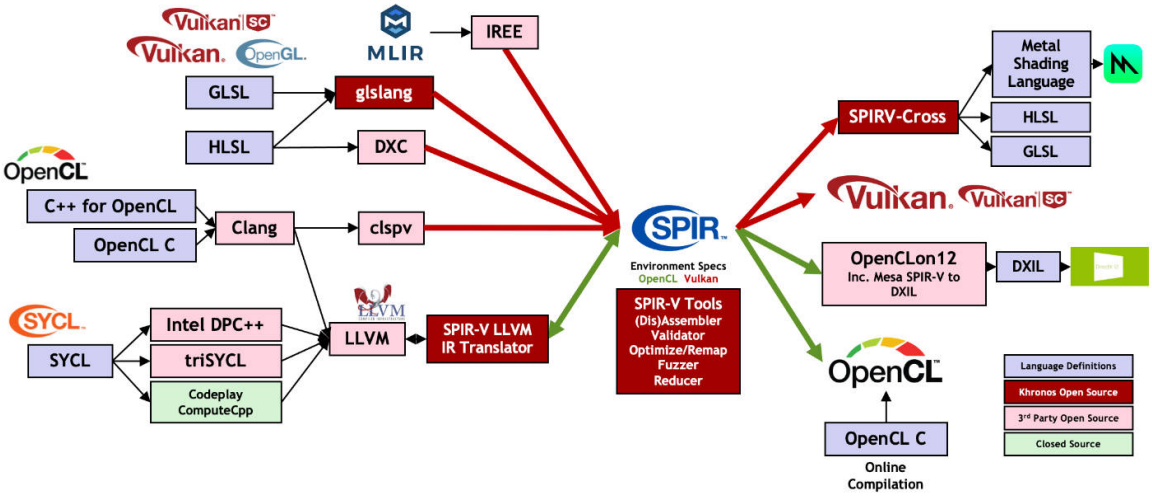


Figure 2.6: The ecosystem of SPIRV.

The Ecosystem of SPIRV

The ecosystem of SPIR-V is illustrated in Figure 2.6. Different frameworks of parallel computing support compiling the accelerated kernels or shaders into SPIRV IR and then lowered by the client drivers. Besides, the command line tool called `clspv` [35] can compile OpenCL kernels into Vulkan computing shaders. However, it is worth noting that SPIR-V IR contains the specific information of target backends, so only the SPIRV IR converted from LLVM IR can launch by OpenCL runtimes[11].

The design of SPIRV enables parallel computing not bound to a particular platform (supported platforms in Figure 2.7) through porting to various backends. Although SPIR-V cannot target directly on Metal [36], `SPIRV Cross` [37] can parse and convert SPIRV IR to Metal Shader Language [38] and then execute in Apple Metal as long as the SPIRV IR is written for compute shaders.

[11] Any OpenCL implementation. OpenCLOn12 is a mapping layer of OpenCL 1.2 specification for Windows.

Figure 2.7: The supported platforms in the ecosystem of SPIRV.

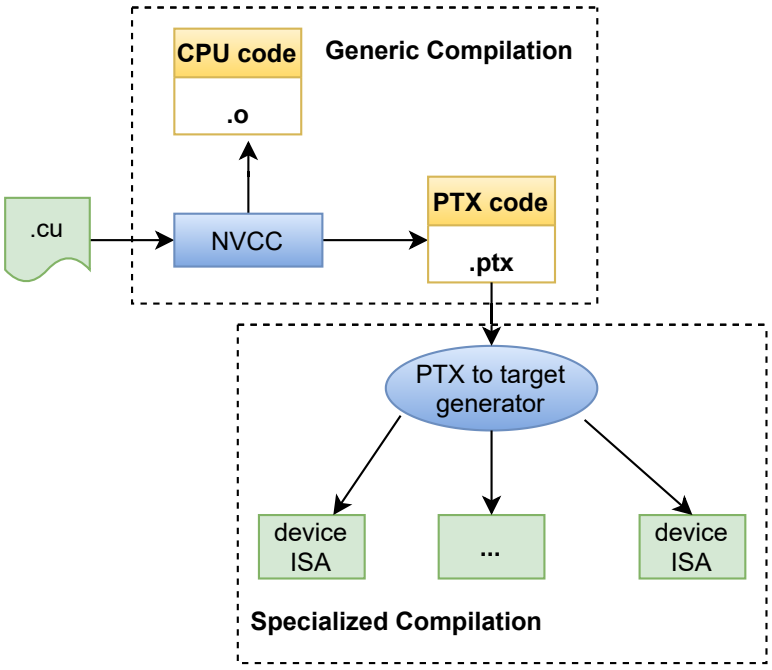


Figure 2.8: The overview of PTX code to native ISA.

2.1.4 Parallel Thread Execution

Parallel Thread Execution (PTX or NVPTX) [39] defines a low-level virtual machine and ISA used in NVIDIA's GPUs for general-purpose parallel programming. It has an assembly-language style syntax and provides a machine-independent ISA for executing across CUDA devices with different architectures. In the CUDA programming environment, kernels are compiled as PTX code and then handled over the CUDA driver at install time to translate to native target-architecture instruction sets (illustrated in Figure 2.8).

Since PTX is dedicated to CUDA devices, more details about CUDA/PTX are described in Section 3.2.

2.2 Recent Research and Development of SYCL

Although CUDA can exploit NVIDIA devices and achieve high performance, it has been exclusive and only supports NVIDIA chipsets. By comparison, SYCL and OpenCL provide an efficient way to support heterogeneous target hardware. Therefore, there are a lot of researches to port CUDA programs with SYCL or OpenCL runtimes. [40] [41] [42] show that the conversion between OpenCL and CUDA also performs well, even though there are some extra overheads compared to native CUDA programs. [43] provides a refactoring tool for transforming CUDA programs into SYCL programs automatically. [44] integrates the cuDNN library via SYCL and thus enables oneDNN to target NVIDIA hardware.

There have many kinds of research to exploit SYCL in machine learning projects. [44] demonstrate that SYCL can work for cuDNN and oneDNN to achieve seamless cross-platform performance portability. [45] utilizes SYCL-DNN, one open-source library based on SYCL, to accelerate neural networks on OpenCL devices and argues that further optimizations on memory prefetching are necessary to make it perform better. Meanwhile, SYCL BLAS [46] and SYCL-DNN are also exploited to deploy computing kernels auto-tuned by machine learning while keeping good performance across heterogeneous accelerators [47] [48]. [49] research on using oneAPI, an SYCL-based programming model, to facilitate co-execution and explore the performance limits of heterogeneous systems and turns out oneAPI can enhance co-execution and improve efficiency when using dynamic load-balancing algorithms.

Apart from the above, SYCL also shows the potential to contribute to autonomous driving since many papers propose computing stacks based on OpenCL and CUDA to address the computing heterogeneity of autonomous driving systems. For example, 4C framework [50] requires defining the top-level APIs to incorporate computing models, such as OpenCL, CUDA and OneAPI, but SYCL could ease their efforts to write the top-level APIs to interact with low-level computing resources. Besides, a modularized computing stack based on OpenCL is depicted in [8] but cannot support

other vendor accelerators except OpenCL, such as NVIDIA GPUs. With the support of SYCL, this computing stack would enable accelerating computations across vendor hardware instead of limiting to OpenCL.

Concerning the performance comparison among computing frameworks, on the one hand, SYCL has significantly improved performance compared to its earlier versions [40] [41], although it still cannot outperform plain OpenCL and CUDA. On the other hand, with the improvement of SYCL implementations, [51] [52] [53] [54] compares SYCL and other popular frameworks and illustrates the negligible gap in performance between them, and SYCL has the potential to be competitive with other conventional computing frameworks. Some research [55] [56] also parameterizes SYCL kernels to demonstrate the performance portability of SYCL across heterogeneous platforms and show that SYCL provides competitive performance against optimized libraries such as CLBlast [57] and MKL-DNN.

Chapter 3

Parallel Frameworks

In this chapter, we try to illustrate the overall architectures of three popular parallel frameworks (OpenCL, CUDA and SYCL), as both OpenCL and CUDA are the two most prevalent frameworks supported by SYCL and have technical supports in QNX environments. By explaining the architectures of OpenCL and CUDA and illustrating how they are related to techniques mentioned in the previous chapter, we can exemplify why SYCL is proposed. Finally, the programmability and kernel construction in three computing frameworks is compared to illustrate how they are integrated as unified programming interfaces.

3.1 OpenCL

OpenCL is a programming framework and standard set from Khronos Group, aiming for heterogeneous parallel computing on cross-vendor and cross-platform hardware. To handle massively parallel code execution on heterogeneous platforms, it layers the programming model as the platform model, the memory model, and the execution model. Therefore, OpenCL enables to scale of code from simple embedded microcontrollers to general-purpose CPUs up to parallel GPGPU hardware pipelines, without rewriting existing code massively.

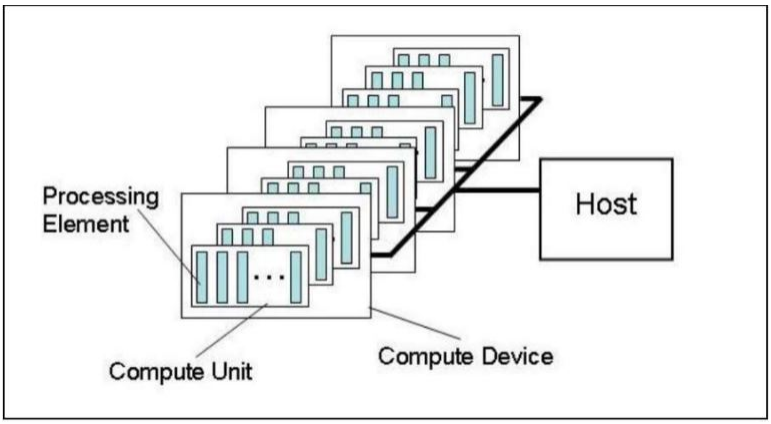


Figure 3.1: OpenCL platform model.

3.1.1 Platform Model

The platform model for OpenCL is a top-level abstraction for low-level hardware resources. It consists of a host connected to one or more OpenCL devices. Each OpenCL device includes one or more Computing Units (CUs), further divided into one or more processing elements (PEs), which perform the actual computation. As shown in Figure 3.1, the OpenCL host, emulated on CPUs, controls multiple OpenCL devices. Therefore, an OpenCL application is implemented as both the host code and device kernel code. The host code runs on a host processor, such as CPU, bound to the OpenCL host, and submits the kernel code as commands from the host to OpenCL devices. At the lowest level, all processing elements and computing units are responsible for scheduling and executing OpenCL kernels[1].

In OpenCL, device kernels are compiled just in time via an online compiler[2] or ahead of time via offline compiler[3] provided by an OpenCL platform. The OpenCL platform also has multiple version identifiers, including the OpenCL C[4] language, the device and vendor-specific platform, as developers may provide programs in the forms of C source strings, the SPIR-V intermediate language, or as implementation-defined binary objects.

[1] A kernel is a function declared in a program and executed on an OpenCL device for acceleration.
[2] An online compiler is available during host program execution using standard APIs.
[3] An offline compiler is invoked outside of host program control, using platform-specific methods.
[4] OpenCL C is a subset of C99 with appropriate language additions.

3.1.2 Execution Model

The OpenCL execution model is conformant to the platform model and is defined in terms of two distinct units of execution: kernels and a host program.

An OpenCL kernel executes within a well-defined context managed by the host. The context is exposed by an OpenCL platform and defines the environment within which kernels execute. It maintains OpenCL devices, kernel objects[5], program objects[6] and memory objects[7] during its lifetime. To enable the host to interact with a targeted device, OpenCL exposes various functions to the host through a command queue associated with the device, which mainly includes three command types:

- Kernel-enqueue commands: Enqueue a kernel for execution on a device.
- Memory commands: Transfer data between the host and device memory, between memory objects, or map and unmap memory objects from the host address space.
- Synchronization commands: Explicit synchronization points that define order constraints between commands.

Regardless of the host side or the device side, the OpenCL runtime handles all commands in a command queue through six states shown in Figure 3.2. Commands can capture the status of other commands through event objects[8] and thus execute relative to each other in either in-order execution or out-of-order execution. Multiple command queues in the same context can also keep consistent via synchronization points established in event objects.

The OpenCL execution model also provides details about how the kernels execute. Once a kernel enqueue command submits a kernel for execution, the kernel is instantiated, and then an index space (called `NDRange`) is created for managing the

[5] A kernel object encapsulates a specific kernel function declared in a program and the argument values to be used when executing on OpenCL devices.

[6] A program object encapsulates a reference to an associated OpenCL context, a program source or binary, the number of kernel objects attached and the latest built program executable.

[7] A memory object is a handle to reference counted region of the global memory.

[8] An event object encapsulates the staus of operations such as commands.

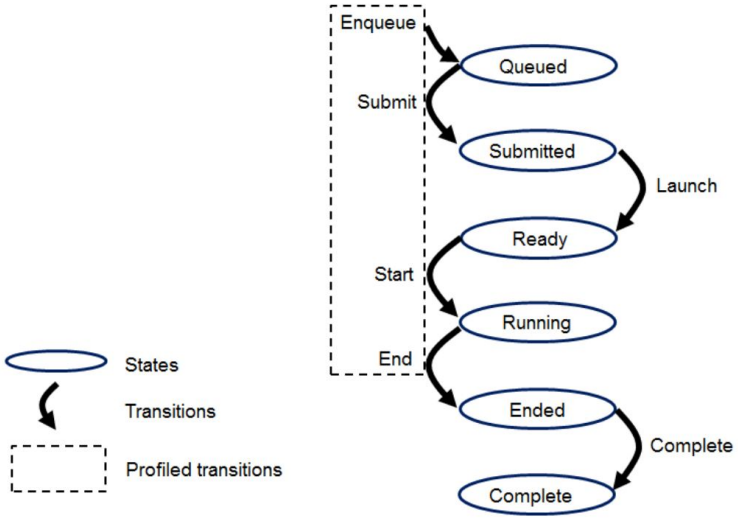


Figure 3.2: The six states and transitions of commands in the execution model.

kernel instance, argument values and parameters. The fine-grained composition of the index space is called work-groups that manage all work-items[9] associated with the kernel instance. Such a mechanism provides some functions to guarantee the synchronization between work-groups and the invisibility between work-items belonging to different work-groups, thereby guaranteeing the correctness of computations on a device.

The kernel instance will not launch until its dependencies - data dependency and event dependency, are captured. The targeted device schedules work-groups in the work-pool for execution on the computing units of the device. After completing all computations, the global memory will update according to global IDs bound to work-groups.

