REX: A SOURCE-TO-SOURCE OPENMP COMPILER FOR PRODUCTIVE RESEARCH OF PARALLEL PROGRAMMING

by

Anjia Wang

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing & Info Systems

Charlotte

2023

Approved by:

Dr. Yonghong Yan

Dr. Harini Ramaprasad

Dr. Dong Dai

Dr. Erik Saule

Dr. Jinpeng Wei

ABSTRACT

ANJIA WANG. REX: A Source-to-Source OpenMP Compiler for Productive Research of Parallel Programming. (Under the direction of DR. YONGHONG YAN)

The growing complexity of high-performance computing (HPC) systems has led to the development of parallel programming models, such as OpenMP and OpenACC, to make it easier to utilize modern HPC architectures. These models provide a higher-level interface for specifying parallelism patterns and reducing programming effort, but performance optimization and customization are left to the compilers. Despite the availability of state-of-the-art OpenMP compilers, including LLVM, GCC, and ROSE, there remains a need for a compiler that is easily usable and extendable by researchers and students who are not in the field of compiler development, supports multiple parallel programming models, and has comparable performance to mainstream compilers. The REX compiler has been proposed as a solution to these challenges. It is built upon the ROSE compiler and uses a unified parallel intermediate representation (UPIR), targeting the LLVM OpenMP runtime for optimal performance. REX provides essential OpenMP 5.0/5.1 constructs and preliminary support for OpenACC 3.2. Its source-to-source transformation capabilities offer flexibility and ease of use with minimal overhead. It can be installed as a Docker image or used through a cloud service. The REX compiler's performance has been evaluated using an enhanced version of the parallel benchmark, Rodinia, which compares GPU offloading performance across different parallel programming models and compilers. In conclusion, the REX compiler provides a unique solution for parallel programming research and education, balancing performance, portability, flexibility, and usability.

ACKNOWLEDGEMENTS

I began my Ph.D. journey at the University of North Carolina at Charlotte in 2019. Over these four transformative years, I have gained invaluable expertise in high-performance computing, forged lasting friendships, and received unwavering support from those around me. This experience has undoubtedly enriched my life and will continue to shape my future positively.

First and foremost, I would like to express my profound gratitude to my advisor, Dr. Yonghong Yan. Dr. Yan has guided my Ph.D. journey from the beginning, providing invaluable assistance with my application, research proposal, completion guidance, and dissertation revisions. I am truly indebted to him for his constant support and mentorship.

I sincerely appreciate my esteemed committee members, Dr. Harini Ramaprasad, Dr. Dong Dai, Dr. Erik Saule, and Dr. Jinpeng Wei. Their valuable insights and guidance have enriched my research and provided direction for my future career.

I am grateful to my teammates, Kewei Yan, Patrick Flynn, Xinyao Yi, and Yaying Shi, for their collaboration and engaging discussions, which have undoubtedly advanced our collective research endeavors.

My deepest gratitude goes to my wife, Xiaoxiao, for her unwavering encouragement and often amusing and endearing attempts to uplift my spirits. Her support has been instrumental in my research and writing process, and her understanding of my prolonged absences from home has been remarkable. I am also profoundly grateful to my parents for their limitless moral support and steadfast belief in my abilities.

Lastly, I sincerely thank those not explicitly mentioned here who have directly or indirectly contributed to my dissertation. I am eternally grateful for your assistance and support have been crucial in my journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AST Abstract syntax tree

FreeCC FreeCompilerCamp

HPC High-performance computing

LLNL Lawrence Livermore National Laboratory

PWC Play-With-Compiler

PWD Play-With-Docker

SPMD Single program multiple data

UPIR Unified parallel intermediate representation

CHAPTER 1: INTRODUCTION

In the last decades, the complexity of the heterogeneous system for high-performance computing (HPC) has increased significantly. To utilize modern HPC architectures efficiently and easily, many parallel programming models have been developed, such as OpenMP and OpenACC. These models are directive-based, and users can specify what and how the source code should be parallelized without knowing the low-level details. While reducing the programming effort effectively, those models do not specify the implementation underneath, so users cannot directly maximize the performance by themselves. Therefore, the compilers play a critical role in many scenarios, such as scientific computing and climate simulation, because performance could highly depend on their different implementations.

Multiple parallel programming models are introduced to ease the difficulty of utilizing more complex HPC systems to help users focus on solving scientific problems instead of programming details. Each model has its advantages and disadvantages. For example, OpenMP requires users to specify more information to control the parallelism precisely. On the other hand, OpenACC is more descriptive so that users can use simpler syntax for parallelization. However, with fewer details of parallelism, the OpenACC compiler may be unable to generate efficient parallel code. Nevertheless, as the arguably de facto standard of parallel programming, OpenMP is supported by most compilers compared to other parallel programming models. It is widely used in academia for both research and teaching. For research purposes, users can evaluate various algorithms and optimizations using OpenMP. Meanwhile, since the fundamental concept of OpenMP is based on SPMD, it is an excellent tool for students to learn parallel programming.

Therefore, to better cover these two subjects, there is a need for an OpenMP compiler that features performance, portability, flexibility, and usability. Specifically, this compiler should meet the following requirements:

- It's open source and free to use/modify/redistribute;
- It supports relatively new OpenMP specifications, if not the latest;
- It helps users quickly experiment with new algorithms, optimization, and prototypes without the need for solid compiler development background;
- Users can easily use it both online and offline;
- It's flexible for extensions to support new features, such as a different parallel model, a new compiler transformation pass, etc.

There have been many state-of-the-art OpenMP compilers. However, none of them fulfill all the points above. LLVM, one of the most popular OpenMP compilers, perfectly addresses performance, portability, and flexibility. Unfortunately, researchers not in this field and students may have difficulty modifying the compiler according to their needs because of the complexity. In addition, understanding the LLVM IR generated from the original OpenMP code and the parallelism it represents could also be a challenge since LLVM IR is very different from regular C/C++ code. GCC is another well-known open-source OpenMP compiler. Compared to LLVM, it also supports the OpenACC parallel programming model. However, it faces the same usability issue as LLVM. GCC's code base is complex, and its GIMPLE IR needs to be more intuitive to showcase the parallelism underneath. Furthermore, GCC presents a much worse performance in many cases than LLVM. Its license could also be problematic for some users wanting to keep their research details private. ROSE, a source-to-source OpenMP compiler developed by LLNL, shows outstanding usability. It lowers the OpenMP code to human-readable generic C/C++/Fortran code. Users

can easily recognize how the transformation is performed and make any changes in the same programming language. Unfortunately, ROSE has its drawbacks. It only has full support for OpenMP 3.0 and partial support up to OpenMP 4.5. In addition, several significant constructs, such as target teams, are not supported. Because ROSE uses GCC as the backend compiler, its performance could be more optimal. There are also several commercial compilers. Intel compiler is based on LLVM and contains its proprietary implementation for certain OpenMP constructs. NVIDIA HPC SDK supports both OpenMP and OpenACC. It can achieve similar performance to LLVM and even better in some cases. However, NVIDIA HPC SDK is proprietary and can only produce its own IR as LLVM and GCC do.

Since those commercial compilers are primarily proprietary and LLVM/GCC requires skills of compiler development to customize, source-to-source compilers like ROSE and OpenUH, mostly from academia, are developed to generate new source code for quick experiments on new algorithms or platforms. Users can focus on high-level programming languages without knowing the compiler's details. Despite the ongoing efforts above, we are still looking for a single approach that achieves all the goals. Thus, to address all the challenges and utilize the advantages of existing compilers for parallel programming research and education purposes, we propose the REX compiler, a source-to-source OpenMP compiler. It's built upon the ROSE compiler. Instead of targeting GCC's GOMP runtime, we lower the OpenMP constructs all to the LLVM OpenMP runtime APIs for the best performance. To provide the most flexibility to users, we adopt unified parallel intermediate representation (UPIR) to support any parallel programming model, including OpenMP. UPIR covers commonly used features among different parallel models and carries as much parallelism information as possible. It makes extending support to other parallel models and transformation passes more accessible.

Our main contributions include the following:

- Portability: REX uses a unified AST for C/C++/Fortran. Users can use the same APIs to manipulate the AST of inputs in different languages. REX compiler supports essential OpenMP 5.0/5.1 constructs, including offloading and tasking. To demonstrate the benefits of utilizing UPIR and unified transformation, REX provides preliminary support to OpenACC 3.2 specification.

- Performance: The OpenMP transformation in the REX compiler targets the LLVM OpenMP runtime, enabling it to achieve the same performance as the original LLVM OpenMP compiler while maintaining the flexibility of source-to-source transformation. We have implemented several optimizations for OpenMP GPU offloading, such as data shuffle and dynamic parallelism. We have developed a comprehensive parallel benchmark based on the classic Rodinia to evaluate different parallel programming models, focusing on assessing compiler GPU offloading performance. Compared to the original Rodinia, our enhanced version adopts the latest OpenMP constructs and supports OpenACC, providing a more robust performance evaluation framework.

- Flexibility: With source-to-source transformation, REX can transform OpenMP and OpenACC code so that users can compile the generated source code using any non-OpenMP/OpenACC compiler with eligible OpenMP runtime. The REX compiler's OpenMP/OpenACC parser and runtime are separate modules. Users can make their customization as long as they keep the API consistent.

- Usability: REX offers a ready-to-use Docker image in addition to the conventional installation, eliminating the need for users to deal with the complexities of setting up the environment. Furthermore, an open-source cloud service, FreeCompilerCamp (FreeCC), is also available, enabling users to experiment with the REX compiler within a browser. This service also serves as a platform for users to learn compiler development. FreeCC can be deployed on local pri-

vate and online public servers, ensuring flexibility and accessibility for users in various settings.

As a research-oriented project, REX has its limitations:

- REX adopts the LLVM OpenMP runtime and benefits from its excellent performance. However, REX also inherits its support status of OpenMP. It means we may not always catch up with the latest OpenMP spec. Additionally, we must internally create corresponding wrappers using LLVM OpenMP APIs to support other parallel models.

- REX's design allows any device code that LLVM supports. However, due to resource limits, currently, we only implement the GPU kernel generation in CUDA.

The remainder of the dissertation is organized as follows: Chapter 2 gives the related work and our motivation. Chapter 3 presents the design of UPIR. We explored the usage of data shuffle on GPU in OpenMP in Chapter 4. In Chapter 5, we discuss the design and implementation details of the REX compiler and evaluate its performance using the enhanced heterogeneous computing benchmark suite Rodinia. Chapter 6 introduces an online learning platform to train researchers and students to develop OpenMP compilers efficiently. Finally, Chapter 7 concludes the dissertation.

CHAPTER 2: MOTIVATION

2.1 Parallel IR

2.1.1 Challenges

Many compiler implementations to support the compilation of parallel programs take the approach of transforming (lowering) IRs of parallel constructs to API calls of parallel runtime systems, and it is often performed at the compiler front-end. The parallel programs are then considered sequential programs in the later stages of compilation. There are, in general, at least two limitations of this approach. First, the parallelism information may not be carried sufficiently through the whole compilation pass to the back-end, and the compilation would then lose some optimization opportunities in the later stage [4]. For example, function outlining is a common compiler transformation step for lowering OpenMP *parallel* code region to runtime API calls. The newly outlined function is separated from the original function. Thus traditional data- and control-flow analysis and optimization cannot apply to the original function body without additional effort to associate the two functions after outlining. Secondly, since the transformation for parallelism is performed at the front-end, operating on language-specific IRs, parallelism-aware compilation becomes language-specific. Each programming model requires implementing the transformation in the compiler front-end. The study of thread programming models such as OpenMP, OpenACC, CUDA, Cilk, etc. shows that programming models share common functionality and even similar syntax for programming parallelism [5]. Thus it is possible to unify compiler transformation for multiple models if a common and language-neutral IR is used by the compiler [6]. Major open-source compilers such as GCC and LLVM have shown

the unification of IRs for multiple languages, their support for unifying parallel models is however limited.

2.1.2 GCC

GCC supports a wide range of programming languages and platforms. It uses a language-independent IR GIMPLE. GCC implements different parsers for each language and generates a generic AST first. Then, it converts the generic AST into GIMPLE for further analysis and transformation. This design reduces the duplicated work of implementing similar compilation passes for multiple languages. GCC supports both OpenMP and OpenACC. A set of IR components are added as extensions to GIMPLE for OpenMP, such as GIMPLE_OMP_TARGET representing an offloading region. All OpenMP constructs are converted to corresponding GIMPLE IR objects. For OpenACC, GCC maps OpenACC constructs to OpenMP GIMPLE IRs. For instance, *acc kernels* and *acc parallel* are both converted to GIMPLE_OMP_TARGET. The transformation for OpenACC then does not need to be implemented separately.

The unification for OpenMP and OpenACC supports in GCC proved that a unified parallel IR reduces the compiler programming effort and an existing transformation module can be extended easily to support another parallel model. However, rather than working as an abstraction for multiple parallel programming models, those GIMPLE IR objects are primarily designed for OpenMP and tightly associated with OpenMP syntax and semantics. Not all the necessary parallelism information is included since they may not be required according to OpenMP specifications. Given a group of optimization passes that need complete data attributes, each of them has to search individually because GIMPLE only contains the information specified explicitly by users.

2.1.3 LLVM

LLVM is another widely used compiler like GCC but with a more flexible modular design. Different languages have their own front-ends that generate unified LLVM IR for later transformation. LLVM IR is a low-level language-neutral IR supporting the same compiler analyses and transformations for multiple programming models. However, the LLVM IR was not designed for parallelism, though it can be enhanced to encode certain parallelism information using metadata or IR attribute (e.g. the *llvm.loop.parallel_ accesses* [7]), or parallel intrinsic functions [8]. However, using the intrinsic or IR metadata has the same limitation of being complicated when the parallel construct becomes complicated. Also, the conventional serial analysis and optimization may not be applicable anymore because the parallel region has been outlined and the compiler is not aware of the semantics of intrinsic calling the outlined function. To bridge the gap between the low-level LLVM IR and high-level language-specific IR and frontend, recently, LLVM adopts the MLIR [9] as the middle-end IR and abstraction to address the fragmentation of compiler development for multiple programming models including domain-specific languages. However, until now, MLIR has mainly been used for defining language-specific IRs dialects for new languages or extensions to existing languages, but not for unifying IRs of common constructs of multiple languages [10].

2.1.4 Enhancing Existing IR for Parallelism

There are proposals for new IRs for parallelism, including as much relevant information as possible. Zhao et al. designed a set of parallel IR including high, middle, and low levels to represent parallel constructs [11], focusing on parallel loop chunking, load elimination, delegated isolation, automatic data privatization, vectorization, and other cases. Benoit et al. presented Gomet and KIMBLE as GCC extensions to support parallel IR [12, 13]. It introduces a hardware abstraction and hierarchy

of parallel IR for parallelism adaption. Doerfert et al. proposed a new interface `TRegion` used in the OpenMP lowering procedure [14]. It postpones the decision of target-specific transformation and finds a better solution rather than solely relying on the user's original OpenMP code. HPVM/approxHPVM is another design of parallel IR [15, 16]. HPVM provides compiler IR, virtual ISA, and runtime libraries, while other studies mainly focus on one of them. Tapir/LLVM [4] adds three instructions *detach*, *reattach*, and *sync* to encode logical parallelism asymmetrically in LLVM IR to support task parallelism. It has been extended to support OpenMP tasking [17] and TensorFlow [18]. However, these state of art lacks the generality for supporting multiple parallelism patterns (e.g., SPMD, data, and task parallelism), data-sharing attributes, and synchronization operations of parallel programs.

2.1.5 Creating Unified Parallel Intermediate Representation (UPIR)

As existing parallel programming models share common parallelism functionality and similar interfaces of essential capability for programming parallelism [5], a language-independent abstraction of the fundamental entities and constructs for parallelism and their connections can be constructed in a unified intermediate representation as the backbone to enable unified and common parallelism-aware analysis and transformation. The UPIR design and specification include 1) the three commonly used parallelism patterns, namely single program multiple data (SPMD), data parallelism, and task parallelism, including offloading tasks; 2) data attributes and explicit data movement and memory management for assisting data-aware optimization for parallel programs; and 3) synchronization operations (e.g., barrier, reduction, mutual exclusion, etc) used in parallel programming for optimizing synchronization cost by the compiler. Table 2.1 shows the UPIR's support and mapping for the language constructs of commonly-used parallel models. In the following three sections, we present our design of the UPIR. Since UPIR naturally unifies IR of different programming models, an additional benefit is that supporting programs of hybrid models or sup-

porting translating programs of one model to another would be made much easier. In comparison with related work such as INSPIRE [6], they share a similar concept, but UPIR's design allows for more parallelism information in the IR and provides more features than INSPIRE. For example, UPIR categorizes data parallelism into work-sharing, SIMD, and taskloop, compared to a generic work distribution in INSPIRE.

2.1.5.1 UPIR extension

Since the goal of UPIR is to cover the common features shared by multiple parallel models, thus the limitation would be that some of the unique features of a parallel model might not be included in UPIR. e.g., memory controls in CUDA. In our design, we address this by including UPIR extension in the form of key-value maps such that language-specific features can be added under the UPIR. For the compiler, a language-unique compiler pass can be added as an extension of the UPIR transformation pass in the UPIR compiler. For instance, *metadirective* in OpenMP is created as a UPIR extension but not part of the UPIR nodes. It will be handled by the compiler only if a corresponding transformation pass is implemented.

Table 2.1: Mapping of parallel model constructs with UPIR design

	SPMD parallelism	**Data parallelism**	**Async task parallelism**	**Data attributes**	**Synchronization**
UPIR	spmd	loop/loop-parallel	task	data	sync
CUDA	<<<...>>>	-	async kernel launching	__shared__, __global__, memory operations	cudaDeviceSynchronize
OpenCL	clEnqueueNDRangeKernel	-	clEnqueueTask	clCreate / Read / WriteBuffer	clWaitForEvents
OpenClik	-	cilk_for, array operations, elemental functions	cilk_spawn, cilk_sync	-	cilk_sync
Kokkos	**parallel**_for	parallel_**for**	task_spawn, host_spawn	View, memory space	fence
SYCL	**parallel**_for	parallel_**for**	single_task, host_task	buffer	wait
OpenMP	teams, parallel	distribute, for, simd	task, target	map(to/from), shared/private/-firstprivate	barrier, atomic, critical, taskwait
OpenACC	parallel	loop gang/worker/vector	async, wait	data(copyin/out), shared/private/-firstprivate	wait, atomic
PThread	pthread_create/join	-	-	-	pthread_join, pthread_cond_wait

2.2 Data Shuffle on GPU

2.2.1 Challenges

In these architectures, data transfer between registers belonging to multiple computing elements, such as cores or vector lanes, can be achieved without using the memory and cache systems by employing shuffle or permutation operations. For instance, starting with the Kepler architecture, NVIDIA incorporated shuffle instructions to enable data transfer among registers of separate threads within a warp. This functionality allows multi-thread kernels to carry out synchronous vector-like operations inside a warp. As register access latency is 10 to 100 times lower than that of SRAM and DRAM, respectively, taking advantage of this data shuffle feature among threads can substantially enhance the computation performance of work-sharing or vector loops. However, mainstream compilers like GCC and LLVM have yet to utilize this feature in OpenMP GPU offloading.

This section explores the data shuffling capability between GPU cores and vector lanes in NVIDIA GPUs, AMD GPUs, and vector architectures.

2.2.2 CUDA `shuffle` instruction for NVIDIA GPUs

Since Kepler architecture, NVIDIA releases the warp `shuffle` instructions to allow data exchange between registers without touching memory. Before that, exchanging data between threads must go through shared memory within a block or global memory. If the operation is not atomic, developers have to synchronize before and after the data transfer, which introduces overhead and increases the programming complexity. The `shuffle` instructions introduced in NVIDIA CUDA include `__shfl_sync`, `__shfl_up_sync`, `__shfl_down_sync`, and `__shfl_xor_sync`. Using those instructions, data in private registers of threads within the same warp could be exchanged directly. They are atomic operations, and the synchronization is enforced naturally by the SIMT execution model of the NVIDIA GPU architecture. The `shuffle` in-

structions are read-only operations to the threads that provide the data.

Table 2.2: CUDA `shuffle` instructions

Instruction	Description	Parameters
`__shfl_sync`	Direct copy from indexed lane	`unsigned mask, T var, int srcLane, int width=warpSize`
`__shfl_up_sync`	Copy from a lane with lower ID relative to caller	`unsigned mask, T var, unsigned int delta, int width=warpSize`
`__shfl_down_sync`	Copy from a lane with higher ID relative to caller	`unsigned mask, T var, unsigned int delta, int width=warpSize`
`__shfl_xor_sync`	Copy from a lane based on bitwise XOR of own lane ID	`unsigned mask, T var, int laneMask, int width=warpSize`

The description of the CUDA's `shuffle` instructions are shown in Table 2.2. They take four parameters and the last one for warp size is optional. `mask` is used to indicate which threads are involved. `var` is the targeting data, which could be an integer, float, double or other types. `srcLane` is an absolute lane ID in the warp while `delta` represents the relative difference to the lane ID of the caller thread. `laneMask` performs a bitwise operation to the lane ID of the caller thread.

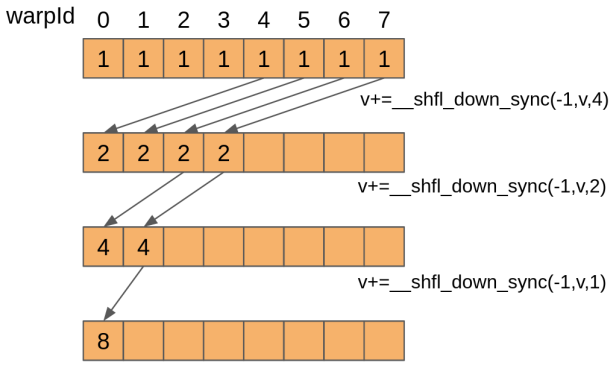


Figure 2.1: Shuffle example using NVIDIA GPU instruction [3]

We use a sum reduction as an example to show how shuffle works. To simplify the case, we ensure only 8 lanes hold their copy of variable `v`. They need to store the value in shared memory without shuffling so that other threads can access it. Between each iteration of reduction, the intermediate result also needs to be maintained in the

shared memory, and the synchronization has to be appropriately handled to avoid a data race. By using shuffle instruction, a thread can directly access the private register of another thread without routing via shared memory. Furthermore, the shuffle operation is atomic and executed in lockstep. In Figure 2.1 [3], initially, the first four threads read `v` from the last four threads and add it up to their copy of `v`. Then the same kind of reduction continued among these four threads. Eventually, the very first thread got the sum of all eight elements. Throughout the whole procedure, only registers are used for computing.

2.2.3 Cross-lane operations of AMD GPUs

The AMD GPUs also provide instructions similar to the `shuffle` instructions in CUDA. Wavefront on AMD GPU plays the same role as warp on NVIDIA GPU. Within a wavefront, 64 lanes can execute the same code simultaneously as a SIMD vector.

Table 2.3: Summary of AMD GPU `shuffle` instruction [1]

Instruction	**Description**	**Parameters**
`ds_permute_b32`	Push `src` data to a lane indicated by `addr`	`dest, addr, src [offset:addr_offset]`
`ds_bpermute_b32`	Pull `src` data from a lane indicated by `addr`	`dest, addr, src [offset:addr_offset]`

There are two instructions related to `shuffle` (Table 2.3). Unlike the read-only operations in CUDA, AMD allows a thread to push its data to another thread's private register using `ds_permute_b32`. `ds_bpermute_b32` reads data from another thread's private register.

