Transpilation Utilizing Language-agnostic IR and Interactivity for Parallelization 🌷

Zujun Tan

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: David I. August

September 2024

Abstract

Migrating codes between architectures is difficult because different execution models require different types of parallelism for optimal performance. Previous approaches, like libraries or source-level tools, generate correct and natural-looking syntax for the new parallel model with limited optimization and largely leave performance engineering to the programmer. Recent approaches, such as transpilation at the compiler intermediate representation (IR) level, can automate performance engineering, but profitability can be limited by not having facts known only to the programmer. Decompiling the optimized program could leverage the strength of existing compilers to provide programmers with a natural compiler-parallelized starting point for further parallelization or refinement. Despite this potential, existing decompilers fail to do this because they do not generate portable parallel source code compatible with arbitrary compilers of the source language.

This thesis provides a method for migrating code such that the compiler and programmer work together to generate code with optimal performance. To achieve this, it introduces Tulip 🌷, a source-to-source code generation framework that operates via IR-level transpilation. Transpilation at the IR level enables Tulip to generalize the transformations applied to retarget parallelism. Furthermore, Tulip integrates the state-of-the-art automatic parallelization framework to explore additional parallelism expressible only in the target parallel programming model. It then generates natural source code through a novel decompiler, SPLENDID, in a high-level parallel programming language (OpenMP), which can be interactively optimized and tuned with programmer intervention. For 19 Polybench benchmarks, Tulip-generated OpenMP offloading programs perform 14% faster than the original CUDA sources on NVIDIA GPUs. Moreover, transpilation to the CPU leads to a 2.9x speedup over the best state-of-the-art transpiler. Tulip-generated portable parallel code is also more natural than what existing decompilers produce, resulting in a 39x higher average BLEU

score.

This thesis includes contributions from Yebin Chon, Ziyang Xu, Sophia Zhang, and David I. August from Princeton Liberty Research Group, Brian Homerding, Yian Su, and Simone Campanoni from Northwestern Arcana Lab, Michael Kruse (AMD), Johannes Doerfert (LLNL), William S. Moses (UIUC), and Ivan R. Ivanov (Tokyo Tech).

Acknowledgements

First, I would like to thank my advisor Prof. David I. August for his support and guidance over the years. I am grateful for his faith in me more than what I have in myself and his constant encouragement during challenging times. Especially at the end of the second year of graduate school, high pressure and the feeling of incompetance made me want to quit the program. It was David's encouragement and help with exploring alternatives in research directions that helped me go through the difficult time. I am grateful for his high standard in paper submission, which trained me to strive for excellence. I am also grateful for the grace simultaneously he displays, when I cannot meet the standard, he will also accept the imperfections and being a strong advocate for my papers. I also appreciate his advice on writing and presentation by showing me how to frame my research clearly and convincingly. Finally, the culture of collaboration and solidarity he has cultivated in the Liberty Research Group and with Northwestern Arcana Lab and Argonne National Lab made my research much more enjoyable and rewarding. I thank the rest of my dissertation committee: Prof. Aarti Gupta, Prof. Zachary Kincaid, and Prof. Amit Levy. I want to additionally thank Michael and Johannes for taking the time to serve as readers on my committee. Their feedback helped improve the quality of this dissertation as well as my research overall.

I thank each member of the Liberty Research Group for all their support and friendship throughout the years. I especially thank Sotiris Apostolaskis, my mentor, friend, and model. He has never been absent from helping and mentoring me, even after his graduation. I thank Greg and Ziyang who went beyond their way to help me in difficult times. I thank Barghav, Ishita, Yebin, Yucan, and Sophia, who have been amazing lab mates and friends.

I would like to thank all my external collaborators, most of whom had no duty but graciously offered help beyond what I could have asked for. I thank all members of

Arcana Lab from Northwestern University, especially Prof. Simone Campanoni, Brian Homerding, Yian Su, and Federico Sossai. I thank all colleagues I met from Argonne National Lab, Johannes Doerfert, William Moses, and Ivan Ivanov. Lastly, I would like to thank my Microsoft internship mentor and supervisor, Dimitrios Prountzos and Aaron Smith. Though the internship did not directly contribute to this thesis, the experience working at Microsoft with them made a hugely positive turn in my PhD journey, as I grew a lot personally and professionally under their supervision. Without your help, this dissertation would not have been possible.

I want to thank all Hope Presbyterian Church members, especially Pastor David Rowe, Pastor Stephen O'Neil, Chris Mills, Felix Yiu, Jess Sauer, Yui Morishima, Emily Lobb, Lucas Ophoff, the Keddies, the Seungs, the Kims, Handa Chun, Chip Lem, Nina Rathbun, Jen and Quinn Peacock, Brian Kook, and Stella Choi. Thank you for your faith and encouragement, for our many theological discussions which greatly deepened my Reformed faith. I want to thank Elizabeth Christianos, Minako Wilkinson, Carrie Louer, Louise Jennewine, Madhu Gammon, Jessica Zakhari, Hitomi Kim, Jiwon Seung, Beth Hastings, and all the church ladies whose unwavering love and support have sustained me throughout the PhD journey. I also want to thank all my friends from the larger Princeton Christian circle, especially Laura Lam, Bob Louer, Allison Huang, Joseph So, Pastor Ken Smith, Pastor Tracy Troxel, Pastor Andrew Zakhari, Pastor Lane Tipton, Joao Castanha, Fady Girgis, Jessica Jin, Yutaka Morishima, Naomi Vaida, the Ramslands, Evelyn Dziedzic, Jasmine Hao, Abigail Sargent, James Loy, Carol and Tom Smith, David and Debbie Monn, Rebecca Petruccelli, Sabrina Sequeira, Chukuemeka Chukuemeka, and Jihye Jeon.

I thank all my peers and cohorts, Themis Melissaris, August Ning, Grigory Chirkov, Marcelo OV, Julian Knodt, Jianan Lu, Nanqingqing Li, and Yushan Su, for many discussions, drinks, and laughter. I want to thank the Demschacks, Maggie Xu, Amber Hokama, Jacob Knight, John Chen, Ben Whelan, Brian Madina, Shirley Liu,

Jim Wall, Chenyang Ye, and Kexin Sun for our friendship for close to or more than a decade. I want to thank my dear fluffy son, bunny Kuromame, who has served as a squeeze ball for stress relief and a great listener to my many troubles. I also would like to thank the theologians and writers who strengthened me through their writings, especially D. Martyn Lloyd-Jones, John Bunyan, John Owen, Ed Welch, and Elisabeth Elliot. I would like to give special thanks to the great French Theologian John Calvin, whose clear and robust theology has clarified many of my questions and wonders and from whom the title of the thesis is inspired (T.U.L.I.P.).

Lastly, my ability to complete this thesis is not my own but wholly the work of the Spirit of Christ. By His grace and for His glory, despite many weaknesses and imperfections, I confidently present this thesis to Him as a thanksgiving to the triune God of the Christian faith for His omnipotent presence and unchanging love.
Soli Deo Gloria[1].

[1]To God alone be the glory - the summarizing doctrine of Reformation.

To the immortal, invisible, only wise God.

Contents

Chapter 1

Introduction

An increase in hardware specialization has led to the birth of numerous distinct parallel execution models. These parallelism models are increasingly specialized to the unique structure of parallelism available in their corresponding hardware target, lest they prevent programmers from leveraging the peak performance of their hardware. This increasing specialization of the parallel programming model (PPM) comes at the cost of generality. As a result, high-performance programs that would merit the use of new hardware cannot effectively run on other architectures.

Numerous strategies have been proposed to address the complexities of this issue. One approach involves writing a kernel for each target platform using highly explicit parallel languages such as Fortran, CUDA, HIP, and SYCL [57]. Although these languages are powerful, they place a significant burden on programmers to effectively harness parallelism. Alternatively, performance portability libraries like RAJA [49] and Kokkos [122] offer backends for various parallel programming models, but they may sacrifice peak performance due to the limited control they provide programmers for performance engineering on each platform. Another option is for programmers to use hardware-agnostic domain-specific languages (DSLs)[20, 103], or sequential languages enhanced with parallel directives that target multiple platforms, such as

OpenMP[93], OpenACC [94], and OpenCL [58]. Given the diversity of parallel programming models (PPMs), substantial legacy code bases always exist, each tailored for specific hardware targets. Porting code from one platform to another often involves manual rewriting. However, this process is labor-intensive, error-prone, and heavily dependent on the programmer's expertise in achieving optimal performance.

Recent tools have been developed to automatically perform performance-aware retargeting by optimizing parallel programs before generating executables in a new model [112, 81]. Compiler-based tools like Polygeist [80, 81, 53] enhance the speed of transpiled code by optimizing both serial and parallel constructs within the compiler itself. Initially, the source code is compiled to an intermediate representation (IR), which is then aggressively optimized and finally compiled to produce executables for various targets. If the selected compiler IR is both language- and hardware-agnostic, this approach not only enhances reusability across different programming models and hardware platforms but also reduces the time required to reimplement these optimizations. Notably, recent breakthroughs in parallelizing compilers [76, 6, 7] can potentially extract additional parallelism beyond what is originally expressed in the parallel program.

However, a significant drawback of this approach is that the generated programs often remain incomprehensible to end-users, particularly when tools directly generate executables or low-level assembly code. Thus, this method necessitates total reliance on the compiler for the entire parallel transpilation process. However, programmers can always play a role in enhancing parallelism by easing constraints on the output, allowing for a broader range of potential optimizations, especially on non-deterministic programs, which can then be optimized more rigorously [30]. This is because, unlike a programmer, the compiler cannot expand the set of valid outputs of a program, even if such additional outputs would produce much better performance and be accepted as valid by the programmer [99, 11, 104]. Additionally, while the programmer

may find the out-of-order printing of diagnostic messages acceptable for some level of performance, the compiler does not know this and cannot unilaterally make this change [99]. Moreover, the programmer may find lower precision for floating-point operations acceptable. The compiler, however, cannot relax the precision of output without the programmer playing a role. Besides this fundamental reason, current parallelizing compilers face many practical challenges, such as limited profitability in the presence of dependences (e.g., the DSWP thread partitioning problem). Thus, additional performance in code migration can always be gained by the programmer and compiler working together, despite recent great advancements in transpilation and automatic parallelization.

As large and legacy applications cannot afford the performance cost associated with manual rewrites into portability libraries or DSLs or the economical cost of developing the most powerful transpiling compilers with all optimizations necessary for all targets, our focus shifts to approaches that aim to deliver "abstraction without regret" — namely, direct and optimized transpilation from one parallel programming model (PPM) to another. The earliest source-to-source transpilers operated at the source code level, directly generating parallel programs in a new model that preserved the semantics and parallelism of the original model [128, 4, 12, 8, 88, 108, 3, 71, 56, 87, 9]). Tools like ROSE [66] maintain parallelism and structure at the source level while allowing for limited optimization at the abstract syntax tree (AST) level. Source-level transpilers that generate high-level source code unlock the use of the full software and tooling ecosystem around the target model, potentially addressing the software fragmentation issue, as many libraries and applications target various parallel programming frameworks. Moreover, the source code generated through this method is natural, enabling programmers to further parallelize the code with the parallelism expressible in the target PPM.

However, this flexibility comes at a cost. Directly translating source code often

results in programs that perform suboptimally on the target hardware. Even devices manufactured by the same company can exhibit significant differences in the number of threads, memory bandwidth, dependency structures, and other characteristics, as commonly seen in new accelerators like Tensor Cores [74]. Consequently, programs generated by such tools cannot be efficiently or directly utilized; they necessitate extensive re-tuning by the programmer. For instance, a programmer with knowledge of the target model could introduce additional parallelism that had been previously reduced in the source model to enhance performance. Although it is theoretically possible for application developers to achieve optimal performance with sufficient rewriting, in practice, this is rarely achieved. For example, [81] observed that many expert-written CUDA benchmarks contained unnecessary parallel synchronize commands and shared memory loads, likely due to the complexity of the programming model. Ideally, transpilation should benefit from both compiler and programmer intervention. This thesis demonstrates that transpilation is best done with a collaboration between a compiler and a programmer[1].

1.1 Dissertation Contributions

This dissertation takes a different approach to code migration, from purely manual and purely automatic to a combined approach that does not compromise either. Taking inspiration from the best practices of prior approaches and practically enabling robust translations between PPMs, we Tulip: a transpilation framework that generates robust and natural code targeting mainstream GPU and CPU heterogeneous systems by transpiling CUDA to primarily CPU-focused PPMs, OpenMP and OpenACC. Tulip proposes a new vision for programmer-compiler interaction, realized via an extensible approach to transpilation that allows easy additions of source and target

[1]Major edits and intellectual contribution credits go to William S. Moses and other coauthors of Tulip and SPLENDID.

PPMs and a parallelism-aware decompiler that generates natural code.

1.1.1 New Vision for Programmer and Compiler Interactive Parallelization

The demand for performance and efficiency drives research to find better program parallelization methods. Most parallelizations are, to some degree, a collaboration between the programmer and a compiler. First, the programmer can parallelize the program using a parallel programming language [89, 119, 15], parallel extensions to sequential languages [86, 93, 42], or by expressing code properties that enable inherent parallelism (i.e., implicit parallel programming [51, 127, 47, 14, 11, 99]). Then, the compiler maps this programmer-expressed parallelism to utilize parallel hardware resources. However, the degree of collaboration is limited in this way, either because the compiler performs only a translation of programmer-expressed parallelism or because the compiler disregards the work of the programmer and parallelizes the code itself (e.g., Polly [45]). In either case, only the programmer or the compiler is ultimately responsible for the parallelization choices.

This thesis introduces a collaborative approach to parallelization, leveraging a combination of compilation techniques—specifically, translating parallelism from the source code to an Intermediate Representation (IR), followed by compiler parallelization, and finally, decompilation. Initially, we translate source-level parallelism into an IR and then conduct aggressive transpilation and parallelization. Given that our selected IR (LLVM-IR) is language-agnostic, it seamlessly integrates multiple sources of parallelism, including those derived from the original parallel source code, robust transpilation processes, and automatic parallelization strategies. Decompiling parallelism from the IR back to the source code provides programmers with an optimized synthesis of parallelization By understanding parallelism at the source level within the target parallel programming model (PPM), programmers can bypass the need to

grasp the intricacies of the original PPM or compiler yet still operate within a familiar framework provided by the target PPM. Moreover, as each programming model may express different levels of parallelism, programmers always have the opportunity to enhance or refine the expression of parallelism using the target PPM. This workflow facilitates a collaboration where programmers of both the source and target PPMs, along with parallelizing compilers, can effectively express parallelism without requiring extensive knowledge in all areas.

1.1.2 Compiler Enhanced Source-to-Source Transpilation Framework

Taking inspiration from the best practices of prior approaches and practically enabling robust translations between PPMs, this thesis proposes Tulip[2]. Tulip does so by performing transpilation at the IR level to fully leverage the state-of-the-art parallelizing compiler and generate natural source code through decompilation to enable source-level toolings and programmer knowledge. Transpilation occurs after standard LLVM frontends. Tulip transforms parallelism expressed in runtime calls to the special hardware environment into metadata, consequently retargeting the IR to multicore systems while separately preserving parallelism. The transpiled IR can then interact with any IR-level automatic parallelization framework. To preserve code naturalness and increase decompiler reusability, instead of directly applying a parallelization plan to the IR, Tulip acquires and interprets only a parallelization plan from the parallelizing compiler. Tulip is extensible because each stage of the transpilation pipeline includes extensively reusable code, from leveraging standard LLVM frontend and state-of-the-art automatic parallelization frameworks to a C decompiler regardless of the parallel extension.

[2]The title is also inspired by the Canons of Dort.

1.1.3 Parallelism-aware Natural Decompiler

Decompilers [1, 68, 46, 26, 40, 23, 107, 29, 130, 129] have great potential to enable collaboration in which better performance can be obtained with less manual effort. However, when it comes to parallel programs, state-of-the-art decompilers cannot produce portable code. Translating parallel IR to portable parallel source code is not a trivial task. First, most parallel programming models impose strict requirements for loop structures. For example, the OpenMP [93] *omp for* construct requires syntactically canonical *for* loops with no additional code between the pragma and the loop. However, most decompilers end up translating low-level parallelized loops into *do-while* loops. This is because parallelization often relies on loop rotation for canonicalization, which converts all loops into *do-while* form. Furthermore, parallelism in the IR is often expressed using parallel runtime library calls. For reverse engineering purposes, code decompiled by previous decompilers exposes these library calls, making the decompiled code not recompilable with compilers using another runtime library.

Since code produced by state-of-the-art decompilers is not portable, it is also not natural. Natural code is informative about what and how a compiler parallelizes, enabling the programmer to improve program performance in any desired workflow. A *do-while* loop compared with a *for* loop is less natural without features like induction variables. Low-level runtime-specific details of parallelization also obfuscate previously decompiled code. While making the decompiled code portable helps with naturalness, code decompiled in previous approaches assigns variables with names corresponding to physical registers. The lack of informative variable names intrudes significant overhead in understanding code semantics.

To overcome the obstacles mentioned above and practically enable collaborative parallelization, this work proposes SPLENDID, the first LLVM-to-C/OpenMP decompiler that provides portable natural translation from parallel LLVM-IR to

OpenMP-parallel source code. SPLENDID explicitly represents parallelism through the widely used parallel programming model, OpenMP [93]. Since using OpenMP directives eliminates compiler-specific implementations of parallel constructs and requires *for*-loops, SPLENDID-produced parallel code is portable and more natural. Moreover, code generated by SPLENDID preserves variable names and is thus closer to manually written code. With variable names that are representative of semantics, SPLENDID significantly reduces the manual effort of interpreting code semantics.