3.1.3 Memory Model

The OpenCL memory model describes the structure, contents and behaviour of the memory exposed by an OpenCL platform as OpenCL programs execute. It is defined

[9]A work-item is executed by Processing Elements as part of a work-group executing on a Computing Unit.

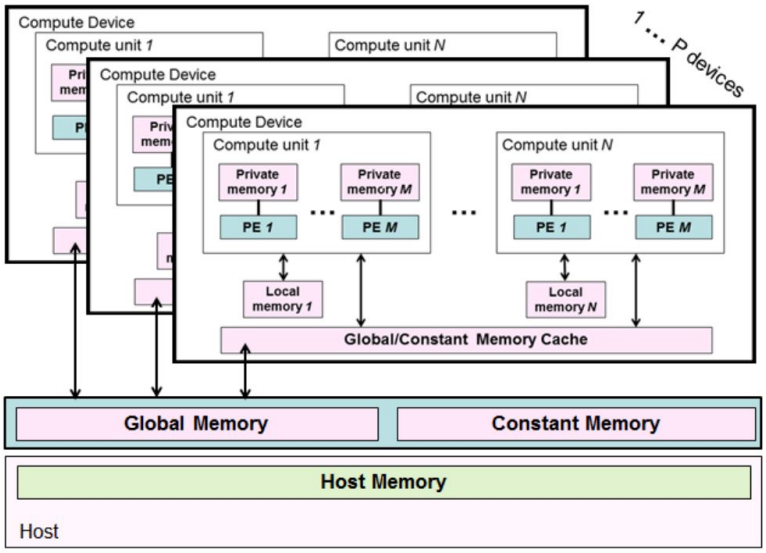


Figure 3.3: OpenCL Memory Model. Global and constant memories are shared between the devices in a context, while local memory and private memory are assoicated with a single device.

in four parts:

- Memory regions. The distinct memories are visible to the host and the devices within the same OpenCL context, called the host memory and device memory. The device memory is further divided into four memory regions (shown in Figure 3.3) -global memory, constant memory, local memory and private memory, to provide fine-grained control and parallelism on devices.
- Memory objects. The objects on the global memory are defined by the OpenCL API and their management by the host and devices. All allocations of memory are done in the host, and the movement of memory objects between the host and devices is through interfaces provided by a shared virtual memory.
- Shared virtual memory (SVM). A virtual address space exposed to both the host and the devices within an OpenCL context. SVM enables the use of pointer-based data structures in OpenCL kernels since it logically extends a portion of the global memory into the host address space and allows work items to access the host address space.
- Consistency model. Rules that define the visibility of data when multiple units of execution load data from the memory.

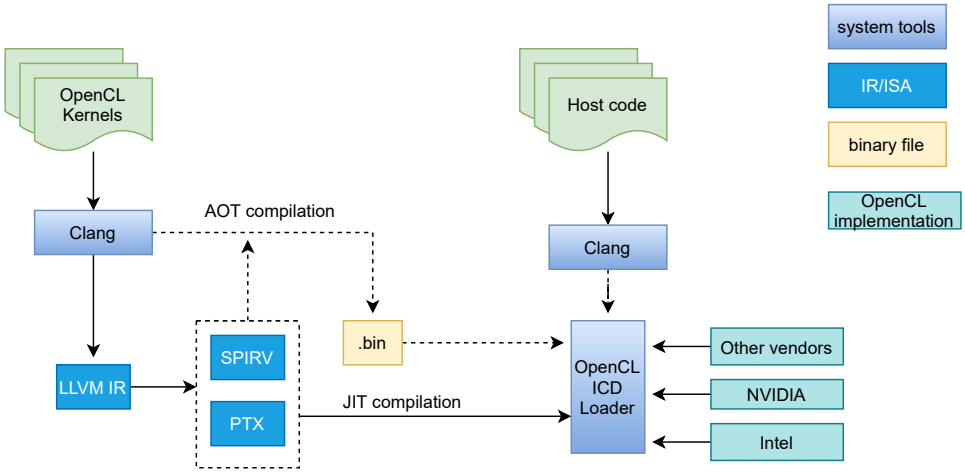


Figure 3.4: The overview of OpenCL compilation.

By mapping device memories into the unified memory model, different devices can work correctly using the consistent OpenCL APIs, thereby facilitating parallel computing on heterogeneous devices. Furthermore, the memory model gives programmers more explicit control over the behaviour of each work item and work-group, while there are no details of target-specific architectures exposed outside of the OpenCL environment.

3.1.4 OpenCL Compilation

Since many vendors have different OpenCL implementations conforming to the OpenCL standard, OpenCL defines the Installable Client Driver Loader [58] mechanism that enables developers to build applications against the ICD loader and load a specific OpenCL implementation (shown in Figure 3.4) when executing. The ICD loader can expose OpenCL API entry points, enumerate vendor-specific OpenCL implementations and forward OpenCL API calls to the correct implementation, while the vendor-specific implementations can expose hardware devices and provides definitions of OpenCL API entry points.

The OpenCL kernels are extracted from one OpenCL program and then translated into LLVM IR (See section 2.1.2) in the pipeline of `clspv` [35] compiler. Users

can provide different compilation options to use different LLVM backends and produce corresponding intermediate representations (SPIRV for Intel devices, PTX for NVIDIA devices, for example). During execution, the vendor-specific OpenCL runtimes can compile IR into device binaries just-in-time or launch device bianries directly compiled ahead-of-time.

3.2 CUDA

CUDA is a general-purpose parallel computing platform introduced by NVIDIA in 2006 that enables NVIDIA GPUs to solve complex computational problems more efficiently than CPUs. Through the extension of CUDA C/C++, developers can develop high-performance parallel programs running on NVIDIA GPUs. To provide fine-grained parallelism of data and threads, the CUDA programming model abstracts its cores as the three keys – a hierarchy of thread groups, shared memories and barrier synchronization – that programmers can utilize simply as a series of programming interfaces.

3.2.1 Execution Model

In NVIDIA hardware, the thread hierarchy (illustrated in Figure 3.5) is similar to the division of work items and work-groups in the execution model of OpenCL. Each independent element that executes a kernel is called CUDA thread (which OpenCL refers to as work-item), and the batch of threads is organized as a grid of basic blocks (also called Cooperative Thread Array aka CTA, and which OpenCL refers to as work-group). To manage hundreds of threads effectively running several different programs, NVIDIA GPUs employ the SIMT architecture to create, manage, schedule and execute kernels. Each thread is mapped to one scalar processor core that owns a private instruction address and register state during execution. Therefore, threads within the same block execute in the SIMT mode: threads execute the same kernel instance but may operate different data region. The unique thread identifier for each thread allows them to work independently on different data or cooperatively on the

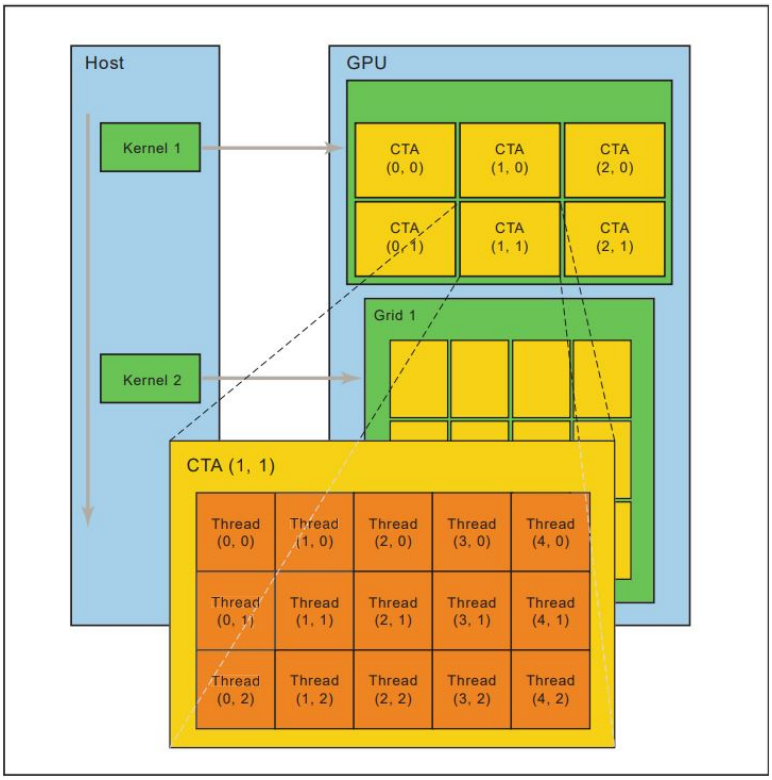


Figure 3.5: The thread hierarchy in CUDA.

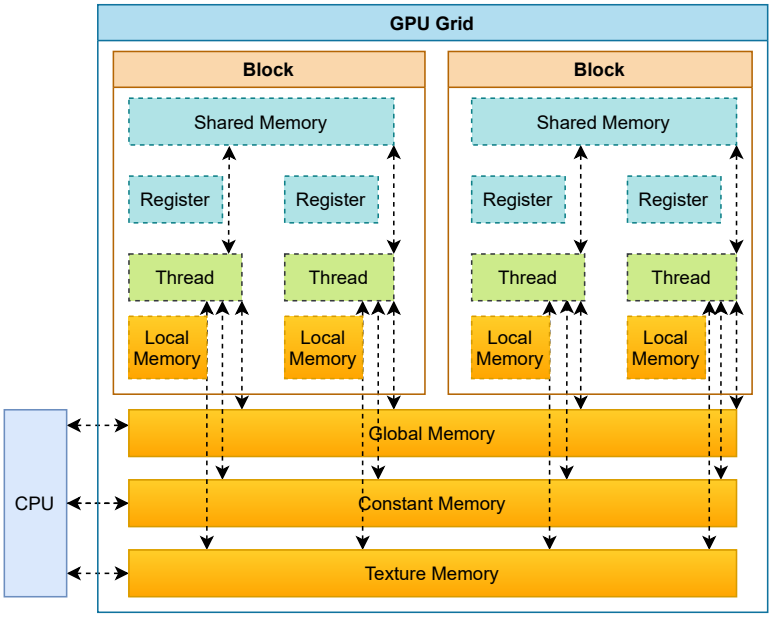


Figure 3.6: The memory hierarchy in CUDA.

same data region through the mechanism of synchronization points specified in the block, thereby peaking the performance of parallelism.

3.2.2 Memory Model

Typically, a system is composed of a host and one or more devices, each of which has a separate memory. Figure 3.6 illustrates the hierarchy of the memory model in the CUDA programming model. Each thread is guaranteed to own private local memory, while each thread block exposes a shared memory to all threads of the block during its lifetime. Apart from the global memory that all threads have access to, additional memory spaces, including constant memory and texture memory, are reserved for optimizing different memory usage and accessible by all threads. It is similar to the memory model of OpenCL except that CUDA provides more explicit mechanisms to regulate and synchronize the behaviour of each thread and block.

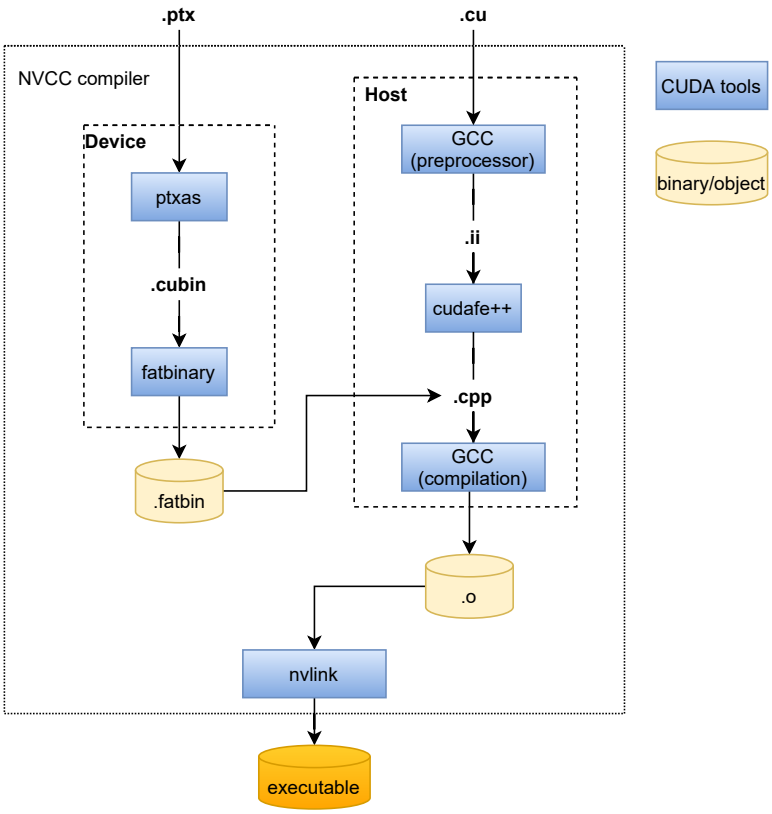


Figure 3.7: The overview of the compilation of CUDA kernels in the CUDA environment. [1]

3.2.3 CUDA Compilation

The separate compilation of CUDA kernel functions is supported since CUDA 5.0 [59], allowing the CUDA driver to run PTX code generated by other compilers. In Figure 3.7, the PTX-to-GPU translator of the CUDA driver can translate PTX instructions (`.ptx`) as executable device code (`.cubin`) for various NVIDIA GPU architectures. Then, the executable device code is wrapped in a host object to output as a fat binary. The embedded fat binary is linked with the host code to generate the final executable. The CUDA driver can unwrap the fat binary (or just-in-time compile PTX into binaries) and load executable device code into the specific device during execution.

On the other hand, LLVM supports targeting LLVM IR to the PTX backend since LLVM 3.0 [60], so PTX code can also be generated from OpenCL kernels in an OpenCL programming environment. The whole process is similar to OpenCL compilation (See Figure 3.4, except that the compilation of OpenCL kernels requires including the `libclc` library to maps OpenCL built-in functions to target-specific functions in the LLVM IR.

3.3 SYCL

SYCL [61] is a single source, high-level, standard C++17 programming model, that can target a range of heterogeneous platforms (listed in **Table** 3.1), developed by Khronos Group, announced in March 2014. It is a royalty-free, cross-platform parallel programming framework designed to abstract the complexity of traditional parallel programming models like OpenCL, allowing developers to use modern C++ features, such as inheritance, templating and operator overloading.

Earlier versions of SYCL [62] were closely aligned with OpenCL and behave like a C++ abstraction layer over OpenCL, aiming to make parallel programming easier based on OpenCL. However, SYCL 2020 [63] transitioned to a generalized backend model, making OpenCL just one of many different potential backends. SYCL is capable of hiding a large amount of the complexities of different backends, substantially

Table 3.1: Features of SYCL

Single source	Developers can write both host and device code in the same C++ source file (executed like in **Figure** 3.8), and then the source file will be compiled through two compilation passes, one for the host code and one for the device code.
High-level	Backend APIs (e.g., OpenCL) is abstracted as common boilerplate code in SYCL, instead of writing different codes for specific backends.
Heterogeneous platforms	SYCL can target any device supported by its backend, including CPU, GPU, ASIC, FPGA and DSP. (The current specification is limited to OpenCL and CUDA.