2.2.4 Shuffle data between SIMD/vector lanes

Vector architectures also provide instructions for cross-lane operations. For example, Intel AVX2 and AVX512 introduced SHUFFLE, BROADCAST, and PERMUTE operations for cross-lane functionality for floating-point and integer operations. Instructions are SHUFPS, VSHUFPS, VPERMI2D, VPERMD, VPERMQ, etc, and

their intrinsics can be found from Intel compiler developer guide [19]. ARM Scalable Vector Extension (SVE) provides permutation and shuffle operations, including reductions across vector lanes. RISC-V vector extensions also have to permute instructions to allow cross-lane data movement. While OpenMP's `simd` directive can instruct the compiler to vectorize a loop, advanced operations such as cross-lane data movement have not yet been supported in the standard.

There are several works that have adopted the CUDA `shuffle` instructions in their studies for performance improvement. CUDA `shuffle` instructions can improve the performance of reduction operation by computing on private registers of multiple threads [3, 20, 21]. Liu and Schmit similarly use warp shuffle functions to develop LightSpMV, a faster algorithm for sparse matrix-vector multiplication [22]. For a more general linear solver, `shuffle` instructions can speed up the computation by directly exchanging values stored on registers as well [21]. Tangram is a high-level programming framework that provides APIs to perform computation on GPU [23]. It has been extended to use atomic and shuffle functions available in the framework [24]. During AST construction, an additional pass is added to determine the opportunity of inserting `shuffle` instruction for loop optimization. With the help of `shuffle` instructions, Chen et al. [25] realize the systolic execution on GPU and demonstrate superior performance for 2D stencil in CUDA than most state-of-the-art implementations. In comparison, our work proposes a high-level interface using shuffle instruction with OpenMP.

2.3 Source-To-Source Compiler

2.3.1 Challenges

As heterogeneous systems evolve to deliver higher performance, their complexity also increases. Various parallel programming models have been introduced to better harness the power of advanced hardware to ease programming difficulties. Consequently, compilers are responsible for producing suitable binaries for different

platforms. While mainstream compilers such as LLVM and GCC perform well in most cases, they have limitations in supporting parallel models, keeping up with the latest hardware features, and providing an accessible development experience for non-compiler developers.

OpenMP and OpenACC are popular directive-based parallel models; however, LLVM does not support OpenACC. Although GCC supports both models, its performance can sometimes be suboptimal. Furthermore, these compilers often lag in supporting the latest hardware features. For example, LLVM's full support for NVIDIA Ampere GPUs arrived a year after its official release in 2020. The latest NVIDIA Ada Lovelace GPUs released in 2022 have yet to be fully supported. To leverage the latest features of these GPUs within parallel models like OpenMP, it is necessary to manually update the compiler and generate new device codes to run on the GPU. This process requires extensive compiler development knowledge, which is unrealistic for many researchers and students.

An alternative approach would be to use CUDA instead of OpenMP to write parallel programs. However, this approach may require migrating an existing codebase from OpenMP to CUDA, resulting in a significant workload. Additionally, as a lower-level parallel programming interface, CUDA demands more effort from users than OpenMP.

This landscape creates a need for source-to-source compilers to address these issues. Source-to-source compilers offer excellent language interoperability by converting code from one language to another, allowing for better compatibility and integration between different languages and parallel programming models. For instance, a source-to-source compiler could transform OpenACC code to OpenMP code, enabling LLVM to handle both models indirectly.

Source-to-source compilers can also utilize the latest hardware features more efficiently due to their extensible language capabilities. For example, data shuffle, a

feature available on recent NVIDIA GPUs, enables multiple threads to exchange data directly through registers, bypassing the need for global and shared memory access. LLVM or GCC does not yet support this feature. OpenMP source-to-source compiler can convert OpenMP code to CUDA, and the data shuffle feature can be utilized through CUDA API calls to improve performance. In addition to leveraging new hardware features, new algorithms can be prototyped directly on the generated code.

Moreover, source-to-source compilers can expose more underlying details to users than mainstream compilers like LLVM and GCC, which convert input code to compiler IR and then to the final binaries. The compiler IR and binaries are difficult to understand without compiler background knowledge. For example, how an OpenMP program is parallelized may be unclear to most users. In such cases, an OpenMP source-to-source compiler can transform OpenMP code into human-readable C code with appropriate runtime API calls, making it more accessible to C users.

2.3.2 Source-to-source Transformation

The process of source-to-source transformation converts code written in one programming language to another instead of converting it to binary code. This approach can improve performance by applying optimizations or reducing the costs associated with coding by generating code in the new programming model with minimal human effort. This is particularly important in parallel programming, as it allows researchers and students to leverage their existing code base and reduces the learning curve associated with new parallel programming models.

One critical use case for source-to-source transformation is auto parallelization, which aims to improve performance. However, transforming traditional serial programs into parallel ones can be challenging. For example, CUDA is a popular parallel programming model developed by NVIDIA that is widely used in HPC systems with NVIDIA GPUs. Despite its performance advantages, CUDA is limited by its device specificity and support for only NVIDIA GPUs. This can hinder its portability, as

many supercomputers use AMD GPUs (such as Corona at LLNL) or have no GPU (such as Fugaku at RIKEN). Additionally, CUDA requires users to learn a new set of extensions to C++ and runtime APIs, which can be a substantial effort.

OpenCL was introduced to address the limitations of device-specific programming models like CUDA. Unlike CUDA, OpenCL code can theoretically be executed on any device, including CPU, GPU, FPGA, and other accelerators. However, this portability is sometimes hindered by vendor-specific implementations and extensions that take advantage of specific platforms. As a result, the market has different open-source and proprietary implementations of OpenCL, which can break its portability.

Multiple directive-based parallel programming models have been proposed to address the second limitation of programming difficulties, such as OpenMP and OpenACC. These models are an intermediate layer between the original sequential code and lower-level languages like CUDA and OpenCL. Users describe how the program should be executed in parallel using directives, and the compiler generates the corresponding parallelized code. While this approach can make writing parallel programs more efficient, it also sacrifices fine control over software and hardware. For instance, CUDA provides a rich set of APIs for kernel execution and data transfer, while OpenMP users mostly rely on implementing the compiler. As a result, these high-level programming models may result in lower performance.

There is no one-size-fits-all solution for parallel programming due to the limitations and trade-offs of the various models. There have been numerous studies on source-to-source transformation to address these issues, focusing on optimizing performance, portability, or a combination of both.

A source-to-source transformation is a popular approach for modern heterogeneous parallel systems, converting generic C code into a parallel programming model. For example, compilers can generate optimized CUDA code for NVIDIA GPUs based on factors such as memory hierarchy, data access pattern, and loop structures [26,

27, 28, 29, 30, 31]. Other high-level programming models, such as OpenMP, can also be generated through source-to-source transformation. Cetus, for instance, is a source-to-source compiler that converts sequential C code into parallel code by inserting OpenMP directives. It performs various compiler analyses and transformations, such as data privatization, reduction recognition, and loop parallelization. Evaluated using the NPB benchmark, Cetus shows an average 1.66x speedup over Intel's ICC compiler [32].

There have been many developments in source-to-source translators for converting between different parallel programming models because each model has its suitable use case. For example, SnuCL and CU2CL address the challenge of automatic direct translation between CUDA and OpenCL [33, 34]. SnuCL supports bidirectional translation, while Swan provides a high-level abstraction that maps API calls to either CUDA or OpenCL internally [35]. NMT uses machine learning techniques to assist in the translation process by pairing commonly used CUDA and OpenCL APIs as the training set and determining usage patterns [36].

There is a strong demand for transforming code between low-level and high-level parallel programming models. For instance, OpenMP is a widely-used high-level parallel programming model that provides portability, but performance depends on the OpenMP implementation in a specific compiler. Researchers have developed tools to translate OpenMP code into low-level models such as CUDA and OpenCL to optimize performance.

Lee et al. proposed an automatic translator from OpenMP to CUDA, making it easier for users to obtain equivalent CUDA code without learning the complex CUDA APIs or writing code from scratch [37]. Once the translation is complete, the compiler can further optimize the code. Kim et al. have also developed a tool that lowers OpenMP code into OpenCL and enables runtime optimizations to minimize data transfer, thus improving performance [38].

The transformation of parallel programming models has significant benefits in terms of reducing programming effort. However, it is challenging to achieve both optimal performance and portability. For instance, automatic parallelization can improve performance over the original sequential code, but it may not be suitable for all scenarios. Furthermore, any optimizations made in the original code must be implemented again. Additionally, these tools often require a learning curve, including the usage and environment setup. REX has been introduced as an OpenMP compiler based on ROSE to address these limitations. It leverages the LLVM OpenMP runtime for improved performance, offers unified source-to-source transformation, and provides a cloud-based integrated solution without needing installation, making it a more robust and user-friendly solution.

2.4 Online Training for Compiler Development

2.4.1 Challenges

Table 2.4 summarizes the main pain points for compiler training. For example, one of the first problems for developers is getting their hands on a machine that is suitable for compiler development. Getting access to a supercomputing cluster could be a challenge and a potential deterrent for many. The second, and most frustrating challenge for beginners is making sure that all the software packages necessary for developing a compiler are met on the said machine. Sometimes users might not have suitable access to install certain dependencies. Or sometimes the dependencies are just too complex to resolve on a particular machine. One solution to these two problems is to provide a free online sandbox terminal that will already have an environment setup for compiler development.

Based on our experiences, traditional text tutorials are not as effective for compiler development, as hands-on tutorials. If a framework is provided which gives its users an option to learn by hands-on practice, and the freedom to dig deep and perform self-experiments, then such a framework will be the most efficacious way of teaching

compilers.

Another problem of creating the content of compiler development, especially for OpenMP, is that no one person or group knows all the details of OpenMP implementations since they involve many compilation and runtime stages including parsing, AST, transformation, as well as runtime support. No one implementation demonstrates all the options of OpenMP. This generally results in incomplete or unproductive tutorials. Having an open-source environment where multiple users can submit tutorials for multiple compilers can resolve such complications.

Finally hosting tutorials on the website costs money. Having containers can result in larger disk space which means more expenses. Having an open-sourced, self-deployable framework can help users host their tutorials for free. This work aims to improve the effectiveness and scalability of compiler development training for re-

Table 2.4: Pain points and solutions for training OpenMP compiler developers

Pain Points	**Description**	**Proposed Solution**
Accessibility	Paperwork to get accounts on suitable machines	Online sandbox terminal open to anyone
Installation	Many software packages are needed	Docker images
Effectiveness	Traditional text tutorials are not effective	Learning by doing, testing, certification
Content	No single person/group knows all details of OpenMP compiler development	Self-made tutorials + crowd-sourcing to accept external contributions
Design trade-offs	One compiler cannot demonstrate all options	Hosting tutorials for multiple compilers
Costs	Hosting websites with containers costs money	Open-source, self-deployable framework
Security	Online websites have inherent risks	Containers + Cloud machines

searchers, developers, and graduate students. We chose two OpenMP compilers, Clang/LLVM and ROSE, as examples. This section gives a brief introduction of background information.

2.4.2 Compilers

Compilers are essential for HPC. Unlike interpreted languages, programs written in compiled languages give a better performance and are more favorable towards high-performance computing. A compiler takes high-level human-readable programs in programming languages, such as C/C++ or Fortran, and converts them into low-level binary machine codes for a specific architecture. The entire process of this transformation is complicated. A compiler must parse the code, check for syntax correctness, gather necessary semantic information (like type checking or variable declaration before use), then convert the source from high-level language to intermediate representation before transforming them into machine codes [39].

Today a compiler can do much more than convert a program into machine instructions. As HPC hardware designs are evolving, machines are becoming more and more complex, and issues that need to be addressed by programmers are also getting convoluted. This raises the question of what more a compiler can do for programmers. Compilers have very complex designs so that the work of an application developer becomes simpler. Owing to the complexity of design, extending a compiler to add a new feature is a very time-consuming job. The development cycle of a compiler is at least 3-5 years. Training programmers to develop compilers is challenging for the trainer and the trainee.

2.4.3 Clang/LLVM

LLVM [40] is the prime environment for developing new compilers and language-processing tools. HPC programmers rely on compilers and analysis tools. LLVM is the environment of choice for developing such tools and, thus, should interest many

HPC programmers. LLVM makes it easier to not only create new languages but to enhance the development of existing ones. Its primary C/C++ compiler frontend is Clang. Today most supercomputing clusters deploy LLVM as one of their compilers due to the following reasons:

1. It provides a high-performance and up-to-date C/C++ compiler frontend Clang.
2. Many researchers in the HPC community enjoy Clang's diagnostic abilities and static-analysis framework.
3. It allows for tapping other languages with an LLVM back-end, like Intel's ISPC [41] and different scripting languages.
4. It makes for compelling compiler research, as evident by the plethora of projects built using LLVM [42].

Ever since its first release in 2003, LLVM has gone through a plethora of changes and updates. With every release, new features are added, and older features are deleted or updated. Owing to these diverse sets of features and many more, using Clang/LLVM for developing a tool or a plugin is a very complex task. Many tutorials are available for Clang/LLVM, but they are all just text-based tutorials and come with their own set of challenges.

2.4.4 ROSE

ROSE is an open-source compiler infrastructure developed at Lawrence Livermore National Laboratory (LLNL). It is designed to build source-to-source program transformation and analysis tools for Fortran, C, C++, OpenMP, and UPC applications[43]. Internally, ROSE generates a uniform abstract syntax tree (AST) as its intermediate representation (IR) for input codes. Sophisticated compiler analyses, transformations, and optimizations are developed on top of the AST and encapsulated as simple function calls, which tool developers can readily leverage. The ROSE AST

can be optionally unparsed to human-readable and compilable source files, which in turn can be compiled into the final executable by a traditional compiler such as GCC or Intel compiler.

However, for users unfamiliar with the ROSE compiler, it's not easy to customize the framework because of the complexity of ROSE. ROSE has more than two million lines of code, including tests, built-in projects, and tutorial examples. Creating a new transformation module could involve multiple functions located in different files far away from each other. Like any other compiler framework, the ROSE compiler exposes its API functions for developers to traverse, analyze, and modify its abstract syntax tree. Users not only need to learn the general knowledge of compilers but also have to understand how ROSE API functions work.

2.4.5 OpenMP

OpenMP is the de-facto portable programming interface in HPC for exploiting node-level parallelism [44]. OpenMP uses C/C++ directives and Fortran comments to annotate base language programs written in C/C++ and Fortran, respectively. These annotations express additional semantics related to parallelism, worksharing, synchronization, tasking, etc. A compiler supporting OpenMP can recognize OpenMP annotations and transform the annotated input code into multi-threaded code by calling some OpenMP runtime functions.

There are multiple compilers implementing OpenMP, such as GCC[45], Intel[46], Cray[47], IBM XL[48], Clang/LLVM and ROSE[49]. Most of the parallel constructs in OpenMP are realized through compiler directives. This allows a serial program to be easily converted into a parallel one by adding the necessary pre-processor directives.

Figure. 2.2 is an OpenMP program to calculate PI in parallel. The user inserted an OpenMP `parallel for` directive at lines 14-15 right above the loop (Figure 2.2a). An OpenMP compiler transforms (or lowers) the program into multi-threaded code with calls to runtime library functions (Figure 2.2b). In the lowered code at lines

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int num_steps = 10000;
5
6  int main() {
7    double x = 0;
8    double sum = 0.0;
9    double pi;
10   int i;
11   double step = 1.0/(
         double) num_steps;
12
13   // Run the code in
         parallel
14   #pragma omp parallel
         for private(i,x)
         reduction(+:sum)
         schedule(static)
15   for (i = 0; i <
         num_steps; i = i +
          1) {
16     x = (i+0.5)*step;
17       sum = sum +
             4.0/(1.0+x*x);
18   }
19
20   pi = step * sum;
21   printf("%f\n", pi);
22 }
```

(a) OpenMP program to calculate PI

```
1  ... // omitted headers and a data
       structure declaration storing
       variable addresses
2  static void OUT__1__2189__(void *
       __out_argv);
3  int main(int argc, char **argv) {
4    ... // omitted variable declarations
5    XOMP_parallel_start(OUT__1__2189__,&
         __out_argv1__2189__,1,0,"demo.c"
         ,10);
6    XOMP_parallel_end("demo.c",15);
7    pi = step * sum;
8    printf("%f\n",pi);
9    XOMP_terminate(status);
10 }
11 static void OUT__1__2189__(void *
       __out_argv) {
12   ... // omitted variable declarations
13   double *sum = (double *)(((struct
         OUT__1__2189___data *)__out_argv)
         -> sum_p);
14   double *step = (double *)(((struct
         OUT__1__2189___data *)__out_argv)
         -> step_p);
15   XOMP_loop_default(0,num_steps - 1,1,&
         p_lower_,&p_upper_);
16   for (p_index_=p_lower_; p_index_<=
         p_upper_; p_index_=p_index_+1) {
17     _p_x = (p_index_ + 0.5) * *step;
18     _p_sum = _p_sum + 4.0 / (1.0 + _p_x *
           _p_x);
19   }
20   XOMP_atomic_start();
21   *sum = *sum + _p_sum;
22   XOMP_atomic_end(); XOMP_barrier();
23 }
```

(b) Transformed (or Lowered) code

Figure 2.2: PI calculation using OpenMP and its corresponding multi-threaded code generated by ROSE

11-23, the loop block is outlined as a function containing the original statements in the loop. Line 15 uses a runtime function call to split loop iterations among several threads. At line 5, the `main` function passes the outlined function's pointer to another

runtime function which will spawn multiple threads to execute the outlined function.

The initial OpenMP standard in 1997 only specified a handful of directives. Since then, many new constructs have been introduced, and most existing APIs have been enhanced in each revision [50]. The latest version of OpenMP 5.0, released in 2018, has more than 60 directives. Compiler support thus requires more effort than before [51]. A complete compiler implementation of the latest OpenMP standard for C/C++ and Fortran would involve many development efforts spanning multiple years. Furthermore, more and more researchers and developers are interested in designing various extensions to OpenMP to tame the increasing complexity of heterogeneous node designs in high-performance computing. Such extensions could be used to enhance the expressiveness, performance, or productivity of OpenMP. Support for those extensions requires a significant amount of compiler development.

2.4.6 Existing Compiler Tutorials

Both ROSE [52] and Clang [53] already have abundant documentation on their official websites, including user guides, tutorials, and Doxygen-generated API webpages, etc. There is also a ROSE wikibook which is open for anyone to contribute. Clang's official page provides documentation ranging from how to obtain and build clang, to how to write plugins and create tools, etc. Additionally, several free and open-source tutorial blogs are available for Clang. OpenMP's official page provides links [54] to several open tutorials available on the internet. However, all the existing documentation is written in the traditional text format. It is still up to the readers to find a machine to install and configure the development environment. The entire preparation phase may take hours to finish. Many learners give up due to the tedious steps or the lack of access to a suitable machine.

2.4.7 Online Education systems

There is a large amount of online learning systems [55], including Khan Academy [56], Coursera [57], edX [58] and so on. These learning systems are mostly aimed at general education and training purposes. They are not especially targeting compiler development. A closely related website is freeCodeCamp [59], which is an online training platform for training web developers. Play-with-Docker is an online sandbox for people to learn docker. Our work builds on top of this framework with customization for compiler training.

Although several cloud-based tools have been leveraged for computer science education, there is an apparent lack of such tools to teach compiler development. Ngo et al. [60] use CloudLab, a national experimentation platform for advanced computing research, to teach cluster computing to students. Bisbal [61] outlines what topics need to be taught to computational scientists in a logical order to train them in open-source software. Shin et al. [62] developed a web-based MOOC system related to computational science education, which could hold various resources and efficient programming practices. Many such tools and resources are available across several computation domains, but compiler development still lacks such online tools.

CHAPTER 3: UPIR: TOWARD THE DESIGN OF UNIFIED PARALLEL INTERMEDIATE REPRESENTATION FOR PARALLEL PROGRAMMING MODELS

3.1 Introduction

The past two decades have seen dramatically increased complexity of computer systems, including the significant increase of parallelism from 10s to 100s and 1000s computing units and cores, the wide adoption of heterogeneous architecture such as CPU, GPUs and vector units in a computer system, and the significant enhancement to the conventional memory hierarchy using new memory technologies such as 3D-stacked memory and NVRAM. Demands from users and applications for computing have also become high and diverse, ranging from computational science, large-scale data analysis, and artificial intelligence that adopts computation-intensive deep neural network methods. Together they have driven the evolution of parallel programming models to become more comprehensive and complex with multifaceted goals including delivering portable performance across diverse architectures, being highly expressible for the wide ranges of users and applications, and allowing for high-performance implementation and tools support.

Compilers have been playing a critical role to meet those goals for parallel programming. Enhancing the conventional compilation technologies and software infrastructure to be parallelism-aware has become one of the main goals of recent compiler development. However, despite the efforts to support parallelism-aware compilation in existing compilers [15, 16, 63, 14, 64, 65] and efforts to augment compiler intermediate representation (IR) with parallelism [4, 66, 67, 68, 12, 13, 9], compilers may still generate sub-optimal parallel code [69]. It is also observed that existing parallel

programming models share common parallelism functionality and use similar interfaces of essential capability for programming parallelism [5]. However, supporting these parallel models in one compiler often has to create language-dependent compiler passes of the same functionality for different models. We believe one of the barriers is the lack of language-independent abstraction of the fundamental entities and constructs for parallelism. This has hindered the research and development of parallelism-aware analysis and transformation across multiple programming models.

In this chapter, we propose *unified parallel intermediate representation (UPIR)* to enable language-neutral parallelism-aware compilation. UPIR specifies 1) three commonly used parallelism patterns, namely single program multiple data (SPMD), data parallelism, and task parallelism, including offloading tasks; 2) data attributes and explicit data movement and memory management for assisting data-aware optimization for parallel programs; and 3) synchronization operations (e.g., barrier, reduction, mutual exclusion, etc.) used in parallel programming for optimizing synchronization cost by the compiler. We create a prototype implementation in the ROSE compiler and demonstrate UPIR for unifying IR for offloading code in both OpenMP and OpenACC and in both C/C++ and Fortran. The demonstration also includes a unified transformation that lowers OpenMP and OpenACC offloading code to LLVM OpenMP runtime. UPIR is also implemented as an LLVM MLIR dialect. Thus the ROSE-based UPIR compiler can export the UPIR of a program to its MLIR dialect.

For the remainder of the chapter, in Section 3.2, we describe the design of UPIR for three kinds of parallelism, i.e., SPMD, data, and task parallelism. Section 3.3 includes the description of UPIR for specifying data attributes, data movement, and memory management used in parallel programming. In Section 3.4, we describe the UPIR for synchronization in parallel programming. Section 3.5 includes the evaluation of the UPIR to support multiple parallel models and unified compiler transformation. Section 3.6 concludes the chapter.

3.2 The UPIR for Parallelism: SPMD, Data, and Asynchronous Tasking

A programming model provides API for specifying different kinds of parallelism that either maps to parallel architectures or facilitates expression of parallel algorithms. We consider three commonly used parallelism patterns in parallel computing: single program multiple data (SPMD), data, and asynchronous tasking. In the specification, we adopt the format of MILR dialect and specified using the EBNF form used for MILR dialect.

3.2.1 SPMD Parallelism

Single program multiple data (SPMD) has been one of the most common styles of parallel programming that a program uses to start parallel execution. A program starts its parallel execution by launching multiple threads or processes on the processing units (hardware threads, cores, processors, or nodes), and they all execute the same program or code region (single program). During the parallel execution, each processing unit works on parts of the program's data (multiple data). Multiple data processing can be implemented via either programming manually, e.g., the domain decomposition method used in MPI, or via explicit data parallelism constructs (see Section 3.3). Examples of the style and language constructs for SPMD include OpenMP *parallel* constructs, most MPI programs, and manycore programming with GPU such as NVIDIA CUDA kernels or OpenCL kernels. For parallelism-aware analysis and optimization, data-race detection [70, 71], optimization of synchronization within an SPMD region [72, 73], data scoping and privatization [74], parallel divergence analysis and reduction (e.g. NVIDIA CUDA warp divergence) [75], etc, are typical techniques for improving the accuracy of data race detection and performance of SPMD regions and programs.

The IR for the SPMD parallelism model includes the notation for SPMD and a code region. To support mapping of SPMD region to the hardware threading hierarchy such

as GPUs and for languages that allow for specifying thread of hierarchy, SPMD units are organized in a two-level hierarchy, namely teams and units. To create a unified IR for different usage and variation of SPMD regions or programs, other important details must be included in the IR specification for the compiler, including for example, the target parallel systems (CPU, GPU or multi-node cluster); the synchronizations used inside and at the end of the region such as barrier or reduction, and the data environment of the region, etc. The SPMD IR in our design is shown in Figure 3.1, which is written in EBNF form. The design enhances it with several important features to cover the common usage of SPMD across different programming models and to enable more advanced analysis and optimization for SPMD code regions.