SPLENDID is designed with the careful consideration of what to de-transform so that key optimizations, such as parallelizations and loop optimizations, are made evident to the programmer. While the goal of this paper is to enable better collaborative parallelization, Readers may find the results useful for other tasks that may benefit from more natural reverse engineering, such as debugging. For example, SPLENDID may be used in debugging and performance tuning of computational kernels automatically parallelized using Polly [45], an LLVM-based parallelizing compiler.

1.2 Summary

In summary, the primary contributions of this dissertation are:

- Presenting a CUDA-to-OpenMP **transpilation framework** that targets multiple mainstream CPU and GPU (i.e., through OpenMP Offloading) systems, Tulip. Tulip-produced code is robust, with parallelism enhanced by the state-of-the-art parallelization framework, NOELLE [76], programmer interactivity, and toolings originally unavailable to the source PPM.

- Presenting the first decompiler targeting OpenMP-parallel IR, SPLENDID [117]. SPLENDID-produced code makes the output of a parallelizing compiler **portable**, recompilable with any host compiler, and **natural** for easy programmer involvement.

- Bringing together the best of the two orthogonal prior approaches and **enabling automatic parallelization, programmer interaction, and software toolings working together**. Unlike prior source-to-source or software rewriting, Tulip's transpilation occurs at the language-agnostic IR level and involves aggressive compiler optimizations. Unlike direct transpilation, instead of directly generating executables after transpilation, Tulip generates natural source code that can further interact with a programmer or be kept as the new golden source for the target machine.

- Realizing a smarter **trade-off** between how close the decompiled code is to the original source code and how instructive it is to compiler parallelization.

- A novel pass that **restores source variable names** by eliminating virtual register to variable naming conflicts and by inferring variable names from another function through inlining.

- Across 19 Polybench benchmarks, **outperforming native CUDA compilation** by 14%, native AMD compilation by 12%, 1.1x-2.93x over Polygeist, the best source-to-many-machine approach, and 12% over Hipify, the best prior source-to-source approach. When Tulip transpiled OpenMP code are run on an AMD GPU, they exceed CUDA native code on NVidia GPUs by a geomean speedup of 10%.

- When SPLENDID-decompiled code is recompiled using GCC, an average speedup of 11x of 16 PolyBench benchmarks is made available universally outside LLVM. The same benchmarks demonstrate an average of 39x improvement on the BLEU score (i.e., a widely-used naturalness metric [95]) over the best prior work. With an average of 3 lines of manual change on top of SPLENDID-generated code, the speedup is doubled relative to both manual and compiler parallelization alone on 7 PolyBench benchmarks, programs simple enough that

either the compiler or the programmer should have easily been able to deliver maximal performance but did not.

The proposed decompilation framework, SPLENDID, is published in [117]. Tulip, the proposed transpilation pipeline, is in preparation for publication.

Chapter 2

Motivation

Source-to-source and direct transpilation approaches hold significant potential for robust code migration across non-native systems with diverse execution models. However, previous methods consistently fail to generate optimally performing code. Furthermore, while these methods may achieve high performance on some platforms, their robustness does not extend to future PPMs and execution models. The subsequent sections will explore these shortcomings in greater detail.

2.1 Partial Performance Enablers

Ultimately, increasing program portability aims to deliver the best performance on a target machine from otherwise incompatible source PPM. We identified three major performance enablers in transpilation: source-level tools, language-agnostic optimizations, and programmer interactivity. As shown in Table 2.1, prior approaches only involve a subset of performance enablers in transpilation, thus not fully exploiting the potential for performance improvement.

		Performance Enablers		
		Source-level Tools	Language-agnostic Aggressive Optimizations	Programmer Interactivity
Direct Transpilation	GPUOcelot [37]	✗	✗	✗
	MCUDA [112]	✗	✗	✗
	Polygeist [81, 80, 53]	✗	✓	✗
Near-source Rewrite	AutoPar-Clava [8]	✓	✓	✓
	Bones [88]	✓	✗	✓
	DPC++ [128]	✓	✗	✓
	ROSE [102]	✓	✓	✓
	hipfy [4]	✗	✗	✓
Source-to-Source Transpilation	Tulip (This work)	✓	✓	✓

Table 2.1: Comparison with prior works categorized based on three approaches. Tulip is a source-to-source transpiler, the only approach that utilize programmer, compiler, and cross platform tools to deliver the best performance.

2.1.1 Source-level Tools

Many source-level tools, such as compilers, only take a source program as input, and frequently include optimizations tailored to benefit only a subset of programs. Therefore, when code is compiled directly for a specific target machine (source-to-many machine), compilers designed for the same target do not impact the performance of the resulting executable. For instance, if Polygeist is used to transpile code from GPU to multicore architecture, other compilers that target multicore systems, like GCC, cannot enhance the optimization of the transpiled code, as shown in Figure 2.1. In the context of near-source rewrite tools, if the target PPM shares the same execution model as the source PPM, tools designed for PPMs of a different execution model will not be applicable, as in the case of Hipify.

2.1.2 Language-agnostic Optimizations

While near-source rewrites typically focus on language-specific optimizations at the AST level, language-agnostic IR-level optimizations are highly reusable. This reusability allows applications to benefit from more optimizations, enhancing performance as the diversity of source and target PPMs grows. Occasionally, automatic parallelization, like that used in ROSE, can be done at the AST-level and potentially enable scalable speedups. However, language-specific constraints impede accurate memory analysis which inhibits meaningful automatic parallelization. Thus, produced parallelization plans are conservative and lack performance gains. Conversely, memory representation at the IR level offers detailed insights essential for optimizing parallel execution. Thus, prior near-source rewrite tools fail to incorporate language-agnostic optimizations to enable greater performance.

2.1.3 Programmer Interactivity

When code is transpiled across different execution models, the target PPM may be capable of expressing parallelism originally unavailable in the source PPM. Portions of what otherwise be manual parallelization in target PPM can be replaced by compiler parallelization. The decompiled parallel code can then be maintained in place of the original code in source PPM. After seeing what the compiler and the original source parallelize in the decompiled source code, the programmer can focus primarily on the loops that have not been parallelized. Moreover, after seeing the already optimized transpiled loop parallelization, the programmer can utilize additional expressible parallelism inherent in the target PPM to further parallelize transpiled code to be more suitable for running on target hardware. This practical use of knowledge of the target PPM and the source code allows for more effective and efficient parallelization. Instead, the prior direct transpilation approach directly generates executables onto different platforms, missing the opportunity to engage programmers to further increase parallelization.

To practically enable programmer to involve in interactive parallelization with the compiler, generating natural source code is a key step. However, to gain the benefit of aggressive compiler optimizations, generating source code is not as trivial as what is done in near-source rewrite, but instead, it requires decompilation. Potentially, we could use any out-of-box decompilers available in industry or academia [1, 107, 28, 114]. Unfortunately, code produced by previous decompilers is neither portable beyond the parallelizing compiler nor sufficiently natural for understanding parallelism, as shown in Table 2.2. This is because the primary goals of previous decompilers, including reverse engineering and analysis, do not rely on decompiled code being syntactically correct, recompilable, or natural. We have identified three core areas that prevent decompilation from enabling collaborative parallelization: lack of explicit parallelism, unnatural control flow translation, and use of artificial variable names.

Table 2.2: Comparison with prior decompiler frameworks. As the first decompiler for collaborative parallelization, SPLENDID emphasizes portability and naturalness for translating parallel code.

Decompiler	**Decompilation Level**	**Primary Goal**	**Explicit Parallelism Translation**				**Control Flow Translation**		**Variable Translation**	
			Parallel Runtime Library Call Elimination	Parallel Loop Restoration	Parallel Code Inlining	Pragma Generation	For-Loop Construction	Loop Rotation De-transformation	SSA De-transformation	Source Variable Renaming
Ghidra [1]	binary	Reverse Engineering	✗	✗	✗	✗	✓	✓	n/a	✗
Gussoni et al. [46]	binary	Security	✗	✗	✗	✗	✗	✗	n/a	✗
Chen et al. [26]	binary	Software Maintainance	✗	✗	✗	✗	✗	✗	n/a	✗
SmartDec [40]	binary	Reverse Engineering	✗	✗	✗	✗	✗	✗	n/a	✗
Phoenix [23]	binary	Security	✗	✗	✗	✗	✓	✗	n/a	✗
Hex-rays IDA Pro [107]	binary	Software Validation	✗	✗	✗	✗	✓	✓	n/a	✗
Relyze [68]	binary	Binary Analysis	✗	✗	✗	✗	✗	✗	n/a	✗
Rellic [114]	LLVM-IR	Security	✗	✗	✗	✗	✓	✗	✓	✗
LLVM CBackend [29]	LLVM-IR	Reverse Engineering	✗	✗	✗	✗	✗	✗	✗	✗
SPLENDID (This Work)	LLVM-IR	Collaborative Parallelization	✓	✓	✓	✓	✓	✓	✓	✓

The rest of this section further describes these three roadblocks.

Lack of Explicit Parallelism

The broad use of OpenMP [93] suggests that it is easier for a programmer to express explicit parallelism through pragmas than controlling threads through calling runtime functions (e.g., pthreads [86]). Prior work lacks the support to encode IR-level parallelism explicitly at the source code level. As a motivating example, Rellic produces code filled with parallelization setup instructions, namely instructions generated to enable parallel execution at lines 3, 7 to 24, and 38 in Figure 4.1. Some of these parallelization setup instructions are runtime-specific. For example, line 3 of the Rellic-generated code shows the runtime fork call from the LLVM/OpenMP runtime [70], *___kmpc_fork_call*, brought directly from the IR to C. Bringing runtime-specific instructions to the source code restricts portability since the decompiled code can now only be compiled with that specific runtime (e.g., libomp [70] in the motivating example). Moreover, these parallelization setup instructions make produced code unreadable. While the fork call suggests some parallelism, it is not explicit. Without specific knowledge of the OpenMP runtime library designed for LLVM, it can be difficult for a programmer to interpret this line. SPLENDID, however, is designed to produce semantic and portable parallelism.

Obfuscated Control Flow Translation

Many parallel programming models constrain the structure of the control flow. For example, OpenMP loop-related pragmas only accept loops in canonical *for*-loop format. Thus, failing to produce a canonical Control Flow Graph (CFG) required by the selected parallel programming model for source-level parallelism will result in syntactic errors in the source code. For example, it is syntactically wrong to apply *omp for* to a *do-while* loop.

For parallel programs, loops generated by previous decompilers are often *do-while* loops. This is because loop rotation [69] is a normalization pass that is commonly applied before optimizations (e.g., LLVM *-O1* or higher, and parallelizing compilers such as NOELLE [76] and Polly [45]). Loop rotation transforms each loop into its rotated form in which the exit condition succeeds the loop body. Without further analysis, rotated loops are, at best, decompiled as *do-while* loops with a guard check, as shown in line 25. The guard check did not exist in the original program; it was created by loop rotation to prevent entry to the loop when the initial state of the loop satisfies the exit condition before rotation. This leads to loop-related OpenMP pragmas being unable to be generated because the original *for* loop has been replaced with a loop that OpenMP does not support. SPLENDID instead fully recovers OpenMP-compatible canonical *for* loops.

Variable Names Irrelevant to Semantics

Prior work produces source code where variable names have no association with the original program semantics [116]. While binaries may still contain debug information that theoretically helps to reconstruct variable names, binary decompilers such as Ghidra were designed for published executables with such data stripped. Even though the IR maps a source code variable to a virtual register with debugging intrinsics, as shown in line 1 of the Parallel LLVM-IR, even fundamental compiler transformations such as Single Static Assignment (SSA) dramatically change the nature of variables in the IR.

First, the number of mappings from a source code variable to virtual registers grows dramatically. This is because promoting a memory reference to a register reference (as done by *mem2reg* in LLVM) may split a single source code variable into multiple instructions connected by a phi instruction to satisfy the SSA form. Moreover, once split, virtual registers may have an overlapping lifetime. That is,

one of two virtual registers mapped to the same source code variable may still be alive after the definition of the other (conflict). Two conflicting virtual registers cannot be mapped back to the same source code variable. Additionally, heavily optimized code regions lose such debugging intrinsics because compiler optimizations are performance-driven, lacking the intention to preserve source information which is thought to be unnecessary for improving performance. SPLENDID introduces a new technique to recover the majority of the source code variable names.

2.2 Transpilation Across Execution Models

Traditionally, near-source rewrites were performed at the AST level, where pattern matching was adequate for translating between two languages that shared the same execution model. This consistency ensured that the degree of parallelism required at the source remained unchanged across such models. For example, although NVIDIA uses CUDA and AMD uses Hip, their GPU execution models are similar, rendering the semantics of these languages nearly identical; their differ only in syntax. Consequently, tools like Hipify are effective, as they can rewrite the AST to facilitate a one-to-one syntactic mapping within the same execution model. However, challenges arise when translating PPMs across different execution models. The semantics of the source and target PPMs often require significant alterations to accommodate different levels of parallelism. For instance, although OpenMP was originally designed for multicore parallelism, its extension to GPU execution via OpenMP offloading has proven less stable, heavily relying on compiler heuristics for performance optimization. Prior studies have indicated that to achieve performance comparable to CUDA, a more explicit set of abstractions is needed beyond standard OpenMP directives (i.e., OpenMPX [35, 50, 54]). The lack of existing CUDA to OpenMP target transpilers underscores the difficulty of translating between source-to-source languages with dis-

parate levels of expressible parallelism. This indicates that merging paradigms across different execution models necessitates advanced analysis and transformations beyond what is possible at the AST level. Thus, with the trend towards increased heterogeneity in computing architectures, the conventional approach of near-source translation is becoming increasingly inadequate.

2.3 Limited Extensibility

Extensibility is essential due to the ever-expanding variety of hardware platforms and the increasing heterogeneity that necessitates further specialization in PPMs. Traditionally, near-source rewrites have been specifically developed for each pair of source and target languages. Tools like ROSE, with their unified AST, provide a middle ground where optimizations can potentially be shared across languages. However, one significant limitation remains: the need to develop a customized parser and unparser for each language pair. Furthermore, as noted in Section 2.1.2, optimizations at the AST level are largely language-specific and thus are less reusable. A new set of AST-level optimizations would likely need to be developed to achieve comparable performance in a new target PPM. If the source and target PPMs operate under different execution models, achieving extensibility at the AST level necessitates finding a suitable abstraction layer and developing coherent strategies for the front and back ends.

On the other hand, direct transpilation essentially functions as a compiler with specialized targets, involving various execution models. Setting aside economic considerations, mainstream compilers like GCC and ICX often exhibit limited extensibility due to their complexity, underscoring the need for more innovative and adaptable solutions. Among existing compilers, LLVM [101] is notable for its relative high level of extensibility. However, achieving its current level of development was a complex

and lengthy process. Consequently, compilers designed to retarget execution models face an even greater challenge: they must navigate the complexities inherent in traditional compilers while also adapting to hardware that supports diverse execution models, thereby exacerbating their extensibility issues.

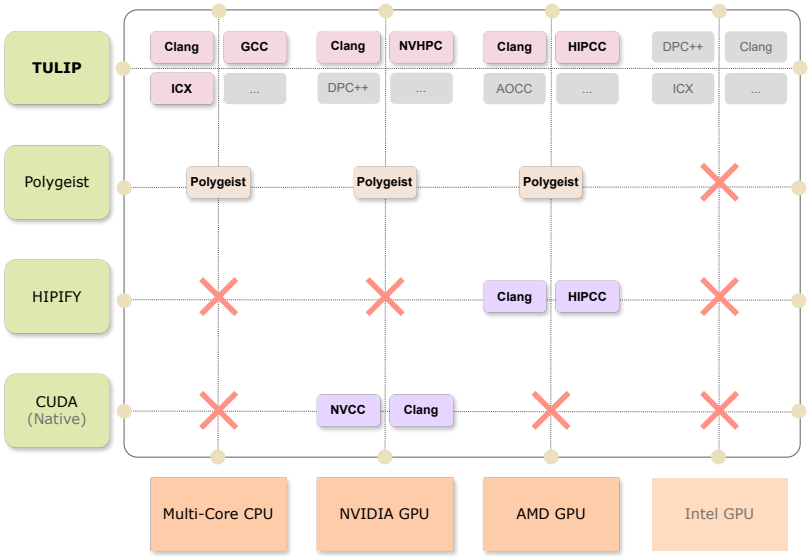


Figure 2.1: Comparison of supported platforms and toolings among prior approaches and Tulip.

Chapter 3

Tulip 🌷

3.1 Tulip Overview

This thesis introduces Tulip, an extensible transpilation framework designed to enhance code portability and performance across increasingly specialized platforms. Tulip improves upon traditional code migration methods by initially compiling the source code into a language-agnostic IR to facilitate robust transpilation and optimization. Subsequently, it decompiles this IR to generate natural source code that supports software tooling and enhances programmer interaction. This method more effectively handles significant variations in language features and execution models compared to either AST-level pattern-matching rewrites or direct transpilation to a non-native system.

As shown in Figure 3.1[1], at the IR level, language-agnostic optimizations, and specifically, state-of-the-art automatic parallelization frameworks, are utilized by Tulip to improve parallelism. Then, unlike the previous approaches that directly transpile and retarget the code, the backend of Tulip decompiles the optimized IR back to the source code in the target PPM. This enables an expert in the target PPM to understand parallelism that otherwise may be obfuscated due to the lack of expertise in

[1]Insights credit goes to Johannes Doerfert, Michael Kruse, and all other coauthors of Tulip.