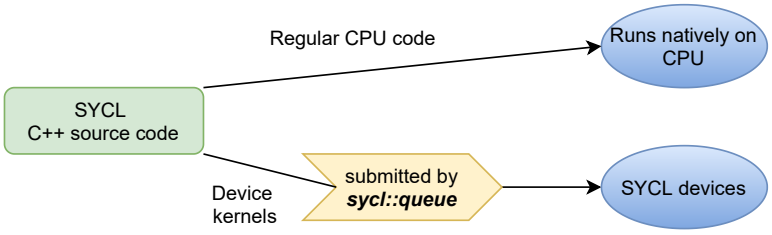


Figure 3.8: Execution of single source file.

reducing the amount of the host-side code needed over. It is designed to abstract low-level resources as different models, which makes it more user-friendly.

3.3.1 Backend Model

The ability of SYCL to target multiple heterogeneous accelerators is from the SYCL backend model. Under the SYCL backend model, SYCL objects can contain one or multiple references to a certain SYCL backend native type. However, not all SYCL objects will map directly to an SYCL backend native type, so the mapping between

SYCL interfaces and a particular SYCL backend is defined either by the documentation of SYCL implementations or by a separate SYCL backend specification document provided by Khronos Group.

Although different code implementations of SYCL specification have different strategies to construct the SYCL backend model, the core philosophy is to abstract a plugin layer to interface different backends with the SYCL runtime. The plugin layer can map specific backend APIs into SYCL generic interfaces, exposing different backends to the SYCL platform model and thus providing unified programming interfaces. Such a mechanism enables SYCL applications to build upon several active backends, thereby guaranteeing the interoperability between SYCL backends.

With the support of SYCL, a system will be able to integrate different accelerators and provide more powerful computing ability. Although existing SYCL implementations do not support all vendor-specific backends, the situation will happend to change with the improvement of backend plugins.

3.3.2 Platform Model

The platform model behaves like a proxy of computing resources exposed by the backend model. An SYCL context, which holds all runtime information interfaced with the SYCL runtime and SYCL backends, can be constructed from the platform model explicitly or implicitly, and then is used to group multiple devices managed by the SYCL platform. As a result, a group of devices in the same SYCL context can have the visibility of each other's memory objects and therefore allow to exchange of data between devices. Nevertheless, developers do not need to manipulate at the level of the SYCL context. All device attached to a particular context will be bound to SYCL queues and then exposed to developers. Therefore, the SYCL platform model, represented as the SYCL runtime in one system, can bridge SYCL backends with SYCL applications.

3.3.3 Execution Model

In SYCL applications, there are three scopes, including application scope, command group scope and kernel scope. Only code in the kernel scope, called kernels, runs on selected devices. Therefore, the SYCL execution model is comprised of the application execution model and kernel execution model. The schedule and execution of one SYCL application are under the control of the SYCL runtime conformant with the SYCL execution model.

- Application execution model. Code in the application scope and command scope, called the host code, runs on the host that is an emulated environment on CPU in SYCL. The SYCL runtime exposes SYCL interfaces, a mapping of low-level resources, to SYCL applications. Therefore, SYCL applications can capture the requisites of kernels and group them into command group objects. Command group objects contain the order execution of kernels under the control of the SYCL runtime and will be submitted to execute on devices bound to SYCL queues. The application execution model depends on the result of the kernel execution model if the SYCL runtime requires to synchronize between the host and devices. Therefore, the host code in an SYCL application can be compiled in any C++ compiler, called the host compiler, reducing a large amount of work to porting SYCL on QNX.
- Kernel execution model. The SYCL kernel execution model abstracts the complexities of parallel computing as unified programming interfaces. Typically, OpenCL devices and CUDA devices use different programming styles to write parallel kernels, while SYCL allows developers to use unified programming interfaces to write kernels for both kinds of devices. Their differences in kernel code are shown in the code snippets.

Besides, built-in kernels in specific backends are also supported in SYCL, because SYCL can directly encapsulate such kernels as object files and send them together with SYCL kernels to targeted devices. However, since different backends may support different instruction set architectures, SYCL is designed to compile kernels as intermediate representations that will be finalized on targeted devices. As a result,

SYCL implementations must provide an SYCL-aware compiler, called the device compiler, to compiler kernels separately.

3.3.4 Memory Model

The SYCL memory model must keep consistency on both the host and devices so that SYCL can be conformant to the platform model and the execution model. It is essential to map the host memory and devices memory to the same global address space, otherwise, the SYCL runtime may fail to synchronize data between the host and devices. Therefore, an SYCL implementation must guarantee that the same consistency model is used across the host and devices.

- Application memory model. SYCL memory objects are not designed to bind with particular devices. Instead, all devices can access SYCL memory objects in one SYCL application, regardless of the SYCL context where they reside. By encapsulating underlying SYCL backend memory objects together with the host memory, SYCL enables the same memory object to be shared between the host and devices in different contexts, platforms or backends. the SYCL memory model enables the device compiler to handle SYCL memory objects correctly and keep them consistent behaviour between the host and devices.
- Device memory model. After SYCL memory objects are instanced as backend-specific memory objects, SYCL also provides unified interfaces for developers to control their consistency on the targeted device, avoiding developers to manage memory using complicated backend-specific APIs. It is also worth noting that backend-specific memory objects are visible to each other only when they have the same SYCL context.

As the high-level abstraction of OpenCL, the memory model of SYCL is also inherited from OpenCL (like Figure 3.3).

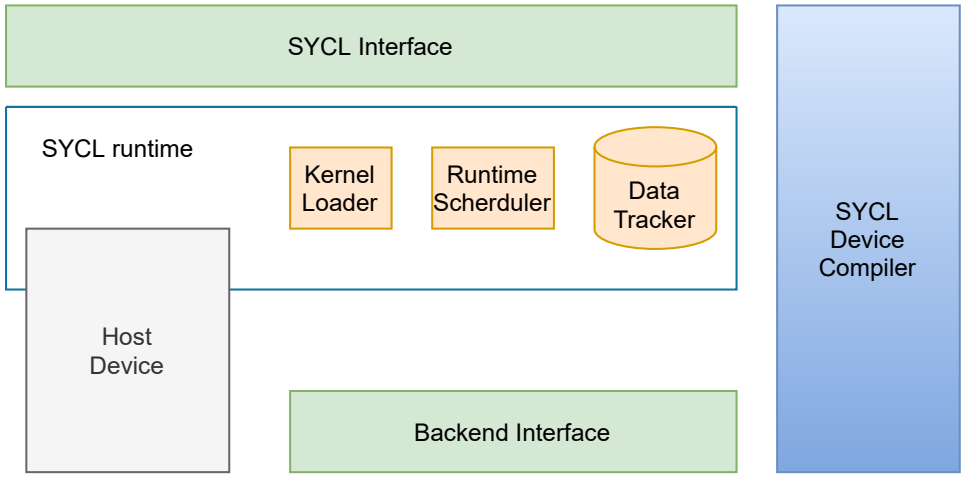


Figure 3.9: Overview of SYCL implementations.

3.3.5 SYCL Architecture and Compilation

SYCL Architecture

Although SYCL implemetations[10] may have different definitions of some SYCL features, all of them have the same SYCL architecture like in **Figure** 3.9:

- SYCL implementations provide the SYCL interface as a C++ template library that developers can use to access the features of SYCL. Both host and device code can use the same interface.
- SYCL runtime is a library that is responsible for scheduling and executing tasks submitted by SYCL command queues[11]. It loads kernels[12], tracks data dependencies and schedules commands in command groups[13].
- The host device is an emulated backend that is executed as native C++ code and emulates the SYCL execution and memory model.

[10]Existing SYCL implementations. See https://www.khronos.org/sycl/.
[11]A SYCL command queue is an object that holds command groups to be executed on s SYCL device
[12]A SYCL kernel which can be executed on a device, including SYCL host device, created by three ways: C++ lambda expression, named function object and interoperability with other languages or APIs
[13]The group of commands for transferring and processing data on a device using kernel

- The backend interface is where the SYCL runtime calls down into a backend to execute on a particular device. (In SYCL 1.2.1, the standard backend is OpenCL but the latest SYCL implementations have supported interfaces from other vendors (like CUDA from NVIDIA).
- The SYCL compiler is a C++ compiler that can identify SYCL kernels and compile them down to an IR or ISA (which can be SPIR, SPIR-V, PTX or any proprietary vendor ISA).

Each SYCL implementation may provide different SYCL-aware C++ compilers, such as DPC++ [64] and ComputeCPP [65], thereby enabling SYCL programs to perform best while developers have access to the full range of capabilities of OpenCL and other backends through the features of the SYCL libraries.

SYCL Compilation

The SMCP (Single-source Multiple Compiler-Passes) design of SYCL offers the power of source integration and remains the flexibility of toolchains. Multiple different compilers can work together as a compilation flow where the source file is passed through and the resulting application combines with the compiled results. Therefore, the compilation of SYCL applications also requires two separate compilation, similar to OpenCL and CUDA compilation, to output the final executable.

Figure 3.10 illustrates how SYCL implementations handle SYCL applications. The device compiler firstly generates the headers to provide the declarations of SYCL kernels for the host code and then emits the intermediate representations that will be handled by the device driver (such as OpenCL runtimes or the CUDA driver) when executing the application (shown in Figure 3.11). The offload wrapper encapsulates the generated IR as a host object, and then the host linker outputs the final executable by linking the wrapped IR and the host code. Particularly, if users provide an option of ahead-of-time compilation (AOT) for the device compiler, the wrapped object file will also include the target-specific binary code. Particularly, most SYCL implementations only support ahead-of-time compile SYCL kernels for CUDA devices.

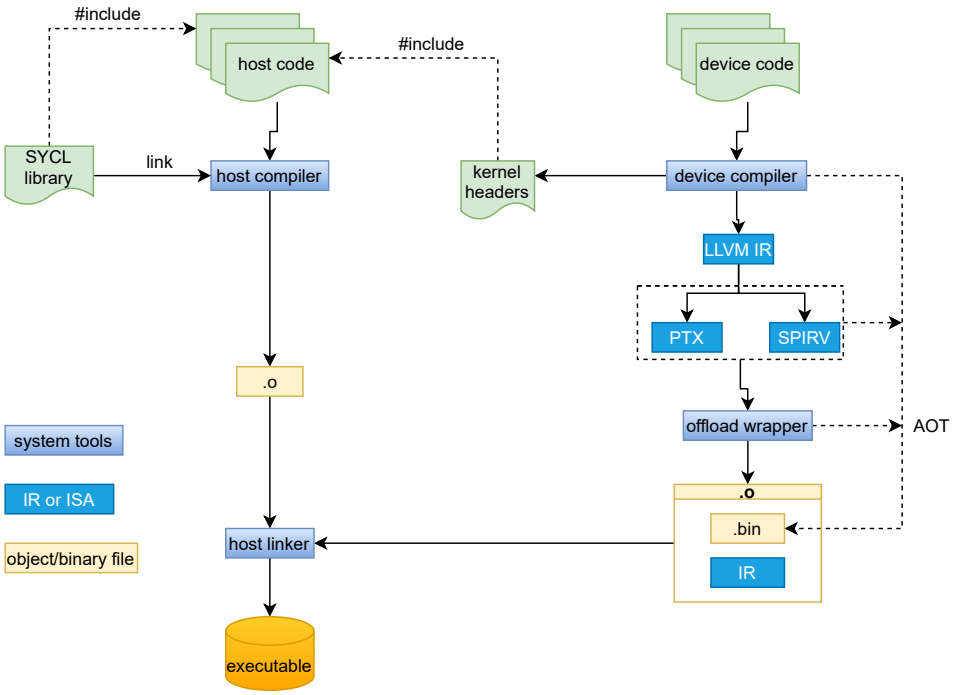


Figure 3.10: The overview of SYCL compilation.

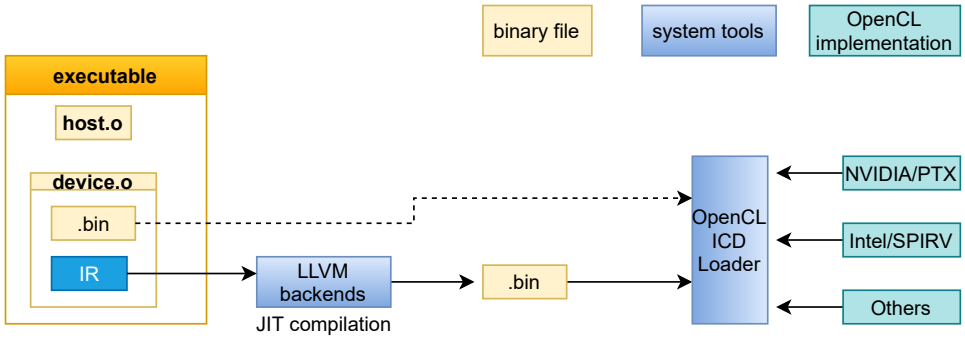


Figure 3.11: Executing SYCL device code.

When running SYCL applications, the OpenCL ICD loader can launch the binary code directly or compile the IR just-in-time for the specific LLVM backend (contained in the device compiler). Then, the API calls in the binary code can be forwarded to the target architecture by the loader.

3.4 The Comparison of CUDA, OpenCL and SYCL

3.4.1 Programmability

For most programmers, the construction of OpenCL and CUDA programs is not easy to deal with. Before executing a kernel, programmers must call host APIs sequentially to construct and execute a kernel in CUDA and OpenCL (See code A.1 and A.2), making the whole program verbose. By comparison, SYCL hides a lot of the complexities of host APIs1 and enables users to focus on writing clean accelerated kernels (see code A.3). Apart from the simplicity of application source code, the design of SYCL empowers the reusability of templated kernels for different data types [51].

Besides, SYCL can manage data access automatically, whereas CUDA and OpenCL require explicit data management in one program. After allocating memory on the host or devices, SYCL runtime can handle data movement and synchronization since SYCL relies on the C++-style RAII and thus can capture data dependencies between devices and the host [51]. In contrast, developers must manually move data to the selected device and manage data access explicitly in a CUDA or OpenCL environment. Particularly, SYCL may be integrated as a C++ standard library class on parallelism and heterogeneous computing [11] [12].