```
1  upir.spmd ::= 'spmd' spmd-field-list
2  spmd-field-list ::= spmd-field | spmd-field spmd-field-list
3  spmd-field ::= target | num_teams | num_units | data | nested-parent |
       nested-child | nested-level | branch | sync
4  target ::= 'target' '(' target-list ')'
5  target-list ::= target-item | target-item ',' target-list
6  target-item ::= 'cpu' | 'gpu' | 'cluster'
7  num_teams ::= 'num_teams' '(' expr-int ')'
8  num_units ::= 'num_units' '(' expr-int ')'
9  data ::= 'data' '(' data-list ')'
10 data-list ::= data-item | data-item ',' data-list
11 nested-level ::= 'nested-level' '(' expr-int ')'
12 nested-parent ::= 'nested-parent' '(' expr-id ')'
13 nested-child ::= 'nested-child' '(' expr-id ')'
14 branch ::= 'branch' '(' expr-id-list+ ')'
15 sync ::= 'sync' '(' expr-id-list ')'
```

Figure 3.1: UPIR MLIR dialect for SPMD parallelism specified using EBNF form

3.2.1.1 Specification for Data and Memory Management

The SPMD IR includes the specification, using the *data* field, for the attributes of data and memory used by the SPMD region. For example, the *data* field can be used to specify whether a variable or an array is shared or private across processing units, which could correspond to the *shared* and *private* clauses of OpenMP for *parallel* di-

rective. Our UPIR design for data specification includes fields and parameters for specifying more detailed information about data access, e.g. read-only, read-write access, data distribution and mapping, and memory management. This facilitates more advanced compiler analysis and optimization that involve data sharing and movement, such as data race detection and enabling overlapping between data movement and computations [76]. The data specification is explained in detail in Section 3.3.

3.2.1.2 Synchronization of an SPMD Region

During the execution of an SPMD region, it is often that the work units interact with each other via synchronization, e.g. barrier, and communications, e.g. shuffling or broadcasting data between units. We use the term "synchronization", in short, "sync", in a broader sense to refer to such interaction. In Section 3.4, we describe the design of synchronization UPIR for various types of sync operations. The UPIR for the SPMD region allows for including a field to point to the UPIR objects for the synchronization used in the region. This facilitates compiler analysis and optimization before the actual sync operation inside the SPMD region, enabling advanced optimization on the global level in coordination with the local level where the actual sync operations are specified. For example, the compiler can fuse a reduction operation with a barrier operation and can eliminate redundant barriers [77, 78] used inside the SPMD region.

3.2.2 Data Parallelism

Data parallelism, by which multiple processing units perform the same operations on different data items, refers to the patterns or parallel APIs of decomposing computation and data among parallel processing units. It is often programmed as parallel loops and often inside an SPMD region. Depending on the target architectures including CPUs or GPUs, SIMD or vector units, and multi-node clusters, parallelization of data parallel loops is often programmed differently.

For CPUs, GPUs or multi-node architecture, the common approach of programming data parallelism is to associate (implicitly or explicitly) the data parallel loops with an SPMD region, and then use the provided language constructs to prescribe how the loop iterations should be distributed to the processing units of the SPMD region. OpenMP worksharing-loop constructs, and OpenACC loop constructs are typical constructs for annotating a loop for data parallel execution. For SIMD or vector units, the "parallelization", known as vectorization of data parallel loops, is performed by the compiler. Some programming languages provide language extensions or APIs for the user to prescribe how the compiler can do vectorization, e.g., the SIMD directive of OpenMP.

The UPIR design for data parallelism, shown in Figure 3.2, includes two IRs: 1) the IR for specifying canonical loops, 2) the IR for specifying loop parallelization target and details such as schedule and chunk size. The separation allows for more flexibility and independence for the compiler to apply different loop transformations (e.g. tiling and unrolling) and parallelization (worksharing-loop or vectorization) passes compared to combining them into one IR. The IR for canonical loop specification includes information for loop trip (induction variable, range, and step), collapsible level, and its data environment and sync operations such as reduction.

The IR for loop parallelization specifies three options of parallelizing canonical loops: worksharing, SIMD, and taskloop. The worksharing parallelization, more specifically, SPMD worksharing, is specified with information such as schedule policy (static, dynamic, guided, etc), chunk size and the distribution target of the SPMD region (teams or units or both). This is similar to the OpenMP standard as OpenMP includes a comprehensive list of options on how a canonical loop can be scheduled. Worksharing-annotated loops must be within an SPMD region. For SIMD parallelization, the IR includes fields and parameters such as simdlen. Clauses for OpenMP SIMD directives represent a rich set of options that we can cherry-pick as fields in

```
upir.loop ::= 'loop' loop-field-list
loop-field-list ::= loop-field | loop-field loop-field-list
loop-field ::= induction-var | lowerBound | upperBound | step | data |
    collapse | sync
induction-var ::= 'induction' '(' expr-id ')'
...
collapse ::= 'collapse' '(' expr-int ')'
sync = ::= 'sync' '(' expr-id-list ')'

upir.loop-parallel ::= 'loop_parallel' lp-list
lp-list ::= lp-list-item | lp-list-item lp-list
lp-list-item ::= worksharing | taskloop | simd

worksharing ::= 'worksharing' '(' ws-field-list ')'
ws-field-list = schedule | distribute | schedule distribute
schedule ::= 'schedule' '(' schedule-parameter ')'
schedule-parameter ::= schedule-policy | schedule-policy ',' chunk-size
schedule-policy ::= 'static' | 'dynamic' | 'guided' | 'runtime' | 'auto
    '
chunk-size ::= expr-int
distribute ::= 'distribute' '(' 'teams' | 'units' | 'teams,units' ')'
simd ::= 'simd' '(' simdlen ')'
simdlen ::= 'simdlen' '(' expr-int ')'
taskloop ::= 'taskloop' '(' taskloop-field-list ')'
taskloop-field-list ::= grainsize | num_tasks | grainsize num_tasks
grainsize ::= 'grainsize' '(' expr-int ')'
num_tasks ::= 'num_tasks' '(' expr-int ')'
```

Figure 3.2: UPIR MLIR dialect for data parallelism

the IR.

The taskloop parallelization is the approach that many programming models use to parallelize a loop using an implicit parallel runtime system, typically tasking. For example, cilk_for of OpenCilk, taskloop of OpenMP, for_each in Rayon, Kokkos::parallel_for, RAJA::forall, tbb::parallel_for. Taskloop allows the runtime to be more flexible for scheduling loop iterations, in comparison to worksharing, since it does not require to be within an SPMD region and has less restriction than worksharing for the user to provide schedule details. There are two important fields for taskloop parallelization, grain size and num_tasks that are used to control the granularity (or the number of tasks) of the taskloop.

The current design includes the common and essential attributes among existing parallel programming models for specifying canonical loops and parallelization options that the compiler can apply. The separation of IRs for loop and parallelization allows for flexible addition of other transformations such as tiling and unrolling before or after parallel transformation of nested loops. As a foundation, more information that is specific to a programming model about loops and parallelization can be added, without comprising the generality of parallelism among programming models.

3.2.3 Asynchronous Task Parallelism

In contrast to data parallelism which involves performing the same operations on a different part of the data, task parallelism is distinguished by running many tasks that perform different operations at the same time on the data. Being asynchronous means that a task (parent task) can spawn another task (child) and then the parent task continues its execution without waiting for the child task to complete. Starting from Cilk with C/C++ language extensions of adding *spawn* and *sync* API for creating and synchronizing asynchronous tasks on shared memory computing systems [79], task parallelism has become popular in mainstream programming models such as the *task* and *taskwait* directives of OpenMP, std::async and std::future starting from C++11. The concept of asynchronous tasking has been extended for supporting offloading computation and data movement on GPU devices, or for launching computation on remote computer nodes. E.g. in CUDA, launching an asynchronous offloading kernel on GPU or an asynchronous memcpy operation can be considered as asynchronous tasking. For OpenMP and OpenACC, the directives used for offloading computation are considered as launching offloading tasks. Efforts for distributed tasking introduced in related work such as PaRSEC [80, 81] have explored programming API and runtime systems of using remote tasking for applications including dense linear algebra or other scientific applications [82].

Tasking parallelism allows users to express the full potential of parallelism that

exists in the application and its algorithm implementation. The approach of compiler transformation and optimization of tasking impacts the overhead of task management as well as the policy and priority of task scheduling. For example, for compiler transformation to support asynchronous tasking, a task code region is often outlined as a function used in tasking management (in many tasking implementation such as LLVM and GNU OpenMP), or a re-entrant function needs to be created by the compiler that contains the task code region (Cilk adopted this approach [79]). The approach of using re-entrant functions in Cilk and cactus stack [83] often incurs lower task management overhead than the outline approach, but requires more sophisticated compiler transformation. Choosing help-first or work-first work-stealing policy of tasking [84] also requires compiler to correctly transform the tasking code region. For analysis and optimization, may-happen-in-parallel analysis has been used often for asynchronous tasking [85, 86, 87]. The analysis gives compiler and users quantitative guidance for tuning the task granularity (thus the amount of prescribed parallelism in a program) to strike the balance between overhead of task management and load-balance for the runtime systems.

In consideration of aforementioned techniques of compiler transformation to support tasking, the UPIR design must consider to include the required attributes in a tasking IR to support those transformation. Existing work such as Tapir [4] demonstrated the successful usage of a tasking IR for compiler to generating high performance tasking code on shared memory systems. The design of UPIR for asynchronous tasking advances these related state of the art in at least three aspects: 1) unifying the three kinds of tasking into one IR: conventional tasking on shared memory systems, offloading tasking for accelerators, and remote tasking for distributed systems; 2) allowing the specification of data attributes of the task data environment; 3) including more fields that are used in many programming models, such as the field for task synchronization, task dependency, and target CPU, device or remote node for

spawning a task. The specification of the UPIR for tasking is shown in Figure 3.3. The tasking parallelism in UPIR has been implemented based on outlining. However, it has not been mapped to all the related constructs in the parallel models, like *taskloop* directive in OpenMP. Users can also specify a task scheduling policy to guide the compiler transformation.

```
1  upir.task ::= 'task' task-field-list | 'task' offload task-field-list |
       'task' remote task-field-list
2  task-field-list ::= task-field | task-field task-field-list
3  task-field ::= depend | data | sync | scheduling-policy
4  offload ::= 'offload' '(' device ':' device-id ')'
5  remote ::= 'remote' '(' device ':' device-id ')'
6  device ::= 'nvptx' | 'amd' | 'fpga' | 'host'
7  device-id ::= expr-int
8  depend ::= ... // similar to OpenMP depend clause
9  scheduling-policy ::= 'policy' '(' policy-item ')'
10 policy-item ::= 'help-first' | 'work-first'
```

Figure 3.3: UPIR MLIR dialect for tasking

3.3 The UPIR for Data Attribute, Explicit Data Movement and Memory Management

With respect to parallelism, data usage by parallel processing units is another dimension of complexity for parallel programming. Fundamentally, there are two kinds of operations that matter to the performance that compiler and users focus on optimizing: data movement (implicit such as paging or caching, or explicit such as memcpy), memory allocation and deallocation (memory management, mm in short). Most language-based programming models, such as OpenMP, OpenACC, and PGAS models, provide language constructs for users to specify data usage attributes and let the runtime determine when and how the data movement and mm operations are performed. Library-based programming models such as MPI, CUDA and OpenCL provide APIs for those two operations that have to be explicitly invoked in a program.

A comprehensive UPIR design should include IR fields for specifying both data attributes, as well as explicit data movement and mm operations to enable compre-

hensive compiler optimization across multiple programming models, such as memory-aware compilation [88], data-aware compilation [89] and compiler-guided data placement [90], overlapping data movement with computation with the help of compiler analysis [91]. Thus, we design UPIR to have three IR classes: data attribute, explicit data movement, and explicit memory allocation and de-allocation.

3.3.1 UPIR for Specifying Data Attributes

For a data item such as a variable or an array (section) that is used during the parallel execution of an SPMD region, a data parallel loop, or a task, the data attribute could include as many as six fields: 1) shared or private attribute if it is used in the shared memory system, 2) mapping attribute if it is used between discrete memory space, 3) attribute for access modes such as read-only, read-write and write-only, 4) memcpy attribute that is used to specify what memcpy API should be used when the data needs to be moved, 5) mm attribute (allocator and deallocator) that specifies what memory allocator and deallocator should be used when a new memory is needed for the data item, e.g. privatizing or mapping the data item, and 6) distribution attribute if an array (section) needs to be partitioned and distributed onto computing units. These data attributes of a data item are used to specify the intention of how data should be used for parallel execution. They do not specify when the data movement and memory allocation should happen. This leaves to compiler and runtime to apply optimization, such as combining memory allocation for multiple data items and scheduling data movement to achieve overlapping computation and movement. The IR for data attribute is described in Figure 3.4.

Programming models such as OpenMP and OpenACC provide language construct for users to specify some of these attributes of a data item, e.g. OpenMP/OpenACC *shared* and *private* clause for specifying the 1) shared-private attribute, OpenMP *map* clause and OpenACC *copyin/copy/copyout* clause for the 2) mapping attribute, and the *allocate* and *alloc* clause in OpenMP and related clause in OpenACC for specifying

```
1  upir.data ::= 'data' '(' data-list ')'
2  data-list ::= data-item | data-item ',' data-list
3  // four dimensions: data-mapping is different from data-sharing
4  data-item ::= expr-id '(' data-sharing ',' data-mapping ',' data-access
       ',' data-distribution-list ',' data-mm-allocator ',' data-mm-
       deallocator ',' data-memcpy ')'
5  data-sharing ::= data-sharing-property | data-sharing-property '('
       visibility ')'
6  data-sharing-property ::= 'shared'|'private'|'firstprivate'|'
       lastprivate'
7  data-mapping ::= data-mapping-property | data-mapping-property '(' data
       -mapping-modifier-list ')'
8  data-mapping-modifier-list ::= data-mapping-modifier | data-mapping-
       modifier ',' data-mapping-modifier-list
9  data-mapping-modifier ::= visibility | data-mapper
10 data-mapper ::= ssa-id
11 data-mapping-property ::= 'to' | 'from' | 'tofrom' | 'allocate' | 'none
       '
12 visibility ::= 'implicit' | 'explicit'
13 data-access ::= 'read-only' | 'write-only' | 'read-write'
14 // three aspects to describe the data distribution
15 data-distribution-list ::= data-distribution-item | data-distribution-
       item ',' data-distribution-list
16 data-distribution-item ::= unit-id | pattern | data-section
17 unit-id ::= 'unit-id' '(' expr-id ')'
18 pattern ::= 'pattern' '(' pattern-item ')'
19 pattern-item ::= 'block' | 'cyclic' | 'linear' | 'loop'
20 data-section ::= 'section' '(' array-section+ ')'
21 array-section ::= '[' expr-id ':' expr-id ':' expr-id ']'
22 data-mm-allocator ::= 'allocator' '(' allocator-attr ')'
23 allocator-attr ::= 'default_mem_alloc' | 'large_cap_mem_alloc' |
       expr_id
24 data-mm-deallocator ::= 'deallocator' '(' deallocator-attr ')'
25 deallocator-attr ::= 'default_mem_dealloc'|'large_cap_mem_dealloc'|
       expr_id
26 data-memcpy ::= 'memcpy' '(' memcpy-attr ')'
27 memcpy-attr ::= expr_id //the memcpy function id
```

Figure 3.4: UPIR MLIR dialect for data attributes

the 5) allocator attribute. For those attributes that are not explicitly specified in a program, the language applies default rules that the compiler can use to append the corresponding attributes.

3.3.2 UPIR for Explicit Data Movement and Memory Management

The IRs for data movement operations are used for specifying the actual operations that would incur moving data from one location to another. Such operation could be explicitly specified in a program using language-provided constructs, such as the *update* directive in OpenACC and *target update* directive in OpenMP, or analyzable by compilers for known data-movement APIs such as *memcpy* or *cudaMemCpy*.

Similar to the data movement operations, IRs for memory management operations specify the memory allocator and de-allocator supported by language constructs or those operations that can be analyzed by compilers for API calls such as *malloc*, *cudaMalloc*, *hbm_alloc*, *pmem_alloc*, etc. Making those operations as part of the IR and to be analyzable would facilitate the optimization and code transformation. Users can specify the information to guide the compiler to allocate the data to desired locations. For example, a huge array can be allocated to the memory space with high capacity by using *allocator (large_cap_mem_alloc)* so that it will not cause the out-of-memory error. The UPIR specification for both data movement and mm are provided in Figure 3.5.

```
1  upir.data_movement ::= 'data_movement' '(' dest-target, dest-ptr, src-
       target src-ptr, dm-size ')' dm-direction data-memcpy dm-field-list
2  dest-target ::= expr_id
3  ...
4  dm-direction ::== forward|backward //to allow two direction of data
       movement
5  dm-field-list ::= dm-field | dm-field dm-field-list
6  dm-field ::= depend | ...
7
8  //upir.data_update is simplified data movement IR.
9  upir.data_update ::= 'data_update' '(' data-list ')' dm-direction data-
       memcpy dm-field-list
10 data-list ::= expr_id-list
11 upir.mm-allocator ::= 'mm_allocator' '(' allocator-attr ')'
12 upir.mm-deallocator ::= 'mm_deallocator' '(' deallocator-attr ')'
```

Figure 3.5: UPIR MLIR dialect for explicit data movement and memory management

In summary, with this UPIR design for data attributes, data movement and memory management operations, they are able to provide adequate information for the compiler to enable advanced data-flow and data-aware analysis and optimization for parallel programs. There are two guidelines of the design to aid that compiler transformation and optimization: 1) data attributes provide a rich set of information about data sharing, but leaving data movement and memory management operations as compiler optimizations for the purpose of achieving maximum overlapping and pipelining of computation and data movement, the fusion of data movement and memory allocation, etc. 2) data movement and memory management operations explicitly specified in the program are also analyzable and optimizable with regards to other operations including computation and implicit operations of them rendered by the data attributes of data items, and synchronizations such as barrier and reductions.

3.4 The UPIR for Synchronization, Communication, and Mutual Exclusion

This group of IR elements is for representing the language constructs and APIs that prescribe the behaviors of communications and coordination operations between parallel work units. We categorize those operations into three sub-groups: 1) those that involve all participating units, referred to as collectives such as barrier (OpenMP barrier, MPI_Barrier, etc), broadcast (MPI_Bcast), reduction in OpenMP/OpenACC/MPI, etc; and 2) those that involve only two units, namely point-to-point (p2p) operations, such as data shuffling between threads and message passing between two MPI processes, and 3) mutual exclusion and locks/unlocks that involves collective participation but one primary unit at a time such as OpenMP single, atomic, etc. For most of those operations, there could be synchronous and asynchronous versions. Using asynchronous operations helps achieve overlapping of synchronization/communication with computations.

Many of those operations are provided or implemented as runtime APIs, thus com-

piler transformation of those operations could be simply converting the languages constructs to runtime API. For optimization, previous work has shown that the compiler can optimize the use of this group of constructs, such as reducing redundant barriers or global synchronizations [77, 78], compiler-assisted optimization of MPI calls [92, 93], and optimizing mutual excluded code sections that use heavy locks [94]. Converting synchronous operations to asynchronous ones by the compiler is also an effective way of optimization for the synchronizations for parallel programs [95, 96]. Thus adequately representing those constructs in compiler IR is necessary for compiler transformation and parallelism-aware optimizations.

The design of UPIR elements for synchronization operations needs to consider two aspects: 1) to unify the IRs for various types of synchronizations, and 2) to unify the IRs for synchronous and asynchronous synchronizations. For the first aspect, we consider four fields that are common among the various types of sync operations: 1) the primary unit that participates in the operation, e.g. the thread that collects the results in reduction operation, or the source process in an MPI_Bcast operation, 2) the secondary unit (s) that participate in the operation, e.g. the receivers of a broadcast or an MPI_Recv, 3) the computation or operation that is performed with the synchronization, e.g. reduction or broadcasting, and 4) the data that is used in the synchronization.

For the second aspect, which is to unify the IRs for the synchronous and asynchronous versions of each operation, we consider two steps when a sync operation, particularly collective syncs, are performed. The first step is arrive-compute, which indicates that a work unit arrives at the sync point and performs the necessary computation or operations such as an addition in an add-reduction, or sending or receiving messages for broadcast operations. The second step is wait-release, which indicates that a work unit waits for the synchronization to be performed by all participating units and then is released to continue. For synchronous synchronization, the two

steps are performed by one API call. For asynchronous version, two API calls, one for each step are performed, thus allowing adding computation between these two calls to achieve overlapping of synchronization with computation. These two steps are also similar to lock and unlock operations for mutual exclusion operations. With these two aspects of unification, we believe the designed IR would facilitate much more uniform compiler passes for the optimization of synchronizations than using independent IRs for each operation. Following this design, we describe the specification of the IRs for the synchronizations in Figure 3.6.