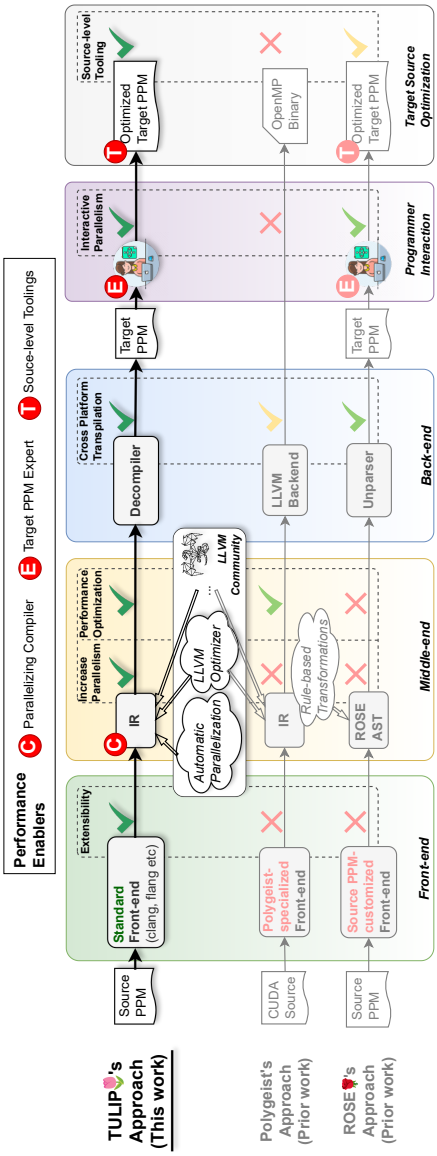


Figure 3.1: Transpilation pipeline comparison between Tulip (this work) and prior works.

either the source PPM or compiler knowledge. Furthermore, the expert can further enhance the parallelism of the transpiled code upon fully understanding the available parallelism already presented at the source in the target PPM.

Another advantage of generating source code rather than executables for different target machines is the ability to utilize numerous source-level toolings. As demonstrated in Figure 2.1, Tulip facilitates this by transpiling CUDA to OpenMP, enabling at least eight compiler selections across three hardware platforms. This number could increase with the variety of tools available for the target PPM. The flexibility to choose any compiler toolchain allows each application to achieve the best possible performance, taking advantage of the specific optimizations that each compiler offers, which are often fine-tuned for particular types of applications. In this manner, all performance enhancers—including language-agnostic optimizations, programmer interactivity, and toolings—collectively contribute to maximizing the speedup on the target hardware platform.

3.1.1 Performance Enablers In Action

Figure 3.2 illustrates how Tulip leverages all performance enablers described in Section 2.1 to optimize performance in code migration. The example provided is a simplified kernel commonly used in High-Performance Computing (HPC) applications, performing batch processes of matrix-vector multiplication. Although the multiplication kernel itself can run in parallel on GPUs, as shown at line 12, the outer batch processing loop at line 9 cannot be parallelized due to an external library call with unknown dependences. This loop-carried dependence on its argument C cannot be verified, as indicated at line 14. Initially, Tulip compiles the CUDA source code to LLVM-IR while preserving the parallel semantics specified by the CUDA expert (Step 1). Subsequently, a parallelizing compiler may enhance the IR by detecting a reduction operation on *dot* at line 9 and vectorizing the loop at line 7 (Step 2).

By generating natural source code in the target PPM, OpenMP, Tulip presents the parallelism preserved from the source PPM (CUDA) and that generated by the parallelizing compiler (NOELLE) to the OpenMP expert in the form of an OpenMP pragma (Step 3).

Next, an expert in OpenMP can quickly understand the Tulip-generated OpenMP code since it appears natural. Such an individual can directly grasp the parallelism initially described in CUDA or automatically generated by the compiler without requiring knowledge of CUDA or automatic parallelization techniques. With an understanding of the acceptable output space—specifically, whether the external library may have dependences across iterations for C at line 13 in Step 3—the programmer and programmer alone can decide whether the outer loop at line 1 can be parallelized. Furthermore, as an expert in OpenMP, they can collapse the nested loops generated in replacement of the original CUDA kernel call into a single, flattened loop, enhancing parallel execution efficiency across iterations. Note that loop collapse is expressible only in OpenMP and not in CUDA since CUDA does not support the concept of loops. This difference exemplifies how source and target PPMs may represent incompatible forms of parallelism, and thus, further manual optimization of the transpiled code using knowledge of the parallelism expressible in the target PPM is highly desirable, as outlined in Step 4. Lastly, any compiler toolchains that support OpenMP can be used to compile the code generated in Step 4 onto any CPU or GPU platform, ranging from out-of-the-box options like GCC, ICX, and NVHPC, to the continuously evolving OpenMP offloading efforts within the LLVM community.

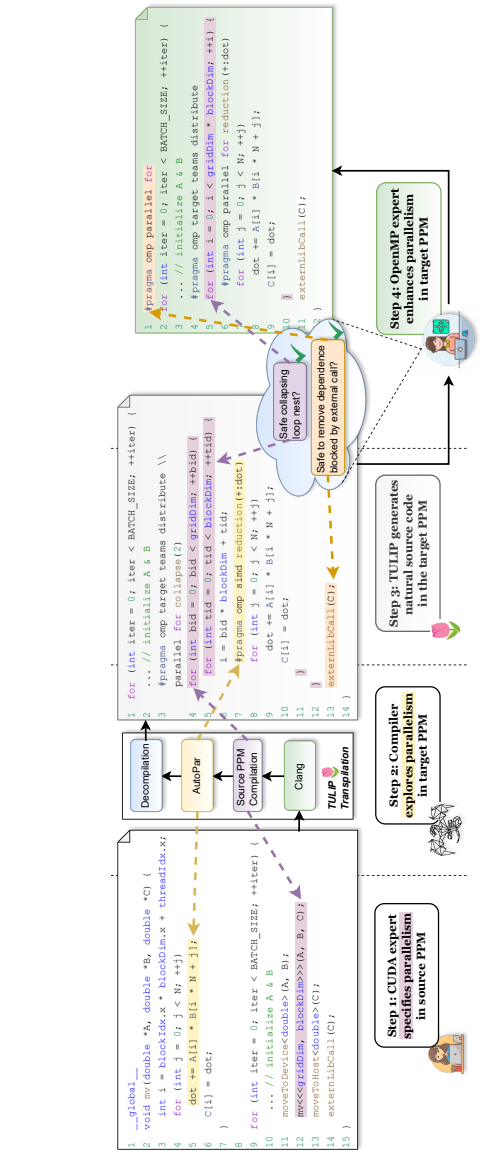


Figure 3.2: Tulip transpilation generates natural source code, in which programmers of the source and target PPMs and the parallelizing compiler exhaustively explore parallelism.

3.2 Design and Implementation

The design of the Tulip transpilation framework is structured to engage programmers specializing in source PPMs, parallelizing compilers, and those specializing in target PPMs in a collaborative process of parallelization. While Figure 3.2 illustrates one possible pipeline where Tulip transpiles CUDA to OpenMP, enhanced by a parallelizing compiler, NOELLE, Tulip's methodology can be adapted for a broader range of source and target PPMs, employing various parallelizing compilers or none at all. This section delves into the technical specifics of how Tulip transpiles CUDA into OpenMP source code and details each stage of the process. It also explains how Tulip's framework can be expanded to incorporate additional source PPMs, parallelizing compilers, performance-enhancing transformations, and target PPMs.

3.2.1 Source PPM Compilation

Tulip's method for integrating a new source PPM capitalizes on the capabilities of well-established compiler frontends within the active LLVM community, augmented by a minimal post-processing pass. This pass normalizes the compiled LLVM-IR for GPU targets into LLVM-IR optimized for CPU targets, a process elaborated upon in Section 3.2.1. For instance, instead of creating a new compiler frontend for CUDA, Tulip utilizes Clang [62], the standard frontend, to compile CUDA source code into GPU-targeted LLVM-IR, as depicted in Figure 3.3. This GPU-targeted LLVM-IR then undergoes a customized lightweight frontend pass, designed specifically for CUDA as the source PPM, converting it into sequential CPU-targeted IR with parallelism represented as LLVM metadata. In contrast to the Clang compiler, which consists of at least 400,000 lines of code, this frontend pass for converting CUDA targets comprises only about 1,000 lines of code, making it exceptionally streamlined and manageable. Several popular PPMs commonly utilized in HPC applications, such

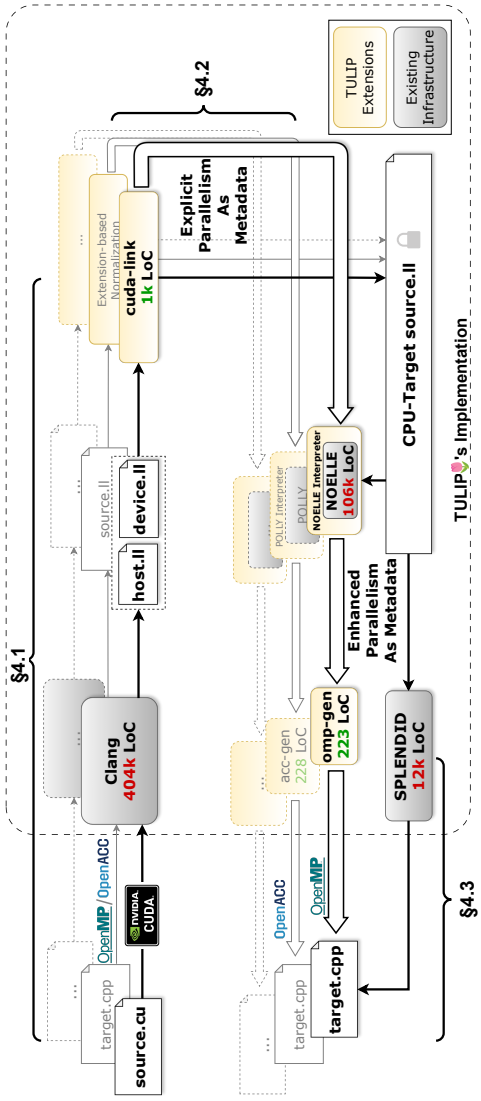
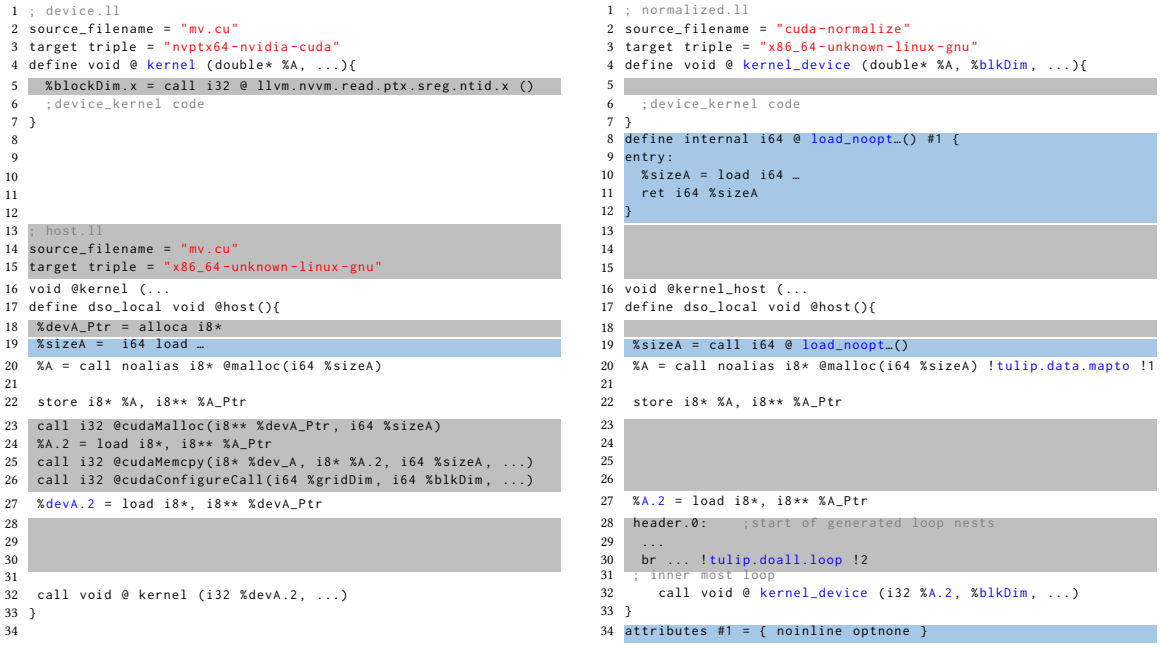


Figure 3.3: Tulip transpilation generates natural source code, in which programmers of source PPM, the parallelizing compiler, and programmers of the target PPM exhaustively explore parallelism.

as Fortran (i.e., Flang [100]), OpenMP (i.e., Clang), CUDA (i.e., Clang), and OpenACC (i.e., Clacc [36]), are already supported by robust LLVM compiler frontends. The following describes the implementation in Tulip of the normalization pass, which converts GPU-target parallel LLVM-IR compiled from CUDA into CPU-target IR, with parallelism encoded as metadata (as shown in cuda-link in Figure 3.3).



(a) IR Before Normalization

(b) IR After Normalization

Figure 3.4: IR before and after CUDA Normalization.

GPU-to-CPU Target Conversion

Clang generates a CPU-target and GPU-target LLVM-IR for host and device code, respectively, for each CUDA source code, denoted by different data layouts and target triples. Functions in Host and device IR can have identical names with different function bodies. For example, shown in Figure 3.4 (a) is a host function call (line 30) to a stub function on the host side (line 14), which in turn calls the actual kernel to be run on the device (line 4). Often, the stub function on the host and device have identical names, as shown at line 14 and line 4. Thus, to link the host and device IR,

we developed a customized linker, *cuda-link*, built on top of *llvm-link* to resolve the name conflicts.

The first step in cuda-link is to unify the target and data layout to be those of a CPU-target IR and create linked.ll from the host and device IR. cuda-link is almost identical to the standard llvm-link except for at link time, i) change the target and data layout triple to be of a CPU machine model and ii) rename functions upon a naming in conflict. Then, the kernel invocation, previously invoked indirectly through a stud on the host, is replaced by a direct call to the kernel on the device. In the example shown in Figure 3.4, the call to stub function at line 30 is replaced by a direct call to the kernel function at line 4, which is now renamed as *kernel_device* after resolving naming conflict. This can be done by searching the actual kernel invocation within the stub function.

Further normalization needs to be done to retarget the linked IR to CPUs. Grid and block dimensions specified in the CUDA configuration call (line 24 in (a)) are used to transpile the kernel into loop nests around the device kernel invocation (line 26-28 in (b)). Since each grid and block can be up to three dimensions, at most, six nested loops can be generated. Within the device kernel function, the CUDA-specific intrinsics to get the grid and block dimensions and indexes (e.g., *llvm.nvvm.read.** as in line 5 in (a)) are replaced by explicit parameter passing (line 4 in (b)). The current implementation optimizes loop nest generation to exclude generating loops for dimensions of only one iteration. Data movements from and to the device are detected from cudaMemcpy calls (line 23 in (a)) and eliminated by replacing the use of device memory objects with the corresponding host objects copied in and out from cudaMemcpy (line 18 in (b)). Once all uses of the device memory objects are erased, all the pointer chasing, casting, allocation and deallocation (e.g., cudaFree) instructions become dead code and easily optimized, a standard 01 level optimization can then remove them all. In the example shown, line 16 and 21-22 can all be removed.

Additionally, special math functions in CUDA are replaced by their equivalents in standard C libraries, such as sqrt and fmax. Lastly, once all the CUDA function calls are removed, the generated IR is no longer library-dependent, CUDA-dependent, and GPU target-dependent.

As the normalization pass is a small building block to showcase a promising future of PPM transpilation via programmer and compiler interaction, the current implementation does not accommodate all CUDA semantics, such as device synchronization (e.g., ___syncthreads() or CUDA shuffle instructions). Prior work [81, 112] has shown how synchronization on the device (e.g., ___syncthreads() or CUDA shuffle instructions) can be transpiled on CPUs, such as applying loop distribution at each barrier. While the engineering required to support synchronization increases the complexity of the normalization pass design compared to the standard Clang compiler, we expect it to still be lightweight compared to the standard frontend, Clang.

Parallelism as Metadata

When grids and blocks of threads are normalized to be loop nests in a CPU-target IR, every loop in that loop nest is DOALL parallel; namely, there is no loop-carried dependence between iterations. The normalization pass, then, upon creating the loop nest, attaches to each terminator of the loop header metadata denoting the loop as DOALL. Information about data being copied to and from the device can be traced through cudaMemcpy, through which the original creation of the memory object on the host is denoted with the direction of movement (i.e., to device, from device, or both) and array dimensions. Additionally, the live range of memory objects on the device per device kernel invocation is identified in topological order, starting with the first memory object copied into the device kernel and ending with the last memory object copied out of the device. The live range of memory objects is important to group statements at source and reconstruct data mapping such as *#pragma omp*

data map in OpenMP and *#pragma acc data* in OpenACC, respectively. Attaching metadata to loops (e.g., llvm.loop) or influencing optimizations using metadata (e.g., loop vectorization and interleaving) are well-adopted as simple solutions to loop-level transformations. Potentially, LLVM canonicalization passes such as *mem2reg*, *instcombine*, or *dce* can remove instructions to which Tulip metadata are attached. This can be resolved by outlining instructions into functions with attribute *optnone* prior to canonicalization. For example, in Figure 3.4 (a) highlighted in blue, *%sizeA* on line 17, is a load instruction that can potentially be promoted as a register value through *mem2reg*. To prevent the removal of Tulip metadata, in Figure 3.4 (b), the instruction is outlined into its equivalent function *%load_noopt* at line 17 with attribute *optnone* at line 32. In this way, Tulip still largely benefits from even peephole optimizations by disabling them on a few instructions to which Tulip metadata are attached. Other forms of parallel representation that come with greater capacity for optimizations also exist within the community, such as Tapir [109] and PS-PDG, and can potentially be adopted by the current Tulip design. Note that at this step, the compiled IR is fully transformed into being CPU-targeted; as it is, it can then be passed into the backend of Tulip to be decompiled into source code, or be compiled into executables.