3.4.2 Kernel Construction

According to the compilation of the three computing frameworks (see section 3.1.4, 3.2.3 and 3.3.5), the essential part of them is to extract the kernels from source code. Both CUDA and OpenCL provide some extensions and macros to handle kernels in

the host code, while SYCL allows writing kernels in standard C++ syntax.

- Marking kernels[14]. CUDA uses three specifiers: `__host__`, `__global__`, `__device__` to mark a kernel and specify on which device the kernel executes, while OpenCL kernels are marked by `__kernel`, but its parameters are bound to different address space qualifiers for a particular device. By comparison, SYCL kernels represent as C++ lambda functions or class functors, not requiring extra macros to mark an SYCL kernel.

- Invoking kernels[15]. The invocation of CUDA kernels is like a regular C++ function call, whereas OpenCL kernels must handle by sequential API calls for execution. In contrast, SYCL queues can submit kernels automatically.

- Special indices[16]. CUDA organizes the threads executing a kernel as a hierarchy level of grid-block-thread (see section 3.2.2) and provides unique identifiers for indexing. On the other hand, due to the similarity of OpenCL and SYCL memory model (see section 3.1.3 and 3.3.4), both provide the `ND-range` mechanism to index each work item and work group.

By comparison, SYCL has a cleaner syntax to write a kernel and execute device code in standard C++ function calls. The three code lists also demonstrate that SYCL makes parallel programming more intuitive and user-friendly than OpenCL and CUDA, thereby providing a simpler abstraction for heterogeneous computing.

[14]See Appendix B.2: device API equivalence for kernel specifiers.
[15]See Appendix B.1: runtime API equivalence.
[16]See Appendix B.3: indexing equivalence.

Chapter 4

Proposed Methods of Porting SYCL on QNX

In this chapter, we briefly compare the current development of SYCL implementations and choose to use DPC++ and SYCL-GTX in our experiments. Then, we present our experiments in two cases and introduce our proposed methods and schemes.

4.1 Possible SYCL Implementations for QNX

There are many mainstream SYCL implementations shown in Figure 4.1. By comparison, ComputeCpp is a closed-source commercial project developed by Codeplay and thus does not allow us to customize in our experiments, and triSYCL is incomplete and requires pre-installing many toolchains (especially LLVM toolchains) before building their SYCL runtime in a system. Most accelerators supported by hipSYCL work through OpenMP and HIP instead of OpenCL. DPC++ includes SYCL libraries and contains all required tools in their source code. Besides, SYCL-GTX, a minimal SYCL implementation developed by a single developer, implements many main features in the SYCL 1.2 specification and can connect with OpenCL runtime directly. Therefore, we decide to use DPC++ and SYCL-GTX in our tests since our primary goal is to demonstrate the feasibility of porting SYCL on QNX, not to build a robust, complete SYCL toolchain at the initial stage.

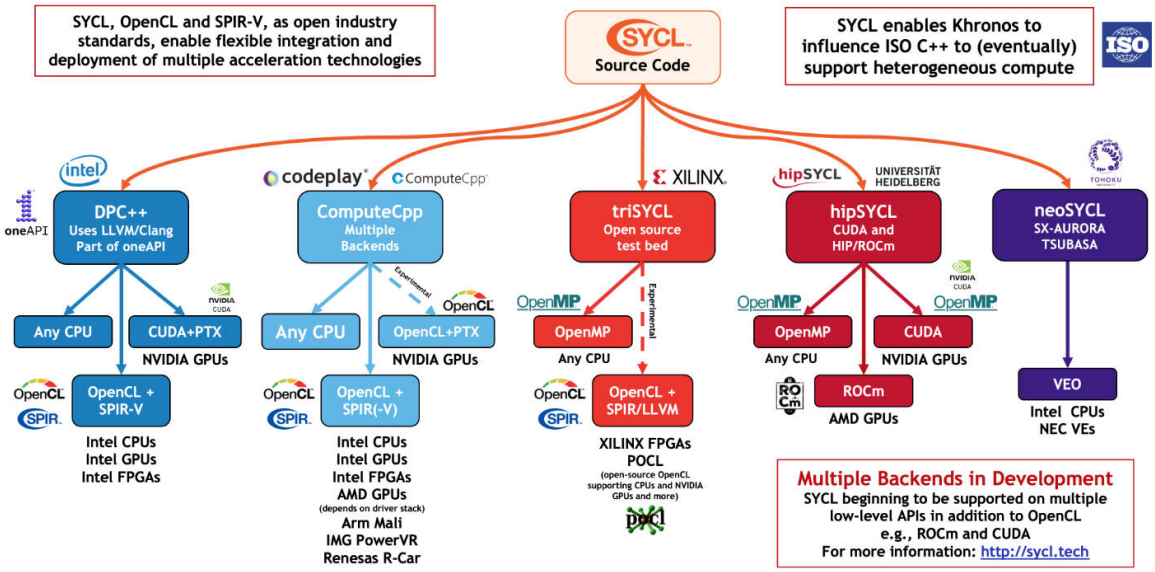


Figure 4.1: Existing mainstream SYCL implementations.

It is essential to compile device code for specific accelerators in SYCL (see section 3.3.5), but there is no ready device compiler on QNX. Consequently, we first conducted our experiment on `Ubuntu 18.04` to figure out the workflow of SYCL architecture and then tried different strategies to build an SYCL environment on QNX. In our Linux machine, there are three SYCL backends: the host backend (Intel CPU), the OpenCL backend (Intel GPU) and the CUDA backend (NVIDIA GPU). We broke our tests into two stages: using DPC++ and SYCL-GTX respectively on Linux and QNX.

4.2 Using DPC++

DPC++ is an SYCL implementation maintained by Intel, which is conformant to SYCL specification 1.2.1 [66] and supports some features of SYCL specification 2020 [67]. It supports different backends, including OpenCL and CUDA, by the Plugin layer – an abstraction between SYCL runtime and specific backend runtime (shown in Figure 4.2). The section SYCL backend model describes why the plugin layer can connect with vendor-specific accelerators.

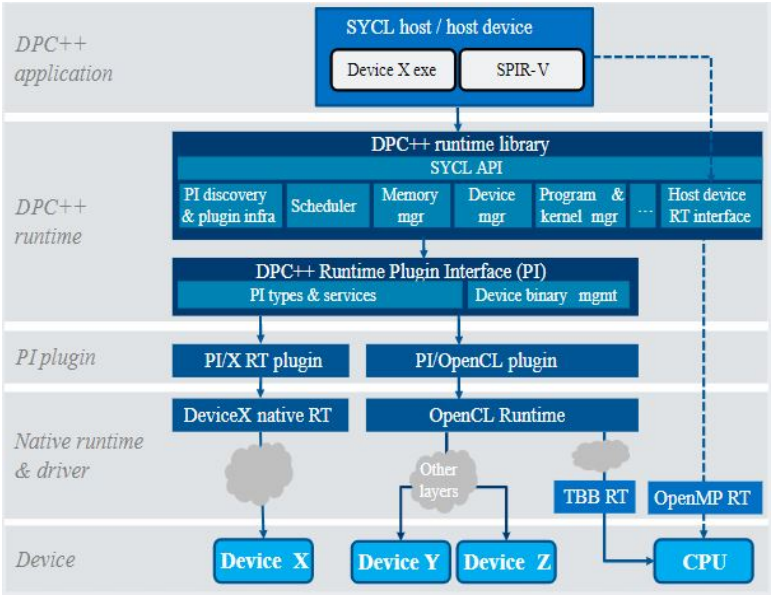


Figure 4.2: The overview of DPC++ architecture.

We mainly test DPC++ on Linux to demonstrate the importance of compiling device code for accelerators. Then, we try DPC++ on QNX and report some existing problems preventing from building DPC++ runtime on QNX.

4.3 Using SYCL-GTX

4.3.1 The Design of SYCL-GTX

SYCL-GTX [68] is an SYCL implementation maintained by a single developer, but now this project is no longer actively maintained by the author [69]. The implementation is far from complete and only covers some features of SYCL specification 1.2. However, this project still can help us try some SYCL features on QNX.

Our tests demonstrated that SYCL-GTX is a high-level encapsulation of OpenCL runtime APIs rather than has a plugin layer that can interact with different backends. Figure 4.3 shows the mapping of SYCL-GTX and OpenCL.

To make SYCL-GTX code workable with any host compiler, some special C++

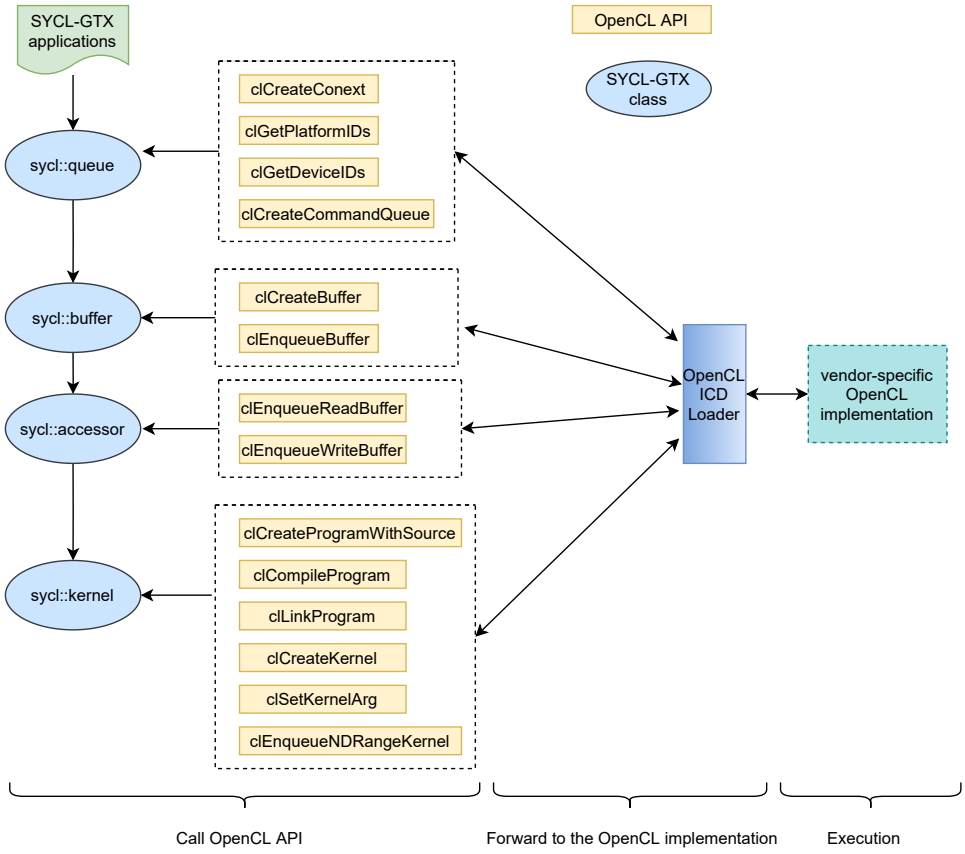


Figure 4.3: The core mapping of interfaces between SYCL-GTX and OpenCL.

classes and macros are used to capture the kernel code:

- `source` class. . The kernel invocation `parallel_for<..>(..)` will create `source` object to capture SYCL kernel code line by line and then combine with the argument information held in `accessors` class.
- `data_ref` class. As the host cannot have access to the private memory region of the accelerators, all variables in SYCL kernels must be derived from `data_ref` class to emulate them as native types in the host but behave as OpenCL variables.
- C++ macros. To enable the host compiler to recognize OpenCL control flow correctly, `SYCL_IF, SYCL_FOR, SYCL_WHILE, SYCL_END` are used to decorate OpenCL control flow so that the SYCL-GTX runtime can translate SYCL control flow as OpenCL C code.

The above unique mechanisms enable SYCL-GTX to generate OpenCL C code at runtime correctly and feed the generated code to the OpenCL C compiler to output the device binary just in time. On the one hand, as a result, the traditional GCC compiler can also compile SYCL-GTX applications. However, on the other hand, compared to the workflow of OpenCL, those mechanisms result in more compilation time (see section 5.2.2) and may introduce extra overhead at runtime (see section 5.2.4 and 5.2.5).

Besides, there are some drawbacks to SYCL-GTX:

- Cannot run SYCL applications on CPU devices. SYCL-GTX does not implement an SYCL host backend required in SYCL specification 1.2.1. Hence, users cannot execute their SYCL kernels on the host backend emulated on CPUs. However, it is still possible to accelerate SYCL kernels on OpenCL devices.
- Can only run SYCL applications on OpenCL devices. As a high-level abstraction of OpenCL, SYCL-GTX does not provide the mapping of interfaces of other vendor-specific backends. Consequently, SYCL-GTX runtime can only offload kernel acceleration to OpenCL devices. It is worth noting that SYCL-GTX can query the information of CUDA devices, but there is no way to interact with the CUDA driver.

we will discuss the possible impacts of those drawbacks to the future work in the next chapter.

4.3.2 Experiment Methods in SYCL-GTX

As most machine learning algorithms require heavy floating-point operations, we determine to benchmark floating-point multiplication in vector (algorithm 1) and matrix (algorithm 2). It is worth noting that we use iterative matrix multiplication algorithm and do not take any optimization, although many optimized algorithms could improve the performance of matrix multiplication. Its time complexity should be `O(N^3)` and would inflate dramatically with the increase of data volume. Consequently, when running real-time application, we must offload computations carefully to guarantee the execution of real-time applications.

Besides, since SYCL-GTX is built upon OpenCL, we test the two algorithms using OpenCL and SYCL-GTX to further demonstrate the performance of SYCL-GTX on Linux and QNX.

Algorithm 1: Vector Multiplication

Input: A, B, C, N
for $i = 0$ *to* N **do**
  $C[i] = A[i] * B[i]$;
end

Since QNX has better support on Intel boards than general laptops and PCs, all cases are tested on `Intel NUC 6i5SYH`. Furthermore, SYCL-GTX does not provide the interface to load pre-compiled device binaries and consequently requires compiling the kernel every time. Therefore, all programs call identical OpenCL APIs to keep consistent between them and thus minimize possible errors. Having a clear comparison of the performance difference between OpenCL and SYCL-GTX on Ubuntu and QNX, the total execution time is divided into five parts:

- Warmup. This part includes querying platform information, preparing execution contexts and creating execution queues.

Algorithm 2: Matrix Multiplication

```
Input: A, B, C, N
for i = 0 to N do
    for j = 0 to N do
        temp ← 0.;
        for j = 0 to N do
            temp ← temp + A[i * N + j] * B[i + j * N];
        end
        C[i][j] ← temp;
        ;            ▷ Or in vectorization-style:  C[i * N + j] ← temp
    end
end
```

- Kernel compilation. Compiling the kernel source for all devices or a specific device.
- Linkage. Linking a set of the compiled program objects and libraries for all the devices and creating the final executable binary.
- Data movement. Copying data from the host memory to dedicated device memories.
- Computation. Executing the compiled kernels after data ready.