```
1  upir.sync ::= sync-name sync-async primary secondary operation data-
       list implicit
2  sync-name ::= 'barrier' | 'reduction' | 'taskwait' | 'broadcast' | '
       allreduce' | 'send' | 'recv' | 'single' | critical | atomic
3  sync-async ::= 'sync' | 'async' step
4  step ::= 'arrive-compute' | 'wait-release'
5  primary ::= 'primary' '(' sync-unit ')'
6  secondary ::= 'primary' '(' sync-unit ')'
7  sync-unit ::= 'task' | 'thread' | 'rank' ':' unit_id
8  unit-id ::= expr_id | '*'
9  operation ::= //sync-specific operation, e.g. add for add-reduction
10 data-list ::= expr-id-list
11 implicit ::= | 'implicit'
```

Figure 3.6: UPIR MLIR dialect for synchronization

3.5 Evaluation

In this section, we present a prototype implementation of the UPIR in the design and describe how the UPIR facilitates compiler transformation. We show how UPIR supports the source code in CUDA, C/C++, and Fortran with OpenMP/OpenACC. The performance evaluation compares our compiler with the OpenMP and OpenACC compilers of LLVM, GCC, and NVIDIA.

Our prototype is implemented in the ROSE source-to-source compiler. ROSE compiler supports C/C++, Fortran, Java, CUDA, and several other languages. It provides source-to-source transformation and a rich set of APIs for program trans-

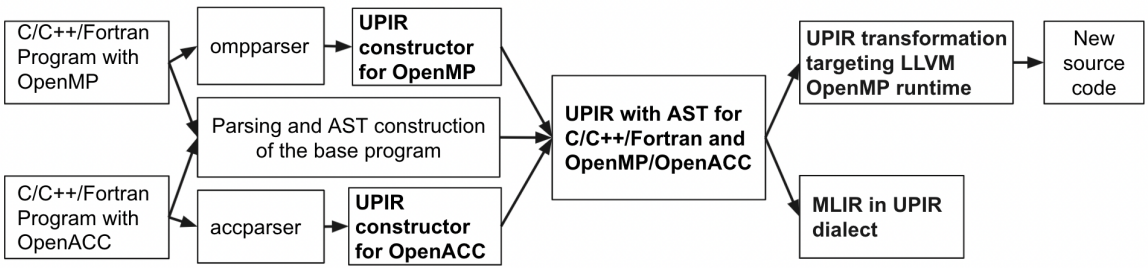


Figure 3.7: UPIR implementation in ROSE compiler to support C/C++/Fortran and OpenMP and OpenACC

formation for quick prototyping with high quality. It enables us to implement the UPIR and the support for multiple parallel models much more productively than any other compiler framework, such as GCC or LLVM. OpenACC is a parallel programming language similar to OpenMP but more focused on accelerators. The original ROSE compiler does not support OpenACC. However, by adding an independent OpenACC directive parser to ROSE, we can handle OpenACC source code and convert it to UPIR. The most significant advantage of using UPIR for both is sharing a unified transformation.

Figure 3.7 shows how UPIR is generated from OpenMP and OpenACC source code, in both C and Fortran, and followed by a unified transformation. ROSE uses EDG and Open Fortran parser to parse the C/C++ and Fortran source code, respectively. We integrate a separate OpenMP parser ompparser [97] and an OpenACC accparser into ROSE to parse OpenMP and OpenACC directives. Their parser IRs are converted to the unified UPIR, regardless of the base language of the source code. A data analysis module is implemented to collect explicit and implicit data usage information and populates the UPIRs with the complete data attribute.

The UPIR is also implemented with LLVM TableGen to produce the UPIR dialects in MLIR, allowing the ROSE implementation of the UPIR to be exported to MLIR (Figure 3.9, 3.12). Thus, besides the transformation for UPIR in the ROSE compiler, developers can also work on the exported MLIR using the LLVM toolchain. The

exported MLIR includes the same parallelism information as our UPIR implementation in ROSE, e.g., data attributes and movements in a parallel region (lines 3-6 in Figure 3.9). However, LLVM and MLIR are more complicated than ROSE and require much more effort to implement the transformation of the supported feature.

```
1 void axpy (float* x, float* y,
      int a, int n) {
2 #pragma omp target parallel for
       num_threads(1024)
3     for (int i = 0; i < n; i++)
4         y[i] = y[i] + a * x[i];
5 }
```

(a) OpenMP

```
1 void axpy (float* x, float* y,
      int a, int n) {
2 #pragma acc parallel loop
      num_workers(1024)
3     for (int i = 0; i < n; i++)
4         y[i] = y[i] + a * x[i];
5 }
```

(b) OpenACC

Figure 3.8: AXPY in OpenMP and OpenACC for GPU offloading

```
1  func @axpy(%arg0: memref<*xi32, 8>, %arg1: memref<*xi32, 8>, %arg2: i32
       , %arg3: i32) {
2    ... // %2, %3, %4, %5 are the data used in the parallel region
3    %2 = upir.parallel_data_info(x, shared, implicit, tofrom, implicit,
         read-only)
4    %3 = upir.parallel_data_info(y, shared, implicit, tofrom, implicit,
         read-write)
5    %4 = upir.parallel_data_info(a, shared, implicit, tofrom, implicit,
         read-only)
6    %5 = ...
7    %c6_i32 = constant 1024 : i32
8    upir.task target(nvptx) data(%2, %3, %4, %5) {
9      upir.spmd num_units(%c6_i32 : i32) data(%2, %3, %4, %5) target(gpu)
           {
10       %c0 = constant 0 : index
11       %c1 = constant 1 : index
12       upir.loop induction-var(%arg4) lowerBound(%c0) upperBound(%arg3)
             step(%c1) {
13         upir.loop-parallel worksharing {
14           ...
15 } } } } }
```

Figure 3.9: AXPY in UPIR MLIR dialect, for OpenMP and OpenACC GPU Offloading of Figure 3.8

Given the OpenMP and OpenACC versions of AXPY in Figure 3.8, identical UPIRs

are generated (Figure 3.9) because they both reveal the same parallelism information. Our implementation produces the same lowered source code from these two inputs. It saves much effort by not developing different lowering modules for the two languages. If necessary, language-specific transformations can be appended after the common ones. The exported MLIR in UPIR dialect can be translated to other dialects or co-operate with them. For instance, the UPIR of AXPY above can be converted to MLIR OpenACC dialect (Figure 3.10). Developers who are more familiar with OpenACC can apply their optimizations upon OpenACC MLIR translated from UPIR.

3.5.1 Using UPIR to Represent CUDA Kernel and Launching

CUDA is another popular parallel programming language designed for NVIDIA GPUs. However, unlike OpenMP and OpenACC, which use simpler directive annotations, it requires additional effort to learn the programming APIs. The language-independent design of UPIR can take both advantages. UPIR can represent the parallelism and data usage of CUDA kernel calls. Figure 3.11 and 3.12 show a CUDA version of AXPY and its UPIR. The `task` IR with `device` attribute indicates that the kernel runs on NVIDIA GPU. `num_teams` and `num_units` attributes of inner `spmd` IR correspond to blocks and threads of the CUDA kernel. The `task` and `spmd` IRs are always perfectly nested since they are converted from one CUDA kernel call.

```
1 func @axpy(%arg0: memref<f64>, %arg1: memref<f64>, %arg2: f64, %arg3:
     i32) {
2   %c6_i32 = constant 1024 : i32
3   acc.parallel num_workers(%c6_i32: i32) {
4     %c0 = constant 0 : index
5     %c1 = constant 1 : index
6     acc.loop worker {
7       scf.for %arg4 = %c0 to %arg3 step %c1 {
8         ...
9 } } } }
```

Figure 3.10: AXPY in OpenACC MLIR dialect translated from UPIR shown in Figure 3.9

The produced UPIR can be unparsed to other parallel programming languages, such as OpenMP. Doing so allows us to run CUDA kernels on the CPU and other computing devices. In addition, the new source code in OpenMP or OpenACC is also easier to program. Based on the same concept, lowering certain UPIRs to the CUDA source code is possible.

3.5.2 Performance Evaluation of using UPIR in Compiler Transformation for OpenMP and OpenACC Offloading

UPIR aims to unify the representation of parallelism across multiple parallel programming languages. It helps compilers conduct a unified transformation for multiple parallel programming models. Therefore, our performance demonstration shows that implementing transformations on top of UPIR in a compiler can support both OpenMP and OpenACC. We pick four offloading kernels for evaluation: AXPY,

```
1 __global__ void axpy_kernel(float* x, float* y, int a, int n) {
2     int i = blockDim.x * blockIdx.x + threadIdx.x;
3     if (i < n) y[i] = y[i] + a * x[i];
4 }
5 void axpy(float* d_x, float* d_y, int a, int n) {
6     axpy_kernel<<<(n+255)/256, 256>>>(d_x, d_y, a, n);
7 }
```

Figure 3.11: AXPY source code in CUDA

```
1 func @axpy_kernel(%arg0: memref<*xi32, 8>, %arg1: memref<*xi32, 8>, %
      arg2: i32, %arg3: i32) { ... }
2 func @axpy(%arg0: memref<*xi32, 8>, %arg1: memref<*xi32, 8>, %arg2: i32
      , %arg3: i32) {
3   ... // %2, %3, %4, %5 are the data used in the parallel region
4   upir.task device(nvptx) data(%2, %3, %4, %5) {
5     upir.spmd num_teams(%1 : i32) num_units(%c256_i32_0 : i32) data(%2,
          %3, %4, %5) target(gpu) {
6       call @axpy_kernel(%arg0, %arg1, %arg2, %arg3) : (memref<*xi32, 8>,
            memref<*xi32, 8>, i32, i32) -> ()
7 } } }
```

Figure 3.12: AXPY in UPIR MLIR dialect, for CUDA of Figure 3.11

matrix multiplication, matrix-vector multiplication, and 2D stencil. They are implemented in both OpenMP and OpenACC. The uses of OpenMP and OpenACC directives in these kernels represent these two models' commonly used parallel constructs in many applications [98]. These examples demonstrate the use of UPIR for SPMD (OpenMP's *parallel* and *teams*, OpenACC's *parallel*), data parallelism (OpenMP's *distribute* and *for*, and OpenACC's *loop*), offloading tasks (OpenMP's *target*, OpenACC's *parallel*), data attributes (OpenMP's *map*, *private*, *shared*, *target data* and OpenACC's *data* and *copyin/copyout/copy*), sync (OpenMP's *barrier*, OpenACC's *wait*), etc. We experimented with an extensive range of problem sizes for each kernel for performance collection but only reported the problem sizes that sufficiently demonstrated the performance trends. Each kernel is executed ten times, and the collected execution time is the average of the ten execution. Thus, we believe the selected kernels are representative enough to serve the purpose. The evaluation compared our ROSE-based UPIR compiler, NVIDIA HPC SDK, and GCC compile, all for both OpenMP and OpenACC, and Clang/LLVM for OpenMP only since LLVM does not support OpenACC. The execution time is presented in a log scale for better readability. Our experimental platform has 2 CPUs (20 cores for each), 512 GB of RAM, and one NVIDIA V100 GPU with 32 GB of HBM. The system runs Clang/LLVM 14.0, NVIDIA HPC SDK 22.1 with CUDA toolkit 11.5, and GCC 11.2 on Ubuntu 20.04. All compilations enable -O3 flag.

The performance results of the four compilers (UPIR, NVIDIA, GCC for OpenMP/OpenACC, and LLVM for OpenMP) are shown in Figure 3.13, 3.14, 3.15, 3.16. For the OpenMP version, our implementation can achieve up to 1.28x speedup over LLVM, and 25.89x speed up over GCC on average for all the problem sizes we selected. For the OpenACC version, UPIR shows up to 235.1x speedup over NVIDIA compiler and 1.15x speedup over GCC on average for all the evaluated problem sizes. We believe LLVM considers more general offloading cases and thus might introduce

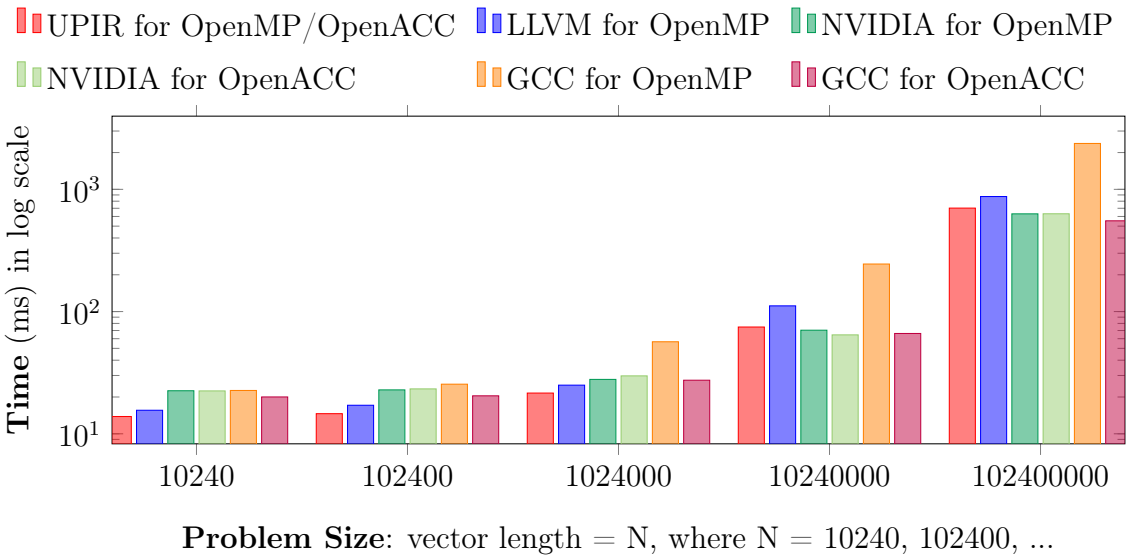


Figure 3.13: AXPY performance of UPIR compiler, LLVM, NVIDIA, and GCC compilers

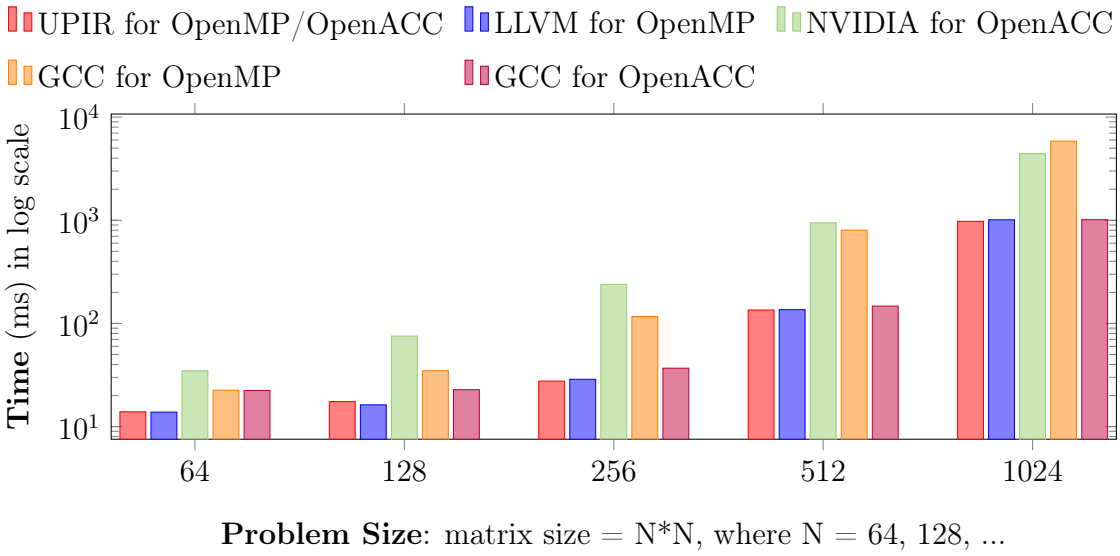
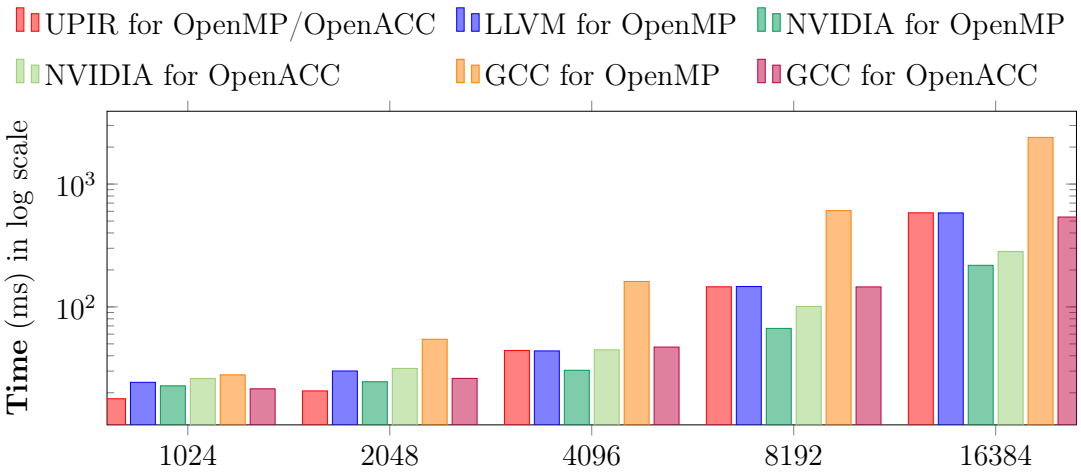


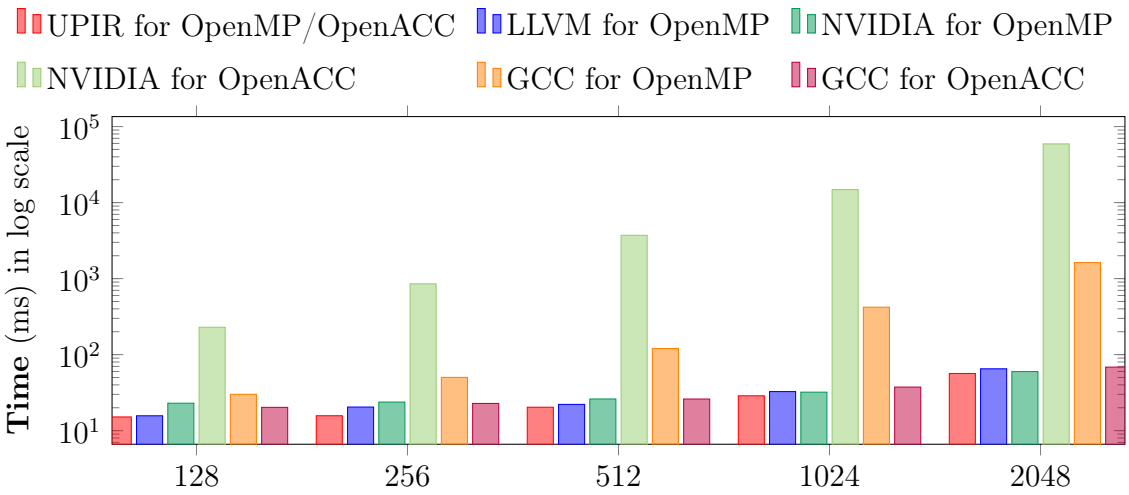
Figure 3.14: Matrix multiplication performance of UPIR compiler, LLVM, NVIDIA, and GCC compilers

more overhead in kernel launching and thread management internally. For example, LLVM uses the state machine [14, 99] to manage dynamic threading on NVIDIA GPU with relatively heavy overhead. Thus, we use dynamic parallelism introduced in CUDA 5.0 for threading management. It directly launches the number of threads for nested kernels as demanded on the device. It prevents the communication over-



Problem Size: vector length = N, matrix size = N*N, where N = 1024, 2048, ...

Figure 3.15: Matrix-vector multiplication performance of UPIR compiler, LLVM, NVIDIA, and GCC compilers



Problem size: filter size = 7, array size = N*N, where N = 128, 256, ...

Figure 3.16: 2D stencil performance of UPIR compiler, LLVM, NVIDIA, and GCC compilers

head to the host and does not need to maintain the state of redundant threads. This demonstration shows that UPIR can assist in the unified compiler transformation. It significantly reduces the compiler programming effort to support multiple parallel programming models because we do not need to maintain several similar versions of compiler transformation. Moreover, UPIR does not sacrifice performance to achieve

this goal.

3.5.2.1 Performance Analysis

We conducted a detailed performance analysis of the compilers for OpenMP and OpenACC programs. GCC utilizes the unified IR GIMPLE and maps both OpenMP and OpenACC code to GIMPLE for later transformation. Theoretically, the compiled executable should lead to the same performance if they are semantic equivalent. However, in all four kernels, the OpenACC version outperforms the OpenMP version significantly. We notice that the number of GPU threads per block used in the kernels is limited to 256, even if 1024 is specified in the source code. This issue only exists in the OpenMP version but not the OpenACC version. GCC always follows the specified number of threads in the OpenACC code. Although GCC adopts the idea of unified parallel IR, it fails to deliver consistent performance for the same kernel in different languages.

A similar problem happens to LLVM. LLVM determines the maximum number of threads supported by GPU. If the specified value exceeds the limit, it will be reduced to that threshold. However, in our experiments, LLVM does not always obtain the threshold correctly. It may use a different number of threads instead.

The latest NVIDIA HPC SDK supports both OpenMP and OpenACC programs. It uses different sets of IR for compilation, such as `__nvomp_*` in OpenMP and `__pgi_uacc_*` in OpenACC. Our implementation of OpenMP and OpenACC code is semantic equivalent. Thus, it should lead to a similar executable generated from the same compiler. However, the NVIDIA compiler does not always show consistent performance. For stencil and matrix multiplication, the OpenACC version is much slower than the OpenMP version and the executable built by other compilers. The profiling shows that the generated GPU code of stencil in OpenACC takes about 71 million cycles while the code generated by LLVM only takes around 50 thousand cycles. About 99% of elapsed kernel time is spent on `__acc_wait` for synchronization.

Additionally, the NVIDIA compiler can compile the OpenMP version of these two test cases, but the execution results in either a kernel launching failure or incorrect computation. It indicates that the transformation for OpenMP and OpenACC in the NVIDIA compiler is not unified.

To sum up, GCC and NVIDIA compilers support both OpenMP and OpenACC, and we notice some unification in their IR and transformations for different parallel programming models. However, they do not deliver consistent performance for the OpenMP and OpenACC programs of the exact computation and parallel semantics. In contrast, the UPIR compiler can support OpenMP/OpenACC code with unified transformation and achieves the same performance.

3.6 Summary

In this chapter, we present UPIR, a unified parallel intermediate representation used for representing parallelism of parallel programming models to assist parallelism-aware compiler analysis, transformation, and optimization. It is designed to support a wide variety of parallel programming models, and the prototype implementation in the ROSE compiler supports C/C++/Fortran, OpenMP, OpenACC, and CUDA. UPIR enables a unified compiler transformation for multiple parallel programming models, including OpenMP and OpenACC. Our experiments show that the UPIR compiler utilizes the unified transformation to compile both OpenMP and OpenACC programs. It achieves promising performance and saves much development effort for supporting new programming models by leveraging the UPIR and the unified transformation. UPIR provides a comprehensive, flexible, and extensible compiler IR designed for compiler development targeting modern heterogeneous parallel systems. The unified IR would enable lots of exciting research and accelerate the implementation of supporting new programming models in a compiler.

CHAPTER 4: SUPPORT DATA SHUFFLE BETWEEN THREADS IN OPENMP

4.1 Introduction

OpenMP has been known for productive shared-memory programming on multi-core, multi-processor, and many-core homogeneous systems. Data movement between computing elements such as cores or CPUs is implicit via memory. The recent specification introduced `target`-family constructs for specifying offloading data and computation to accelerators whose memory is physically separate from the host CPU memory. E.g., the `map` clause can explicitly specify data movement between the host's memories and an accelerator GPU. From OpenMP users' perspective, data sharing and movement between parallel threads and tasks must go through the memory system, implicitly or explicitly.

For manycore accelerators such as GPUs and vector architectures, data can be copied between registers of multiple computing elements such as cores or vector lanes without going through the memory and cache system, using shuffle or permutation operations. For example, NVIDIA introduced shuffle instruction from Kepler architecture to conduct data transfer between registers of different threads in a warp. The feature enables a multi-thread kernel to synchronize vector-like operations within a warp. When shared data is small and can reside in the register within a warp of threads (32 threads), those threads can access registers from each other. Considering that register access latency could be 10x and 100x smaller than SRAM and DRAM, respectively, this data shuffle feature between threads could significantly improve the computation performance of worksharing or vector loops.

In this chapter, we present two approaches to using shuffle in OpenMP. First, we provide a high-performance runtime implementation of the reduction clause using

shuffle. Then, a new directive and a new clause, named `shuffle`, are introduced for programmers to specify explicit data movement between threads. The `shuffle` clause is used to specify the data that can be shuffled between threads and the directive to specify when and how the data are transferred. While the motivation is to support shuffling data between cores or vector lanes via registers on many-core and vector architectures, the support generally is designed to bypass slow memory for data movement between threads via explicit data shuffle operation. We develop and evaluate a prototype implementation of the proposed support using reduction and stencil algorithms. The shuffle implementation always delivers the best performance with up to 2.39x speedup compared with other high-performance implementations. Compared with standard OpenMP offloading code for 2D stencil, our shuffle implementation delivers superior performance for as many as 25x better. We also provide a study of simulated shuffle using shared memory on NVIDIA GPUs to demonstrate how to support this extension on hardware without native shuffle support.

In the rest of the chapter, Section 4.2 shows the high-performance implementation of reduction using shuffle instruction. Section 4.3 presents the `shuffle` extension to OpenMP with syntax details and how to use it for 2D stencil. Then we show the performance evaluation in Section 4.4. At last, we conclude this chapter in Section 4.5.

4.2 Using shuffle to implement the `reduction` clause

In parallel computing, reduction is a typical operation for aggregating partial results. For multi-thread programming, it repeatedly applies the same operation by multiple threads with partial results. The final result resides in one thread. Figure 4.1 shows the sum reduction using OpenMP. The task is offloaded onto an accelerator that has multiple teams of threads to perform the reduction operation. Within a team, data from all threads are accumulated. Then those partial results are reduced into one final result and can be copied back to the host. Data and operations are

performed off the global DRAM memory on GPUs by default. An optimized implementation can use NVIDIA's shared memory (SRAM) to accelerate the reduction operations, e.g., the reduction from the official CUDA examples of NVIDIA.