3.2.2 Interaction with a Parallelizing Compiler

We define in the Tulip pipeline the interaction with a parallelizing compiler to be, instead of applying the actual parallelization onto the CPU-target IR, the parallelizing compiler adopts Tulip's API in the form of metadata, and while making no modification to the IR itself, emits more parallelism as metadata. This design choice first ensures that the interaction with any parallelizing compiler is easily extensible, that the Tulip backend, namely the decompiler, does not need to understand the engineering effort and design choices made by each parallelizing compiler, such as which

parallel runtime library to use (e.g., libomp [70] or libgomp [43] for OpenMP), and scheduling policies (e.g., dynamic, static, and chunking sizes). In this way, Tulip decompiler logic needs no change to whenever a new parallelizing compiler comes into the play, and it can always assume it receives a CPU-target IR without parallel runtime library calls as the input, and parallelism can always be interpreted in the defined API to the parallelizing compiler to be involved. Beyond hindering extensibility, the decision to express parallelism as metadata instead of applying parallelization to IR prevents further code obfuscation and thus produces unnatural code after decompilation. Prior work [117] has shown that unnatural code not only prevents programmer interaction, but when it comes to parallel programming models, unnaturalness can result in incorrect code. For example, since OpenMP requires loops to be in the canonical form, when a rotated loop in IR is decompiled as a do-while loop, applying OpenMP pragmas to it results in compilation error. To demonstrate how a parallelizing compiler participates in the Tulip pipeline, we integrated the state-of-the-art compiler, NOELLE, to generate additional Tulip metadata identifying DOALL loops with parallelization enablers such as reduction. Polly, PLUTO, and other parallelization frameworks in LLVM can similarly be integrated with the extension of understanding and emitting Tulip metadata.

Interaction with NOELLE

Internally, NOELLE relies upon the Program Dependence Graph (PDG) [39] to determine the applicability of a particular parallelization scheme. For a loop to be DOALL-parallelizable, all loop-carrying dependences of that loop need to be disproven. This is done through state-of-the-art memory analysis frameworks such as SCAF [7] and SVF [113]. If the only loop-carried dependences existing after all the analysis are the ones resulting from reduction operations such as sum, min, and max, a remedy can be applied so that the loop is DOALL with reduction. Compared to LLVM, which

supports only single-use reducible variables, NOELLE identifies reducible variables of a loop independently on its uses. Normally, once NOELLE determines that DOALL is applicable to a certain loop, it then outlines the loop body, inserting parallel runtime function calls to spawn threads and dispatching the loop body onto available threads to run in parallel. This is a standard approach to most parallelizing compilers, first identifying parallelism, then injects its own runtime to realize such parallelism. To participate in the Tulip pipeline, instead of generating the parallel code for the DOALL loop, NOELLE generates Tulip DOALL metadata at the terminator of the loop header while leaving the loop as it is. Since NOELLE provides APIs to query whether a loop is DOALL-, DSWP [105]-, or HELIX [24]-parallelizable, integrating NOELLE into Tulip requires no modification to the parallelizing compiler itself but simply adding a pass to add Tulip metadata using the query API of NOELLE. With 264 LoC to build the query pass, Tulip integrates with NOELLE, whose parallelization tools along with the abstractions are built with 106k LoC.

Beyond the primary form of interaction by querying NOELLE, collaboration can be enhanced by the parallelizing compiler using the metadata generated from Tulip frontend (i.e., parallelism from source PPM) to improve parallelization. Specifically, in NOELLE, Tulip's metadata that denotes parallelism from the frontend can be used, in addition to the analysis frameworks, to disprove dependences. For example, if a loop is denoted as Tulip DOALL from the frontend, such as loops created from CUDA normalization, all loop-carried dependences of the loop can be disproven. Though this work includes the implementation for this form of interaction since the frontend PPM is CUDA, CUDA parallel kernels can be easily analyzed as DOALL as they are often simple affine loops. Thus, the implementation on which the evaluation section is based uses the formerly described query pass.

3.2.3 Source Code Generation

Source code generation consists of IR decompilation and parallel construct generation. The decompiler of Tulip is adopted from SPLENDID, which reconstructs all counted loops with induction variables into loops and restores the majority of variable names using the source variable detection algorithm described in SPLENDID. Code generated by SPLENDID has shown great improvement in naturalness compared to the best prior approach. Unlike SPLENDID, which decompiles IR with lower-level parallel runtime function calls, Tulip's transpilation and interaction with a parallelizing compiler does not engage in parallel code generation. Since in Tulip, no parallelization is actually applied to the IR itself, no parallel runtime function calls are generated to make the IR library dependent, and no parallelization-enabling transformation such as outlining occurs, obfuscating the IR to be decompiled. Thus, unlike SPLENDID, which includes complex analysis to detransform outlining parallel regions and remove parallel runtime calls, the complexity of Tulip decompilation is greatly reduced since it receives an IR with almost no optimization applied as an input. Thus, the code generated by Tulip is as natural as that of SPLENDID but with a simpler decompilation design.

Tulip decompiles the CPU-target IR separately from parallelism as metadata. A specialized pragma generator is developed for each targeted parallel extension. For instance, *omp-gen* is implemented to interpret parallelism and generate OpenMP pragmas, as illustrated in Figure 3.3. The current metadata interface accommodates DOALL parallelism transpiled from CUDA, translating it into OpenMP parallel loops or SIMD, depending on the applicability. When multiple loops within the same nest are DOALL, the Tulip backend adds pragmas to the outermost DOALL loop to enhance data locality and expand the parallel region. For loop nests exceeding two levels, Tulip employs the *collapse* directive in OpenMP to boost the parallelism visible to the compiler. Although CUDA lacks a CPU-equivalent reduction code

pattern (requiring special libraries or more sophisticated synchronization for GPU reductions), a parallelizing compiler can identify additional reduction operations after the IR is re-targeted to the CPU. These reduction operations are also captured as metadata and converted into OpenMP reductions. Data transfers between the host and device (e.g., line 18 in Figure3.4(b)) are converted into *#pragma omp data* if OpenMP offloading is utilized. If the target PPM is OpenMP without offloading, namely multicore targeted OpenMP, since the code is already normalized for CPU-targeting with all data movement eliminated at the IR level, metadata related to data movement can be disregarded.

The advantage of separately generating sequential code and parallel construct is that the generation of sequential code can be shared for PPMs designed as extensions to C/C++, especially those that are directive-based (e.g., OpenMP and OpenACC). In the case of generating OpenMP and OpenACC code, the sequential code generation portion is entirely shared, while the metadata encoding parallelism is used to construct OpenMP or OpenACC pragmas depending on the desired output.

Chapter 4

SPLENDID

4.1 SPLENDID Overview

This work introduces SPLENDID, a decompiler framework that produces portable OpenMP code natural for programmer involvement. As shown in Table 4.1, SPLENDID includes all features necessary for producing portable code. Loop-related transformations, such as Loop Rotation De-transformation, restore loops in IR to canonical *for* loops in source code. Features related to transforming runtime library calls, such as Parallel Runtime Elimination, remove runtime-specific constructs and explicitly express parallelism as OpenMP pragmas at the source level. Together, these techniques enable SPLENDID to produce code not only syntactically correct but also compilable universally with any runtime library. In addition to being portable, the same set of techniques also makes SPLENDID-produced OpenMP code more natural, because they restrict code structures (e.g., making loops more natural), and represent parallelism using OpenMP (e.g., eliminating obfuscated runtime function calls). Moreover, SPLENDID chooses variable names that reflect code semantics. The more natural the decompiled code is, the less effort is required from a programmer to understand what and how a compiler parallelizes, and the better chance of additional

Table 4.1: Techniques of SPLENDID to produce portable code that is also natural for manual improvement.

Techniques	**Portability**	**Naturalness**
Parallel Runtime Elimination	●	●
Loop Parameter Restoration	●	
Loop Rotation De-transformation	●	●
For Loop Construction	●	●
Parallel Code Inlining		●
Pragma Generation	●	●
SSA Detransformation	●	●
Source Variable Renaming		●

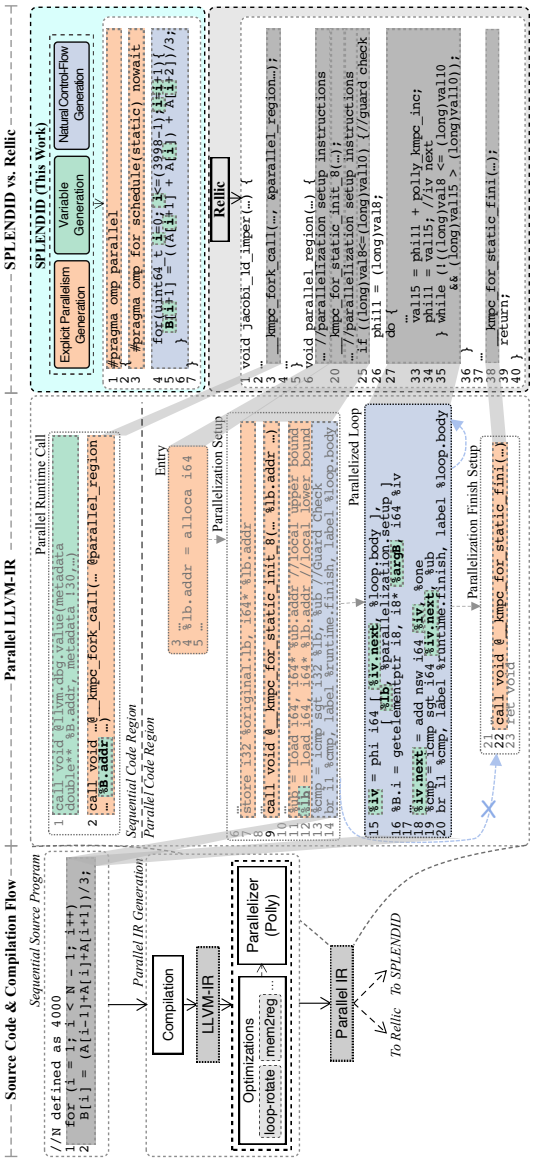


Figure 4.1: A comparison of code decompiled using Rellic [114] and SPLENDID. The motivating example is a simplified hot loop from `jacobi-1d-imper` in PolyBench [98]. The original sequential code is compiled into LLVM-IR, optimized by LLVM `-O2`, and parallelized using Polly [45]. The resulting parallel LLVM-IR then serves as the input for decompilation. Compared to code produced by Rellic, code produced by SPLENDID is portable to any host compiler and natural.

profitable collaborative parallelization.

By using SPLENDID, a programmer *i)* is freed from parallelizing that which the compiler is capable of parallelizing, *ii)* can make incremental improvements to what the compiler can parallelize, and *iii)* can focus on parallelizing what the compiler cannot parallelize. The rest of this section starts by describing techniques developed to overcome each hindrance identified in Section Section 2.1.3. Then, we present an example demonstrating how each technique in SPLENDID contributes to producing portable and natural source code. Lastly, we present case studies to show how *i)* SPLENDID successfully supports collaborative parallelization, a promising direction in bringing performance in the post-Moore's Law era, and *ii)* despite the enabling power of collaborative parallelization, SPLENDID advances the state of decompilation by making critical optimizations clearer to programmers.

4.1.1 Explicit Parallel Translation using OpenMP

As far as we know, SPLENDID is the first decompiler that translates parallel IR into portable OpenMP code [93]. This is achieved by first finding loops parallelized from a parallel code region with extraneous parallel execution setup code. Then, the parallel execution setup is removed, namely, instructions that enable a code region to run in parallel, including parallel runtime function calls. With no implementation-specific code at the source code, the decompiled code is portable to any compiler and more natural. After that, SPLENDID generates OpenMP pragmas to replace IR-level parallelism since OpenMP is widely accepted and can be easily understood by developers. Lastly, since the parameters of a parallel loop (e.g., loop bounds and step sizes) are thread-local, loop parameters are restored to the original sequential loop parameters.

4.1.2 Enhanced Natural Control Flow Translation

Unlike many loop optimizations that bring considerable performance gain, loop rotation, though critical for loop normalization, does by itself not improve program performance but simplifies the implementation of other loop optimizations. Thus, SPLENDID is designed to de-transform rotated counted loops to *for* loops. SPLENDID-generated loops are much more comprehensible since the loop structure is closer to the original source code. Moreover, the well-structured loops generated by SPLENDID make pragma selection easier. As depicted in the motivating example, when the loop is restored to a DOALL *for* loop (with no dependence across iterations), simply applying *#pragma omp for* parallelizes the loop. However, SPLENDID intentionally does not de-transform optimizations other than loop rotation as they do not hinder portability and are critical to performance.

4.1.3 Natural Variable Reconstruction

Since SSA separates what was originally one source variable into multiple virtual registers (instructions), SPLENDID collapses instructions connected through a phi instruction into one variable. Further, SPLENDID beautifies variables by assigning names to them that are indicative of code semantics. This work proposes relating each instruction with a source variable through mappings extracted from debug information. Unfortunately, because of conflicts, mappings of virtual registers to source variables cannot be directly used to assign variable names. SPLENDID provides a novel verification module that detects and removes conflicts. Details about conflict elimination are described in Section Section 4.2.3.

If debug information is missing for an instruction, SPLENDID takes one step further and attempts to associate this instruction with debug information from another code region. Specifically, while debug information is preserved throughout transformations to the compiler backend in LLVM, optimizations such as automatic paral-

lelization [6, 76, 45] developed on top of LLVM are not designed with decompilation in mind. Thus, when these optimizations insert new instructions, they may not have precise debug information. To overcome this constraint, SPLENDID assigns source variables to a code region without source information by relating it to a region where source information is present through inlining.

4.1.4 SPLENDID in Action

Figure 4.1 shows how each aforementioned technique contributes to the final translation of the example loop. First, all the parallel runtime setup instructions, which are at lines 2–12 and 22 in the parallel LLVM-IR, are used to *i)* restore the parallelized loops to a sequential loop and *ii)* generate OpenMP pragmas for the sequential loop. The parallel code region and its input are some of the inputs to one of the runtime calls, *___kmpc_fork_call* at line 2, through which multiple workers are spawned to execute the parallel region specified as inputs. Parallel regions are functions containing the outlined and parallelized loops from the original code. By recognizing inputs to a fork call, the loops that are parallelized and the original sequential loop parameters are recognized by SPLENDID. The parallelized loop from lines 15 to 20 is then found between the runtime initialization call at line 9 and the finish call at line 22. Since each instance of the parallel region only executes a portion of the parallelized loop, loop parameters such as loop bounds are unique to each worker and are calculated using the original loop parameters as inputs to the initialization call. To restore each parallelized loop to the combined iteration space of all threads, SPLENDID replaces the loop parameters with the values before the initialization call. For example, the lower bound at line 12 is replaced with its original value of 0 at line 7. Once the parallel region is transformed into a sequential loop with OpenMP pragmas, SPLENDID removes all parallelization setup instructions as depicted in gray and inlines the outlined parallel region into the sequential code region.

At this point, the inlined loop is still rotated, and SPLENDID, after verifying the loop is counted by finding the induction variable at line 15, transforms the sequential rotated loop into the *for* loop shown at lines 4 to 6 in the final produced code in Figure 4.1. Note that SPLENDID removes the guard check at lines 13 and 14 in the parallel LLVM-IR by proving that it is equivalent to the initial exit condition of the transformed *for* loop. Constructing the *for* loop not only improves naturalness but also validates the use of the *#pragma omp for* at line 3, as the pragma can only be used on a *for* loop.

Lastly, SPLENDID collapses and renames virtual registers. SPLENDID removes each phi instruction by replacing its inputs with itself or an expression containing itself. In the motivating example, $iv.next$ is replaced with iv, the name of the phi instruction, which is then detected as an induction variable and generated as the variable i at lines 4 and 5 in the produced code. As the final step, if there is debug metadata relating a source variable name to a virtual register, an instruction, SPLENDID assigns each instruction the source variable name it is related to unless there is a lifetime overlap (such a conflict is described in Section Section 4.2.3). When an instruction has no debug metadata, such as *argB* from the parallel region at line 16, it can indirectly relate to a source variable through inlining in the following way. First, the sequential code region contains debug metadata that maps instruction *B.addr* to variable B at line 1. Meanwhile, since the parallelized loop is inlined, *argB* is replaced by the input to the parallel region at line 2, which is *B.addr*. Thus, as *B.addr* is mapped to the source variable B, so does *argB* after inlining. As SPLENDID finds no conflict with this mapping, variable B can safely replace *argB* as the generated variable name at line 5 in the produced code.

4.1.5 Case Studies

This section presents case studies to demonstrate two use cases of SPLENDID. First, SPLENDID enables compiler-programmer collaborative parallelization, as shown in Table 4.2 and Figure 4.2. Second, SPLENDID advances the state of decompilation by presenting natural code in the presence of aggressive compiler transformations, as shown in Figure 4.3.

Compiler-Programmer Collaboration

Instead of letting the programmer try to optimize the whole program, we propose an alternative approach. That is, before any manual optimization, let the compiler present its parallelization plan to the programmer through SPLENDID.

As shown in Table 4.2, there is a large overlap between what the compiler and the programmer alone can parallelize. By first letting SPLENDID produce parallel code that is portable to any compiler, programmers are freed from parallelizing loops that are parallelizable by a compiler. By knowing what the compiler can parallelize, a programmer can focus on parallelizing loops that the compiler could not parallelize. This way, SPLENDID enables the parallelization of all loops proposed by the programmer and the compiler.