Besides, we also tried to find some methods to improve the performance of SYCL on QNX.

Chapter 5

Result Analysis and Discussions

In this chapter, we test DPC++ and SYCL-GTX separately according to our proposed methods in chapter 4. Then, we analyze the results of the two cases and discuss some possible solutions and future directions for their problems. Besides, we also show the improvement of our new method in SYCL-GTX and give some suggestions for QNX to port SYCL in the future.

5.1 Testing DPC++

As we finally demonstrate that DPC++ is not compatible with QNX and cannot compare its performance difference on both Linux and QNX, we divide the experiments of DPC++ into two sections and introduce them separately.

5.1.1 DPC++ on Linux

Our experiment illustrates that the plugin layer in DPC++ does not directly map CUDA driver APIs. Currently, DPC++ only maps OpenCL API entry points in the plugin layer, and the OpenCL ICD loader can forward requests from those entry points to CUDA device runtime APIs (see section 3.3.5 and Figure 3.11).

After modifying code List A.3, we compile and run the simple SYCL application

Table 5.1: The relation of DPC++ compilation and execution.

Compilation option	Selected backend	Execution error
`-fsycl`	Host backend	N/A
`-fsycl`	OpenCL backend	N/A
`-fsycl`	CUDA backend	`CL_INVALID_KERNEL_NAME`
`-fsycl`, `-fsycl-targets=nvptx64`	Host backend	`CL_INVALID_KERNEL_NAME`
`-fsycl`, `-fsycl-targets=nvptx64`	OpenCL backend	`CL_INVALID_KERNEL_NAME`
`-fsycl` `-fsycl-targets=nvptx64`	CUDA backend	N/A

by different compilation options and specify different SYCL backends. The result is shown in Table 5.1.

DPC++ compiles the device code after specifying the option `-fsycl` and output the SPIRV code in default, whereas the option `-fsycl-targets=nvptx64` sets for outputting PTX code. Since users must specify the compilation option for CUDA devices, it must be compiled ahead of time when targeting CUDA devices.

The execution error `CL_INVALID_KERNEL_NAME` occurs when the kernel name cannot be found in a program. The host compiler is responsible for converting the SYCL kernel as LLVM IR and just-in-time finalize it as the target-specific instructions during execution. If the compilation option of the device compiler is not matched to the selected backend, the host code cannot distribute the compiled kernels to the selected backend. Consequently, it will report the execution error `CL_INVALID_KERNEL_NAME`.

For the OpenCL backend, the compilation option `-fsycl-targets` specifies the device compiler to generate PTX code for the CUDA backend. Consequently, the device binary cannot be recognized by OpenCL devices during execution. On the other hand, the wrong device binary can pass the compilation since the device compiler wraps all kernels as host objects, and the internal properties of wrappers are invisible

to the OpenCL runtime before execution. Therefore, OpenCL can create a program by `clCreateProgramWithBinary()` but fail to instantiate the wrong device binary as an OpenCL kernel and create the correct kernel name. When it comes to the CUDA backend, the cause is also due to unmatched device binaries, and the CUDA error is mapped as the standard OpenCL error.

Hence, it is essential to generate the device binary for the selected backend. Otherwise, it will crash down during execution, even though the SYCL application could pass ahead-of-time compilation.

5.1.2 DPC++ on QNX

We failed to build SYCL runtime on QNX from DPC++ source code since there are many incompatible issues between DPC++ and QNX.

The SYCL libraries of DPC++ only support Windows and Linux, so QCC reported errors about system-dependent services, such as accessing local files and directories. Mainly, QNX does not have an LLVM stack available currently, and consequently, some functions bound to the toolchains of the device compiler cannot pass the compilation. Besides, the OpenCL interfaces used in DPC++ is inconsistent with those defined in the legacy version of OpenCL header files on QNX. As a result, the plugin layer of DPC++ fails to map OpenCL APIs as SYCL interfaces correctly. Particularly, DPC++ may not be compatible with the internal OpenCL archive provided by QNX since DPC++ includes some OpenCL extensions exclusive to Intel. It is also worth noting that those errors are parts of SYCL core libraries. We could not continue compiling the rest of the SYCL libraries since the above errors terminated the compilation. Therefore, there may have many unknown problems to build DPC++ SYCL runtime on QNX.

To summarize, our tests on Linux demonstrated that it is essential for executing SYCL applications to correctly generate the device code for the selected backend and devices. It also turns out that device binaries only have impacts when executing and do not influence the compilation of an SYCL application. Although it may not be feasible to reuse DPC++ source code to build SYCL runtime on QNX due to compatibility issues, it is still possible to compile the host code and device code of an

Figure 5.1: The summary of vector and matrix multiplication.

SYCL application with different compilers built for different systems separately. We will give more discussions in the section 5.3

5.2 Testing SYCL-GTX

In our research, SYCL-GTX is the only one SYCL implementation that we can refactor and then apply to QNX. Therefore, we put the results of SYCL-GTX experiments on both systems together to illustrate their performance gap on difference operating

systems.

The overall results are shown in Figure 5.1. As we can see, warmup virtually takes no time compared to the other aspects during execution. However, on the one hand, the increased size of vectors leads to the decreased percentages of kernel compilation and linkage, while data movement and computation take more and more time. On the other hand, matrix multiplication inclines to be stable in computation whose percentage keeps steady even with bigger matrix sizes. Meanwhile, there is a prominent increment in the percentage of data movement on QNX. It is also worth noting that both OpenCL and SYCL-GTX have difficulty in matrix multiplication when feeding more and more data. For example, on Ubuntu, the usage of GPU reached 100% for `3000*3000` matrix in SYCL-GTX and `4000*4000` matrix in OpenCL, and feeding more data will not get any response in a long time.

5.2.1 Warmup

Both Ubuntu and QNX do not take much time on warmup (shown in Figure 5.2), and the warmup time only has slight fluctuation with the increased size of vectors and matrixes.

As the OpenCL we used on QNX is just an internal archive and not formally released, QNX has no dedicated optimizations on OpenCL, requiring more time to construct OpenCL and SYCL-GTX context than Ubuntu. Besides, the difference in system features between them may also result in a gap in warmup. However, despite the performance difference, their warmup time keeps under 3ms, significantly superior to the time used on kernel compilation and linkage (illustrated in Figure 5.1).

5.2.2 Kernel Compilation

Compared to warmup, both OpenCL and SYCL-GTX take more time on compiling kernels (shown in Figure 5.3). In particular, the performance of OpenCL and SYCL-GTX (about 150ms) on Ubuntu is almost twice that of QNX (about 300ms).

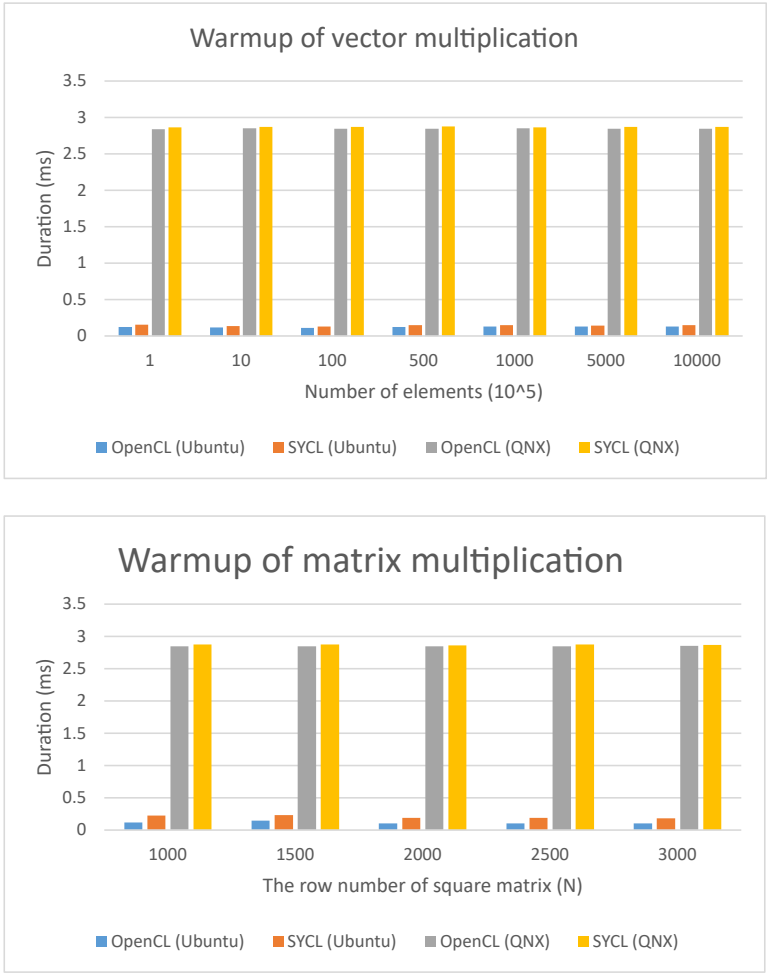


Figure 5.2: The warmup time of vector and matrix multiplication.

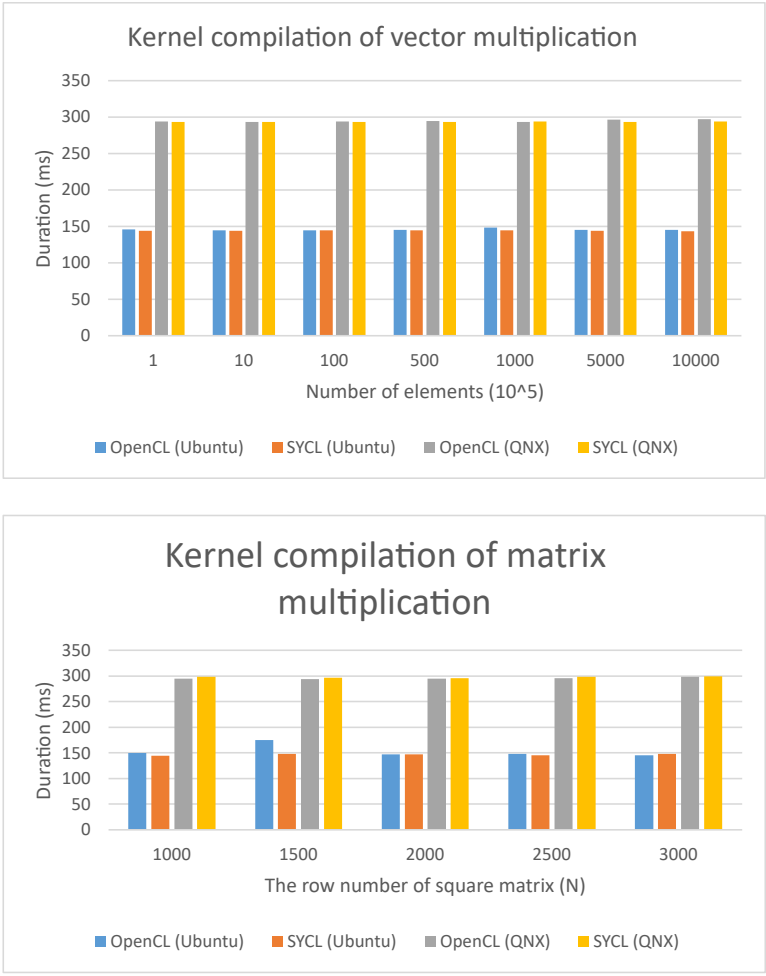


Figure 5.3: The kernel compilation time of vector and matrix multiplication.

One reason is that the OpenCL C compiler on QNX has not been optimized yet, while Linux currently has better optimizations on the compiler. Moreover, it may not be easy to support Clang/LLVM in QNX better, making it hard to optimize the OpenCL C compiler on QNX. Besides, the design of SYCL-GTX requires generating OpenCL C code line by line at runtime and then feeding it to the OpenCL C compiler. As a result, we can also observe a similar performance difference of SYCL-GTX kernel compilation on Ubuntu and QNX.

Besides, both OpenCL and SYCL kernels are compiled just in time, so it is inevitable to compile them repeatedly in our tests. Theoretically, we could compile SYCL kernels once and reuse them in subsequent tests. For example, the latest

DPC++ and ComputeCpp support compiling SYCL kernels as separate files and then loading them at runtime, and programmers can also use pre-compiled OpenCL kernels `clCreateProgramWithBinary()` to reduce the compilation time. Therefore, Comparing five performance aspects in Figure 5.1 illustrates that reducing the time on kernel compilation is a powerful method to optimize SYCL-GTX.

Nonetheless, SYCL-GTX only covers parts of features of legacy SYCL specification and does not provide such a mechanism to load pre-compiled kernels and thus reduce the total execution time significantly. From this perspective, it is possible to add more features to SYCL-GTX through code refactoring, thereby improving its performance on QNX. In section 5.2.6, we tried another way to compile SYCL and OpenCL kernels and found some surprising results, even though those kernels are still compiled just in time.

5.2.3 Linkage

In our latest experiments, we provide strict exception mechanism to guarantee the execution of SYCL applications. It turns out that both OpenCL and SYCL-GTX require enough time to link requisite OpenCL libraries and compiled objects on both systems (illustrated in Figure 5.4) and then emit the final executable. Since compilation is usually twisted with linkage in the workflow of a compiler architecture, the reason of the performance gap in the linkage between Ubuntu and QNX should be the same as we analyzed in the kernel compilation 5.2.2.

However, our new way to compile SYCL-GTX kernels could avoid linkage and directly emit the compiled kernels, thereby saving much time on kernel compilation and linkage.