```
1  // prerequisite data declaration and computing
2  #define BLOCK_SIZE 64
3  float src[N] = ...;
4  #pragma omp target teams distribute parallel for map(to: src[0:N]) map(
       from: sum) num_teams(N/BLOCK_SIZE) num_threads(BLOCK_SIZE) reduction
       (+: sum)
5    for (i = 0; i < N; i++)
6      sum += src[i];
```

Figure 4.1: Sum reduction using OpenMP

Reduction in OpenMP can be implemented in CUDA using shuffle operations and other optimization techniques. Such implementation can be done in the runtime system, thus requiring minimum compiler transformation. In Figure 4.3, we show an implementation similar to the one presented in [3]. This algorithm divides the whole input in the global memory into multiple tiles, and each block on GPU reads a tile to its shared memory. Using shuffle, threads in the same warp share their partial results directly between private registers as soon as they are available. Only the results from warps will be reduced in the shared memory.

For comparison, the same algorithm can be implemented using CUDA-shared memory, and the algorithm can be used for GPUs with no native shuffle instruction. The implementation is shown in Figure 4.3. For each variable that needs to be shuffled, an array of block sizes is created underneath so that each thread in that block can maintain a copy of the variable in that array. From the user's point of view, the simulated shuffle can still directly access the private data of another thread even though they didn't declare the shuffle variable as shared data. Comparing the two implementations in Figure 4.2 and Figure 4.3, it is shown that their algorithms are identical, and the only difference is the implementation of the shuffle function.

```
1  template <class T>
2  __inline__ __device__ T warpReduceSum(T val) {
3    for (int offset = warpSize/2; offset > 0; offset /= 2)
4      val += __shfl_down_sync((unsigned int)-1, val, offset);
5    return val;
6  }
7  template <class T>
8  __global__ void reduce(T *g_idata, T *g_odata, unsigned int n) {
9    T mySum = ...; // prepare the local partial sum per thread
10   mySum = warpReduceSum<T>(mySum);
11   int lane = threadIdx.x % warpSize;
12   int wid = threadIdx.x / warpSize; // warp id
13   if (lane == 0) sdata[wid] = mySum; // the partial result of a warp
14   ... // rest of reduction
15 }
```

Figure 4.2: Reduction implementation using native shuffle

4.3 Proposing `shuffle` clause and directive for OpenMP

As discussed in Section 4.1 for the current OpenMP memory model, sharing data between threads must go through memory systems. This is defined based on the fact that most existing multi-core and many-core architectures only allow data sharing between functional units via memory. Shuffle primitives enable direct data movement between threads, hence function units of a system, allowing data sharing by bypassing memory system. For the second contribution of this work, we experiment with high-level language support of data sharing between threads without using any kind or level of memory. We introduce a `shuffle` clause and a `shuffle` directive to OpenMP for such an experiment.

First, the `shuffle` clause can be used with `parallel` and `teams` directives to declare the shuffling variables. Its syntax is simply as "`shuffle (`*src-variable-list*`)`", in which the *src-variable-list* specifies the variables that can be shuffled. Compared with the two similar clauses that are used in OpenMP to specify data sharing attributes, the `shared` or `private` clauses, variables that are annotated to be shuffled are read-only shared variables to other threads and access to the variable must use the

```
1  template <class T>
2  __inline__ __device__ T warpReduceSum(T val) {
3    T *buffer = SharedMemory<T>();
4    int lane = threadIdx.x % warpSize;
5    int wid = threadIdx.x / warpSize;
6    buffer[threadIdx.x] = val;
7    __syncthreads();
8    for (int offset = warpSize/2; offset > 0; offset /= 2)
9      if (lane + offset < warpSize) {
10         val += buffer[wid*warpSize + lane + offset];
11         buffer[threadIdx.x] = val;
12         __syncthreads();
13     }
14   return val;
15 }
16 template <class T>
17 __global__ void reduce(T *g_idata, T *g_odata, unsigned int n) {
18   T mySum = ...; // prepare the local partial sum per thread
19   mySum = warpReduceSum<T>(mySum);
20   int lane = threadIdx.x % warpSize;
21   int wid = threadIdx.x / warpSize; // warp id
22   if (lane == 0) sdata[wid] = mySum; // the partial result of a warp
23   ... // rest of reduction
24 }
```

Figure 4.3: Reduction kernel using simulated shuffle

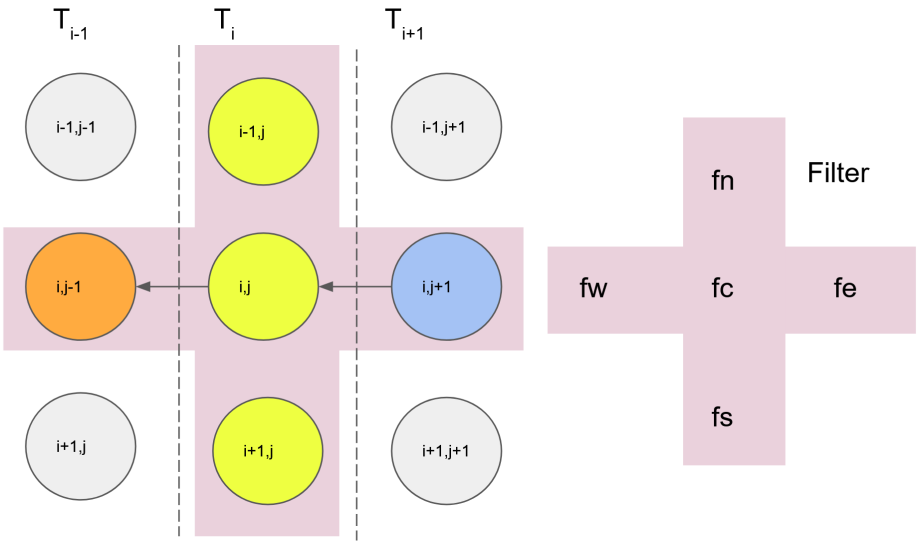


Figure 4.4: 2D 5 points stencil using shuffle. Each circle indicates an original pixel. Each thread loads three pixels. Pixels in color are involved with the computation of pixel (i,j). Arrow represents the shuffle direction

```
1  // prerequisite data declaration and computing
2  float src[N], dst[N], fw, fc, fe, fn, fs, sum, BLOCK_SIZE = ...;
3  #pragma omp target teams map(to: src[0:N], fw, fc, fe, fn, fs) map(from
       : dst[0:N]) num_teams(N/BLOCK_SIZE)
4  #pragma omp parallel num_threads(BLOCK_SIZE) shuffle(sum) // declare
       sum for shuffle
5      {   // prepared needed data, such as global index of src item and
           dst item
6          int global_index[3], index = ...;
7          sum = src[global_index[1]] * fe; // partial sum1
8          #pragma omp shuffle down(-1, 1, sum, sum) // thread n shuffles
               sum from thread n+1 and replace its own sum copy
9          sum += src[global_index[0]] * fn;
10         sum += src[global_index[1]] * fc;
11         sum += src[global_index[2]] * fs; // partial sum2
12         #pragma omp shuffle down(-1, 1, sum, sum)
13         sum += src[global_index[1]] * fw; // partial sum3
14         dst[index] = sum; // write the final result to output array dst
15     }
```

Figure 4.5: 2D 5 points stencil using shuffle OpenMP extension

```
1  // prerequisite data declaration and computing
2  float src[N], dst[N], fw, fc, fe, fn, fs, sum, BLOCK_SIZE = ...;
3  int N = width*height;
4  #pragma omp target map(to: src[0:N], fc, fn0, fn1, fw1, fw0, fe1, fe0,
       fs1, fs0, height, width) map(from: dst[0:N])
5  #pragma omp teams distribute parallel for num_teams(N/BLOCK_SIZE)
       num_threads(BLOCK_SIZE) collapse(2) schedule(static, 1) shuffle(sum)
6  for (int i = 0; i < height; i++) {
7    for (int j = 0; j < width; j++) {
8      sum = src[i*width+j+1] * fe;
9      #pragma omp shuffle down(-1, 1, sum, sum)
10     sum += src[(i-1)*width+j] * fn;
11     sum += src[i*width+j] * fc;
12     sum += src[(i+1)*width+j] * fs;
13     #pragma omp shuffle down(-1, 1, sum, sum)
14     sum += src[i*width+j-1] * fw;
15     dst[i*width+j+1] = sum;
16   }
17 }
```

Figure 4.6: 2D 5 points stencil using worksharing and shuffle OpenMP extension

```
1  __global__ void stencil(const float* src, float* dst, ...,
2         float fc, float fn, float fw, float fe, float fs) {
3    // prepared needed data, such as global index of src item and dst
         item
4    int global_index[3], index = ...;
5    sum = src[global_index[1]] * fe; // partial sum1
6    sum = __shfl_down_sync(0xFFFFFFFF, sum, 1);
7    sum += src[global_index[0]] * fn;
8    sum += src[global_index[1]] * fc;
9    sum += src[global_index[2]] * fs; // partial sum2
10   sum = __shfl_down_sync(0xFFFFFFFF, sum, 1);
11   sum += src[global_index[1]] * fw; // partial sum3
12   dst[index] = sum; // save the result back to the output array
13 }
```

Figure 4.7: 2D stencil kernel using `shuffle` instructions

`shuffle` directive proposed. The `shared` clause indicates read-write sharing among all threads, while the `private` clause indicates that the data are only available to the thread itself.

Second, the proposed `shuffle` directive is an executive directive to specify how exactly the data should be shuffled between registers of different threads. It must be used within a `parallel` or `teams` region. The syntax is: "`shuffle` *clause*", and the *clause* must be in the following format:

"`sync|up|down(`*mask-modifier[,] src-modifier[,] dst-variable [operator], shuffle-variable*`)`"

The shuffle directive operates moving data of a shuffled variable from a source thread or lane (specified by the *src-modifier*) and then accumulating the data using a specified operation (the *operator*) with a variable (the *dst-variable*, and then storing the result in the variable. The *mask-modifier* is a mask to indicate which threads to participate in shuffle operation, similar to the first parameter in the CUDA's `shuffle` primitives. The *src-modifier* is used to specify the threads or lanes that supply the data. For the `sync` shuffling, which stipulates that all participating threads shuffle data from a single source thread, the *src-modifier* is the absolute warp or lane ID, such as 25 or 31. For `up` and `down` clauses, the *src-modifier* is used to indicate the

relative distance between the participating thread and the source thread. *operator* is the operation to be applied to the shuffled data, which could be =, +=, -=, \=, and so on. It equals *dst-variable = dst-variable operator shuffle-variable*. The default operator is = if none is specified. The *shuffle-variable* must be the variables specified by the `shuffle` clause.

Currently, the most usage of `shuffle` operation is for using GPUs because of its availability on NVIDIA and AMD GPUs. On CPU and other platforms, shuffle can be easily implemented using shared memory, and performance can be optimized by taking advantage of the last level of shared cache. While our proposal is one approach to exposing these features to users, shuffle can be used in other approaches, such as via runtime function or used with `metadirective` or `declare variant` for performance optimization. Yet those approaches require knowledge and skills in CUDA and OpenCL programming. One limitation of this proposal is that the use of `shuffle` directive may render incorrect execution of the OpenMP code if OpenMP compilation is turned off since the use of `shuffle` requires parallel SIMD-type of data movement between variables of the same symbol.

4.3.1 Stencil Example

In stencil, a filter is applied to each pixel and several pixels around it to get a new value for that position. Since a pixel can be involved multiple times during computing, loading several pixels to register once in one thread and completing all the computations would be faster than multiple threads all loading the pixels from global memory repeatedly. Taking a 2D 5 points stencil as an example, to compute the pixel (i, j), it needs four adjacent pixels and itself. We consider these 5 pixels as three columns handled by three threads (Figure 4.4). Each thread calculates a partial sum and passes it to the left neighbor. The leftmost thread collects all partial results and gets the final result. In this example, thread T_{i+1} computes $sum1_{i,j} = P_{i,j+1} \times fe$ and passes it to thread T_i. Thread T_i computes $sum2_{i,j} = P_{i-1,j} \times fn + P_{i,j} \times$

$fc + P_{i+1,j} \times fs$ and passes $sum1_{i,j} + sum2_{i,j}$ to thread T_{i-1}. Then thread T_{i-1} computes $sum3_{i,j} = P_{i,j+1} \times fw$. As the last step, thread T_{i-1} stores the final result $sum_{i,j} = sum1_{i,j} + sum2_{i,j} + sum3_{i,j}$ to a proper location.

Figure 4.5 shows a simplified 2D 5 points stencil using `shuffle` constructs. As we described above, each thread reads 3 points. It generates two partial results and passes them to neighbors. Two partial sums are retrieved back as well. At last, three partial results corresponding to three filter columns are combined as the final result. Figure 4.6 presents a worksharing version of 2D stencil. The nested loop is flattened by `collapse`. `schedule` clause ensures that the threads next to each other process continuous pixels to pass intermediate results correctly.

We create a prototype implementation in CUDA to demonstrate how the compiler would transform the OpenMP code in Figure 4.5. It doesn't perform shuffle operations across the whole team. Instead, the operation is mapped to a warp on NVIDIA GPU, which means the shuffle is conducted within a warp. For other platforms, it depends on what native shuffle instruction is available and how it works on the hardware level. The `shuffle` operations are implemented using native `shuffle` instructions and shared memory. In Figure 4.7, the intermediate results of a column of pixels are shuffled between adjacent threads at lines 6 and 10, which correspond to lines 8 and 12 in Figure 4.5. Each thread makes the maximum use of the pixels and produces all the possible results from them. Then it exchanges the partial results among the private register of neighbors via shuffle to avoid shared memory access. The shuffle instruction can be simulated using shared memory at lines 8-14 and 18-24 so that the code will support the devices without native shuffle (Figure 4.8). They still share the same kernel function.

4.4 Experimental Results

The experimental platform used for reduction has a 12 cores Intel Xeon W-2133 CPU, 32 GB DRAM, and one NVIDIA Quadro P400 GPU with 2 GB of memory.