Moreover, whatever is already parallelizable by the compiler may also benefit from the knowledge of a programmer. As shown in Figure 4.2, the example loop can be conditionally parallelized by the compiler (e.g., Polly [45]) when A and B do not alias. Thus, an aliasing check is injected to provide a fallback to the sequential version of the code when A and B alias. The compiler can emit such aliasing check because *i)* alias analysis is limited (e.g., because it is limited to intra-procedural analysis) to proving that A and B do not alias even if the programmer has only called *MayAlias* with separately allocated units, or *ii)* alias may indeed occur at runtime as the programmer may pass the same pointer to both arguments A and B, as in the

```
void MayAlias(double* A, double* B, double *C) {
//... initializations
 for (i = 0; i < N-1; i++){
  A[i+1] = M_PI*B[i] + exp(C[i]);
 }
}
void main(...){
 double *A = (double*) malloc(n * sizeof(double));
 double *B = (double*) malloc(n * sizeof(double));
 double *C = (double*) malloc(n * sizeof(double));
 MayAlias(A, B, C);
 MayAlias(A, A, C);
}
```

(a) Original Code

```
void MayAlias(double* A, double* B, double *C) {
  if ((A+1000) <= B | (B+999) <= (A+1) & (A+1000) <= C | (C+999) <= (A+1))
       {//Aliasing check

  #pragma omp parallel
  {
   #pragma omp for schedule(static) nowait
   for(uint64_t i = 0; i<=998; i = i + 1){
    A[i+1] = (exp(C[i]) + B[i] * 3.1415926535897931);
   }
  }

 } else {
  for(uint64_t i = 1; i < 999;   i = i + 1){
   A[(i+1)] = (exp(C[i]) + B[i] * 3.1415926535897931);
  }
 }
}
```

(b) SPLENDID Output

```
void NoAlias(double* restrict A, double* restrict B, double* C) {
 for (i = 0; i < N-1; i++){
  A[i+1] = M_PI*B[i] + exp(C[i]);
 }
}
void <func>_InPlace(double* A, double* C);

NoAlias(A, B, C);
<func>_InPlace(A, C);
```

(c) Specialized Optimization by Programmer

Figure 4.2: An example of the programmer removing compiler generated aliasing checks.

case of line 12 in the original code. In the first scenario, a programmer, knowing A and B cannot alias, can remove the sequential version of the code, eliminating the computational overhead from the aliasing check. If the programmer is a compiler writer, he/she no longer needs to search in IR to be informed of the limitation in its compiler analysis. The alias analysis can be improved by looking at decompiled source code. In the second scenario, the programmer, after knowing that the compiler can parallelize the cases when A and B do not alias, can improve the original code by simply restricting the accessing of example code to only when A and B do not alias (i.e., line 1 in (c)) and focus on optimizing cases when A and B must alias in a separate function (i.e., line 6 in (c)). In both scenarios, the programmer is freed from manually parallelizing the example loop under the condition that A and B do not alias. More importantly, such interaction greatly improves the final produced code in both naturalness and performance. Such interaction is only enabled through SPLENDID, as any unnaturalness introduced at the assembly level or by other decompilers can make finding the aliasing check extremely difficult.

Advancement in Decompilation

Natural decompilation goes beyond producing code identical to the original source code. An advanced decompiler should present performance-enabling optimizations the compiler applies in a human-readable way. SPLENDID achieves this by making a trade-off of what to de-transform. Figure 4.1 has already shown the natural representation of the parallelization of SPLENDID. Instead of aggressively applying de-transformations to all compiler optimizations, SPLENDID chooses to de-transform only peep-hole optimizations that intrude unnaturalness while having little or no influence on performance (e.g., SSA and loop rotation). This design choice results in aggressive optimizations, such as loop transformations, remaining untouched and presented to the programmer, as shown in Figure 4.3. Performance engineers can then

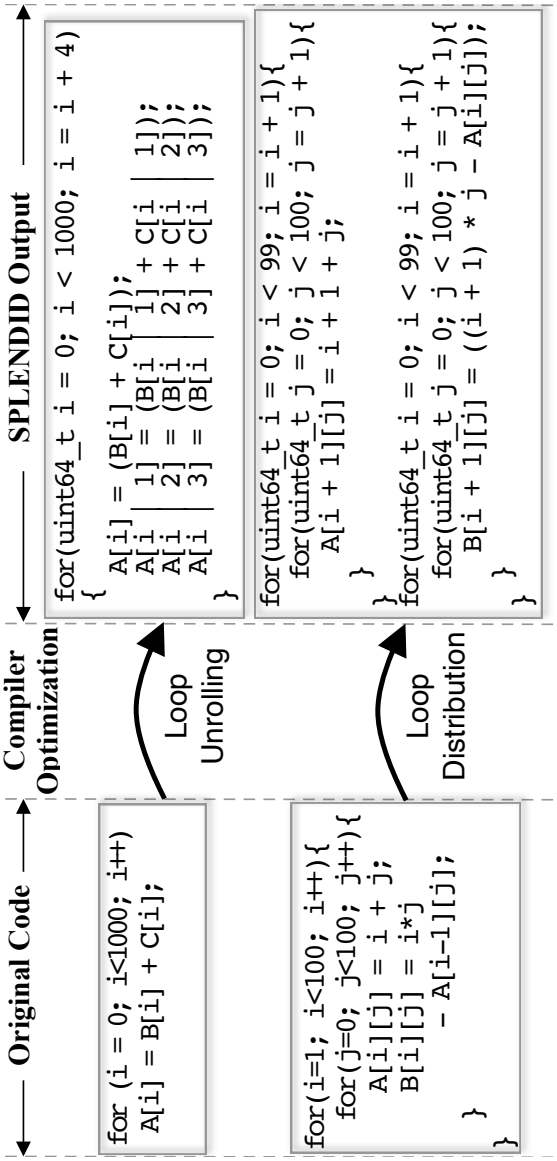


Figure 4.3: Decompiling loop optimizations using SPLENDID.

read the output of SPLENDID and quickly find relevant code properties, such as unrolling factors.

Table 4.2: SPLENDID enables the parallelization of all loops that the compiler and programmer can parallelize alone, with reduced manual effort.

Benchmarks	Programmer Parallelized [44]	Compiler Parallelized [45]	Total Parallelizable	Eliminated Manual Parallelization
2mm	2	2	2	2
3mm	3	6	6	3
adi	6	2	7	1
atax	2	1	3	0
bicg	2	1	3	0
doitgen	1	1	1	1
fdtd-2d	4	2	4	2
floyd-warshall	1	1	2	0
gemm	1	4	4	1
gemver	4	4	4	4
gesummv	1	1	2	0
jacobi-1d-imper	2	2	2	2
jacobi-2d-imper	2	2	2	2
mvt	2	3	3	2
syr2k	2	4	4	2
syrk	2	4	4	2
Total	37	40	53	24

4.2 Design and Implementation

This section describes how SPLENDID obtains the features described in Section Section 4.1.

4.2.1 Parallel Source Code Generation

SPLENDID explicitly represents parallel code regions using OpenMP, which consists of the OpenMP pragmas and sequential code regions to which the pragmas are applied. OpenMP requires loops to be strictly structured in counted *for*-loop fashion with no loop-carried dependences. However, beyond the fact that LLVM loops vastly differ from OpenMP-compatible loops, parallel IR contains a large body of low-level OpenMP runtime setup code, making it extremely difficult to extract the parallelized loop. Nevertheless, SPLENDID transforms highly obfuscated IR into portable and natural parallel C code. It starts with the OpenMP Semantic Analyzer.

Parallel Semantic Analyzer

Parallel Semantic Analyzer first collects the runtime calls and then extracts the parallel code regions from the runtime fork calls: *__kmpc_fork_call*. The fork function gets an outlined function as an argument, and when it is called, it creates multiple workers to work on the outlined function simultaneously. The Parallel Analyzer then finds the parallel code region through the outlined function.

Parallel Region Detransformer

Parallelized Region Detransformer *i)* extracts each parallelized loop from the parallel code region, *ii)* restores loop parameters, and *iii)* removes all the parallelization setup instructions such that the output only contains the original loop. To detect a parallelized loop, the Parallelized Region Detransformer searches for loops between a

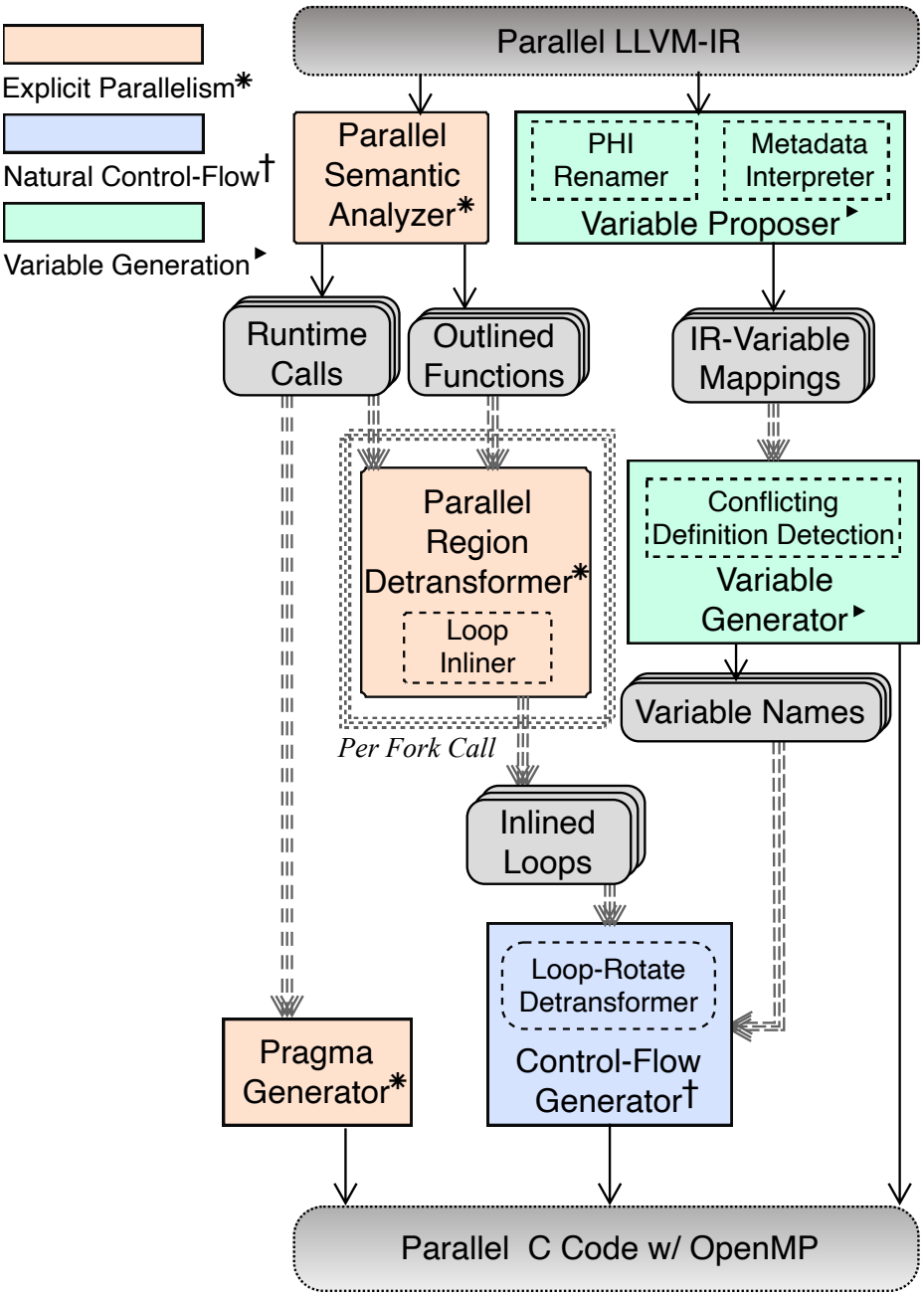


Figure 4.4: The design and workflow of SPLENDID.

pair of runtime function calls that initializes and ends a parallelized region, such as *___kmpc_for_static_init_8* and *___kmpc_for_static_fini* in Figure 4.1. Then, loop parameters are restored by replacing them with those used as arguments for the initialization call. Since extraneous instructions can cause an error when placed between *#pragma omp for* and the parallelized loop, the parallelization-related instructions are removed.

The loop is then inlined into the sequential code region. Since the runtime fork call indirectly calls the outlined function, inlining requires that the Loop Inliner replaces arguments passed into the runtime fork call with their corresponding arguments of the outlined function. Lastly, the Loop Inliner replaces the runtime fork call in the sequential region with the transformed sequential loop, eliminating the last runtime-dependent instruction.

Pragma Generator

Explicit parallelism is presented using sequential loops with OpenMP directives. Pragmas are generated from runtime function calls through one-to-one or many-to-one mappings or from static analysis (e.g., private clause). Scheduling policy and chunk size are determined by extracting and interpreting parameters of runtime initialization calls. When more than one correct translation exists, the Pragma Generator uses the most performing pragmas. For example, a pair of *___kmpc_static_for_init_8* and *___kmpc_for_static_fini* with no barrier calls can be transformed into both *#pragma omp for schedule(static)* and *#pragma omp for schedule(static) nowait*. The Pragma Extractor, in this case, produces the latter since there is no implicit barrier.

While most pragmas are generated through runtime calls, the Pragma Generator minimizes the use of clauses to reduce the knowledge required for programmers to interpret code produced by SPLENDID. For example, if the earliest definition of a variable is inside the parallel region, declaring it inside the parallel region by default

makes the variable private, thus eliminating the need of using the *private* clause.

4.2.2 Natural Control-Flow Generation

SPLENDID generates *for* loops which OpenMP requires through de-transforming loop rotation. The Loop Rotate Detransformer first attempts to produce a *for* loop from a well-structured rotated loop by searching for loop parameters, including the induction variable and loop upper and lower bounds. Traditionally, inverting a rotated loop is done by loop peeling, which separates the first iteration from the rest of the loop so that the exit condition can be moved before the loop body. A loop initially rotated from a *for* loop is counted and thus can omit loop peeling by directly changing the upper bound to allow execution of one more iteration since the exit condition is checked before executing the loop body. Inverting the rotated loop creates a *for* loop, but within a guard check created by loop rotation to ensure that if the exit condition before loop rotation fails, the loop is not executed once by mistake. This guard check can be removed if it is verified to be equivalent to the initial exit condition of the transformed *for*-loop. For example, in Figure 4.1, the guard check prevents entering the loop if the lower bound is greater than the upper bound in line 13 in the IR. The rotated loop exits if the induction variable initialized with the lower bound incremented by one is greater than the upper bound in line 19. Since loop rotation examines the exit condition one iteration later than the original loop, transforming this exit condition to be used for a *for* loop will make it equivalent to the guard check. Therefore, the guard check in the motivating example can be safely removed by the Loop-Rotate Detransformer.

4.2.3 Variable Generation

This section describes how variables are generated by combining phi instructions, detecting and utilizing conflict-free debug information, and inlining parallel code re-

gions.

Variable Proposer

Variable generation starts by proposing to replace instructions with variables that reflect original code semantics. The incoming values of phi instructions are proposed to be combined and named with the phi instruction itself. The Metadata Interpreter leverages LLVM-IR metadata containing source variable debug information. The relationship between an IR and a source variable is contained in debug intrinsics encoded as LLVM metadata. As shown in Figure 4.5, %1 and %2 are both associated with the variable *var* through a debug intrinsic function containing metadata !30. While debug information can be invalidated as optimizations are applied, LLVM guarantees that debug information are correct throughout all the mid-level and backend passes [101], including *mem2reg*. Thus, the Metadata Interpreter can safely rely on debug information. A Metadata Extraction table is built with the debug information, as shown in Figure 4.5. Since a phi instruction may also be mapped to a source variable when the incoming values are combined for a phi instruction, they are mapped together to the associated source variable.

Variable Generator

While many instructions can be mapped to the same variable, mappings are invalid if instructions mapped to the same variable have a conflict. A conflict in variable naming occurs when a pair of instructions have overlapping lifetimes. For example, a conflict exists between %1 and %2 in Figure 4.5 since they map to the same variable *var*, and instruction *F* uses %1 after %2 is defined. Renaming them with the same variable results in incorrect execution, as %1 will wrongfully use the value of %2. Thus, the Conflicting Definition Detection module inside the Variable Generator is designed to remove such conflicting mappings. The module detects the most recent

variable definitions at every point in the program, as described in Algorithm 1. This is a forward data flow analysis in which a most recent variable to an instruction mapping is generated at this instruction if metadata containing a source variable is available, indicating that the most recent definition of the variable is the current instruction. Simultaneously, the old definition of the variable to which a new definition is generated is killed (i.e., their lifetime ends), as depicted in the Most Recent Variable Definition table in Figure 4.5.

Once the most up-to-date variable definitions are established at each point of the program, the module then uses it to remove conflicting definitions from the proposed instruction-to-variable map, as described in Algorithm 2. At each use of a proposed variable definition, the algorithm checks if the most up-to-date definition of the proposed variable is indeed the used definition. If not, a conflict is detected, and SPLENDID chooses to remove the most recent mapping to eliminate the conflict arbitrarily. For example, at instruction *F*, the definition at use, %1, is mapped to variable *var* according to the Metadata Extraction Table generated by the Variable Proposer. However, according to the Most Recent Variable Definition table, the most recent definition for *var* is %2 at instruction *F*. Thus, only the %1-to-*var* mapping is valid, and the %2-to-*var* mapping is removed. After Conflicting Definition Detection, the rest of the instruction-to-variable mappings are valid for generating variable names. For example, %3, which also maps to *var*, is not defined before any use of 1 or 2, so it can also be mapped to *var*.

The Variable Generator then generates declarations using the resulting mappings. At the definition or use of a variable, the Variable Generator returns the same value as its declaration by referring to the exact IR-variable mapping shown in Figure 4.5. As for variables without a mapping, such as %2, they are given the virtual register name as it is unique and somewhat meaningful (e.g., *indvar* tells the programmer this variable is an induction variable).