5.2.4 Data Movement

As shown in Figure 5.5, the increased data volume requires more and more time to move data between the host and accelerators. Owing to the totally different ways to allocate and access memory between Linux and QNX, Figure 5.5 shows a momentous

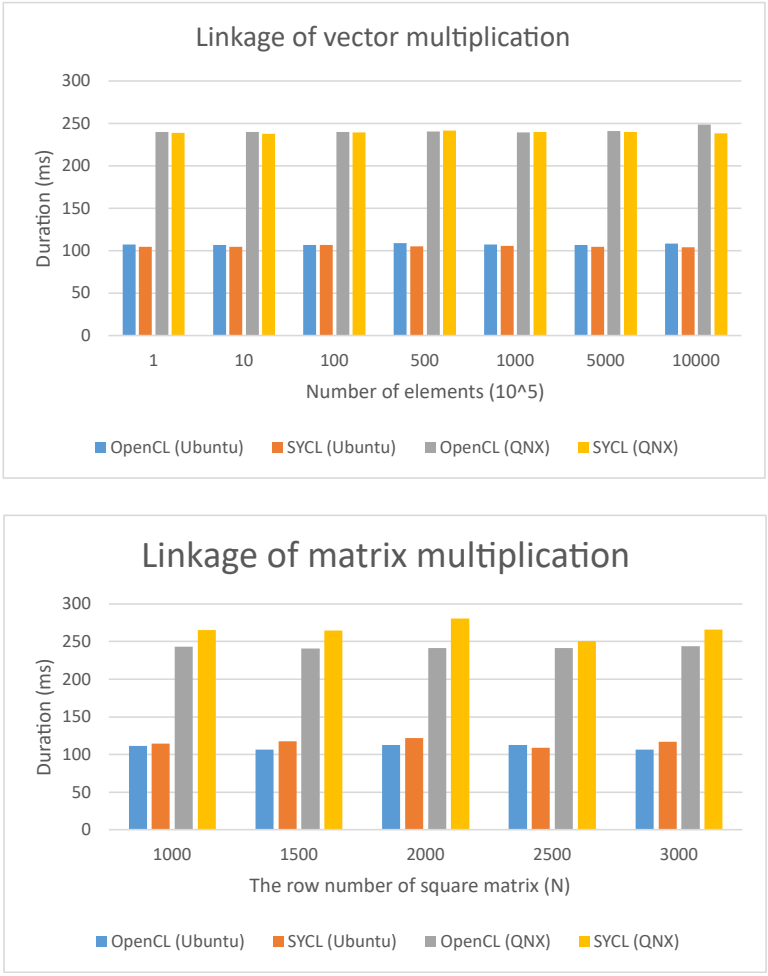


Figure 5.4: The linkage time of vector and matrix multiplication.

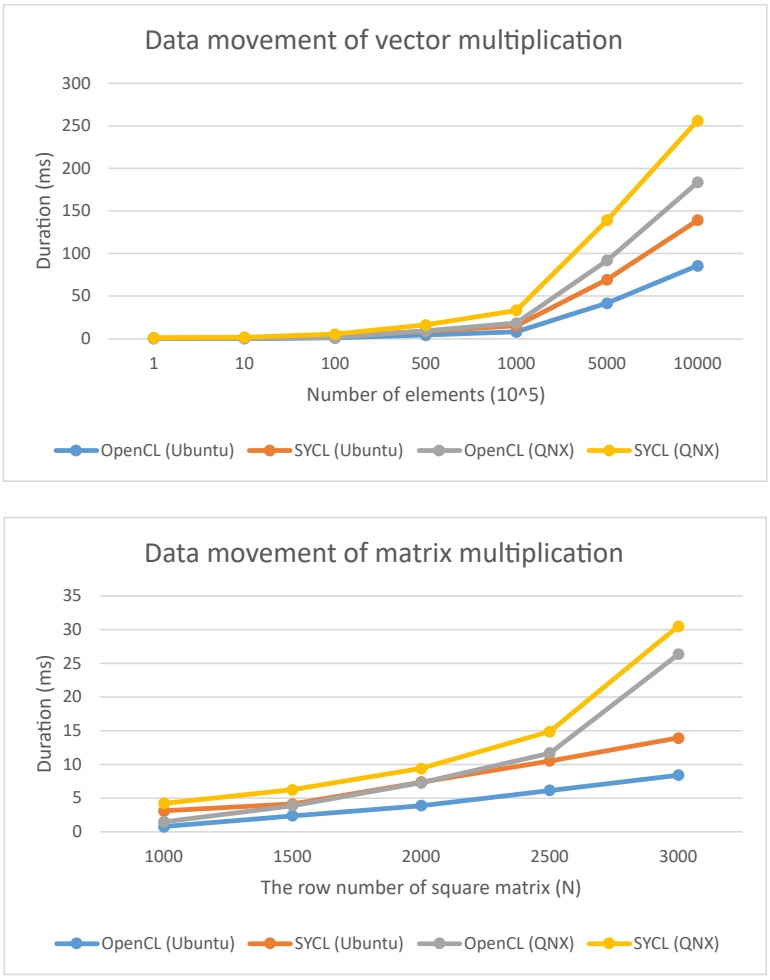


Figure 5.5: The data movement of vector and matrix multiplication.

gap in data movement between the two systems. The reason may be because integrated GPUs share the physical memory with the CPU and must compete for access memory with the CPU, and thus the gap in memory operations between Linux and QNX is enlarged. Consequently, we can see the data movement of SYCL-GTX and OpenCL encounters the bottleneck earlier on QNX than Ubuntu.

Generally, `clEnqueueWriteBuffer()` copies data from the host memory to the dedicated memory if OpenCL devices have dedicated device memories, but our machine only has one integrated Intel GPU that shares the memory with the CPU. So, instead of operating self-owned memory regions as a dedicated GPU does, the integrated GPU

must compete for memory access with the CPU. As a result, the gap in data movement between Ubuntu and QNX inflated with the rise of the number of elements. In particular, the execution of host code on the CPU also experienced significant interference due to the increased data volume on both systems. The situation may change if QNX and SYCL-GTX support connecting with dedicated GPUs, such as NVIDIA GPUs.

5.2.5 Computation

Concerning the computation time (see Figure 5.6), the figure for matrix multiplication is weird compared to that for vector multiplication. Unlike the trend in vector multiplication, matrix multiplication does not take increasing time on computing when feeding more data. We used `intel-gpu-tools` to monitor the usage of our GPU on Ubuntu. The maximum GPU usage for vector multiplication was approximately 24%, whereas matrix multiplication ate GPU fiercely until reaching 100% and no response. A possible reason is that the GPU had kept working in a state of full load, although it only indicated a quarter of usage at the beginning. And most time is used to access the host memory based on the virtual address space mapped by `clEnqueuWriteBuffer()`. We believe that it has a similar situation on QNX.

Figure 5.7 could be proof of our guess because the bigger size of vectors also shows a similar trend compared to matrix multiplication. When the size of vectors rises to 10^9, the float-point operations in vector multiplication is the same as matrix multiplication with the size of 1000*1000, as the computational complexity of matrix multiplication is $O(N^3)$. As we can see, the computation time of them (10^9 for vector vs. 1000 for matrix) is at the same level in Figure 5.6. If we feed more data in vector multiplication, the computation time only fluctuates slightly and maintains a similar level compared to matrix multiplication, but the data movement time enlarges further.

By our analysis, we tend to consider that SYCL-GTX introduces extra overhead during kernel execution, and its encapsulation of variables also results in more time accessing data than OpenCL.

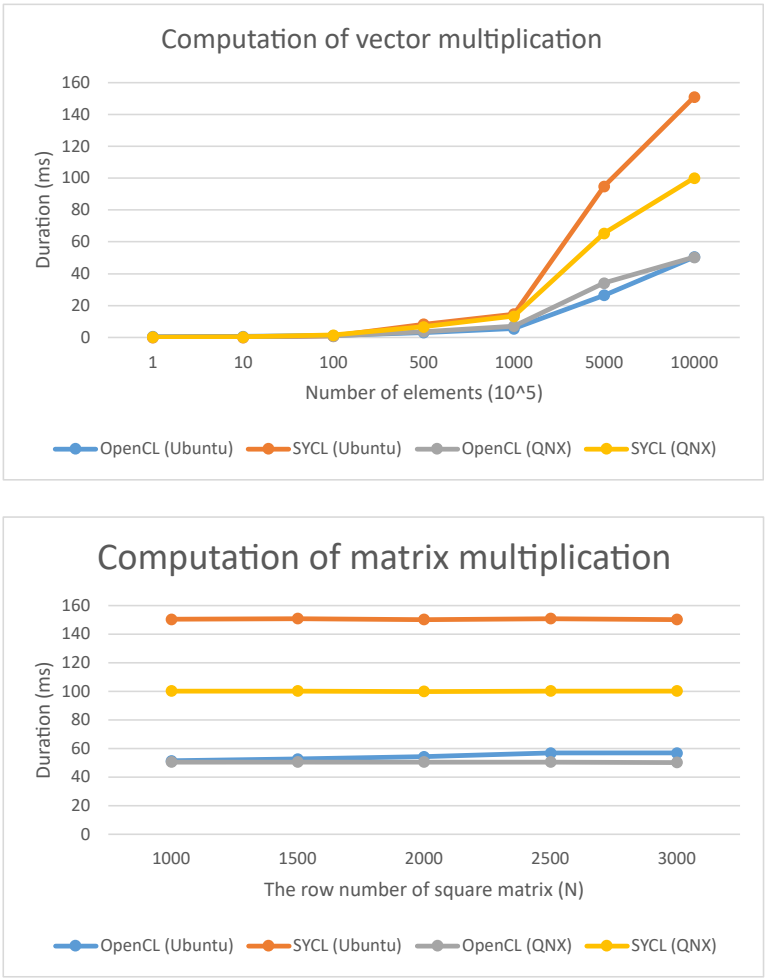


Figure 5.6: The computation time of vector and matrix multiplication.

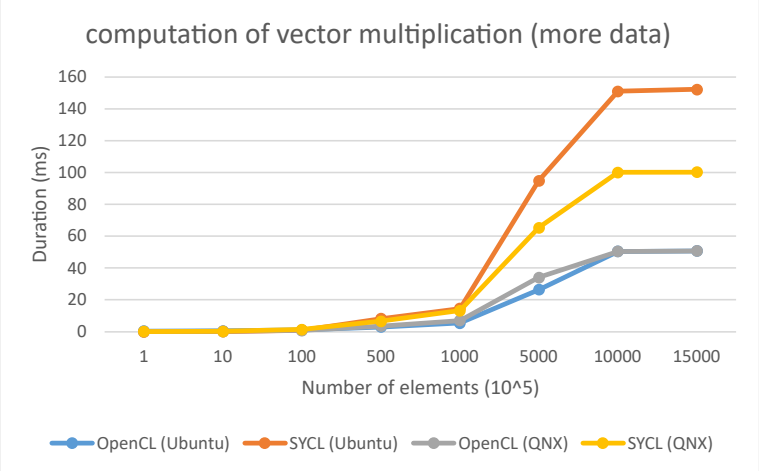


Figure 5.7: The computation time of vector multiplication (more data) on QNX.

However, we need to emphasize that although we somehow made the separation of the time on data movement and computation in SYCL-GTX, we cannot define the boundary of data movement and computation precisely. The same SYCL-GTX application even has a different execution order but returns correct calculations on both QNX and Ubuntu. For example, SYCL-GTX only takes one queue to access data and execute the kernel on Ubuntu, whereas it splits two sub-queues implicitly to complete all computing tasks on QNX. As a result, the computation time and data movement time shown in the figures should not be considered accurate data but approximate somewhat the changes and the gap between OpenCL and SYCL-GTX on both systems.

5.2.6 Another Way of Kernel Compilation and Linkage in SYCL-GTX

As the author replied[1], we could try another path to build OpenCL programs directly. Therefore, instead of compiling SYCL/OpenCL kernels in two-step calls: `clCompileProgram()` and `clLinkProgram()`, we replaced them with one OpenCL API: `clBuildProgram()` and created a new function `compile_once()` in SYCL-GTX. The new function somehow compiles kernels for OpenCL and SYCL-GTX more efficient

[1]The issue in github. See https://github.com/ProGTX/sycl-gtx/issues/24.

on QNX.

Using the new function, we found that the total compilation time (kernel compilation + linkage) on QNX somehow shrinks to about 0.149 ms (OpenCL) and about 0.317ms (SYCL-GTX) in both vector and matrix multiplication. However, this new function did not take effect on Ubuntu, and the total compilation time just decreased slightly. Therefore, to avoid the cause of capturing the compilation time wrongly, we average the total execution time and compare the results before and after using the new function (shown in Figure 5.8). The valid time includes warmup, data movement and computation before using the new function, while the total execution time after using the new function is regarded as the valid time since the compilation only takes a little time.

As we can see, if we remove kernel compilation and linkage, the valid time is similar. Thus, it turns out that the one-step compilation reduces the total execution time dramatically on QNX through diminishing kernel compilation and linkage. However, we have not figured out why it happens and why it only happens to QNX, and it requires more research to verify the correctness of such a case.

To summarize, on the one hand, float-point multiplication of SYCL-GTX in vector and matrix turns out to be viable to port SYCL to QNX, although the differences at the system level lead to a significant performance gap in data movement between Ubuntu and QNX. Besides, the design of SYCL-GTX also requires more time on data movement and computation compared to OpenCL. Nevertheless, on the other hand, it is surprising that the new way to compile SYCL/OpenCL kernels shrinks the total execution time and thus makes QNX outperform Ubuntu. It requires more research to figure out why it happens and verify its correctness. Even then, future optimizations should focus on reducing the compilation time of SYCL-GTX and OpenCL on QNX and enabling reusing the compiled kernels.

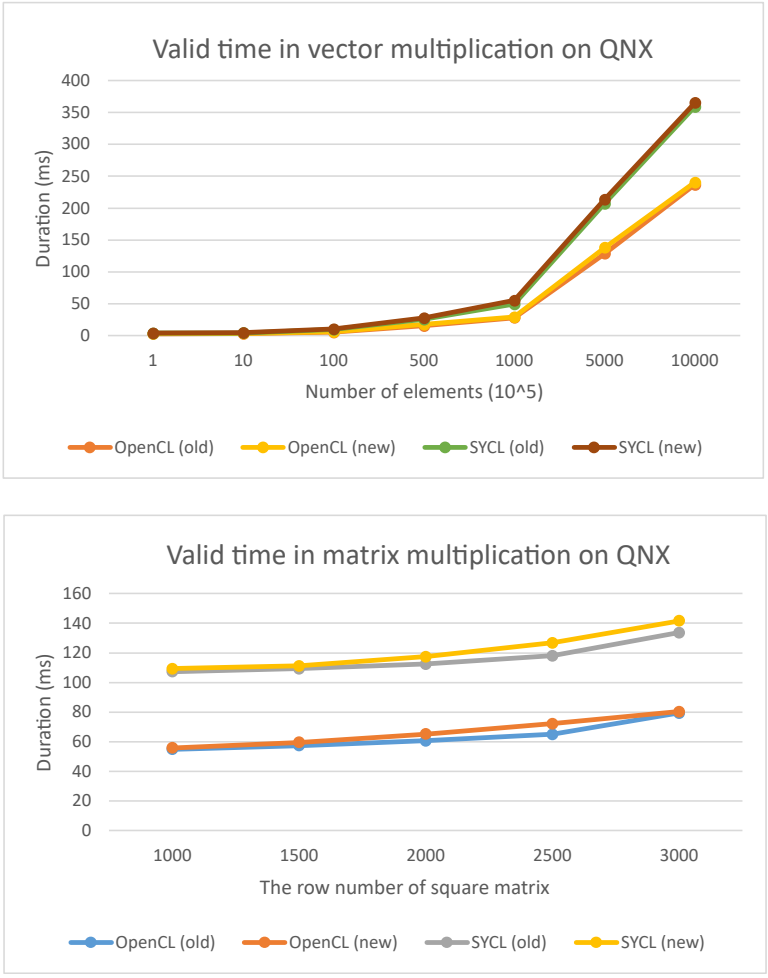


Figure 5.8: The valid execution time of vector and matrix multiplication.