```
1  __global__ void stencil(const double* src, double* dst, ...,
2         double fc, double fn, double fw, double fe, double fs) {
3    // prepared needed data, such as global index of src item and dst
         item
4    int global_index[3], index = ...;
5    // an array shared in a block to exchange sum between threads
6    __shared__ double shared_sum[BLOCK_SIZE];
7    float sum = src[global_index[1]] * fe;
8    shared_sum[thread_id] = sum;
9    __syncwarp();
10   if (lane_id < warpSize) { // lane_id is the thread id within a warp
11     shared_sum[thread_id] = shared_sum[thread_id+1];
12     __syncwarp();
13     sum = shared_sum[sumId];
14   }
15   sum += src[global_index[0]] * fn;
16   sum += src[global_index[1]] * fc;
17   sum += src[global_index[2]] * fs;
18   shared_sum[thread_id] = sum;
19   __syncwarp();
20   if (lane_id < warpSize) {
21     shared_sum[thread_id] = shared_sum[thread_id+1];
22     __syncwarp();
23     sum = shared_sum[thread_id];
24   }
25   sum += src[global_index[1]] * fw;
26   dst[index] = sum; // save the result back to the output array
27 }
```

Figure 4.8: 2D Stencil kernel using shuffle simulated by shared memory

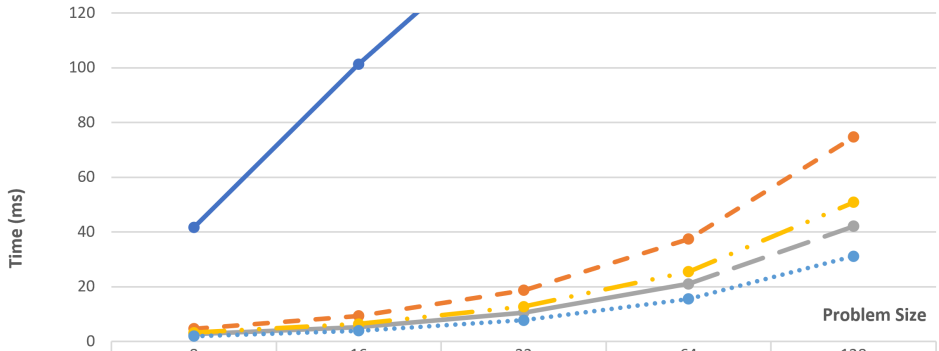
The other platform that is used for stencil has two 18-core Intel Xeon E5-2699 v3 CPUs, 256 GB DRAM, and two NVIDIA Tesla K80 GPUs with 24 GB of memory. Both systems run Ubuntu 18.04 LTS and NVIDIA CUDA SDK 10.2.

As a baseline, `omp target teams distribute parallel for` is used to implement reduction and stencil, and then compiled by Clang/LLVM 10.0.1 with `-O3` parameter. Thus the baseline performance completely depends on the transformation and optimization by Clang/LLVM compiler. The kernel time on GPU is measured as execution time, the time cost of data transfer is not included. There are four more versions of implementation to be evaluated, including accessing global memory

directly, using shared memory as software cache for a tile of loop tiling, using shared memory to simulate shuffle, and using native shuffle. The version using shared memory as software cache for loop tiling is considered a highly optimized implementation on NVIDIA GPUs [100].

In both tests, the baseline OpenMP version is much slower than the rest four versions. The native shuffle version is about 20x faster than the baseline. Besides the shuffle instruction, the reason could be that the manually transformed CUDA code and the baseline OpenMP code compiled by LLVM have a different mechanism of parallelization. It may lead to various memory access behaviors, such as coalesced memory access versus uncoalesced memory access.

4.4.1 Reduction



	8	16	32	64	128
OpenMP	41.672	101.3	154.62	266.95	528.96
Global Memory	4.67	9.36	18.69	37.38	74.69
Shared Memory	2.68	5.27	10.52	21.02	41.99
Shared-Memory Simulated Shuffle	3.23	6.37	12.76	25.48	50.86
Native Shuffle	1.98	3.93	7.79	15.56	31.13

Figure 4.9: Performance of reduction

The input is an array of a given size that is filled with randomly generated numbers. We can see the native shuffle version is the fastest, as expected since it has the least amount of access to slower memories (Figure 4.9). It shows up to 25x better performance than the standard OpenMP version and 2.39x speedup over the global memory version. The version using shared memory to simulate the shuffle instruction is slower than the second version that uses shared memory without shuffle. It's

reasonable because the simulated shuffle requires more resources to maintain an array to share data and it performs more synchronizations in the block to keep atomic and data consistency.

Table 4.1: Memory accesses for reduction. N: problem size=G*B, G: grid size=32768, B: block size=256, W: warp size=32

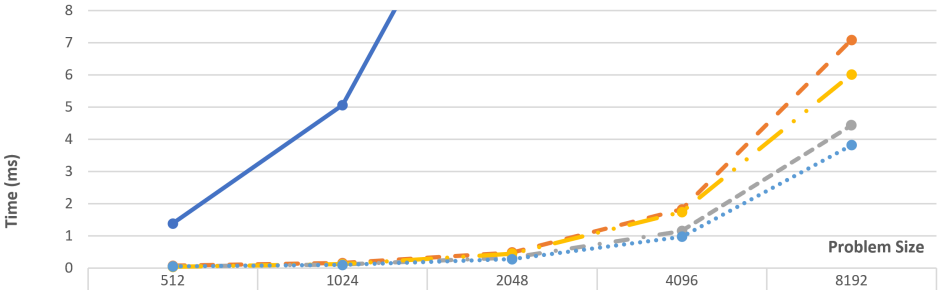
	Global Memory	**Shared Memory**	**Shared Memory Simulated Shuffle**	**Native Shuffle**
Global memory access	2*G*B + 2*G*f(B)	2*G*B	2*G*B	2*G*B
Shared memory access	0	2*G*f(B)	2*G*f(B)	2*G*W
Cross-bock synchronization	2*G	2*G	2*G*(f(B)+1)	2*G

The memory access of reduction can be modeled in Table 4.1. $f(x)$ is the amount of memory operations for reducing x numbers, where $f(x) = \sum_0^k 2^k$ and $k = \log_2 x$. Different versions incur different amounts of memory access to each memory. In the global memory version, all those accesses occur in the global memory. In the shared memory version, the elements are reduced in the shared memory. It reads and writes this memory location $f(B)$ times, respectively. Since there are G blocks, the total number of shared memory access is $2 * G * f(B)$.

For the native shuffle version, within a warp, the elements are reduced among registers directly. Then the partial results from all warps in the same block are reduced in the shared memory as usual or via shuffle again. The simulated shuffle shares the same operations. However, it accesses shared memory $2 * G * f(B)$ times to shuffle data. It also requires two more synchronizations to make the operation atomic and prevent data race. Given one shuffle operation per iteration of reduction, the additional amount of cross-block synchronization is $2 * G * f(B)$.

According to the analysis above, the performance improvement of native shuffle over the shared memory version is from the much less access to shared memory. The time overhead of the simulated shuffle is caused by excessive cross-block synchronization, which is a trade-off between performance and compatibility.

4.4.2 2D Stencil



	512	1024	2048	4096	8192
OpenMP	1.386	5.06	14.683	50.124	181.64
Global Memory	0.074	0.157	0.486	1.826	7.086
Shared Memory	0.06	0.116	0.326	1.156	4.432
Shared-Memory Simulated Shuffle	0.041	0.123	0.449	1.743	6.014
Native Shuffle	0.052	0.1	0.281	0.977	3.82

Figure 4.10: Performance of 2D 9 points stencil

Table 4.2: Memory accesses for 2D stencil N: problem size=4*G*B, G: grid size=N/4B, B: block size=128, W: warp size=32

	Global Memory	**Shared Memory**	**Shared Memory Simulated Shuffle**	**Native Shuffle**
Global memory access	G*B*4*10	G*B*9	G*B*9	G*B*9
Shared memory access	0	G*B*(8 + 4*9)	G*B*4*16	0
Cross-bock synchronization	0	1	4*4*2	0
Shared memory size used	0	B*8	B	0

The input of this test is an automatically generated image by random numbers. The results present a similar trend between the four versions of reduction experiments (Figure 4.10). The native shuffle version has the best performance. According to the breakdown of memory accesses, this version has the least amount of slower memory accesses and cross-block synchronizations (Figure 4.2). The average speedup of the native shuffle over the hand-written tiled shared memory version, which has been highly optimized, is 1.11. While sharing the same source code, the simulated shuffle version suffers from the overhead of cross-block synchronization.

4.5 Summary

Data shuffling between threads or lanes of many-core GPUs allows data copy between threads without involving the memory system. It could be exploited to im-

prove computing performance when there is a large amount of data communication between threads. In this work, we experiment with two approaches of using shuffle in the OpenMP high-level programming model, 1) a high-performance runtime implementation of reduction clause; and 2) a proposed `shuffle` extension to OpenMP to let users specify when and how the data should be moved between threads. Superior performance improvement has been achieved and demonstrated when using shuffle to implement the reduction and 2D stencil kernels. While the effort of correctly programming using shuffle primitive is significant, our language extension to allow users to use it in high-level programming models can reduce its complexity. These explorations and experiments prove that shuffle instructions should be exploited in compiler code generation and application optimization for performance improvement.

CHAPTER 5: REX: A SOURCE-TO-SOURCE OPENMP COMPILER FOR PRODUCTIVE RESEARCH OF PARALLEL PROGRAMMING

5.1 Design

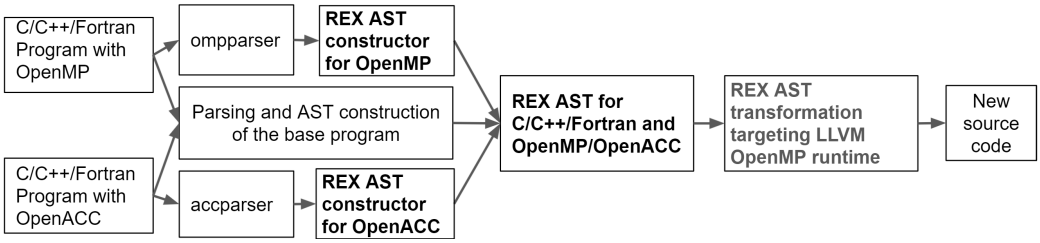


Figure 5.1: REX: a source-to-source OpenMP compiler based on ROSE

The REX compiler is a source-to-source compiler that targets OpenMP and provides preliminary support for OpenACC (as illustrated in Figure 5.1). It is built on top of the ROSE compiler, enhancing its OpenMP capabilities with features such as using the LLVM OpenMP runtime instead of GOMP, supporting more OpenMP constructs, and improving existing transformations like the `target parallel for` directive. The differences between REX and ROSE are thoroughly discussed in Section 5.2.

The REX compiler is designed with modularity in mind. The OpenMP parser, OpenACC parser, and LLVM OpenMP runtime library are integrated as submodules, allowing users to build their version of the OpenMP/OpenACC parser and OpenMP runtime library as long as they adhere to the API. This flexible design enables users to customize REX to their specific needs and requirements.

To concentrate solely on the research and education of OpenMP, we have rebranded the ROSE compiler. The focus of the rebranding effort is to create a C/C++/Fortran OpenMP compiler that facilitates research and education in the field. To achieve this

goal, we have removed support for languages such as PHP/Java and binary analysis. The repository has also been cleaned up to minimize unnecessary dependencies.

5.1.1 Intermediate Representation

ROSE provides a unified Intermediate Representation (IR) for OpenMP that supports both C/C++ and Fortran, streamlining the process of mapping OpenACC constructs. This enables a consistent transformation targeting the LLVM OpenMP runtime, regardless of the parallel programming model used in the input, eliminating the need for separate implementation of lowering modules for OpenMP and OpenACC.

To enhance the performance of Single Instruction Multiple Data (SIMD) operations, REX employs different compiler intrinsics tailored to the specific platform. These intrinsics transform the OpenMP `simd` directive, ensuring that the implementation of SIMD operations is optimized for each platform.

To achieve this, REX has designed a new IR set specifically for SIMD operations called the SIMD IR. This IR set enables the conversion of for loops with the SIMD clause into a three-address format, where the left-hand operand is either a memory location or a scalar variable. The SIMD IR nodes are based on binary nodes, allowing for both left and right operands. This IR set is kept in a vector, as it is not meant to represent concrete code but instead to provide an efficient implementation method.

The SIMD IR has been proven highly effective, with an almost 1:1 mapping to the underlying architecture. This mapping ensures that the SIMD operations are optimized for the specific platform, whether Intel's AVX2/AVX-512 or ARM's SVE.

5.1.2 Front End

5.1.2.1 Parsers

The REX compiler incorporates two distinct parsers, ompparser and accparser, explicitly designed for OpenMP and OpenACC. The ompparser is a standalone parser

for OpenMP, which can work as a crucial component of compilers [97]. To support OpenACC, the REX compiler has developed accparser based on ANTLR4 and follows a similar concept as the ompparser. The accparser is responsible for converting OpenACC directives into language-independent OpenACCIR, which is then passed on to the REX compiler.

Once the REX compiler receives the OpenACCIR, it is converted into the internal unified AST and integrated with the ones generated from other source codes in the base language. The conversion of OpenACCIR to the internal unified AST ensures that the REX compiler can work seamlessly with both OpenMP and OpenACC directives and provide the required parallelism analysis and optimization accordingly.

5.1.3 Middle End

Implementing a data analysis and normalization module specifically for parallelism in the REX AST is a crucial step after the construction of the REX AST. This module is essential in the REX compiler because it includes several passes that must be performed before the lowering phase. The information obtained through these passes enables REX to optimize the code efficiently while ensuring that the transformed code complies with the user's specified data attributes.

5.1.3.1 Forward Declaration

OpenMP has powerful features to aid in parallel programming, including the declarative directives `declare target` and `declare mapper`. The `declare target` directive allows users to declare variables or functions for use on devices, which can then be used directly in kernels. For the compiler to correctly find these symbols in the symbol table during analysis, the symbols specified in `declare target` must be explicitly declared on the device by the compiler. To achieve this, REX stores all symbols specified in `declare target` and inserts them into the symbol table whenever any analyses or transformations are about to be performed for targeting devices.

The `declare mapper` directive is used to specify a particular mapping pattern between host and device memory. The compiler must implement the specified pattern in the runtime to facilitate this mapping. REX helps to support this by creating a separate helper function to transfer the data according to the pattern listed in `declare mapper`. By doing so, the data transfer can be easily managed and optimized.

5.1.3.2 Data Analysis

To provide parallelism-aware analysis and optimization, REX implements a specialized pass to gather information about parallel data usage. The REX compiler is capable of extracting data attributes, such as symbol name, access pattern (write, read, or both), sharing property (shared, private, firstprivate), mapping property (from host to device, from device to host, or bidirectional), and whether the information is explicit or implicit, directly from the OpenMP and OpenACC directives or through compile-time analysis of symbols that appear within parallel regions.

The collection of this information plays a crucial role in the later stages of the compiler's analysis and transformation processes, such as data dependency analysis and kernel overlapping analysis. With the data attributes obtained from this pass, REX can make informed decisions about handling the parallel data, ultimately leading to a more efficient and optimized parallel code.

5.1.3.3 Clause normalization

As a source-to-source compiler, REX aims to transform both OpenMP and OpenACC code into human-readable, lowered source code that generic non-OpenMP compilers can handle. However, transforming OpenMP and OpenACC code into C/C++ code is not always straightforward, as it involves following implicit rules for data sharing and mapping.

The implicit rules for data sharing and mapping are specified in the OpenMP and OpenACC specifications, which can be difficult to understand or locate for users. As

a result, REX implements a clause normalization pass to explicitly specify all the symbols used in the parallel region. This lets users understand the data and its use in the generated source code.

To handle the data appropriately and follow the implicit rules, the compiler must perform various analyses during the transformation process. These analyses can interrupt the lowering process and make it more complex. By implementing the clause normalization pass, REX simplifies the transformation process and ensures that the generated code will be more transparent and easier to understand for users.

5.1.3.4 Lowering

We focus on GPU offloading in the REX compiler. The transformation details are discussed in Section 5.3 to 5.4.

5.1.4 Back End

The REX compiler is designed to provide a solution for generating human-readable and optimized source code from OpenMP and OpenACC code. The output from REX is compatible with generic non-OpenMP compilers, making it easy for users to integrate their code into existing projects or build environments.

One of the critical features of REX is its reliance on the LLVM OpenMP runtime for linking. This guarantees that the produced code will work efficiently across various platforms and systems. Furthermore, the new source code generated by REX only requires a few files from the compiler rather than the entire tool. This makes it easy for users to create a portable, self-contained collection of source code that can be used on other systems without needing REX installed.

Currently, REX supports offloading to NVIDIA GPUs via the CUDA toolkit. This makes it ideal for those looking to take advantage of the high performance and scalability GPU computing offers. In addition, it is worth noting that future releases of REX may expand its support to other platforms and architectures, such as AMD and

Intel GPUs, providing even greater flexibility and compatibility for users.

5.2 OpenMP support status

Table 5.1: Comparison of essential OpenMP support in ROSE and REX compiler

OpenMP	**ROSE**	**REX**
target	partial	add the support to a complex target region and dynamic parallelism
teams	no	add the support using LLVM OpenMP runtime
distribute	no	add the support using LLVM OpenMP runtime
target teams distribute parallel for	no	add the support using LLVM OpenMP runtime
metadirective	no	partial
simd	no	support AVX512/SVE using compiler intrinsics
task	yes	use LLVM OpenMP runtime instead
parallel	yes	use LLVM OpenMP runtime instead
map	yes	use LLVM OpenMP runtime instead
barrier / atomic / single	yes	use LLVM OpenMP runtime instead
for	yes	minor changes
reduction	yes	remove dependency on CUDA host APIs
orphaned constructs	no	no

The REX compiler is built on top of the ROSE compiler, which already has extensive support for OpenMP. With ROSE, many key OpenMP constructs, such as `parallel`, `for`, and `reduction`, can be appropriately handled. However, it lacks support for crucial directives like `teams` and related combined directives.

REX addresses these limitations by updating and enhancing the OpenMP transformations in ROSE to target the LLVM OpenMP runtime. The intermediate layer XOMP has been mostly removed, enabling users to directly modify the LLVM runtime calls in the generated source code without learning a new set of APIs.

Previously, ROSE only supported a `target` directive followed by an immediate `parallel for` directive. Any code regions not attached to this `target` directive

could not be handled. This limitation is not problematic in simple cases, such as AXPY and Sum, but it can lead to failure when compiling more complex kernels, such as the one shown in Figure 5.2.

5.3 Supporting OpenMP parallel, teams and worksharing constructs

This section only focuses on using the `teams` and `parallel` constructs on the host. GPU offloading is a separate topic in Section 5.4. In OpenMP, the `teams` directive binds a group of threads together, and each team operates as a single program multiple data (SPMD) unit. On the other hand, the `parallel` directive considers each thread as an SPMD unit. ROSE implements the transformation of the `parallel` directive through its interface, XOMP, using GOMP. However, it does not support the `teams` directive.

REX replaces XOMP and GOMP with the LLVM OpenMP runtime to support both `teams` and `parallel` directives, eliminating the need for an intermediary interface like XOMP. Users can work directly with the LLVM OpenMP runtime APIs in the new source code generated by REX. REX outlines the enclosing region first and then replaces the original constructs with the appropriate LLVM runtime calls. The `__kmpc_fork_teams` and `__kmpc_fork_call` are required for the `teams` and `parallel` constructs, respectively, while the `__kmpc_push_num_teams` and `__kmpc_push_num_threads` set the number of teams and threads. The LLVM runtime calls are inserted into the original code location, while the host-related outlined functions are stored in a separate file. This allows users to optimize or modify the outlined functions without affecting the original code.

5.4 Support OpenMP target GPU constructs

LLVM uses a state machine to manage the parallelism on GPUs. When a `target` region is encountered, LLVM starts with the maximum number of threads required, then modifies the number of threads that are actually used accordingly. In Figure 5.2,

```
1  void compute(double *x,double *y,int n,double a) {
2    int i;
3  #pragma omp target map(tofrom: y[0:n]) map(to: x[0:n])
4    { printf("Loop␣1.\n");
5  #pragma omp parallel for num_threads(1024)
6      for (i = 0; i < n; i++)
7        ...;
8      printf("Loop␣2.\n");
9  #pragma omp parallel for num_threads(128)
10     for (i = 0; i < n; i++)
11       ...;
12   }
13 }
```

Figure 5.2: An example of having mixture of parallel and sequential code in the kernel on GPU

there is a `target` region including two `parallel for` regions and two serial statements. LLVM creates a `target` region with up to 1024 threads. Of these, only one thread will execute the serial statements, while the remaining threads will stay idle. In the first `parallel for` region, all 1024 threads will be running as specified in the `num_threads` clause. In the second `parallel for` region, only 128 out of 1024 threads will be used, leading to high overhead due to excessive thread management.

To improve the performance, REX utilizes dynamic parallelism to transform such a `target` region. Dynamic parallelism is a feature provided by NVIDIA since CUDA 5.0. It allows a CUDA kernel to call another kernel using multiple child threads. The host does not need to be involved in the launching process, thus reducing communication overhead. Additionally, thread management is simplified by explicitly specifying the kernel configuration.

5.4.1 Scheduling

To achieve the best performance in an SPMD region having loop worksharing, REX will assign one iteration to each thread when the number of iterations is less than the allowable number of threads on the GPU. If the user does not specify any scheduling policy, REX will use the default scheduling policy for the transformation. If a

`schedule` clause is provided, REX will honor it and use the specified scheduling policy to assign iterations to threads. REX automatically adjusts the number of threads used to optimize performance, making sure that all threads are utilized efficiently. This helps ensure that the GPU is working at maximum efficiency and maximized performance.

5.4.2 Kernel Generation

ROSE has already implemented an outliner that meets our needs, and the loop transformation in ROSE works very well in terms of the performance and readability of the lowered source code. Therefore, we chose not to change the original implementation. REX compiler utilizes dynamic parallelism to implement the kernel generation in OpenMP. Given a `target` region, it starts with one team having one thread, and the kernel is offloaded to the GPU with this configuration. If any SPMD region is encountered, another kernel with more blocks and threads is launched directly from the device side. After the execution, the original serial kernel continues, and only the master thread runs.

Figure 5.3 presents the transformation of the source code shown in Figure 5.2. The whole `target` region of function `compute` is replaced with an LLVM runtime call at line 29 and necessary settings. The data analysis module in REX determines the data usage based on the explicit information from `map` clause and implicit rules from OpenMP specification. Then the required data and outlined function information are passed to the LLVM runtime. It calls function `OUT__3__compute__67__kernel__` on the device (line 7). Inside this base function for the `target` region, two outlined functions for the loops are called using dynamic parallelism with different kernel configurations (lines 11 and 16). Compared to the original LLVM OpenMP transformation, REX does not use the state machine, which can slow down the kernel execution. While being able to modify the extent of parallelism easily, dynamic parallelism also brings minor overhead. We measured the total kernel execution time