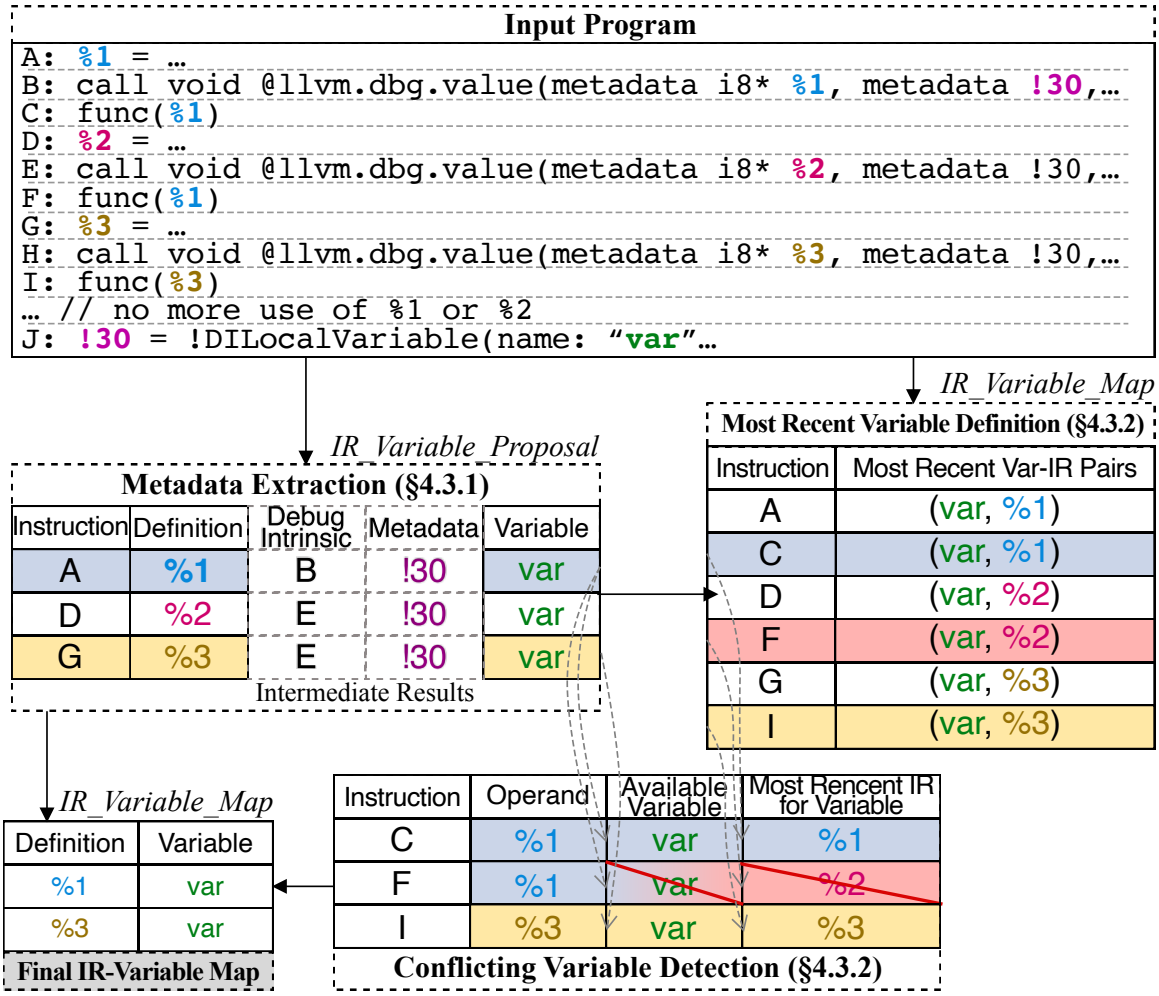


Figure 4.5: An example of how SPLENDID associates an IR to a source variable (Final IR-Variable Map) through the Metadata Interpreter and Conflicting Definition Detection.

Algorithm 1: *Most Recent Variable Definitions*

Input: IR_Variable_Proposal - Proposed instruction to variable mappings
Result: MR_Var_IR_Maps - Most recent variable definitions at each instruction

```
Var ← getVariable(I, IR_Variable_Proposal);
GEN[I] ← (Var, I);
Kill[I] ← (Var, I_old);
IN[I] ← ∪_{PI∈Pred(I)} OUT[PI];
OUT[I] ← GEN[I]∪(IN[I]-Kill[I]);
MR_Var_IR_Maps ← OUT;
return MR_Var_IR_Maps;
```

Algorithm 2: *Conflicting Definition Removal*

Input: IR_Variable_Proposal - proposed instruction to variable mappings
MR_Var_IR_Maps - Output of Algorithm 1
Result: IR_Variable_Map - validated mappings with conflicting definitions removed

```
for Instruction I in F do
    for Operand op in I.Operands() do
        var ← getVariableName(op);
        if op != MR_Variable_IR_Maps[I][var] then
            IR_Variable_Proposal.erase(pair(op, var)) ;
    end
end
IR_Variable_Map ← IR_Variable_Proposal;
return IR_Variable_Map;
```

Chapter 5

Evaluation

This chapter provides an empirical evaluation of the thesis, focusing on the effectiveness of enhancing code quality—specifically in terms of robustness and readability—through the proposed Tulip pipeline equipped with the natural decompilation technique, SPLENDID. TULIP and SPLENDID are based on the LLVM Compiler Infrastructure [101] (versions 9.0.0 and 10.0.1, respectively), with SPLENDID additionally utilizing the LLVM C Backend [29] for basic decompilation support of C syntax. The C backend closely approximates a one-to-one translation from IR instructions to C statements, translating IR branch instructions into C *goto* statements. All modules described in Chapters 3 and 4 were developed in-house for Tulip and SPLENDID.

The evaluation of Tulip and SPLENDID is structured as follows:

- Section 5.1 describes the experimental setup for this evaluation, including the selection of benchmarks, tools, and metrics used.
- Section 5.2 details the established pipelines enabled by this thesis and demonstrates the extensibility of the proposed Tulip transpilation, which achieves complex integrations with only a few lines of code.
- Section 5.3 presents a detailed analysis of the performance improvements in

code generated by Tulip, comparing it to native compilers and the best previous approaches.

- Lastly, Section 5.4 assesses the naturalness of the code using the BLEU score and lines of code as metrics. This section also highlights a code snippet from Tulip's code generation, emphasizing the reduction in explicit data movement achieved through SPLENDID decompilation and the recovery of variables by SPLENDID.

5.1 Experiment Setup

We performed 4 experiments to evaluate Tulip. First, we demonstrate the performance improvement of Tulip with all performance enablers in action against the best prior approaches. Second, we showcase Tulip's performance improvement when enhanced by NOELLE. Third, we show the performance numbers of OpenACC and Line of Code (LoC) for the components of Tulip. Lastly, we show a code snippet of Tulip-generated code to make a case for naturalness.

We performed 4 experiments to evaluate SPLENDID. First, we show that the SPLENDID-generated code is natural and enables collaborative parallelization. Second, naturalness is further depicted by portability. Namely, SPLENDID is recompilable using other compilers and libraries such as GCC and libgomp. Third, we provide BLEU score and LoC to measure code naturalness. Lastly, we show the percentage of variables reconstructed by SPLENDID.

5.1.1 Benchmarks

We evaluated Tulip against 20 Polybench [98] benchmarks handwritten in CUDA, with their OpenMP and Hip equivalents as baselines. CUDA, OpenMP and Hip implementation of the same benchmarks have no algorithm level changes. 10 bench-

marks are eliminated for they need support from *thrust* library. Since Tulip focuses on PPM transpilation itself, library function support is out of the scope of this paper. Cholesky failed in its Hip implementation and our baseline transpiler, Polygeist. Thus, we also excluded Cholesky [1] from most graphs for a fair comparison and simple calculation.

We measure the end-to-end runtime of each benchmark. Speedups are calculated by averaging 5 runs of each benchmark to reduce variance. For each benchmark, we verified the output for correctness by making sure the output of their sequential equivalent or within an acceptable range of small floating point precision errors due to optimizations

Similarly, SPLENDID is evaluated using 16 benchmarks from the PolyBench benchmark suite. Other benchmarks in PolyBench are excluded due to the lack of industrial-level robustness in CFG transformations implemented in SPLENDID. To generate parallel IR (i.e., the input to SPLENDID), a benchmark is first compiled to LLVM-IR, optimized with LLVM *-O2*, and parallelized using Polly [45]. Comparing original sequential code to parallel code generated from parallel LLVM-IR is counter-intuitive when evaluating code naturalness since even manually parallelized code should look different from the sequential code. Thus, we define code naturalness to be how close the decompiled parallel code is to a piece of semantically equivalent hand-written parallel code. To obtain reference code fair for comparison, OpenMP pragmas are manually added into the original sequential source code according to how Polly parallelizes them to simulate the most natural parallel code that a decompiler can generate using OpenMP without divergence in semantics.

[1] Reported here are the raw speedup numbers against sequential CPU run time.
clang.cpu: 15.73, gcc.cpu: 15.09, clang.cpu.noelle: 15.70, gcc.cpu.noelle: 15.12, icx.cpu: 16.67, icx.cpu.noelle: 16.52, nvhpc.nvidia: 173.67, nvhpc.nvidia.noelle: 174.72, clang.nvidia: 98.51, clang.nvidia.noelle: 99.30, clang.amd: 78.08, clang.amd.noelle: 78.33, aomp.amd: 76.39, aomp.amd.noelle: 76.58

5.1.2 Hardware Systems

We evaluated Tulip transpilation performance on mainstream GPU and CPU systems, including multicore shared memory, NVIDIA and AMD GPU. The CPU evaluation was done on a machine with two Intel Xeon E52697 v3 processors, each with 14 cores (28 cores total) and a total of 252 GB of memory. The operating system is 64-bit Ubuntu 20.04 LTS. The GPU evaluation for NVIDIA was done on a system with Intel Xeon Gold 6252 $\times$ 2 with 192 GB memory and an NVIDIA A100 (40 GB), and for AMD on a system with Intel Xeon Silver 4215 $\times$ 2 with 384 GB Memory and an AMD MI210, both of which use AlmaLinux 8.7 for their operating system.b Further specification for the GPUs are in Figure 5.5a (b).

For SPLENDID, performance of all programs are evaluated on a commodity shared-memory machine with two 14-core Intel Xeon CPU E5-2697 v3 processors (28 cores total) running at 2.60GHz (turbo-boost disabled) with 250GB of memory. The operating system is 64-bit Ubuntu 20.04 LTS with GCC 9.4.0.

5.1.3 Baselines

We use Polygeist [81, 53] (commit `11265cd8`) as a baseline, a state-of-the-art transpilation framework that targets the same hardware as Tulip. Polygeist uses a common parallel-optimization-friendly intermediate representation to retarget CUDA programs to run on either the originally intended NVIDIA GPU, AMD GPU, or CPU while applying target-specific optimizations. MCUDA as a potential baseline for source-to-source transpilation was eliminated because, as prior work [81] suggests, it handles the earliest release and only a subset of input semantics, and we were unable to compile Polybench with it. Since there is no direct CUDA-to-OpenMP or OpenACC source-to-source transpiler, we use Hipify to translate CUDA to Hip to compare performance on AMD GPUs. Since Hip is the official programming language for AMD GPUs and HipCC the native AMD compiler, Hipify is used as a baseline

comparison against Tulip's transpilation to OpenMP AMD Offloading.

For SPLENDID, Rellic, the state-of-the-art LLVM-to-C decompiler, and Ghidra, a widely used binary-to-C decompiler, are used as baselines. Rellic is the fairest comparison since its input is at the same level as SPLENDID, while the input of Ghidra is binary. Nevertheless, we found Ghidra to be a competitive baseline as it is an industrial standard.

5.1.4 Tools Selection

As described in Figure 2.1, Tulip enables using toolings that are otherwise incompatible with the source PPM, among which we selected the most widely used compilers for compilation, namely, Clang (v9.0.0), GCC (v9.4.0), and ICX (2024.0.2) for multicore systems, NVCC (native, v12.3), Clang (v18.1.2), and NVHPC (v24.3-0) for NVIDIA GPUs, and HipCC (v19.0.0), Clang (v18.1.2) and HipCC(native, v5.4) for AMD GPUs. Additionally, NVHPC is also used to compile Tulip transpiled OpenACC.

5.1.5 Metrics

Tulip focuses its evaluation on performance and extensibility with metrics commonly seen in literature such as speedup and LoC. For SPLENDID, the first metric used is speedup, which shows that SPLENDID-produced code is portable to other host compilers. Then, several naturalness metrics are used to evaluate the claim that since explicit parallelism is expressed through OpenMP directives in SPLENDID, it also brings naturalness on top of portability. First, naturalness is measured using the number of code lines (LoC) since eliminating low-level parallel implementation dramatically reduces LoC. Then, the percentage of variable names restored to the source is provided to show the effectiveness of the variable renaming of SPLENDID (Section Section 4.1.3).

Lastly, the BLEU score (BiLingual Evaluation Understudy) [95] is used to measure overall code naturalness. The BLEU score measures the similarity between a reference text to a set of manual translations of the same text. It correlates highly with the human-evaluated quality of natural-language translations [95], which also has been explored recently to evaluate programming languages, specifically in machine learning-based code migration (e.g., source-to-source compilers [61, 38, 60, 59, 2]). The use of the BLEU score for formal languages is well established in the literature. CodeXGLUE [72] project developed by Microsoft, for example, uses it to evaluate every source-to-source compiler infrastructure submitted for testing. As in this paper, others [106, 121] use BLEU as the gold standard. However, no other proposed metrics have been found to be practically better. For example, codeBLEU [106] by design is biased towards longer inputs because it can find more matches from the reference. Originally, BLEU score ranges from 0 to 1 with 1 the translated text being identical to a reference. To conform with other literature, this paper uses BLEU-4 score with the score also reported on a scale between 0 and 100. Appendix A.1 illustrates in detail how BLEU-4 score is calculated and its capability in measuring code naturalness. As already pointed out by Tran et al. (in [121]), BLEU does not enforce rigorous word ordering following the syntactic rules of a programming language and thus does not evaluate the correctness of code. Whether code emitted by SPLENDID is syntactically and semantically correct by construction is confirmed by showing a similar speedup to the parallelizing compiler.

5.2 Translation Pipelines

Table 5.1 presents the current transpilation pipelines supported by Tulip. C, OpenMP, and CUDA all utilize Clang as a common frontend. For CUDA, Clang generates NVPTX-related calls, address space, and PTX intrinsics, whereas for OpenMP, it

<table>
<tr><th>Existing Tulip Pipeline</th><th>Standard Fronend: Clang</th><th>Normalization (LoC)</th><th>Parallelizing Compiler (LoC)</th><th>AutoPar Interpreter (LoC)</th><th>C Decompiler: SPLENDID</th><th>Pragma Decompilation (LoC)</th></tr>
<tr><td>C/OpenMP→Polly→OpenMP</td><td rowspan="4">404k LoC</td><td rowspan="2">None (0)</td><td>Polly (98k)</td><td>polly-interpret (509) (192x)</td><td rowspan="4">12k LoC</td><td rowspan="2">omp-gen (223) (53x)</td></tr>
<tr><td>C/OpenMP→NOELLE→OpenMP</td><td rowspan="3">NOELLE (106k)</td><td rowspan="3">noelle-interpret (264) (401x)</td></tr>
<tr><td>CUDA→NOELLE→OpenMP</td><td rowspan="2">cuda-normalize (1k) (404x)</td></tr>
<tr><td>CUDA→NOELLE→OpenACC</td><td>acc-gen (228) (52x)</td></tr>
</table>

Table 5.1: LoC comparison of different components of experimented Tulip pipeline. Shaded in columns are existing tools that do not change across PPMs. The numbers in parentheses are the LoC numbers.

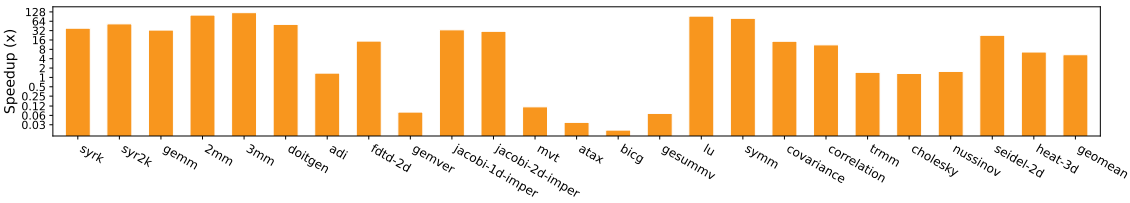


Figure 5.1: Performance of Tulip generated OpenACC source code.

produces KMPC-related calls. NVPTX calls, without normalization, are not executable on CPUs. However, no normalization is required for OpenMP since the backend, SPLENDID, includes KMPC decompilation. Consequently, no additional lines of code (0 LoC) are needed for normalization in the compilation of C or OpenMP, as pragmas are ignored during the compilation process. Remarkably, Tulip achieves normalized CPU-targeted IR from various input PPM within 1,000 lines of code, which is 400 times fewer than required by the complex Clang compiler.

We experimented with integrating two parallelizing compilers into the proposed transpilation pipeline: Polly [45] and NOELLE, each equipped with its own analysis-based interpreter at the IR level. Both compilers are complex and feature extensive code analysis capabilities. Specifically, NOELLE utilizes numerous profilers and analysis frameworks and creates advanced abstractions on top of LLVM-IR, such as program dependence graphs, loop forests, and environments. However, the interpreter developed within Tulip is 400 times simpler than these complex parallelizing compilers. In the case of Polly, which was developed before the introduction of Tulip metadata, significant effort is devoted to decompiling KMPC runtime function calls. This task could be transferred to the Tulip frontend as a normalization pass for OpenMP. Despite the challenges of decompiling runtime functions instead of interpreting Tulip-generated metadata, the integration of Polly was achieved with approximately 500 lines of code, making it 192 times simpler than the Polly parallelizer itself.