5.3 Discussions about SYCL Device Compiler on QNX

As we have analyzed, SYCL-GTX is not mature and has problems on performance. Therefore, we compare the compilation mechanisms of the two mainstream SYCL implementations (DPC++ and ComputeCpp) and consider that reusing existing SYCL implementations may benefit QNX to provide a product-ready SYCL environment.

On the one hand, apart from SYCL-GTX that compiles SYCL kernels just in time by the OpenCL runtime, other mainstream SYCL implementations require an SYCL device compiler to generate device binaries from SYCL kernels. As the sole compiler on QNX, QCC cannot compile device code and generate acceptable intermediate representations or device binaries, and consequently, users cannot compile SYCL applications correctly only using QCC. On the other hand, it is possible to use existing SYCL device compilers to generate device binaries and wrap them with the host code using QCC. Therefore, we research how existing SYCL implementations generate and integrate device binaries to output the final executable and see if this strategy is viable.

In DPC++, compiling and wrapping the device code are shown in Figure 5.9. If SYCL kernels execute on OpenCL devices, it requires `clang-offload-bundler` to bundle some system-dependent libraries with the generated IR, making it impossible for QCC to wrap the generated IR directly. By comparison, the CUDA backend only utilizes the toolchains of the CUDA driver, isolating the generated IR from system-dependent environments. However, there is little information about accessing and interacting with the CUDA driver and devices on QNX. As a result, DPC++ may not be an appropriate way to compile SYCL kernels.

The whole process of compiling SYCL kernels in ComputeCPP is similar to that in DPC++, but ComputeCPP emits the compiled device code and integrated header files as an SYCL stub file (`.sycl`). The generated SPIRV IR is stringified, and the integrated header is like standard C/C++ header files. In our tests, the host code can include the SYCL stub file directly without any modification, and then GCC can link the entire source code with the SYCL runtime (`libComputeCpp.so`) and output

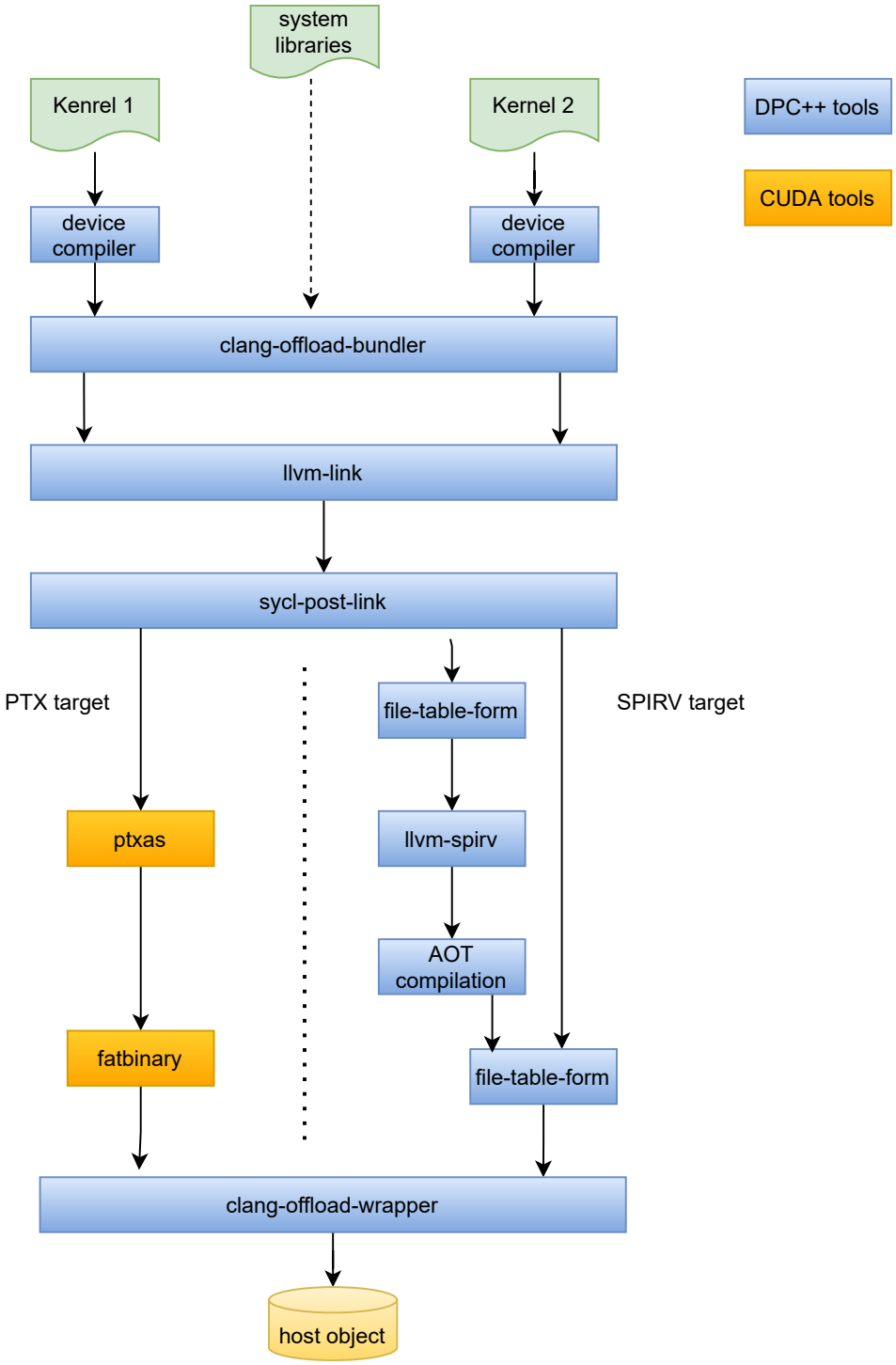


Figure 5.9: The workflow of compiling DPC++ applications.

the correct executable.

To summarize, ComputeCPP may be more suitable to generate device binaries for QNX than DPC++. As we summarized in DPc++ experiments (see section 5.1, the core feature of SYCL is to compile SYCL kernels correctly in a host environment and enables executing the compiled kernels in different accelerators. By comparison, the host environment can affect ComputeCpp less than DPC++ and allow an isolated compilation process in ComputeCpp. Therefore, it may be a viable solution for QNX to work with ComputeCpp and then build commercial-grade SYCL toolchains.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Based on our experiments and research, we can answer the research questions:

- **RQ1**: Most SYCL implementations are not compatible with QNX since they require various toolchains that QNX could hardly provide currently. Only SYCL-GTX, as a high-level abstraction of OpenCL, can work on QNX without rewriting source code significantly.
- **RQ2**: Although SYCL implementations may cover different features of SYCL, they are conformant to the SYCL specification that is backward-compatible. Moreover, considering SYCL-GTX, it is limited to OpenCL devices and cannot support discrete accelerators (e.g., NVIDIA GPUs). Besides, SYCL-GTX is not maintained actively any more, so it could hardly go through some new features in the latest SYCL specification.
- **RQ3**: Generally, SYCL-GTX does not perform well on QNX than Ubuntu. The major performance differences is due to kernel compilation and data movement, while the time on computation has little difference in both operating systems. Therefore, by optimizing the SYCL/OpenCL compiler, we could reduce the time on kernel compilation. With respect to the data movement, the differences of QNX and Linux may cause performance degradation, so it requires more

research to explore possible solutions. For example, we could accelerate SYCL kernels on discrete GPUs to compare the impact of integrated and discrete GPUs on both systems and then find possible optimizations.

In summary, our experiments based on SYCL-GTX illustrate the feasibility of having SYCL on QNX. However, at the initial stage, our results only can demonstrate that there is a significant gap in SYCL-GTX performance between QNX and Linux, but we cannot arbitrarily conclude SYCL on QNX has no potential to surpass SYCL on Linux through dedicated optimizations. Besides, by analyzing the experiment results, we consider that improving kernel compilation and data movement should be the next stage to enhance SYCL on QNX. Moreover, the new way to compile SYCL kernels on SYCL-GTX also demonstrates the importance to optimize kernel compilation, although we need to take more research why it does not happen to Ubuntu.

6.2 Future Work

Since experiment conditions are not mature and there are not abundant technical references about SYCL on QNX, it needs to take further research to evaluate the performance and significance of SYCL on QNX. First of all, better optimizations on OpenCL are required to explore the potential performance of SYCL on QNX. Then, future experiments could focus on multi-dimensional data (e.g., tensor) and advanced machine learning algorithms to monitor how it perform well. Specially, to maintain the real-time of QNX, we will also try more methods to decrease the time complexity of advanced algorithms and thereby exploit the benefits of SYCL on QNX.

Besides, it is also essential to enable SYCL-GTX to interact with dedicated accelerators or choose a more mature SYCL implementation supporting more backends, as integrated GPUs can affect the CPU execution significantly with a big data volume and thus may make SYCL adverse to the security and real-time of QNX.

List of References

[1] Y. Arafa, A. A. Badawy, G. Chennupati, N. Santhi, and S. J. Eidenbenz, "Instructions' latencies characterization for NVIDIA gpgpus," *CoRR*, vol. abs/1905.08778, 2019.

[2] M. Zahran, *Heterogeneous Computing: Hardware and Software Perspectives*. New York, NY, USA: Association for Computing Machinery, 2019.

[3] S. Kang, H. J. Choi, C. H. Kim, S. W. Chung, D. Kwon, and J. C. Na, "Exploration of cpu/gpu co-execution: From the perspective of performance, energy, and temperature," in *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, (New York, NY, USA), p. 38–43, 2011.

[4] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communication management and optimization," *SIGPLAN Not.*, vol. 46, p. 142–151, June 2011.

[5] N. P. J. et al., "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, p. 1–12, June 2017.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[7] E. S. Ruud van der Pas and C. Terboven, *Heterogeneous Architectures*, ch. 6, pp. 253–329. MIT Press, 2017.

[8] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot, "Computer architectures for autonomous driving," *Computer*, vol. 50, no. 8, pp. 18–25, 2017.

[9] L. Liu, S. Lu, R. Zhong, B. Wu, Y. Yao, Q. Zhang, and W. Shi, "Computing systems for autonomous driving: State-of-the-art and challenges," *CoRR*, vol. abs/2009.14349, 2020.

[10] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," *SIGPLAN Not.*, vol. 53, p. 751–766, Mar. 2018.

[11] Michael, A. Wong, M. Richards, R. Rovatsou, and Reyes, "Khronos's opencl sycl to support heterogeneous devices for c++." http://www.open-std.org/jtc1/

sc22/wg21/docs/papers/2016/p0236r0.pdf, 2016. [Online; Accessed 15-June-2021].

[12] Ronan, J. Keryell, and Falcou, "A c++ standard library class to qualify data accesses." http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0367r0.pdf, 2016. [Online; Accessed 15-June-2021].

[13] S. Lin, K. Chen, and A. Karimoddini, "SD-VEC: software-defined vehicular edge computing with ultra-low latency," *CoRR*, vol. abs/2103.14225, 2021.

[14] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, 2019.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, p. 451–490, Oct. 1991.

[16] L. Foundation, "LLVM official website." https://llvm.org/. [Online; Accessed by 1-March-2021].

[17] Microsoft, "An llvm based msil compiler." https://github.com/dotnet/llilc, 2015.

[18] R. Foundation, "The rust programming language." https://github.com/rust-lang/rust, 2021.

[19] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, 2004.

[20] L. Foundation, "LLVM Language Reference Manual." https://llvm.org/docs/LangRef.html#abstract. [Online; Accessed by 1-March-2021].

[21] C. Lattner and V. Adve, "The llvm instruction set and compilation strategy." https://llvm.org/pubs/2002-08-09-LLVMCompilationStrategy.pdf, 2003. [Online; Accessed by by 9-August 2021].

[22] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, pp. 467–468. Pearson India Education Services, 2015.

[23] A. Zakai, "Emscripten: An llvm-to-javascript compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, (New York, NY, USA), p. 301–312, Association for Computing Machinery, 2011.

[24] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," *SIGPLAN Not.*, vol. 52, p. 185–200, June 2017.

[25] K. Groups, "Llvm/spir-v bi-directional translator." https://github.com/KhronosGroup/SPIRV-LLVM-Translator, 2021.

[26] C. Lattner, "The design of LLVM." http://aosabook.org/en/llvm.html. [Online; Accessed by 1-March-2021].

[27] G. team, "Ssa fro trees." https://gcc.gnu.org/projects/tree-ssa/. [Online; Accessed by by 31-May-2021.

[28] J. Merrill, "Generic and gimple: A new tree representation for entire functions jason," *2003 GCC Developers' Summit*, p. 171–180, Jun 2005.

[29] D. Novillo, "Tree ssa a new optimization infrastructure for gcc." https://gcc.gnu.org/projects/tree-ssa/, 2003. [Online; Accessed by 28-May-2021].

[30] L. Foundation, "LLVM Bitcode File Format." https://llvm.org/docs/BitCodeFormat.html. [Online; Accessed by 1-March-2021].

[31] K. G. O. W. G. S. subgroup, "Spir 1.0 specification for opencl." https://www.khronos.org/registry/SPIR/specs/spir_spec-1.0-provisional.pdf, 2012. [Online; Accessed by by 31-May-2021].

[32] M. Bailey, "Introduction to the vulkan® computer graphics api," in *SIGGRAPH Asia 2019 Courses*, SA '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[33] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th ed., 2013.

[34] R. K. John Kessenich, Boaz Ouriel, "SPIR-V Specification." https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html. [Online; Accessed by 1-March-2021].

[35] G. Inc., "A prototype compiler for a subset of opencl c to vulkan compute shaders." https://github.com/google/clspv, 2021.

[36] A. Inc., "Metal overview." https://developer.apple.com/documentation/metal/. [Online; Accessed by 31-May-2021].

[37] K. Group, "A tool designed for parsing and converting spir-v to other shader languages.." https://github.com/KhronosGroup/SPIRV-Cross. [Online; Accessed by 31-May-2021].

[38] A. Inc., "Metal shading language specification v2.3." https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf. [Online; Accessed by 31-May-2021].

[39] N. Inc., "Parallel thread execution isa v7.3." https://docs.nvidia.com/cuda/pdf/ptx_isa_7.3.pdf, 2021. [Online; Accessed by 31-May-2021].

[40] G. Martinez, M. Gardner, and W.-c. Feng, "Cu2cl: A cuda-to-opencl translator for multi- and many-core architectures," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pp. 300–307, 2011.