```
1  __global__ void OUT__1__compute__75__kernel__(int *np__,double *ap__,
       int *ip__,double *_dev_x,double *_dev_y) {
2    ...; // transformation for loop 2
3  }
4  __global__ void OUT__2__compute__70__kernel__(int *np__,double *ap__,
       int *ip__,double *_dev_x,double *_dev_y) {
5    ...; // transformation for loop 1
6  }
7  __global__ void OUT__3__compute__67__kernel__(int *np__,double *ap__,
       int *ip__,double *_dev_x,double *_dev_y) {
8    printf("Loop␣1.\n");
9  { int _threads_per_block_ = 1024;
10   int _num_blocks_ = 1;
11   OUT__2__compute__70__kernel__<<<_num_blocks_,_threads_per_block_>>>(
         np__,ap__,ip__,_dev_x,_dev_y);
12   }
13   printf("Loop␣2.\n");
14 { int _threads_per_block_ = 128;
15   int _num_blocks_ = 1;
16   OUT__1__compute__75__kernel__<<<_num_blocks_,_threads_per_block_>>>(
         np__,ap__,ip__,_dev_x,_dev_y);
17   }
18 }
19 void compute(double *x,double *y,int n,double a) {
20   int i;
21   int _threads_per_block_ = 1;
22   int _num_blocks_ = 1;
23   void *__host_ptr = (void *)(&OUT__3__kernel__67__id__);
24   void *__args_base[] = {&n, &a, &i, x, y};
25   void *__args[] = {&n, &a, &i, x + 0, y + 0};
26   int64_t __arg_sizes[] = {((int64_t )(sizeof(int ))), ((int64_t )(
         sizeof(double ))), ((int64_t )(sizeof(int ))), ((int64_t )(sizeof(
         double ) * n)), ((int64_t )(sizeof(double ) * n))};
27   int64_t __arg_types[] = {33, 33, 33, 32, 35};
28   int32_t __arg_num = 5;
29   __tgt_target_teams(OUT__3__kernel__67__id__,__host_ptr,__arg_num,
         __args_base,__args,__arg_sizes,__arg_types,_threads_per_block_,
         _num_blocks_);
30 }
```

Figure 5.3: Transformation of the source code shown in Figure 5.2

of AXPY, matrix multiplication, and matrix-vector multiplication compiled by the REX compiler and found that using dynamic parallelism only leads to 8% overhead at most.

5.5 Extending the Infrastructure to Support OpenACC

REX compiler provides preliminary support for OpenACC. At the front end, we designed an OpenACC parser based on ANTLR4, called accparser, to support the latest OpenACC 3.2 syntax and parse C/C++ and Fortran. We also developed OpenACCIR as an intermediate representation to store grammatical information.

Instead of using a dedicated OpenACC runtime, we target the LLVM OpenMP runtime in OpenACC source code transformation. Essential directives and clauses in OpenACC are mapped to the unified parallel IR of REX, which supports both OpenMP and OpenACC, making the whole process transparent to end users. They don't need to change their code significantly, only needing to link a generic LLVM OpenMP library. We also improved a commonly used benchmark Rodinia to test its effectiveness, which will be introduced in Section 5.6.

A point-to-point translation can map many essential OpenACC constructs to REX AST for OpenMP. Only *acc kernels* and *acc loop* need to be handled carefully. *acc kernels* informs the compiler that the enclosed kernel should be offloaded to the GPU, and any proper optimizations can be applied. For instance, the compiler may parallelize some loops in the given kernel. Therefore, we must comprehensively analyze the kernel and convert *acc kernels* to one or more directives for better performance. Nested loop worksharing is another problem. Unlike OpenMP, OpenACC does not require the user to specify which parallelism level the loop should be assigned. *acc loop* could be associated with gangs, workers, or vectors, and the association could be implicit and decided by the compiler.

5.6 Evaluation

We aim to demonstrate the efficiency of the REX compiler by highlighting its minimal overhead and consistent performance across different parallel programming models. To facilitate this evaluation, we have enhanced the existing Rodinia bench-

Table 5.2: Conversion between OpenMP and OpenACC essential constructs

OpenMP	**OpenACC**
target teams/parallel	parallel
distribute, for	loop
map(to/from/tofrom/alloc)	copyin/copyout/copy/alloc
target data	data
target update to/from	update device/host
atomic/critical	atomic

mark to support both OpenACC and OpenMP GPU offloading, enabling a comprehensive comparison of the compiler's performance in handling these parallel models. This evaluation's primary purpose is to validate REX's effectiveness in managing both programming models, proving that adopting UPIR and unified transformation strategies allows the compiler to deliver reliable performance with a negligible increase in overhead. By showcasing the consistency and low overhead of the REX compiler, as well as our contribution to extending the capabilities of the Rodinia benchmark, we seek to emphasize its potential to facilitate seamless integration and optimization of various parallel programming models for high-performance computing applications.

5.6.1 Experimental Platform

The hardware and software configurations for the evaluation are listed in Table 5.3.

5.6.2 Enhanced Rodinia

Rodinia is one of the most popular benchmark suites targeting multi-core CPU and GPU, developed by the University of Virginia (Table 5.4). It includes 24 benchmarks in the latest release (v3.1 in Dec. 2015).

While Rodinia is widely used for evaluation in the HPC domain, it has a few limitations. Its OpenMP variant could improve because most programs are only

Table 5.3: Experimental platform

CPU	Intel Xeon Gold 6230N CPU 2.30GHz
Cores	2 sockets × 20 physical cores
Vector Length	512-bit
RAM	512 GB
GPU	NVIDIA Tesla V100 32 GB
OS	Ubuntu 20.04 LTS
Compilers	Clang/LLVM 15.0, GCC 12.2, NVIDIA HPC SDK 22.1
CUDA	11.5

Table 5.4: Rodinia applications and kernels [2]

Application/Kernel	**Algorithm**	**Domain**
K-means	Dense Linear Algebra	Data Mining
Needleman-Wunsch	Dynamic Programming	Bioinformatics
HotSpot	Structured Grid	Physics Simulation
Back Propagation	Unstructured Grid	Pattern Recognition
SRAD	Structured Grid	Image Processing
Leukocyte Tracking	Structured Grid	Medical Imaging
Breadth-First Search	Graph Traversal	Graph Algorithms
Stream Cluster	Dense Linear Algebra	Data Mining
Similarity Scores	MapReduce	Web Mining

implemented for CPU execution. The ones that support GPU offloading are not optimized. For example, only `parallel for` is used in a `target` region. Without `teams distribute`, compilers probably use only one CUDA block, which has 1024 threads, to run the kernel. It will significantly lower the performance. Furthermore, Rodinia doesn't have an OpenACC variant. It is arguably another important parallel programming model primarily focusing on GPU offloading. In NVIDIA HPC SDK, besides CUDA, OpenACC is the only parallel programming model it supports

natively. OpenMP support is implemented with OpenACC runtime APIs and is limited to more recent NVIDIA GPUs (SM70 or newer). We notice there are unofficial versions of Rodinia porting for OpenACC. Unfortunately, they are incomplete and barely working.

These limitations are pretty understandable since Rodinia hasn't been updated since 2015. Therefore, we implemented two more variants in Rodinia: OpenMP GPU and OpenACC. We take the official CUDA and OpenMP versions as references and implement these two versions according to the latest OpenMP and OpenACC specifications.

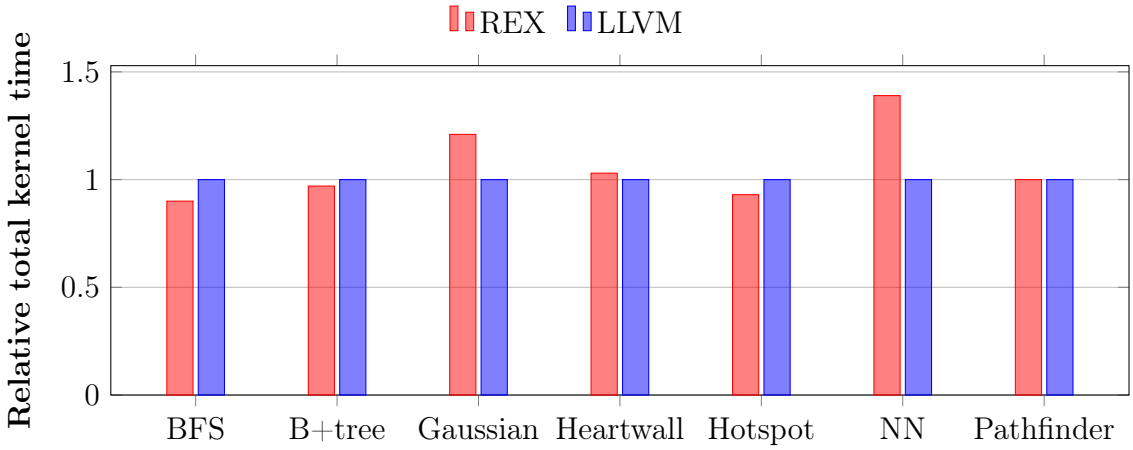


Figure 5.4: Kernel execution time of 7 benchmarks from Rodinia, including computation and data transfer. **LLVM is taken as the baseline, and its total kernel time is normalized to 1.**

Besides the original 24 benchmarks, we implemented a group of commonly used kernels to evaluate OpenMP GPU offloading [1], as a compliment to Rodinia. These kernels utilize essential OpenMP and OpenACC directives for GPU offloading, such as `omp target teams distribute parallel for` and `acc parallel loop`.

On average, REX incurs only a 6% overhead compared to the LLVM compiler, and in some cases, REX is even faster (Figure 5.4 and 5.5). REX uses the same OpenMP runtime as the LLVM compiler but can perform source-to-source transformations

[1] `https://github.com/passlab/Benchmarks`

that the latter cannot. Despite this additional functionality, REX maintains high performance without sacrificing efficiency.

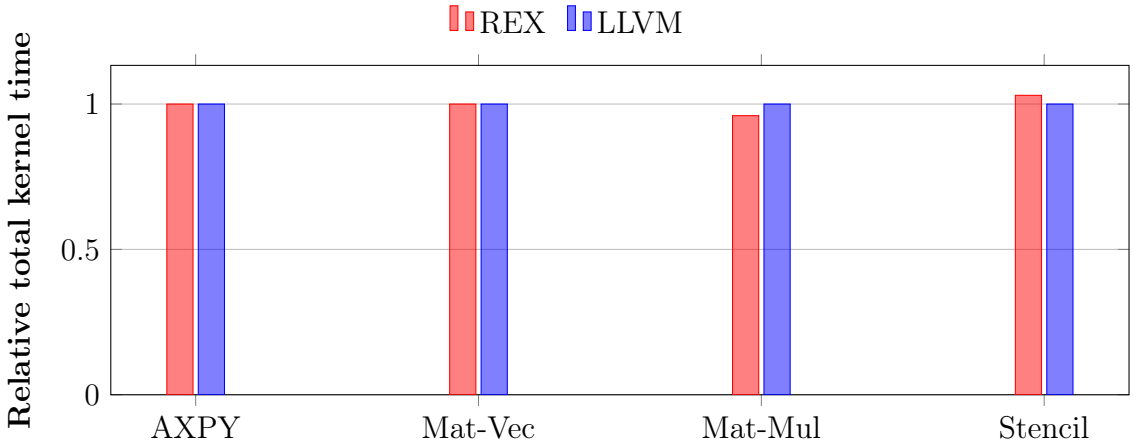


Figure 5.5: Kernel execution time of 4 home-brewed benchmarks, including computation and data transfer. **LLVM is taken as the baseline, and its total kernel time is normalized to 1.**

REX, NVC, and GCC are all compatible with OpenMP and OpenACC. Upon evaluating the OpenMP GPU and OpenACC versions, we observed performance inconsistencies and inappropriate GPU offloading configurations in NVC and GCC (Table 5.5).

Firstly, our implementations of OpenMP GPU and OpenACC variants have equivalent semantics. When using a compiler that supports both OpenMP and OpenACC, the generated binary and its performance should be identical, provided that unified transformation and runtime are employed. In the AXPY example, this trend is evident. However, the execution time for Matrix-Vector multiplication slightly differs. A closer examination of the execution time reveals that the OpenMP computation time with NVC is considerably longer than the OpenACC computation time. Similar issues are present in Matrix-Multiplication and Stencil as well. Matrix-Multiplication demonstrates that the GCC OpenMP and NVC OpenACC programs are significantly slower than the other variant compiled by the same compiler. For Stencil, the NVC OpenACC program consumes substantially more time than its OpenMP counterpart.

Table 5.5: Kernel execution time of 4 home-brewed benchmarks. Time unit: ms.

REX								
Benchmark	**OpenMP/OpenACC**							
	Compute		**HtoD**		**DtoH**		**Total**	
AXPY	44.4		4730		2574.9		7304.9	
Mat-Vec	147.8		3049.9		0.08		3049.98	
Mat-Mul	590.1		184.5		590.1		1364.7	
Stencil	6.4		83.6		111.9		201.9	
GCC								
Benchmark	**OpenMP**				**OpenACC**			
	Compute	**HtoD**	**DtoH**	**Total**	**Compute**	**HtoD**	**DtoH**	**Total**
AXPY	406.7	5715.5	2829.8	8952	42.1	5811.3	2602.3	8455.7
Mat-Vec	183.5	3057.4	1	3241.9	148.8	3211.5	1	3361.3
Mat-Mul	12357	285.7	119.6	12762.3	598.8	235.4	114.8	949
Stencil	151.5	138.4	111.6	401.5	476	117.2	115.1	708.3
NVC								
Benchmark	**OpenMP**				**OpenACC**			
	Compute	**HtoD**	**DtoH**	**Total**	**Compute**	**HtoD**	**DtoH**	**Total**
AXPY	34.6	1324	690	2048.6	36.2	1324.2	662.5	2022.9
Mat-Vec	38.8	868.6	1	908.4	13.8	868.5	1	883.3
Mat-Mul	317	54.3	28.2	399.5	3941.1	54.3	28.3	4023.7
Stencil	10.5	27.2	28.1	65.8	6607.5	27.2	28.1	6662.8

Profiling reveals that most of the extra time is spent on synchronization. Taking matrix multiplication as an example, the computing time for the NVC OpenACC version is 3583.7 ms, significantly higher than its OpenMP counterpart, which takes 307.4 ms. However, 3597.2 ms of the OpenACC computing time is spent on synchronization at the beginning of the computation, specifically on the runtime API `acc_wait`. In contrast, the NVC OpenMP version only spends 13.4 ms on `acc_wait` at the start of the computation. Moreover, the profiling results reveal that the synchronization call in the OpenACC version comprises two parts. One part stems from the Ope-

nACC pragma in the source code, taking about 14 ms. The other part, marked as unknown by the profiling tool, is generated by the compiler, not directly from the source code. This latter portion is the primary cause of the significantly longer execution time for the OpenACC version. This additional synchronization is absent in the NVC OpenMP version. Excluding this portion of time cost, the kernel time of NVC OpenMP and OpenACC becomes similar.

Since NVC and GCC utilize a single runtime for both OpenMP and OpenACC, the performance discrepancy can only be attributed to compiler transformation. They do not implement a unified transformation for inputs with identical semantics. Continuing with the matrix multiplication example, printing out the NVC compiler IR in both OpenMP and OpenACC versions reveals that the compiler IR differs despite using the same runtime. The NVC OpenMP version employs `__nvomp_*`, while the OpenACC version uses `__pgi_uacc_*`. These different IRs lead to distinct transformations and code generations.

Secondly, all benchmarks employed for evaluation specify the GPU offloading configuration, such as 256 teams and 1024 threads per team. While this configuration may not be optimal in every scenario, the compiler should respect these user-defined values as long as they are valid. However, this is not the case for GCC and NVC. Despite specifying the number of teams and threads, and the GPU supporting them on the hardware level, GCC and NVC still adopt different configurations. For example, the GCC OpenMP program consistently uses 240 teams and 256 threads per team. Conversely, the NVC OpenMP program reduces the specified 1024 threads per team to 128. While they alter the configuration in OpenMP programs, GCC and NVC adhere to the user-defined configuration in OpenACC.

We argue that this approach to compiler transformation is suboptimal. Although the modified configuration could offer some performance advantages, the execution result may not always be accurate. For instance, the user's program may rely on team

and thread indices with specific offloading configurations. The compilers risk introducing errors and unintended behavior by disregarding the user-specified settings.

5.7 Summary

In this chapter, we have introduced REX, an OpenMP source-to-source compiler that leverages the unified parallel intermediate representation (UPIR) to support essential features of OpenMP 5.1 and OpenACC 3.2. The output generated by REX is designed to target the LLVM OpenMP runtime, with the option to produce MLIR instead of new source code. Additionally, we have expanded the Rodinia parallel benchmark by implementing two more variants: OpenMP GPU and OpenACC, enabling the evaluation of GPU offloading performance across various parallel programming models and compilers.

REX uses the same OpenMP runtime as LLVM but offers additional source-to-source transformation and OpenACC support. Remarkably, REX introduces an average overhead of only 6%. Compared to GCC and NVC, which also support OpenMP and OpenACC, REX consistently delivers performance across both parallel models, thanks to the UPIR. In contrast, semantically equivalent programs compiled by GCC and NVC may exhibit significant performance discrepancies.

Moving forward, we plan to incorporate additional constructs from the latest OpenMP and OpenACC specifications, further enhancing the capabilities of REX and expanding its applicability across a broader range of high-performance computing scenarios.

CHAPTER 6: FREECOMPILERCAMP.ORG: TRAINING FOR OPENMP COMPILER DEVELOPMENT FROM CLOUD

6.1 Introduction

Due to the increasing complexity of supercomputer node architectures for high-performance computing (HPC), high-level programming models are used to improve the productivity of using supercomputers. OpenMP is considered by many as the de-facto portable programming model for exploiting node-level parallelism for supercomputers. Compiler support for OpenMP has been added in many open-source compilers, such as GNU compiler collection, Clang/LLVM, and ROSE source-to-source compiler frameworks, as well as vendor compilers from Intel, Cray, NVIDIA, and AMD. More and more researchers are interested in researching using OpenMP as a vehicle in parallel programming models, compiler technologies, and computer systems. However, one of the significant challenges in developing an OpenMP compiler and extending OpenMP language is the steep learning curve of compiler implementation and the development efforts of adding compiler support for language extensions.

Fundamentally, compiler development is a complex and time-consuming task. Although many cloud-based, online learning platforms [55, 56, 58, 60, 62] have been created for computer science education, focusing on entry-level programming courses, there is an apparent lack of such resources to teach compiler development. Even with the developer manuals of a compiler framework, it is difficult for beginners to teach themselves how to modify compilers that contain millions of lines of code. Training beginners by proficient compiler developers consume lots of time, human effort, and cost, which is not scalable in the long term.

In this chapter, we introduce an ongoing effort, FreeCompilerCamp.org, a free

and open online learning platform aimed at training researchers to quickly develop OpenMP compilers and help them learn the skills of compiler development. FreeCompilerCamp.org has several distinct features: 1) It allows anyone interested in developing OpenMP compilers to learn the necessary skills for free; 2) A live training website is set up, so a web browser and an Internet connection are the only requirements for anyone to take the training; 3) It enables those who have the relevant skills to contribute new tutorials; and 4) The entire training system is open-source so it can be deployed on a private server, workstation or even personal laptop.

The remainder of the chapter is divided as follows: Section 6.2 presents the implementation of the framework. Section 6.3 gives an overview of the design of the tutorials with a few examples. Finally, Section 6.4 consists of the conclusion and our plans.

FreeCompilerCamp.org is aimed to build a free and open cloud-based training platform integrating the solutions mentioned above. This platform aims to facilitate the training of researchers to quickly develop compilers for OpenMP and help them learn the skills of compiler development. We will elaborate the design and implementation of this platform in the next sections.

6.2 FreeCompilerCamp.org Platform

FreeCompilerCamp.org is a learning system with several distinct design principles:

- It aims to allow any developer who is interested in understanding the internal working of OpenMP compilers to learn the necessary skills for free.
- It provides a pre-configured compiler development environment in an online sandbox, eliminating the burden of beginners' tedious and error-prone software installation processes.
- A live training website based on the system is set up, so a web browser and an Internet connection are the only requirements for anyone to get the training.

- The entire training system is open-source, so it can also be deployed by anyone on a private server, workstation, or even personal laptop.

- It enables anyone with the relevant skills to contribute new tutorials as well.

There are two components in the FreeCompilerCamp.org platform (or FreeCC as an abbreviation) as displayed in Figure 6.1 – a web-based framework with all tutorials and a Play-With-Compiler (PWC) engine for the sandbox environment. The website provides a browser-based interactive interface with two panels: the left panel contains the training instructions in text, and the right panel connects with the PWC engine, which creates a live terminal sandbox for real-time practice.

Figure 6.1: Two components of FreeComplierCamp.org

6.2.1 Tutorial Website

The tutorial website is created as the major interface of FreeCC. It provides easy-to-understand documents in multiple tutorials organized by categories. Users can choose any entry on demands or learn in order.

6.2.2 Play-With-Compiler Engine

The Play-With-Compiler engine is based on Play-With-Docker (PWD) [101], which is an online sandbox platform for visitors to learn basics about container techniques using Docker [102]. Docker uses OS-level virtualization to deliver software, libraries, and configuration files in packages called containers, which are isolated from one another though there are defined channels to enable their communication. Containers on the same machine share a single operating-system kernel and are thus more lightweight than virtual machines.

Play-With-Docker uses a so-called Docker-in-Docker technique. While the host service is running in an outer docker, the component of this service runs in an isolated inner docker so that multiple components won't affect each other[103, 104]. In the case of PWD, each user has their own sandbox and won't get interrupted by others' activities. PWD uses Apline Linux, which is widely used in docker images due to its lightweight and security.

6.2.3 Customization

We encountered several technical issues during the development of FreeCompilerCamp.org and subsequently resolved them. Most of these issues may not be new in web development, but our target audience is mostly people with a HPC background, who may not have a flair for web development. Also these issues are common and will be faced by anyone who would like to deploy our framework. Hence mentioning these issues here is vital.

6.2.3.1 Same-Origin Policy

The same-origin policy [105] restricts resources loaded from one origin to interact with resources from another origin. This prohibits training website and PWC to be deployed on different servers. We had to apply Cross-Origin Resource Sharing [106] mechanism that uses additional HTTP headers to enable resources on PWC server

to be accessed by training website.

6.2.3.2 Port Conflict

Later to simplify management and lower the cost, we decided to deploy both the training website and PWC on the same server. This caused port conflict since they both use port 80 by default. We set up an HTTP server using Apache and non-default ports redirection to resolve this conflict.

6.2.3.3 Alpine Linux

The PWD sandbox had dockers built from Alpine Linux, which was unfit for compiler training. Compilers are sensitive to the host system environment. Alpine Linux is not supported for the development of either ROSE or LLVM. Therefore, we created new docker images based on Ubuntu for better compatibility with both ROSE and LLVM. Ubuntu has a much wider application support, hence if future even more compilers can be added in the tutorial.

6.2.3.4 Security

The PWD sandbox by default gives users root access inside the terminal. This is a security risk since a malicious user may hack into web hosting directories where they are not supposed to access. As a solution we create a user/group (freecc/freecc) in

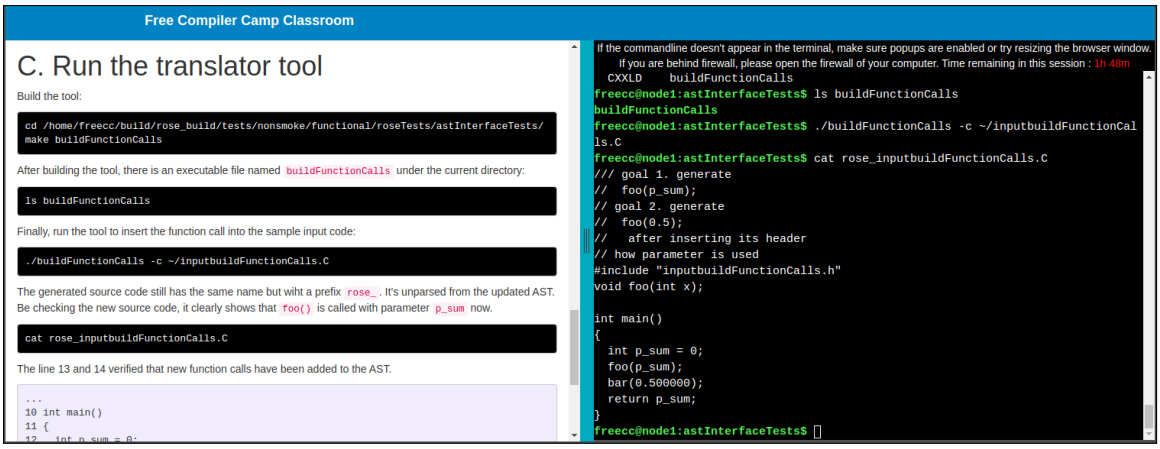


Figure 6.2: The tutorial for teaching AST modification

our sandbox and let all process run in that user account instead of root. This way we have more control over what access we want to provide the users.

6.3 Tutorial Design

We have created several initial tutorials to take advantage of FreeCompilerCamp. The goal is to have a good mix of text and commands for users to read and practice essential compiler skills.

6.3.1 Concepts

Tutorials of FreeCC are designed based on the principle of experimental learning or learning by doing. John Dewey introduced learning-by-doing, and it promotes the idea that students should learn by actively interacting with environments[107]. Kolb reviewed the major experimental learning models and created his comprehensive structural model[108]. He also explored the application of experimental learning in higher education. Students read static texts and apply theoretical knowledge to practical cases. They learn the skills by solving problems, working on small projects, etc.

Under the guidance of this theory, FreeCC hosts tutorials to let users start from any point they like with a ready environment, with the following major features:

- We make users practice as much as possible with detailed instructions by providing an easy-to-use sandbox for users to test given code or conduct their experiments.
- FreeCC covers different topics in compiler development, including parsing, AST generation, OpenMP programming, compiler extension, and so on.
- We split larger learning tasks into smaller ones to fit each tutorial into a 10-15 minutes session. The goal is to ensure that we can grab sufficient attention from visitors.

- The tutorial lists the steps and explains why each step should be conducted and how it works.
- FreeCC supports clickable code snippets, which can be tested in the sandbox immediately by clicking.
- Video instructions are not included currently because more students prefer static tutorials to video tutorials[109]. Using static tutorial is easier to seek and pick different sections of tutorials and learn at a comfortable pace for themselves.

6.3.2 Example Tutorials

FreeCompilerCamp.org provides a flexible learning experience based on the concepts mentioned above. In particular, we split the training content into several tutorials with incremental complexity so visitors can jump to the right levels they are comfortable with. We start with simple ones to let visitors play with the input and output of compilers and get familiar with compilers' internal representations for input programs. After that, we let them try out how to traverse the tree representations and finally how to change the tree for writing transformations.

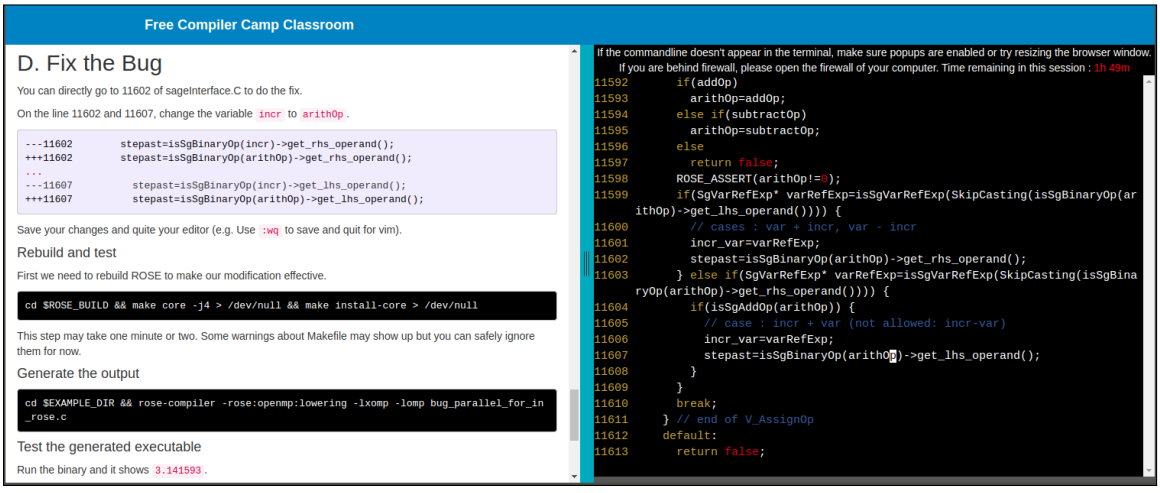


Figure 6.3: The tutorial for fixing an OpenMP translation bug in ROSE

6.3.2.1 Tutorial for Learning AST

Taking ROSE as an example, we designed the following tutorials:

- AST/IR Generation. An AST will be generated and represented visually in a graph for a given input source file. This tutorial shows how information is retrieved from source code and organized internally inside ROSE for future use.
- AST/IR Traversal. After AST generation, this tutorial shows how to traverse the tree to search for certain information of interest, such as loops or functions.
- AST/IR Modification. This tutorial demonstrates the method to add function call nodes into AST. Unparsing the AST will result in an output source file with the inserted function calls.

For example, the AST modification tutorial teaches users how to insert a functional call node into AST and check the updated AST by looking into the corresponding unparsed source code (Figure 6.2). Users can click the corresponding code snippets to download those files without leaving the page. All necessary source files can be downloaded in the sandbox on demand. The sample input has no function calls in the `main` function. The tutorial explains how a function call subtree is constructed in the compiler and shows all steps to create the subtree and attach it to the AST to complete the task. The input and the expected output are provided in the tutorial so that users can compare their results with the correct solution.

6.3.2.2 Tutorial of Fixing a Compiler Bug

Developers often learn many things by fixing actual bugs. Figure 6.3 is an example tutorial to fix a user-reported bug in ROSE. A PI calculation program in OpenMP compiled by ROSE generated some wrong values. Upon debugging, it was found that during ROSE's transformation of the loop body of `'omp parallel for'`, the loop stride was miscalculated due to incorrect operand nodes retrieved in the AST. The tu-

torial first highlights the bug and describes the steps to reproduce it. It then explains how compiler transformation and a runtime library function collaborate to schedule loop iterations among multiple threads. After that, it gives specific instructions on which source files should be modified to fix the bug. At last, with a few simple clicks, the modified ROSE is rebuilt to compile the test program, and the correct execution output is generated. Thus in a wholesome way, this tutorial gives an example of a real OpenMP implementation bug and explains how to reproduce, debug and resolve it.

6.3.2.3 Tutorial for Writing a Clang Plugin

We take Clang as another example to show our tutorials. This is a self-contained tutorial about writing a short plugin in Clang that modifies the source code as required.

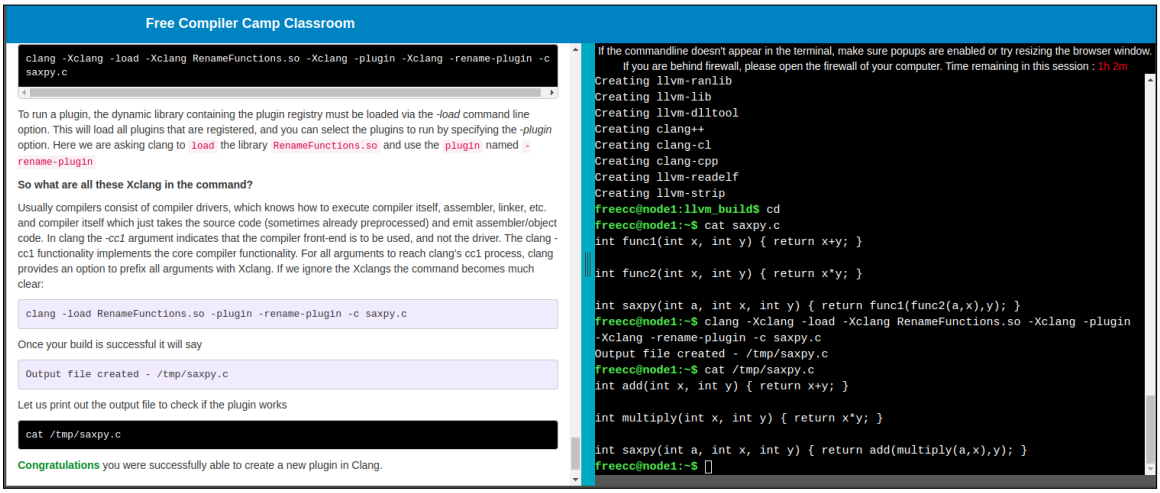


Figure 6.4: The tutorial for writing a Clang Plugin

Let's say we want to analyze a simple C file as shown in Listing 6.1. Suppose we want to do some simple fixes on this C file. We want to change the name of `func1` to `add` and `func2` to `multiply`. Then we would also like to change the function calls of `func1` and `func2` to `add` and `multiply`, respectively. This will result in a code as shown in Listing 6.2. We can write a plugin that will parse through the AST and

make the above changes to the file.

Listing 6.1: Example input code