In addition to the OpenMP backend, we have implemented emitting OpenACC directives to demonstrate the extensibility of the Tulip backend. OpenACC [94], like

OpenMP, is a parallel extension for C/C++ but is prescriptive in nature and thus gives the compiler a higher degree of freedom for optimization. The generation of both OpenMP and OpenACC pragmas is distinct from the natural C code generation process. While the decompilation of 12k lines of C code is substantial, the generation of parallel extensions for OpenMP and OpenACC involves only about 200 lines of code each, primarily consisting of a direct mapping from Tulip metadata to pragmas. Figure 5.1 illustrates a significant performance improvement, showing a geometric mean speedup of 5.36x for Tulip-generated OpenACC code, validating its correctness and efficiency within the current transpilation framework. The generated code was compiled using the NVHPC OpenACC compiler, the primary support of OpenACC, to ensure a direct translation.

5.3 Migrated Code Performance

Figure 5.2 shows the performance of all the transpilation pipelines enabled by Tulip and Figure 5.3 shows inside of Tulip the performance differences with and without NOELLE. We produced three plots for the three targets we evaluated: multicore CPU, NVIDIA GPU, and AMD GPU. When the best configuration is picked, Tulip outperforms the original programming model, native compilers, and best prior work. This is largely due to its flexibility with tools, automatic parallelization, and programmer interaction. With all performance enablers described in Section 3.1 in action, against Polygeist, TULIP achieves a geomean speedup of 2.93x, 37%, and 11% on multicore, NVIDIA, and AMD systems, respectively. Moreover, compared to native compilation, TULIP achieves 85%, 14%, and 12% against handwritten OpenMP, CUDA, and Hip, respectively.

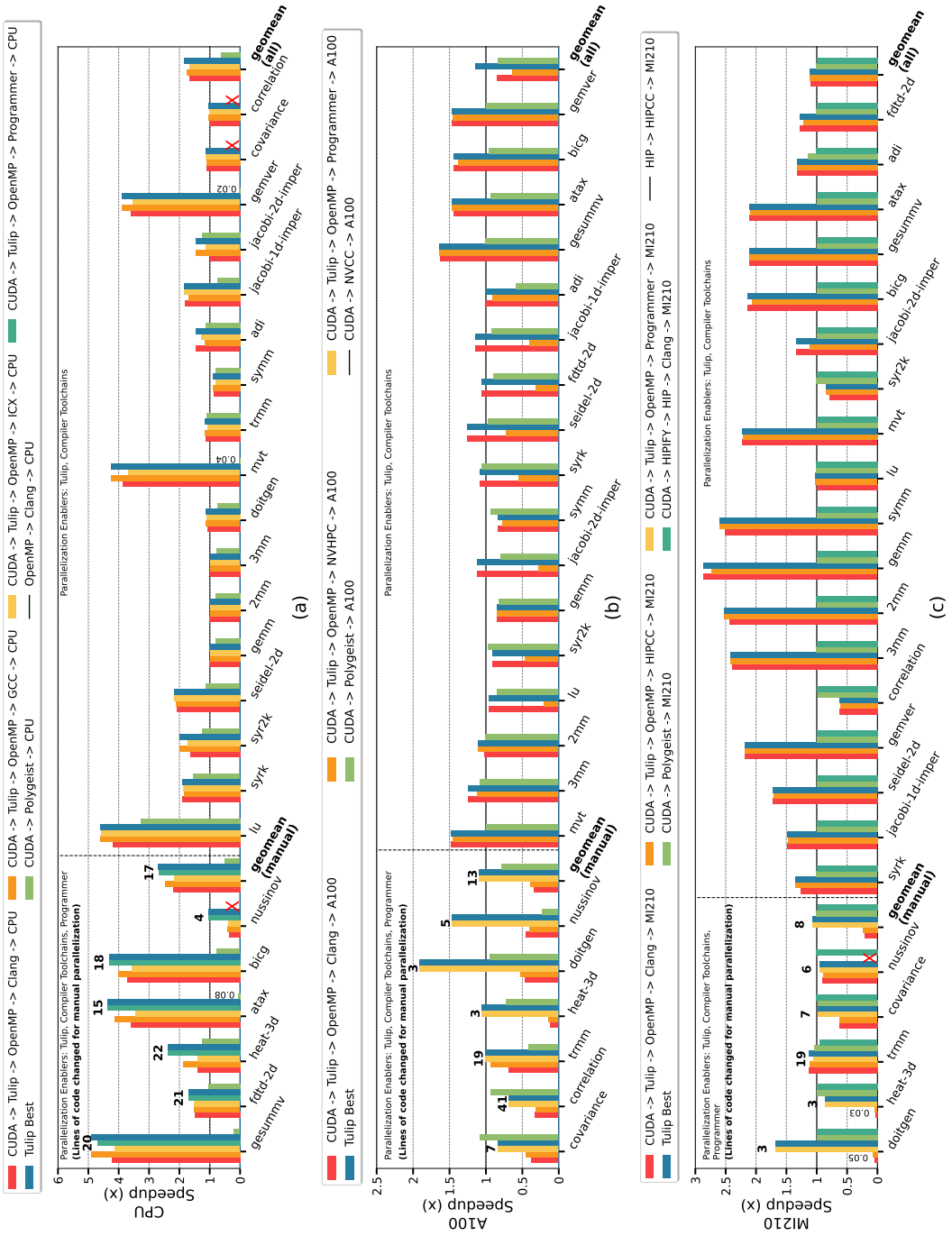


Figure 5.2: Performance enhancements comparing Tulip's transpilation to the best previous methods, including Polygeist and Hipify, relative to native compilation. The '-O3' optimization level is consistently applied across all compilers. Each pipeline's performance is evaluated both with and without the NOELLE framework, and the superior configuration is selected for comparison. 'Tulip Best' represents the optimal result for each benchmark, utilizing the most effective tools and, where applicable, manual performance optimizations at the source level.

5.3.1 Freedom of Choosing the Tools

We first show the performance difference of various tooling. In (a), GCC almost always outperforms other compilers due to its robust in-house optimization. GCC, however, is of no use in the case of direct transpilation such as Polygeist, as shown in Figure 2.1. In (c), For some benchmarks like *gemm* or *bicg*, Clang-compiled code performs better, and for others like *2mm* or *syr2k*, HipCC-compiled code runs faster. Since Tulip provides the freedom to choose tools tailoring performance for each benchmark, the *Tulip Best* speedup consists of the maximum speedup out of all tools for each benchmark.

Automatic Parallelization

Figure 5.3 shows across all platforms, whenever a benchmark noticeably performs better than the native compiler is when NOELLE is in the pipeline. On both NVIDIA and AMD GPUs, many benchmarks perform at best as the native compiler, as shown as vertically aligned at 1x along the x-axis. With NOELLE, many benchmarks perform significantly better, up to 5x, than the speedup gained without NOELLE on the CPU.

5.3.2 Programmer Interactivity

For 6 benchmarks that exhibited suboptimal performance on each system, as shown on the left side of Figure 5.2, we manually parallelized them because Tulip generates naturally comprehensible source code, making it easier for programmers to modify, as detailed in Section 5.4. On average, all manual modifications involved no more than 20 lines of code. Most benchmarks benefited from straightforward adjustments, such as altering loop scheduling, chunking policies and imposition of thread limits in GPU-targeted code. Additional enhancements included optimizing data transfer pragmas to improve efficiency beyond the conservative original translations (e.g.,

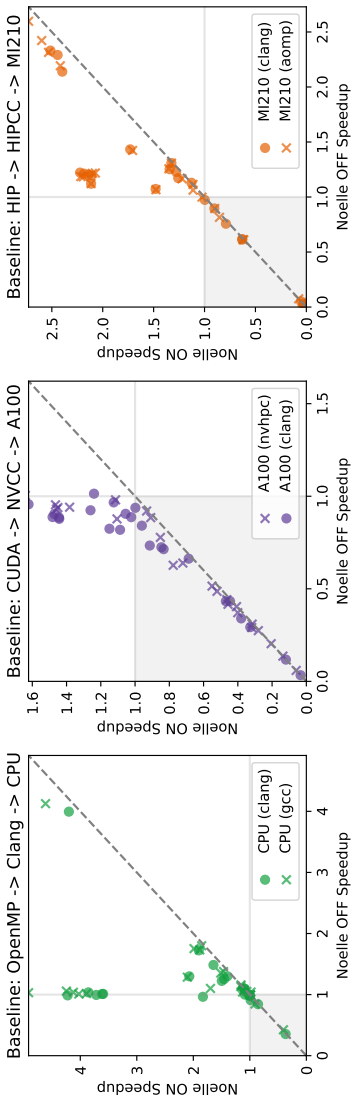


Figure 5.3: Effect of NOELLE on performance with the native programming model and compiler for each of the NVIDIA, AMD, and Multicore CPU platforms.

changing 'tofrom' to 'from' for result arrays initialized within the kernel). However, implementing these adjustments on the compiler side is challenging, as no single set of optimizations yields optimal performance across all benchmarks and target platforms.

In cases like correlation, where more modification is required, Tulip's static approach to ordering nested loops sometimes causes unwanted memory access patterns. This involves manually changing the tiling effect created by CUDA's execution model (i.e., collapsing loops along each grid and block dimension or changing the tiling size). Doing so also eliminates the expensive calculation of the index from the block and thread IDs that involve multiplication. Creating a heuristic to automatically explore the impact of different placements and scheduling strategies on performance optimization is an interesting future work direction[2].

```
dim3 block{threadsPerBlock, 32, 1};
dim3 grid{blocksPerGrid, 8, 1};
kernel<<<grid, block>>>(...);
```

(a) CUDA

```
#pragma omp parallel for collapse(2)
for(int i = 0; i < blockPerGrid;   i = i + 1)
for(int j = 0; j < threadsPerBlock;   j = j + 1)
      //loop tiling
      for(int k = 0; k < 8;   k = k + 1)
      for(int  l = 0; l < 32;   l = l + 1){
            m = 8 * i + k;
            n = 32 * j + l;
            //kernel
      }
```

(b) Tulip Generated OpenMP

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        //kernel
```

(c) Handwritten OpenMP

Figure 5.4: Simplified loop nests from *gemver*.

Loop Tiling Most Polybench benchmarks see performance improvements from loop tiling, as indicated in Figure 5.4, which shows an overall speedup in the majority of benchmarks running on CPUs (Figure 5.2 (a)). Due to the CUDA programming model, which aligns closely with the hierarchically parallel structure of GPUs, programmers must define the grid and block dimensions. These dimensions are analogous to loop nests in CPUs and often follow widely accepted best practices for optimal performance. Typically, the block dimension should be multiples of 32, while the grid size should correspond to the size of the processed elements. This adherence to 'golden rules' for setting block and grid sizes is not primarily about optimizing

[2]Insights credited to Yebin Chon and all other coauthors of Tulip.

code but rather about following established guidelines. However, when this structure is translated to CPU code, it results in loop tiling, a technique critical for enhancing computational performance through better cache locality. Thus, loop tiling is a performance optimization naturally gained by transpilation.

5.3.3 Better Speedups on AMD than on NVIDIA

Figure 5.5a demonstrates that Tulip Best achieves a better geometric mean speedup on AMD MI210 compared to native CUDA speedup on NVIDIA A100, while native HIP compilation underperforms against the same NVIDIA baseline. One contributing factor is that when Hipify translates CUDA code to HIP, it maintains the original computation schedule—meaning the same number of blocks and threads are specified, and the same workload is assigned to each thread for AMD as for NVIDIA. However, hardware differences between NVIDIA and AMD GPUs, such as the number of threads per warp (32 for CUDA, 64 for AMD), suggest that the CUDA scheduling may not be optimal for AMD GPUs.

When we transpile CUDA code to OpenMP, we remove the explicit scheduling (block and thread numbers) defined in the original code. This adjustment allows the OpenMP runtime to employ compiler heuristics to optimize scheduling for blocks and threads, potentially assigning a different number of work items per thread, unlike the fixed one work item per thread in the CUDA model. This indicates that AMD's OpenMP compilers are better optimized for such dynamic scheduling decisions than tools such as Hipify which does a one-to-one translation from CUDA to Hip. By default, the Clang OpenMP runtime for AMD launches as many blocks and threads as needed to fully utilize the parallelism available on the target device then assigns multiple work items per thread both statically and serially. Tulip capitalizes on the advanced scheduling decisions integrated into AMD's OpenMP implementation. In contrast, NVIDIA has focused more on enhancing its CUDA programming

model compiler (nvcc), which accounts for the observed performance disparities. As illustrated in Figure 5.5b, even though the A100 and MI210 GPUs are from roughly the same generation and have similar capabilities, matching performance on AMD hardware would likely not be achieved without the programming model changes and optimizations provided by Tulip[3].

5.4 Naturalness

Generated code must be natural to allow meaningful programmer engagement. Section 5.4.1 shows that the Tulip pipeline generates natural code through SPLENDID by comparing the BLEU score and LoC with the best of prior work, and an example of the generated code. Section 5.4.3 shows how, by making the code portable, it is also made natural. Section 5.4.4 shows the effectiveness of the variable renaming technique introduced by SPLENDID.

5.4.1 Naturalness Overview

Table 5.2 shows that, by eliminating low-level run-time specific code, parallel representation in SPLENDID-produced code uses less than 13 lines of OpenMP pragmas, including brackets, at least 35x less than naively decompiling parallel execution setup instructions to the source level. The LoC produced by SPLENDID is within 18 LoC difference from the reference code for every benchmark, almost identical to LoC in total with only 0.1x difference, 45x less than the better of the baselines.

The BLEU scores presented in Figure 5.7 evaluate the overall code naturalness as described in Section 5.1.5. Using the generated code as the translation under evaluation and the reference code as an instance of natural translation, the BLEU score indicates how close the parallel translation is to manual translation.

[3]Insights credited to Ivan R. Ivanov and all other coauthors of Tulip.

Table 5.2: Comparison of LoC similarity to reference code. Programs decompiled by SPLENDID contain LoC highly similar to the reference code.

Benchmark	LoC				Parallel Representation (LoC)		
	Ghidra	Rellic	SPLENDID	Ref	Ghidra	Rellic	SPLENDID
2mm	534 (8.1x)	381 (5.8x)	74 (1.1x)	66	343	171	4
3mm	813 (9.0x)	624 (6.9x)	105 (1.2x)	90	620	425	12
adi	311 (5.0x)	371 (6.0x)	70 (1.1x)	62	129	122	4
atax	155 (3.7x)	173 (4.1x)	41 (1.0x)	42	46	49	2
bicg	154 (3.0x)	202 (4.0x)	52 (1.0x)	51	53	52	2
doitgen	442 (9.6x)	307 (6.7x)	58 (1.3x)	46	296	123	2
fdtd-2d	405 (6.8x)	322 (5.4x)	67 (1.1x)	60	132	118	4
floyd-warshall	153 (4.8x)	150 (4.7x)	33 (1.0x)	32	48	49	2
gemm	455 (7.7x)	373 (6.3x)	63 (1.1x)	59	362	262	8
gemver	433 (5.8x)	410 (5.5x)	81 (1.1x)	75	295	275	4
gesummv	130 (2.6x)	155 (3.1x)	41 (0.8x)	50	57	59	2
jacobi-1d-imper	153 (3.8x)	217 (5.4x)	58 (1.4x)	40	70	92	4
jacobi-2d-imper	460 (10.7x)	276 (6.4x)	53 (1.2x)	43	361	142	4
mvt	229 (4.5x)	258 (5.1x)	54 (1.1x)	51	166	185	6
syr2k	458 (7.6x)	379 (6.3x)	62 (1.0x)	60	369	278	8
syrk	400 (7.5x)	357 (6.7x)	59 (1.1x)	53	324	261	8
Total	5685 (6.5x)	4955 (5.6x)	971 (1.1x)	880	3671	2663	76

We create two variants of SPLENDID to quantify explicit parallelism for naturalness. The first variant, SPLENDID v1, only enables the natural control-flow construction, which contains basic CFG analysis and the novel Loop-Rotation Detransformer proposed in Section 4.2.2. On top of natural control-flow construction, SPLENDID v2 enables explicit parallelism translation, representing parallel code regions using inlined sequential loops applied with OpenMP pragmas. Thus, SPLENDID v2 produces code recompilable with any host compiler. All counted loops are generated as *for* loops using SPLENDID v1, yielding an average BLEU score of 1.4, 3.4x higher than the best prior approach in terms of BLEU score, Ghidra. The improvement is not significant because BLEU score focuses on word matching and any improvement in control flow only results in a few keyword differences. SPLENDID v2, however, achieves a much higher BLEU score, indicating that what comes with portability is the massive benefit of naturalness. Code produced by portable SPLENDID scores 21x higher than Ghidra and 43x higher than Rellic due to removing a considerable amount of parallel execution setup code unrelated to original code semantics.

Lastly, Figure 5.6 shows the actual transpiled code compared to the original CUDA source program for the motivating example discussed in Section 3.1.1 with limited manual changes of parenthesis spacing. All the data movement from and to the device in the CUDA source program is replaced by OpenMP data mapping, as highlighted in grey. In addition to the natural pragma generation, OpenMP pragmas are, by nature, a minimal extension to sequential CPU code, more readable than reading code with two devices in mind, as in the case with CUDA. The actual kernel code, highlighted in blue, is almost identical on both sides by fully leveraging naturalness-enhancing detransformations such as variable renaming and loop detransformation mentioned in SPLENDID. Note that though for simplicity NOELLE parallelization is left out in this direct transpilation, we see only additional pragmas being added for additional parallelism discovered by parallelizing compilers, as parallelism applies not directly

to the IR.

5.4.2 Naturalness by Effective Interaction

Portability automatically frees the programmer from parallelizing what Polly can parallelize. As shown in Table 4.2, among the loops parallelizable by Polly, 60% of what a compiler parallelizes is what the programmer can also parallelize but is freed from doing so. An additional 40

By adding or modifying on average 13 lines of code in the Tulip generated code, a geomean of 1.08-4.43x speedup over the unmodified version is achieved across 5-6 benchmarks for each platform, 1.06-5.37x over the transpiling down to many targets approach Polygeist took. It took in total roughly 8 hours for an engineer with 3 years of experience in OpenMP to manually optimize the 15 benchmarks evaluated in Figure 5.2, some for multicore and some for GPUs. The actual time taken to understand the code is negligible. Most time was spent on performance debugging, specifically fine-tuning the scheduling policy and the chunking size since GPU workloads per thread can be tiny compared to the overhead of thread spawning on multicores.

A similar engineering effort was demanded for SPLENDID-enabled parallelization. We found 7 benchmarks among the 16 simple and highly parallelizable PolyBench benchmarks where, surprisingly, neither the programmer nor the compiler achieved the best result, as shown in Figure 5.9. The manually parallelized PolyBench benchmarks were found on Github by the Cavazos Lab [44]. SPLENDID restores all counted affine *for*-loops, enabling simple manual parallelization on top of SPLENDID-generated parallel code. By simply applying loop distribution (in the case of bicg and atax) and DOALL parallelism to loops that Polly does not parallelize, the speedup is doubled compared to both the compiler and programmer parallelization on its own. This collaborative parallelization is only made practical with SPLENDID.

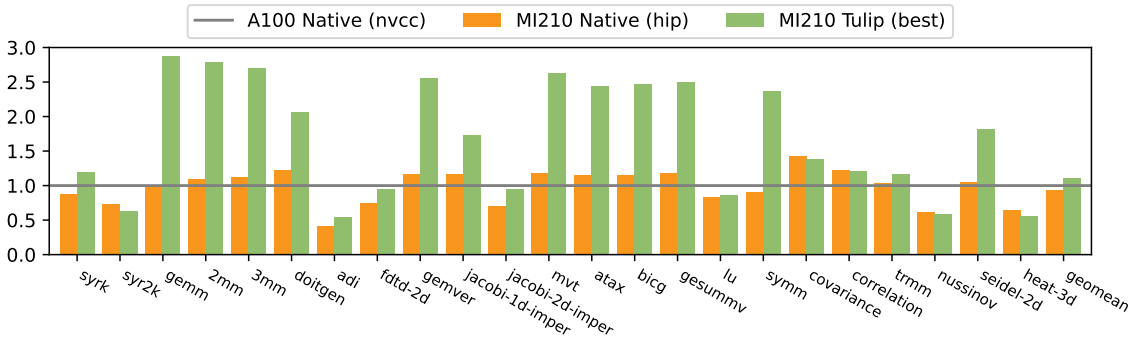
5.4.3 Portability

One way to ensure that code is natural is that it is portable. Namely, the generated code can be recompiled with any library, compiler, or target platform. Figure 5.2 effectively shows the Tulip-generated code enables source-level toolings of the target PPM that cannot be used by a direction transpilation approach. Not only is Tulip-generated code portable, but it is portable across platforms.

SPLENDID, likewise, practically reduces the involvement of programmers in parallelization by replacing original sequential source code with portable parallel source code. Since previous compilers, such as Ghidra and Rellic emit low-level runtime-specific code, the work of a parallelizing compiler, such as Polly, cannot be automatically made available at the source level. Figure 5.8 shows the result of comparing speedup obtained by Polly-generated binaries and SPLENDID-generated OpenMP code recompiled using Clang and GCC. All binaries were produced with the same optimization level (-O3) and were run 5 times on a near-idle machine. The result shows that SPLENDID-generated code produces identical speedup as Polly, indicating SPLENDID faithfully represents the complete work of Polly. A similar average speedup is obtained when recompiled using GCC and its standard runtime library for OpenMP, libgomp [43], indicating SPLENDID-generated code is compiler-independent. Through SPLENDID, the programmer is freed from parallelizing what a parallelizing compiler like Polly can parallelize, and an average of 11x speedup is made universally available outside of LLVM. The programmer can then choose a compiler with optimizations capable of achieving the best performance. For example, for benchmarks such as *mvt*, GCC produces a noticeable speedup over Clang on the decompiled code.

5.4.4 Variable Renaming

SPLENDID-generated code is much more readable because of intuitive variable names. In more detail, Figure 5.10 shows that, on average, 87.3% of variables are either reconstructed from metadata or inferred through inlining using source variables. Variables that are not reconstructed are because of the loss of source information even before parallelization during the optimization pipeline, such as loop invariant code motion promoting registers and hoisting memory accesses out of the loop. This code hoisting creates an intermediate instruction not associated with any source variable. Since neither Rellic nor Ghidra creates intuitive variable names related to original code semantics, no numbers are provided for prior work. With variable renaming enabled on top of control flow and parallel translation, SPLENDID-generated code achieves an average of 16.4 in BLEU score, 39x times higher than Ghidra and 82x times higher than Rellic.



(a) Performance comparison of native CUDA code on an NVIDIA GPU against two approaches of executing the same code on an AMD GPU.

GPU	NVIDIA A100	AMD MI210
Compute Capability	8.0	gfx90a
SMs	108	104
FLOPs (f64)	9.75T	22.60T
FLOPs (f32)	19.49T	22.60T
Memory Bandwidth	1555 GB/s	1638 GB/s
Global Memory	40 GB	64 GB
L2 Cache	40 MB	16 MB
L1 Cache (Per SM)	192 KB	16 KB

(b) Specification of similarly graded A100 and MI210.

Figure 5.5: Tulip-generated code outperforms NVIDIA on AMD.