[41] J. Kim, T. T. Dao, J. Jung, J. Joo, and J. Lee, "Bridging opencl and cuda: a comparative analysis and translation," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2015.

[42] H. Perkins, "CUDA-on-CL: a compiler and runtime for running NVIDIA® CUDA™ C++11 applications on OpenCL™ 1.2 Devices," *IWOCL 2017: 5th International Workshop on OpenCL*, 2017.

[43] T. Stauber and P. Sommerlad, "ReSYCLator: Transforming CUDA C++ source code into SYCL," *IWOCL'19: International Workshop on OpenCL*, 2019.

[44] M. Goli, K. Narasimhan, R. Reyes, B. Tracy, D. Soutar, S. Georgiev, E. M. Fomenko, and E. Chereshnev, "Towards Cross-Platform Performance Portability of DNN Models using SYCL," *Proceedings of P3HPC 2020: International Workshop on Performance, Portability, and Productivity in HPC, Held in conjunction with SC 2020: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 25–35, 2020.

[45] R. Burns, J. Lawson, D. McBain, and D. Soutar, "Accelerated neural networks on opencl devices using sycl-dnn," *Proceedings of the International Workshop on OpenCL*, May 2019.

[46] J. I. Aliaga, R. Reyes, and M. Goli, "Sycl-blas: Leveraging expression trees for linear algebra," in *Proceedings of the 5th International Workshop on OpenCL*, IWOCL 2017, (New York, NY, USA), Association for Computing Machinery, 2017.

[47] J. Lawson, "Performance portability through machine learning guided kernel selection in SYCL libraries," *CoRR*, vol. abs/2008.13145, 2020.

[48] J. Lawson, "Towards automated kernel selection in machine learning systems: A SYCL case study," *CoRR*, vol. abs/2003.06795, 2020.

[49] R. Nozal and J. L. Bosque, "Exploiting co-execution with oneapi: heterogeneity from a modern perspective," *CoRR*, vol. abs/2106.01726, 2021.

[50] L. Liu, S. Liu, and W. Shi, "4c: A computation, communication, and control co-design framework for cavs," *CoRR*, vol. abs/2107.01142, 2021.

[51] A. Doumoulakis, R. Keryell, and K. O'Brien, "SYCL C++ and OpenCL interoperability experimentation with triSYCL," vol. Part F127755, may 2017.

[52] A. Doumoulakis, R. Keryell, and K. O'Brien, "Sycl c++ and opencl interoperability experimentation with trisycl," in *Proceedings of the 5th International Workshop on OpenCL*, IWOCL 2017, (New York, NY, USA), Association for Computing Machinery, 2017.

[53] T. Deakin and S. McIntosh-Smith, "Evaluating the performance of hpc-style sycl applications," in *Proceedings of the International Workshop on OpenCL*, IWOCL '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[54] R. Keryell and L.-Y. Yu, "Early experiments using sycl single-source modern c++ on xilinx fpga: Extended abstract of technical presentation," in *Proceedings of the International Workshop on OpenCL*, IWOCL '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[55] T. Sabino and M. Goli, "Toward performance portability of highly parametrizable trsm algorithm using sycl," in *International Workshop on OpenCL*, IWOCL'21, (New York, NY, USA), Association for Computing Machinery, 2021.

[56] J. W. Lawson, M. Goli, D. McBain, D. Soutar, and L. Sugy, "Cross-platform performance portability using highly parametrized SYCL kernels," *CoRR*, vol. abs/1904.05347, 2019.

[57] C. Nugteren, "Clblast," *Proceedings of the International Workshop on OpenCL*, May 2018.

[58] K. Group, "The khronos official opencl icd loader." https://github.com/KhronosGroup/OpenCL-ICD-Loader. [Online; Accessed by 31-May-2021].

[59] N. Corporation, "Using separate compilation in cuda." https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#using-separate-compilation-in-cuda. [Online; Accessed by 31-May-2021].

[60] L. Foundation, "User guide for nvptx back-end." https://llvm.org/docs/NVPTXUsage.html. [Online; Accessed by 31-May-2021].

[61] K. G. Inc, "SYCL official website." https://sycl.tech/. [Online; Accessed by 1-March-2021].

[62] K. O. W. G. S. subgroup, "The sycl 1.2 specification." https://www.khronos.org/registry/SYCL/specs/sycl-1.2.pdf, 2015. [Online; Accessed 11-July-2021].

[63] K. O. W. G. S. subgroup, "The sycl 2020 specification." https://www.khronos.org/registry/SYCL/specs/sycl-1.2.pdf, 2021. [Online; Accessed 11-July-2021].

[64] B. Ashbaugh, A. Bader, J. Brodman, J. Hammond, M. Kinsner, J. Pennycook, R. Schulz, and J. Sewall, "Data parallel c++: Enhancing sycl through extensions for productivity and performance," in *Proceedings of the International Workshop on OpenCL*, IWOCL '20, (New York, NY, USA), 2020.

[65] C. S. Ltd., "ComputeCpp Guide." https://developer.codeplay.com/products/computecpp/ce/guides/. [Online; Accessed by 1-March-2021].

[66] K. S. working Group, "Sycl integrate opencl devices with modern c++ v1.2.1." https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf, 2020. [Online; Accessed 15-June-2021].

[67] K. S. wWorking Group, "Sycl specification (revision 3)." https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf. [Online; Accessed 15-June-2021].

[68] P. Zuzek, "An overview of sycl-gtx," *PPoPP 2016, 1st SYCL Programming Workshop*, 2016.

[69] P. Zuzek, "Sycl gtx git reporsitory." https://github.com/ProGTX/sycl-gtx. [Online; Accessed 10-August-2021].

Appendix A

Code Lists

Listing A.1: Vector Addition in CUDA

```
#include <stdio.h>
#include <cuda_runtime.h>

float *generate(int size);
//CUDA kernel
__global__ void cudaAddition(float* a, float* b, float* c, int nums) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < nums) {
        c[i] = a[i] + b[i];
    }
}

int main() {
    int length = 10;
    size_t size = length * sizeof(float);
    int threadsPerBlock = 256;
    int blocksPerGrid = (length + threadsPerBlock - 1) / threadsPerBlock;

    //generate host data
    float *host_a = generate(size);
    float *host_b = generate(size);
    float *host_c = generate(size);
```

```
23      //declare pointers to device data;
24      float device_a = nullptr;
25      float device_b = nullptr;
26      float device_c = nullptr;
27      // memory allocation on devices
28      cudaMalloc((void **)&device_a, size);
29      cudaMalloc((void **)&device_b, size);
30      cudaMalloc((void **)&device_c, size);
31      // copy host data to devices
32      cudaMemcpy(device_a, host_a, size, cudaMemcpyHostToDevice);
33      cudaMemcpy(device_b, host_b, size, cudaMemcpyHostToDevice);
34      // [ass kernel parameters and execute the CUDA kernel
35      cudaAddition<<<blocksPerGrid, threadsPerBlock>>>(device_a, device_b,
            device_c, length);
36      //copy the result to the host memory
37      cudaMemcpy(host_c, device_c, cudaMemcpyDeviceToHost);
38      // release unused resources
39      cudaFree(device_a);
40      cudaFree(device_b);
41      cudaFree(device_c);
42      free(host_a);
43      free(host_b);
44      free(host_c);
45  }
```

Listing A.2: Vector Addition in OpenCL

```
1   #include <stdio.h>
2   #include <math.h>
3
4   #define MAX_SOURCE_SIZE (0x100000)
5
6   #ifdef MAC
7   #include <OpenCL/cl.h>
8   #else
9   #include <CL/cl.h>
10  #endif
```

```
11
12 float *generate(int size);
13
14 int main(int argc, char* argv[]) {
15     unsigned int length = 10;
16     size_t bytes = length * sizeof(float);
17     size_t local_size = 64;
18     size_t global_size = math.ceil(length / local_size) * local_size;
19     // generate host data
20     float *host_a = generate(length);
21     float *host_b = generate(length);
22     float *host_c = generate(length);
23     // declare pointers to device data;
24     cl_mem device_a;
25     cl_mem device_b;
26     cl_mem device_c;
27     // declare OpenCL objects to construct a program
28     cl_platform_id platform;
29     cl_device_id device_id;
30     cl_context context;
31     cl_command_queue queue;
32     cl_program program;
33     cl_kernel kernel;
34     // read the kernel file as bytes
35     FILE *kernelFile = fopen("openclKernel.cl", "r");
36     char *kernelSource = (char *)malloc(MAX_SOURCE_SIZE);
37     size_t kernelSize = fread(kernelSource, 1, MAX_SOURCE_SIZE,
           kernelFile);
38     fclose(kernelFile);
39     // acquire computing resources from the OpenCL runtime
40     clGetPlatformIDs(1, &platform, NULL);
41     clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
42     context = clCreateContext(0, 1, &device_id, NULL, NULL);
43     queue = clCreateCommandQueue(context, device_id, 0);
44     program = clCreateProgramWithSource(context, 1, (const char
           **)&kernelSource, NULL);
45     clBuildProgram(program, "openclKernel");
```

```
46      kernel = clCreateKernel(program, "openclKernel");
47      // create tasks to copy data to devices
48      device_a = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL
            ,NULL);
49      device_b = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL
            ,NULL);
50      device_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL
            ,NULL);
51      clEnqueueWriteBuffer(queue, device_a, CL_TRUE, 0, bytes, host_a, 0,
            NULL, NULL);
52      clEnqueueWriteBuffer(queue, device_b, CL_TRUE, 0, bytes, host_b, 0,
            NULL, NULL);
53      // set kernel parameters for devices
54      clSetKernelArg(kernel, 0, sizeof(cl_mem), &device_a);
55      clSetKernelArg(kernel, 1, sizeof(cl_mem), &device_b);
56      clSetKernelArg(kernel, 2, sizeof(cl_mem), &device_c);
57      clSetKernelArg(kernel, 3, sizeof(unsigned int), &length);
58      // execute the kernel and finish
59      clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size,
            &local_size, 0, NULL, NULL);
60      clFinish(queue);
61      // copy the result back the host memory
62      clEnqueueReadBuffer(queue, device_c, CL_TRUE, 0, bytes, host_c, 0,
            NULL, NULL);
63      // release unused resources
64      clReleaseMemObject(device_a);
65      clReleaseMemObject(device_b);
66      clReleaseMemObject(device_c);
67      clReleaseProgram(program);
68      clReleaseKernel(kernel);
69      clReleaseCommandQueue(queue);
70      clReleaseContext(context);
71      free(host_a);
72      free(host_b);
73      free(host_c);
74  }
75
```

```
// "openclKernel.cl"
__kernel void openclKernel(__global const float *a, __global const float
    *b, __global float *c) {
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Listing A.3: Vector Addition in SYCL

```
#include <Cl/sycl.hpp>

using namespace cl::sycl;

float *generate(int size);

int main() {
    int length = 10;
    size_t size = length * sizeof(float);
    // create a SYCL queue to schedule computing tasks
    queue Q;
    // generate host data
    float *host_a = generate(length);
    float *host_b = generate(length);
    // allocate device memory
    float *device_a = malloc_device(size, Q);
    float *device_b = malloc_device(size, Q);
    float *shared_c = malloc_device(size, Q);
    // copy host data to devices
    auto event1 = Q.memcpy(device_a, host_a, size);
    auto event2 = Q.memcpy(device_b, host_b, size);

    Q.submit([&](handler &cgh) {
        // ensure data copy has completed
        cgh.depends_on({event1, event2});
        // execute the kernel as a C++ lambda function
        cgh.parallel_for(range<1>(length), [=](id<1> i) {
            shared_c[i] = device_a[i] + device_b[i];
```

```
29          });
30      });
31      // wait SYCL runtime synchronize the result
32      // and copy the result to the host memory (shared_c) implicitly
33      Q.wait();
34      // release unused resources
35      free(device_a, Q);
36      free(device_a, Q);
37      free(device_a, Q);
38      free(host_a);
39      free(host_b);
40  }
```

Listing A.4: Vector Addition in SYCL-GTX

```
1   #include <Cl/sycl.hpp>
2
3   using namespace cl::sycl;
4
5   float* generate(int size);
6
7   class Addition;
8
9   int main() {
10      int length = 10;
11      // create a SYCL queue to schedule computing tasks
12      queue Q;
13      // generate host data
14      float* host_a = generate(length);
15      float* host_b = generate(length);
16      float* host_c = generate(length);
17      // create 1-dimension buffer
18      buffer<float, 1> buffer_a{host_a};
19      buffer<float, 1> buffer_b{host_b};
20      buffer<float, 1> buffer_c{host_c};
21
22      Q.submit([&](handler &cgh) {
```

```
23          // enable the accelerator to access the buffers
24          auto accessor_a = buffer_a.get_access<access::mode::read>(cgh);
25          auto accessor_b = buffer_b.get_access<access::mode::read>(cgh);
26          auto accessor_c = buffer_c.get_access<access::mode::write>(cgh);
27
28          // execute the kernel as a C++ lambda function
29          cgh.parallel_for<class Addition>(range<1>(length), [=](id<1> i) {
30              accessor_c[i] = accessor_a[i] + accessor_b[i];
31          });
32      });
33      // waiting for writing the result back host_c vector
34      Q.wait();
35  }
```

Appendix B

Nomenclature Difference in CUDA, OpenCL and SYCL

Table B.1: Runtime API equivalence

CUDA	SYCL	OpenCL
...	*nd_range* class	*global_work_size* *local_work_size*
Kernel Name... ()	*queue::submit()*	*clCreateProgramWithSource/Binary()* *clBuildProgram()* *clCreateKernel()* *clCreateKernelArg()* *clEnqueueNDRangeKernel()*

Table B.2: Device API equivalence for kernel specifiers

CUDA	SYCL	OpenCL
__golbal__	N/A	*__kernel*
__device__	N/A	N/A
__constant__ variable declaration	N/A	*__constant* variable declaration
__device__ variable declaration	N/A	*__global* variable declaration
__shared__ variable declaration	N/A	*__local* variable declaration

Table B.3: Indexing equivalence

CUDA	SYCL	OpenCL
N/A	*nd_item* class	N/A
gridDim.{x, y, z}	*nd_item::get_num_group({0, 1, 2})*	*get_num_group({0, 1, 2})*
blockDim.{x, y, z}	*nd_item::get_local_range({0, 1, 2})*	*get_local_size({0, 1, 2})*
blockIdx.{x, y, z}	*nd_item::get_group({0, 1, 2})*	*get_group_id({0, 1, 2})*
threadIdx.{x, y, z}	*nd_item::get_local_id({0, 1, 2})*	*get_local_id({0, 1, 2})*
N/A	*nd_item::get_global_id({0, 1, 2})*	*get_global_id({0, 1, 2})*