A Method for Compiling Domain-Specific Languages into SYCL through Abstract Syntax Trees

Cao Jiawei[1]

[1]Xi'an Jiaotong University, Xi'an, 710000, China

Abstract

The escalating demand for computing resources, particularly in the realm of artificial intelligence (AI), necessitates efficient utilization of heterogeneous parallel systems. This study focuses on compiling domain-specific languages, specifically data-centric computation models, into SYCL for heterogeneous many-core systems. SYCL, based on C++17, offers a unified programming model for various hardware accelerators, promoting code reusability across different architectures. Leveraging SYCL's cross-hardware compatibility and performance optimization capabilities, this project aims to enhance programming efficiency and performance on diverse hardware backends. Through the translation of domain-specific languages into SYCL (DPC++), developers can harness the simplicity and usability of domain-specific languages while achieving high-performance parallel computing. This approach addresses the challenges of complex programming interfaces and poor program portability across heterogeneous systems. By enabling domain-specific languages to run in parallel on heterogeneous systems, this research contributes to advancing the development of heterogeneous computing systems and providing programmers with more flexible and efficient programming tools. The significance of this work lies in its potential to facilitate broader application scenarios and higher execution efficiency, ultimately promoting the widespread adoption of domain-specific languages and driving innovation in parallel computing.

Keywords: SYCL, Domain-Specific Languages, Abstract Syntax Trees.

1. Introduction

The rapid advancement of artificial intelligence (AI) technology and the continuous growth in its application demands have led to increasingly substantial requirements for computing resources, alongside various existing computationally intensive disciplines. In response to these demands, hardware technologies have continuously evolved, with various heterogeneous processors, notably GPUs, spearheading breakthroughs in parameters such as core

counts and floating-point computation speeds. Particularly in recent years, the AI field has witnessed the emergence of a series of massive models, often comprising billions of parameters. The training of these models often necessitates complex computing systems involving multiple nodes and devices to provide computational power. Consequently, the necessity for heterogeneous parallel systems research has escalated.

In these supercomputing systems, coprocessors or heterogeneous accelerators are often utilized as their acceleration devices. In the latest (November 2023) TOP500 list of high-performance computing systems, nine out of the top ten supercomputing systems utilize coprocessors or heterogeneous accelerators as their acceleration devices, with NVIDIA's GPUs being used in seven of these supercomputers. Simultaneously, to meet the diverse needs of different higher-level applications, the variety of architectures for heterogeneous systems continues to expand. These systems leverage coprocessors and accelerators to execute computing tasks, thereby enhancing computational efficiency and performance.

There are four methods for constructing such systems: single-node single-device, single-node multiple-devices, multiple-node single-device, and multiple-node multiple-devices. The simplest and most common among these systems is the single-node single-device configuration (e.g., CPU+GPU setups in personal computers), while the more complex method involves multiple-node multiple-device parallel systems (e.g., supercomputing systems).

The utilization of different architectural heterogeneous systems spawned by various higher-level applications has revealed significant issues. The massive parallelism of heterogeneous systems and the differences between architectures have made programming interfaces overly complex and inconsistent across different heterogeneous systems. Consequently, parallel programming on heterogeneous systems not only becomes challenging to learn and optimize for efficiency but also suffers from poor program portability between different heterogeneous systems. This problem greatly restricts the value of parallel programming.

Therefore, as a leading industry player, Intel introduced the vision of OneAPI during the Architecture Day in 2018, swiftly realizing it in the subsequent years. OneAPI comprises a set of unified application programming interfaces that can be used for various computing accelerators (coprocessors), including GPUs, AI accelerators, and field-programmable gate arrays (FPGAs). Its aim is to eliminate the need for developers to maintain separate codebases, multiple programming languages, tools, and workflows for each architecture. In fact, prior to this, the Khronos Group had publicly proposed a programming model called SYCL in March 2014, for similar purposes.

SYCL is an advanced programming model designed to improve programming efficiency on various hardware accelerators. The current version of SYCL 2020 is based on a pure C++17 single-source embedded domain-specific language (DSL), with several implementations available. Among these, the most comprehensive and outstanding is Intel's OneAPI DPC++ compiler. DPC++ is built on top of the Khronos Group's SYCL spe-

cification and aims to enable developers to reuse code across hardware targets (CPUs and accelerators such as GPUs and FPGAs), as well as to customize optimizations for specific accelerators. DPC++ includes C++17 and SYCL language features and integrates open-source community extensions to make SYCL easier to use.

Domain-specific languages are languages tailored to specific application domains.(1) Compared to general-purpose programming languages used in their application domains, they offer significant benefits in terms of expressiveness and usability. DSLs come in various forms, from widely used languages in common domains, such as HTML for web development, to languages used by only one or a few pieces of software.