```
1  int func1(int x, int y) { return x+y; }
2  int func2(int x, int y) { return x*y; }
3  int saxpy(int a, int x, int y) {
4      return func1(func2(a,x),y);
5  }
```

Listing 6.2: Expected output code

```
6  int add(int x, int y) { return x+y; }
7  int multiply(int x, int y) { return x*y; }
8  int saxpy(int a, int x, int y) {
9      return add(multiply(a,x),y);
10 }
```

This tutorial explains in detail the steps that need to be taken to write this plugin. It starts with giving an overview of what is a Clang plugin. Then it goes on to explain what this plugin intends to do. Then it explains how to set up the source code structure of the plugin and which files need to be written or modified to write this plugin. The tutorial also allows users to download a reference plugin or write it themselves. Ultimately, it helps the user build and test out the plugin. Figure 6.4 is a screenshot of this tutorial where the user tests the plugin.

6.3.3 Trial and Feedback

We invited six Ph.D. students majoring in Computer Science, with only basic compiler knowledge, to participate in a trial of FreeCC. No pre-training was provided before the trial to ensure the most accurate feedback. Each student chose a tutorial based on their interests and completed it independently without additional guidance. After the tutorial, they completed a survey form to share their experiences using

FreeCC. The feedback from the survey is summarized as follows:

- They feel comfortable with the length of each tutorial of 10-15 minutes.
- All steps of the tutorial are completed without any issues.
- Students prefer using clickable code snippets rather than manually typing them.
- Providing a choice from multiple code editors will be helpful.
- Additional video instructions are not needed.
- The sandbox and clickable code snippets attracted the most attention. They make FreeCC unique compared to conventional tutorials.
- Some students tried to conduct their experiments in PWC as we expected.
- The overall appearance of FreeCC could be improved.
- They want to retrieve files from the sandbox (ssh or git might help).
- Support for X11 forwarding might be needed to display graphics.
- The tutorials can use some links to external courses for fundamentals about OpenMP and compilers.
- GPU support is needed for extending tutorials running on GPU.

Based on the feedback, we conclude that the current design of the FreeCC tutorial is a very good starting point. All testers are satisfied with the features of FreeCC. The sandbox, PWC, is highly rated since students don't need to configure any complicated environment but a modern browser on any system. Criticism mostly came from the website appearance, customization, and cloud-machine resources for GPUs, which can be addressed in the future.

6.4 Summary

In this chapter, we have introduced an ongoing effort, FreeCompilerCamp.org, a free and open online learning platform to train researchers to develop OpenMP compilers quickly. FreeCompilerCamp.org is built on the Play-with-Docker platform to relieve learners' difficulty finding suitable machines and installing software. The tutorials of FreeCompilerCamp are entirely web-based with both text content and a live embedded sandbox terminal in which learners can immediately practice compiler development skills. Instructors or students can customize this platform quickly and deploy it on any local server, workstation, or even personal laptop.

In the future, we will include more tutorials on developing OpenMP compilers for HPC. We will also design online examinations to help learners evaluate the effectiveness of their learning process. We welcome anyone to try out our system, give us feedback, contribute new training courses, or enhance the training platform to make it a practical learning resource for the HPC community.

CHAPTER 7: CONCLUSIONS

In this dissertation, we present UPIR, a unified parallel intermediate representation designed to facilitate the representation of parallelism in parallel programming models, thereby enabling parallelism-aware compiler analysis, transformation, and optimization. Developed to support a wide range of parallel programming models, UPIR's prototype implementation within the ROSE compiler is compatible with C/C++/Fortran, OpenMP, OpenACC, and CUDA. This approach allows consistent compiler transformation applicable to multiple parallel programming models, such as OpenMP and OpenACC.

Moreover, we investigate the application of data shuffle in many-core GPUs, which allows data to be copied between threads without relying on the memory system. This technique can improve computing performance when significant data communication occurs between threads. In this research, we explore two methods of integrating shuffle within the OpenMP high-level programming model: 1) an efficient runtime implementation of the reduction clause; and 2) a proposed 'shuffle' extension for OpenMP, enabling users to control when and how data is moved between threads. We have achieved notable performance enhancements by implementing shuffle for reduction and 2D stencil kernels. Although programming with the shuffle primitive can be challenging, our language extension allows users to leverage it in high-level programming models with reduced complexity. Our findings demonstrate the potential for shuffle instructions to be utilized in compiler code generation and application optimization to boost performance.

We then introduce REX, an OpenMP source-to-source compiler that employs the unified parallel intermediate representation (UPIR) to support critical features of

OpenMP 5.1 and OpenACC 3.2. REX generates output targeting the LLVM OpenMP runtime and offers the option to produce MLIR instead of new source code. Additionally, we have extended the Rodinia parallel benchmark by implementing two new variants: OpenMP GPU and OpenACC, which facilitate evaluating GPU offloading performance across various parallel programming models and compilers.

REX operates on the same OpenMP runtime as LLVM, providing additional source-to-source transformation and OpenACC support. Impressively, REX incurs an average overhead of only 6%. Compared to GCC and NVC, which also support OpenMP and OpenACC, REX delivers consistent performance across both parallel models, thanks to UPIR. Conversely, semantically equivalent programs compiled by GCC and NVC may display considerable performance differences.

Lastly, we have initiated FreeCompilerCamp.org, a complementary and accessible online educational platform to accelerate researchers' training in OpenMP compiler development. FreeCompilerCamp.org is built on the Play-with-Docker platform to mitigate learners' difficulties when seeking appropriate machines and installing the required software. The platform's web-based lessons combine textual content with an interactive embedded sandbox terminal, allowing learners to apply their compiler development skills immediately. This versatile platform can be easily customized by instructors or students and deployed on any local server, workstation, or even an individual's laptop.

REFERENCES

[1] B. Sander, "AMD GCN assembly: cross-lane operations," 2016.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*, pp. 44–54, Ieee, 2009.

[3] J. Luitjens, "Faster parallel reductions on Kepler," *Parallel Forall. NVIDIA Corporation. Available at: https://devblogs. nvidia. com/parallelforall/faster-parallel-reductions-kepler*, 2014.

[4] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into llvm's intermediate representation," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 249–265, 2017.

[5] S. Salehian, J. Liu, and Y. Yan, "Comparison of threading programming models," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 766–774, IEEE, 2017.

[6] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer, "Inspire: The insieme parallel intermediate representation," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pp. 7–17, IEEE, 2013.

[7] "LLVM Compiler Infrastructure."

[8] X. Tian, H. Saito, E. Su, A. Gaba, M. Masten, E. Garcia, and A. Zaks, "Llvm framework and ir extensions for parallelization, simd vectorization and offloading," in *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pp. 21–31, IEEE, 2016.

[9] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, IEEE, 2021.

[10] "MLIR Dialects."

[11] J. Zhao and V. Sarkar, "Intermediate language extensions for parallelism," in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pp. 329–340, 2011.

[12] N. Benoit and S. Louise, "Extending gcc with a multi-grain parallelism adaptation framework for mpsocs," in *GCC for Research Opportunities Workshop*, 2010.

[13] N. Benoit and S. Louise, "Kimble: a hierarchical intermediate representation for multi-grain parallelism," in *Proceedings of the Workshop on Intermediate Representations*, pp. 21–28, 2011.

[14] J. Doerfert, J. M. M. Diaz, and H. Finkel, "The tregion interface and compiler optimizations for openmp target regions," in *International Workshop on OpenMP*, pp. 153–167, Springer, 2019.

[15] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "Hpvm: Heterogeneous parallel virtual machine," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 68–80, 2018.

[16] H. Sharif, P. Srivastava, M. Huzaifa, M. Kotsifakou, K. Joshi, Y. Sarita, N. Zhao, V. S. Adve, S. Misailovic, and S. V. Adve, "Approxhpvm: a portable compiler ir for accuracy-aware optimizations.," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 186–1, 2019.

[17] G. Stelle, W. S. Moses, S. L. Olivier, and P. McCormick, "Openmpir: Implementing openmp tasks with tapir," in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–12, 2017.

[18] T. B. Schardl and S. Samsi, "Tapirxla: Embedding fork-join parallelism into the xla compiler in tensorflow using tapir," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8, IEEE, 2019.

[19] Intel, "Intel C++ compiler 19.1 developer guide and reference," 2019.

[20] NVIDIA, "CUDA programming guide," 2020.

[21] M. Bernaschi, M. Carrozzo, A. Franceschini, and C. Janna, "A Dynamic Pattern Factored Sparse Approximate Inverse Preconditioner on Graphics Processing Units," *SIAM Journal on Scientific Computing*, vol. 41, pp. C139–C160, Jan. 2019.

[22] Y. Liu and B. Schmidt, "LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 82–89, July 2015.

[23] L.-W. Chang, I. El Hajj, C. Rodrigues, J. Gómez-Luna, and W.-m. Hwu, "Efficient kernel synthesis for performance portable programming," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[24] S. G. D. Gonzalo, S. Huang, J. Gomez-Luna, S. Hammond, O. Mutlu, and W.-m. Hwu, "Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs," in *2019*

IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 73–84, Feb. 2019.

[25] P. Chen, M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka, "A versatile software systolic execution model for gpu memory-bound kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–81, 2019.

[26] M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA code generation for affine programs," *Compiler Construction. Lecture Notes in Computer Science*, vol. 6011, p. 244.

[27] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 51–61, 2010.

[28] C. Nugteren and H. Corporaal, "Introducing'bones' a parallelizing source-to-source compiler based on algorithmic skeletons," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pp. 1–10, 2012.

[29] M. Amini, F. Coelho, F. Irigoin, and R. Keryell, "Static compilation analysis for host-accelerator communication optimization," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 237–251, Springer, 2011.

[30] G. Mendonça, B. Guimarães, P. Alves, M. Pereira, G. Araújo, and F. M. Q. Pereira, "Dawncc: automatic annotation for data parallelism and offloading," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[31] A. Mishra, M. Kong, and B. Chapman, "Kernel fusion/decomposition for automatic GPU-offloading," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 283–284, IEEE, 2019.

[32] H. Bae, D. Mustafa, J.-W. Lee, H. Lin, C. Dave, R. Eigenmann, and S. P. Midkiff, "The cetus source-to-source compiler infrastructure: overview and evaluation," *International Journal of Parallel Programming*, vol. 41, pp. 753–767, 2013.

[33] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 341–352, 2012.

[34] 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, IEEE, 2011.

[35] M. J. Harvey and G. De Fabritiis, "Swan: A tool for porting CUDA programs to OpenCL," *Computer Physics Communications*, vol. 182, no. 4, pp. 1093–1099, 2011.

[36] Y. Kim and H. Kim, "Translating CUDA to OpenCL for hardware generation using neural machine translation," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 285–286, IEEE, 2019.

[37] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 101–110, 2009.

[38] J. Kim, Y.-J. Lee, J. Park, and J. Lee, "Translating OpenMP device constructs to OpenCL using unnecessary data transfer elimination," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 597–608, IEEE, 2016.

[39] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[40] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[41] M. Pharr and W. R. Mark, "ispc: A spmd compiler for high-performance cpu programming," in *2012 Innovative Parallel Computing (InPar)*, pp. 1–13, IEEE, 2012.

[42] LLVM, "Projects built with llvm," Aug. 2019.

[43] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011, p. 1, 2011.

[44] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *Computing in Science & Engineering*, no. 1, pp. 46–55, 1998.

[45] "Gcc support for the openmp language," 2019.

[46] "Intel c++ compiler code samples," Mar. 2019.

[47] C. Cray, "C++ reference manual, s-2179 (8.7). cray research," 2019.

[48] "Openmp support in ibm xl compilers," 2019.

[49] C. Liao, D. J. Quinlan, T. Panas, and B. R. De Supinski, "A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries," in *International Workshop on OpenMP*, pp. 15–28, Springer, 2010.

[50] B. R. de Supinski, T. R. W. Scogland, A. Duran, M. Klemm, S. M. Bellido, S. L. Olivier, C. Terboven, and T. G. Mattson, "The ongoing evolution of openmp," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2004–2019, 2018.

[51] I. Leontiadis and G. Tzoumas, "OpenMP C Parser," Dec 2001.

[52] "Rose documentation," 2019.

[53] "Clang documentation," 2019.

[54] "Openmp tutorials & articles," 2019.

[55] T. R. Liyanagunawardena, A. A. Adams, and S. A. Williams, "Moocs: A systematic study of the published literature 2008-2012," *The International Review of Research in Open and Distributed Learning*, vol. 14, no. 3, pp. 202–227, 2013.

[56] C. Thompson, "How khan academy is changing the rules of education," *Wired Magazine*, vol. 126, pp. 1–5, 2011.

[57] "Coursera," 2019.

[58] "edx," 2019.

[59] "Learn to code with free online courses, programming projects, and interview preparation for developer jobs," 2019.

[60] L. B. Ngo and J. Denton, "Using cloudlab as a scalable platform for teaching cluster computing," *Journal of Computational Science Education*, vol. 10, pp. 100–106, Jan. 2019.

[61] P. Bisbal, "Training computational scientists to build and package open-source software," *Journal of Computational Science Education*, vol. 10, pp. 74–80, Jan. 2019.

[62] J. Shin, J. C. Jang, H. Chae, G. Rvu, J. Yu, and J. R. Lee, "A web-based mooc authoring and learning system for computational science education," in *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, pp. 1028–1032, IEEE, 2018.

[63] D. Majeti and V. Sarkar, "Heterogeneous habanero-c (h2c): a portable programming model for heterogeneous processors," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 708–717, IEEE, 2015.

[64] E. Tiotto, B. Mahjour, W. Tsang, X. Xue, T. Islam, and W. Chen, "Openmp 4.5 compiler optimization for gpu offloading," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 14–1, 2019.

[65] I. K. Prabhu and V. K. Nandivada, "Chunking loops with non-uniform workloads," in *Proceedings of the 34th ACM International Conference on Supercomputing*, pp. 1–12, 2020.

[66] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, "Array-data flow analysis and its use in array privatization," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 2–15, 1993.

[67] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

[68] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[69] G. Georgakoudis, J. Doerfert, I. Laguna, and T. R. Scogland, "Faros: A framework to analyze openmp compilation through benchmarking and compiler optimization analysis," in *International Workshop on OpenMP*, pp. 3–17, Springer, 2020.

[70] J. Mellor-Crummey, "Compile-time support for efficient data race detection in shared-memory parallel programs," *ACM SIGPLAN Notices*, vol. 28, no. 12, pp. 129–139, 1993.

[71] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar, "An extended polyhedral model for spmd programs and its use in static data race detection," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 106–120, Springer, 2016.

[72] C.-W. Tseng, "Compiler optimizations for eliminating barrier synchronization," *ACM SIGPLAN Notices*, vol. 30, no. 8, pp. 144–155, 1995.

[73] M. F. P. O'Boyle, L. Kervella, and F. Bodin, "Synchronization minimization in a spmd execution model," *Journal of parallel and distributed computing*, vol. 29, no. 2, pp. 196–210, 1995.

[74] Y. Lin, C. Terboven, D. an Mey, and N. Copty, "Automatic scoping of variables in parallel regions of an openmp program," in *International Workshop on OpenMP Applications and Tools*, pp. 83–97, Springer, 2004.

[75] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in gpu programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pp. 1–8, 2011.

[76] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 1–10, 2008.

[77] D.-K. Chen and P.-C. Yew, "Redundant synchronization elimination for doacross loops," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 5, pp. 459–470, 1999.

[78] S. P. Midkiff and D. A. Padua, "Compiler algorithms for synchronization," *IEEE Transactions on computers*, vol. 100, no. 12, pp. 1485–1495, 1987.

[79] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pp. 212–223, 1998.

[80] "PaRSEC." `https://icl.utk.edu/parsec/index.html`.

[81] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pp. 1151–1158, 2011.

[82] A. Danalis, H. Jagode, G. Bosilca, and J. Dongarra, "Parsec in practice: Optimizing a legacy chemistry application through distributed task-based execution," in *2015 IEEE International Conference on Cluster Computing*, pp. 304–313, IEEE, 2015.

[83] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson, "Using memory mapping to support cactus stacks in work-stealing runtime systems," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 411–420, IEEE, 2010.

[84] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–12, IEEE, 2009.

[85] J. K. Lee, J. Palsberg, R. Majumdar, and H. Hong, "Efficient may happen in parallel analysis for async-finish parallelism," in *International Static Analysis Symposium*, pp. 5–23, Springer, 2012.

[86] C. Chen, W. Huo, L. Li, X. Feng, and K. Xing, "Can we make it faster? efficient may-happen-in-parallel analysis revisited," in *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 59–64, IEEE, 2012.

[87] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar, "May-happen-in-parallel analysis of x10 programs," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 183–193, 2007.

[88] P. Grun, N. Dutt, and A. Nicolau, "Aggressive memory-aware compilation," in *International Workshop on Intelligent Memory Systems*, pp. 147–151, Springer, 2000.

[89] V. Bala, J. Ferrante, and L. Carter, "Explicit data placement (xdp) a methodology for explicit compile-time representation and optimization of data movement," *ACM SIGPLAN Notices*, vol. 28, no. 7, pp. 139–148, 1993.

[90] C. Li, Y. Yang, Z. Lin, and H. Zhou, "Automatic data placement into gpu on-chip memory resources," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 23–33, IEEE, 2015.

[91] L. Carter, J. Ferrante, and V. Bala, "Xdp: A compiler intermediate language extension for the representation and optimization of data movement," *International journal of parallel programming*, vol. 22, no. 5, pp. 485–518, 1994.

[92] A. Danalis, L. Pollock, M. Swany, and J. Cavazos, "Mpi-aware compiler optimizations for improving communication-computation overlap," in *Proceedings of the 23rd international conference on Supercomputing*, pp. 316–325, 2009.

[93] A. Friedley and A. Lumsdaine, "Communication optimization beyond mpi," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pp. 2018–2021, IEEE, 2011.

[94] D. Novillo, R. C. Unrau, and J. Schaeffer, "Optimizing mutual exclusion synchronization in explicitly parallel programs," in *International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pp. 128–142, Springer, 2000.

[95] S. Prakash, M. Dhagat, and R. Bagrodia, "Synchronization issues in data-parallel languages," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 76–95, Springer, 1993.

[96] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar, "Chunking parallel loops in the presence of synchronization," in *Proceedings of the 23rd international conference on Supercomputing*, pp. 181–192, 2009.

[97] A. Wang, Y. Shi, X. Yi, Y. Yan, C. Liao, and B. R. de Supinski, "Ompparser: A standalone and unified openmp parser," in *International Workshop on OpenMP*, pp. 140–152, Springer, 2019.

[98] J. Löff, D. Griebler, G. Mencagli, G. Araujo, M. Torquati, M. Danelutto, and L. G. Fernandes, "The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures," *Future Generation Computer Systems*, vol. 125, pp. 743–757, 2021.

[99] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O. Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating gpu threads for openmp 4.0 in llvm," in *2014 LLVM Compiler Infrastructure in HPC*, pp. 12–21, IEEE, 2014.

[100] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, IEEE Press, 2008.

[101] M. Nils and J. Leibiusky, "Play with docker," June 2019.

[102] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[103] T. Goethals, D. Kerkhove, L. Van Hoye, M. Sebrechts, F. De Turck, and B. Volckaert, "Fuse : a microservice approach to cross-domain federation using docker containers," in *Proceedings of the 9th International Conference on Cloud Computing and Services Science*, pp. 90–99, Scitepress, 2019.

[104] C. Yong, G.-W. Lee, and H. Eui-Nam, "Proposal of container-based hpc structures and performance analysis," *Journal of Information Processing Systems*, vol. 14, no. 6, pp. 1398–1404, 2018.

[105] J. Schwenk, M. Niemietz, and C. Mainka, "Same-origin policy: Evaluation in modern browsers," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 713–727, 2017.

[106] A. Van Kesteren and et al., "Cross-origin resource sharing," *W3C REC-cors-20140116, latest version available at< https://www.w3.org/TR/cors/*, 2014.

[107] J. Dewey, *Experience and Education*. Kappa Delta Pi, 1938.

[108] D. A. Kolb, *Experiential Learning: Experience as the source of learning and development*. Pearson FT Press, 2014.

[109] L. S. Mestre, "Student preference for tutorial design: a usability study," *Reference Services Review*, vol. 40, no. 2, pp. 258–276, 2012.