Given the broad concept of domain-specific languages, this project primarily focuses on domain-specific languages related to parallel computing, such as domain-specific languages in data-intensive disciplines.

Therefore, the significance of this project emerges: SYCL (DPC++) can provide reusable code for different hardware, while domain-specific languages can offer simple and easy-to-use programming interfaces for domain-specific programmers. Thus, designing a compiler from a domain-specific language to SYCL enables domain-specific languages to run in parallel on different hardware backends (heterogeneous systems), thereby improving programming efficiency and performance. The development of such a compiler allows developers to leverage the simplicity and usability of domain-specific languages while achieving high-performance parallel computing on different hardware platforms. By translating domain-specific languages into SYCL (DPC++) code, we can utilize SYCL's cross-hardware compatibility and performance optimization capabilities, thereby enabling broader application scenarios and higher execution efficiency. The development of such a compiler not only promotes the widespread use of domain-specific languages but also drives the development of heterogeneous computing systems, providing programmers from different domains with more flexible and efficient programming tools.

2. Background

Introduction to Existing Research Backgrounds from Two Perspectives: Domain-Specific Languages and SYCL Compilation

2.1 Domain-Specific Languages

Domain-specific languages (DSLs) are closely tailored to specific domains, offering a higher-level, convenient programming interface for that domain. Consequently, DSLs related to parallel computing can serve as excellent higher-level interfaces for heterogeneous parallel programming. The design of such DSLs allows programmers to focus more on solving specific domain problems without needing to concern themselves with low-level hardware details. Particularly in heterogeneous computing environments, such high-level

interfaces effectively conceal the complexity of underlying hardware, simplifying the programming process and enhancing development efficiency.

In the domain of image processing, several outstanding DSLs exist. ImageCL, for example, abstracts performance optimization details, allowing the compiler to handle specific performance optimizations, thus enabling programmers to focus primarily on algorithm design.(2) Halide achieves significant improvements in image processing efficiency compared to CUDA through methods such as separating computation and scheduling and loop optimizations.(3) Moreover, Halide supports deployment on various hardware backends and optimizes for different hardware, achieving good efficiency across many hardware platforms. Inspired by Halide's approach, subsequent research by Mullapudi et al. automated the generation of high-level scheduling strategies previously required by the Halide compiler, achieving efficiency close to manual implementation.(4) Building upon this, Adams et al. trained a large dataset to model runtime costs, achieving over twice the efficiency of automatic tuning.(5)

Beyond the realm of image processing, domain-specific languages have emerged in other data-intensive or compute-intensive domains. Hu Yuanming's development of the Taishi language provides a high-level, data-structure-agnostic interface for storing and applying sparse data structures, resulting in an average performance improvement of 4.55 times.(6) Xu Kai et al. converted dynamic core components from the Weather Research and Forecasting (WRF) model domain into a new domain-specific language called SWSLL, deployed on the Sunway TaihuLight supercomputing system, achieving a 4.7-fold speedup in widely used benchmark tests with a horizontal resolution of 2.5 kilometers.(7)

2.2 SYCL Compilation Related

According to the SYCL official website, there is a mature tool called SYCLomatic that can migrate CUDA code to SYCL. However, there are relatively few automatic compilers that directly compile existing code into SYCL, but some efforts involve manual migration of existing code to SYCL. In Sobol's research, the trajectory reconstruction algorithm for particle physics experiments was compiled to SYCL for execution.(8) Angus's research deployed Open Neural Network Exchange (ONNX) using SYCL as the backend on edge computing platforms.(9) Naiouf's research compared the performance of Python with SYCL and introduced the dpctl Python library under development, enabling writing extensions in Python and supporting asynchronous SYCL kernel execution.(10) An noteworthy achievement with a similar approach to SYCL is TVM, which addresses the deployment of various neural network frameworks on heterogeneous backends, providing support for different hardware and optimizing at the intermediate representation stage.(11)

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat

quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula. 147 148

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula. 149 150 151 152 153 154

3. Compiler Construction 155

In this section, we will discuss the process of compiling domain-specific languages to SYCL. Firstly, speaking holistically, the process of compiling from domain-specific languages to SYCL involves first compiling them into abstract syntax trees, and then compiling them into the SYCL language through these abstract syntax trees. Therefore, we can consider these two parts as two components of the compiler, namely the frontend and the backend. The frontend of the compiler is responsible for lexical analysis, syntax analysis, and ultimately generating abstract syntax trees. 156 157 158 159 160 161 162

3.1 Compiler Frontend 163

In this section, we need to build the frontend of the compiler. As discussed earlier, the main tasks of the compiler frontend are lexical analysis, syntax analysis, and generating abstract syntax trees. These tasks can be accomplished using tools such as lex and yacc, although their more common modern versions are flex and bison. By utilizing flex and bison together, this task can be relatively easily completed. 164 165 166 167 168

After outlining the approach to completing this task, we also need to determine the source language of our compiler, i.e., which domain-specific language we need to compile into SYCL. Here, we have chosen a data associated computation(DAC) model as our domain-specific language.(12) We won't delve into detailed introduction of this language here; instead, we'll introduce its basic characteristics through a code snippet.1 169 170 171 172 173

Through this illustration, we can observe the distinctive characteristics of Data Associated Computation (DAC). Firstly, the model features relaxed data declarations, allowing for shape modifications post-declaration. Following this is an explanation of DACrw, a function executing parallel reads and writes. The "|" symbol indicates variables on the left are subject to read-only operations, while those on the right involve value modifications. Lastly, the model's most notable aspects are highlighted: DACcalc establishes the computation pattern, DACshell forms the associative structure, and their combination generates the associated computation expression. 174 175 176 177 178 179 180 181

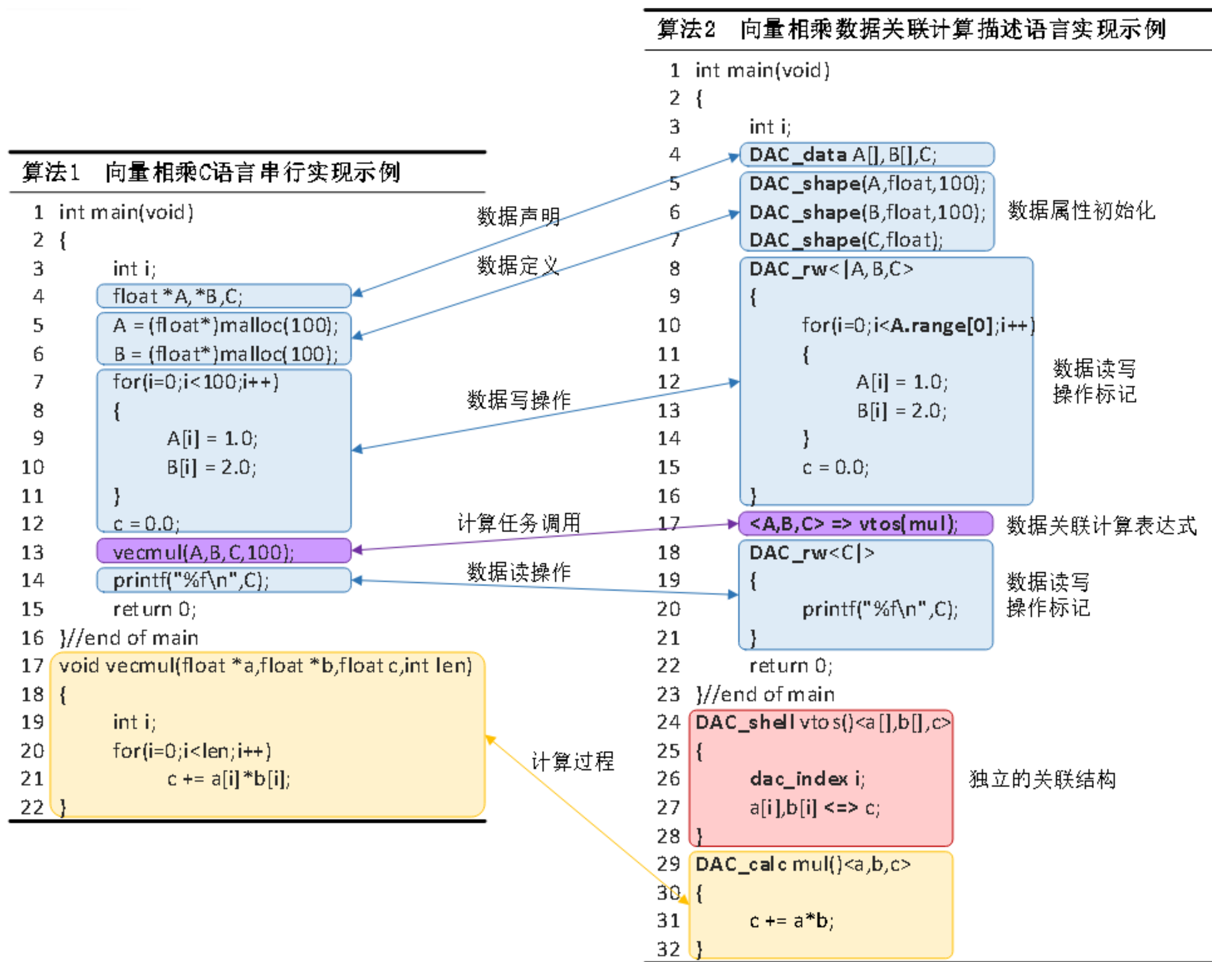

Figure 1: The features of DAC model.

3.2 Compiler Backend

In this section, we need to complete the backend of the compiler, which involves generating code from the abstract syntax tree. In this part, we refer to the generation method of LLVM. In the backend stage of the compiler, LLVM traverses the syntax tree and generates corresponding intermediate or target code based on the type of each node. This traversal method allows LLVM to effectively transform the syntax structure of high-level languages into low-level instruction sequences. Therefore, by traversing the abstract syntax tree nodes in a preorder manner, we generate code for each type of abstract syntax tree node, thus obtaining the final code.

During this translation process, the correspondence between the SYCL language and the data parallel language can be observed. The unified shared memory in SYCL corresponds to the variables in the data-associated model. The lambda expressions in SYCL queues correspond to the associative structures in the model. The code generation for reading and writing in the example is shown below in SYCL code.2

```
#include <sycl/sycl.hpp>
using namespace sycl;
int main()
{
  queue q;
  double *A = malloc_shared<int>(100, q);
  double *B = malloc_shared<int>(100, q);
  double *C = malloc_shared<int>(1, q);
  for (int i = 0; i < N; i++) data[i] = i;
  q.parallel_for(range<1>(100), [=](id<1> i) { A[i] = 1.0; B[i] = 2.0;}).wait();
  free(data, q);
  return 0;
}
```

Figure 2: The code generated.

References

[1] Mernik, M., Heering, J., & Sloane, A.M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4), 316–344. DOI: 10.1145/1118890.1118892.

[2] Falch, T.L., & Elster, A.C. (2016). ImageCL: An Image Processing Language for Performance Portability on Heterogeneous Systems. In W.W. Smari (Ed.), *Proceedings of the 2016 International Conference on High Performance Computing & Simulation (HPCS 2016)*, 562–569.

[3] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6), 519–530. DOI: 10.1145/2499370.2462176.

[4] Mullapudi, R.T., Adams, A., Sharlet, D., Ragan-Kelley, J., & Fatahalian, K. (2016). Automatically scheduling Halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4), 83. DOI: 10.1145/2897824.2925952.

[5] Adams, A., Ma, K., Anderson, L., Baghdad, R., Li, T.M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., & Durand, F. (2019). Learning to optimize Halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4), 121. DOI: 10.1145/3306346.3322967.

[6] Hu, Y.M., Li, T.M., Anderson, L., Ragan-Kelley, J., & Durand, F. (2019). Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)*, 38(6), 201. DOI: 10.1145/3355089.3356506.

[7] Xu, K., Song, Z.Y., Chan, Y.D., Wang, S.D., Meng, X.X., Liu, W.G., & Xue, W. (2019). Refactoring and optimizing WRF model on Sunway TaihuLight. In *Proceed-*

ings of the 48th International Conference on Parallel Processing (ICPP 2019), DOI: 10.1145/3337821.3337923. 220 221

[8] Sobol, B., & Korcyl, G. (2023). Particle track reconstruction on heterogeneous platforms with SYCL. In *Proceedings of IWOCL '23: International Workshop on OpenCL*, 3. DOI: 10.1145/3585341.3585344. 222 223 224

[9] Angus, D., Georgiev, S., Arroyo Gonzalez, H., Riordan, J., Keir, P., & Goli, M. (2023). Porting SYCL accelerated neural network frameworks to edge devices. In *Proceedings of IWOCL '23: International Workshop on OpenCL*, 4. DOI: 10.1145/3585341.3585346. 225 226 227 228

[10] Faqir-Rhazoui, Y., & Garcia, C. (2023). Exploring Heterogeneous Computing Environments: A Preliminary Analysis of Python and SYCL Performance. In M. Naiouf, E. Rucci, F. Chichizola, & L. De Giusti (Eds.), *Cloud Computing, Big Data & Emerging Topics: 11th Conference, JCC-BD&ET 2023, Proceedings* (Vol. 1828, pp. 3-16). DOI: 10.1007/978-3-031-40942-4_1. 229 230 231 232 233

[11] Chen, T.Q., Moreau, T., Jiang, Z.H., Zheng, L.M., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., & Ceze, L. (2018). TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 579-594. 234 235 236 237

[12] Wu, S. (2021). Research on High-level Unified Parallel Programming Architecture for Heterogeneous Many-core Systems. Doctoral dissertation, Xi'an Jiaotong University. 238 239