```cuda
__global__ void mv(double *A, double *B, double *C, int m, int n){
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < m) {
    double dot = 0.0;
    for (int j = 0; j < n; j++)
      dot += A[i] * B[i*n + j];
    C[i] = dot;
  }
}

host(){
    double *dev_A;
    double *dev_B;
    double *dev_C;
    cudaMalloc(&dev_A, n*m*sizeof(double));
    cudaMalloc(&dev_B, m*n*sizeof(double));
    cudaMalloc(&dev_C, n*n*sizeof(double));
    cudaMemcpy(dev_A, A, n*m*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_B, B, m*n*sizeof(double), cudaMemcpyHostToDevice);

    mv<<<gridD, blockD>>>(dev_A, dev_B, dev_C, m, n);

    cudaMemcpy(C, dev_C, n*n*sizeof(double), cudaMemcpyDeviceToHost);

    cudaFree(dev_A);
    cudaFree(dev_B);
    cudaFree(dev_C);
}
```

(a) Original CUDA source

```openmp
void mv(double* A,  double* B, double* C, uint32_t m, uint32_t n,
  uint32_t gridDim_x, uint32_t gridDim_y,  uint32_t gridDim_z,
  uint32_t blockDim_x, uint32_t blockDim_y, uint32_t blockDim_z,
  uint32_t blockIdx_x, uint32_t blockIdx_y, uint32_t blockIdx_z,
  uint32_t threadIdx_x, uint32_t threadIdx_y, uint32_t threadIdx_z) {
    uint32_t i = blockDim_x * blockIdx_x + threadIdx_x;
    if (i < m) {
      double dot = 0;
      for(int j = 0; j < n;  j = j + 1){
        dot = dot + A[i] * B[j + i * n];
      }
        C[i] = dot;
    }
    return;
}
host(){


#pragma omp target data map(to: A[0:n * m], B[0:m * n]) map(from: C[0:n * n])
  {
  #pragma omp target teams distribute
  for(int32_t i = 0; i < gridD;  i = i + 1){

    #pragma omp parallel for
    for(int32_t j = 0; j < blockD;  j = j + 1){
      mv(((double*)A), ((double*)B), ((double*)C), m, n, gridD,
        1, 1, blockD, 1, 1, i, 0, 0, j, 0, 0);
    }
  }
  }
}
```

(b) Transpiled OpenMP source

Figure 5.6: CUDA source and Tulip generated OpenMP code (without NOELLE) for the motivating example in Figure 3.2.

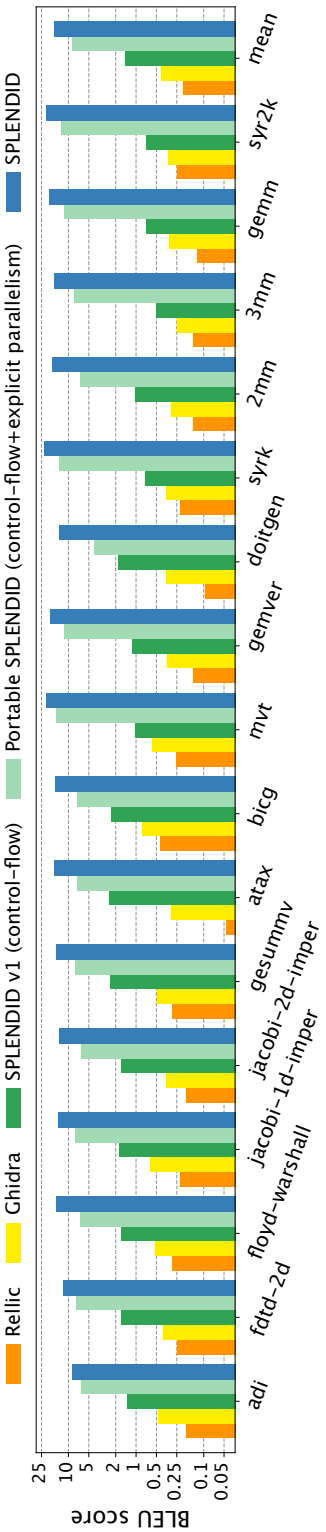


Figure 5.7: BLEU score comparison of code decompiled using Rellic, Ghidra, and SPLENDID (this work).

Polly	Polly→ Ghidra→ Clang	Polly→ Rellic→ Clang	Polly→ SPLENDID→ Clang	Polly→ Ghidra→ GCC	Polly→ Rellic→ GCC	Polly→ SPLENDID→ GCC
✓	✗	✗	✓	✗	✗	✓

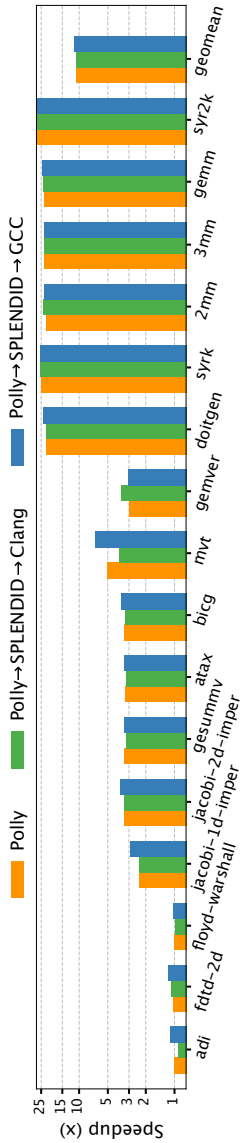


Figure 5.8: Performance of code decompiled from benchmarks automatically parallelized by Polly using SPLENDID. SPLENDID allows parallelization of Polly to be used by GCC. Polly achieves a geomean speedup of 10.7x on 28 cores. With SPLENDID, GCC also achieves a 11.3x geomean speedup.

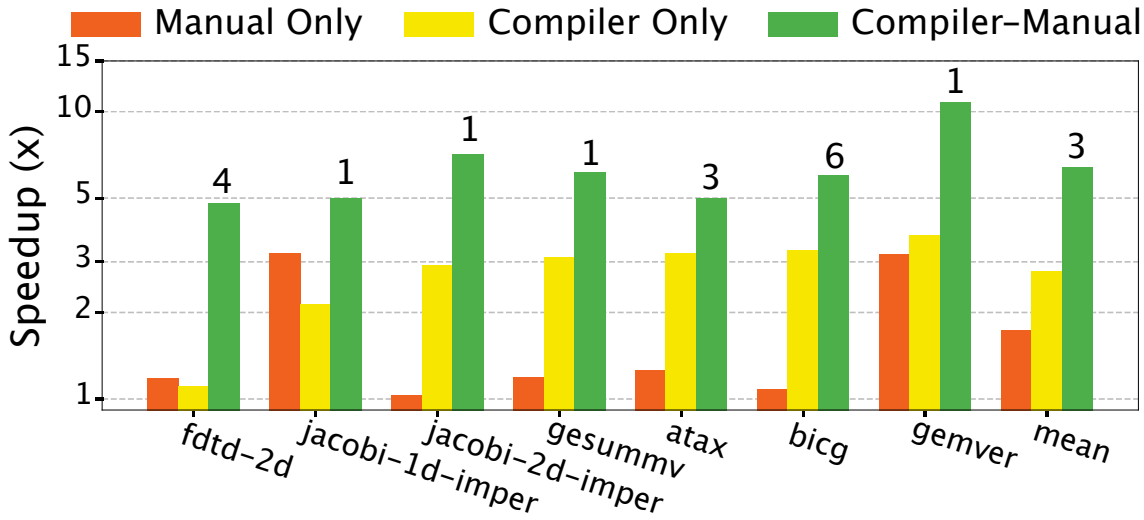


Figure 5.9: Performance of code with additional manual parallelization after decompiling Polly-parallelized IR using SPLENDID. The numbers represent LoC used to manually parallelize SPLENDID-generated code.

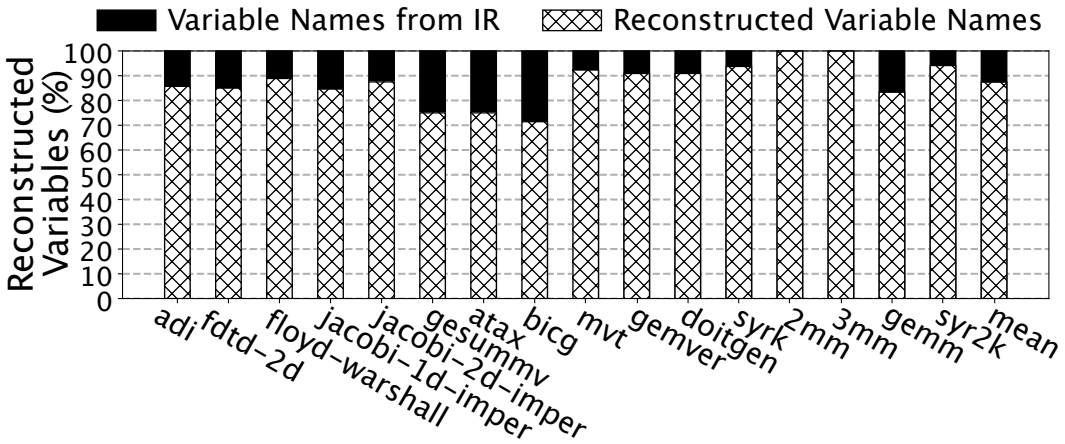


Figure 5.10: Percentage of variables whose names are reconstructed by SPLENDID.

Chapter 6

Related Work

This chapter explores various strands of research related to this thesis. Section 6.1 describes PPMs proposed in the past and their approach to performance engineering. Section 6.2 delves into historical approaches to source-level rewrites and source-to-source translation. Section 6.3 reviews established works on direct transpilation found within the scholarly literature. Section 6.4 examines studies on OpenMP offloading and their contributions to this field. Section 6.5 details significant breakthroughs in automatic parallelization that have improved the effectiveness of this work. Finally, Section 6.6 discusses decompilation frameworks that relate to the SPLENDID project.

6.1 Parallel Programming Models

Over the years, a variety of parallel programming models and libraries, both general-purpose and domain-specific, have been developed to leverage parallelism from hardware architectures. Extremely explicit parallel languages such as Fortran, CUDA, HIP, and SYCL [57] have been designed to provide precise control over parallelism. However, they require programmers to deeply engage with the parallelization process, which can be burdensome. In contrast, languages such as OpenMP, OpenACC, and OpenCL [58] use compiler directives to abstract parallelism, allowing programmers to

maintain a predominantly sequential approach to coding while managing parallel execution indirectly. Yet, this model falters when explicit synchronization is necessary, forcing programmers to contend with low-level hardware details once more. To reduce the complexities of performance engineering, libraries like RAJA [49], Thrust [90], and Kokkos [122] have been introduced. These libraries encapsulate the intricacies of parallelism, simplifying the developer's task of optimizing code across varied computing platforms. However, they may not always provide the appropriate level of abstraction for every application or deliver perfect optimization for each platform. In academic settings, several implicit parallel programming styles have also gained traction, affording compilers or runtime systems greater control over parallelism and further alleviate the burden of a programmer by eliminating the need for explicit programming directives. Notable examples include COMMSET [99], Sequoia, StreamIt [120], and Cilk [16], among others [75, 73, 31, 125, 22, 115]. While these models closely resemble sequential programming paradigms, they still demand thorough understanding to maximize performance. Despite the varying challenges and advantages of each parallel programming model (PPM), the necessity to port legacy code for future hardware platforms without manually rewriting large code bases underscores the importance of robust transpilation frameworks like Tulip.

6.2 Source Level Rewrite

Manual rewriting is one technique to facilitate code portability across different platforms. However, it is labor-intensive, error-prone, and heavily dependent on the programmer's expertise to optimize performance effectively [75, 73, 31, 125, 22]. Pioneer works in source-to-source transpilers were designed using sublanguages of the original source and target languages that comprise additional concepts that are not directly compatible with the source language but can be mapped into them, such

as the Ada-Pascal Converter [3]. Transpilation via sublanguages is limited in extensibility for each pair of source and target language; novel sublanguages need to be designed without reusability [82].

Source-to-source translation of modern languages relies largely on AST, which provides extensibility due to its language-agnostic representation and preserves naturalness, as source features can be implemented as separate AST nodes. AST-level transformations add minor language features(e.g., backward compatibility and type safety) within the same software toolchain, as often seen in commodity compilers such as the Closure Compiler [17], Babel [118], TSC [13], CoffeeScript [10], and Polyglot [91]. Stratego/XT [21], ROSE, and Haxe [41], for example, facilitate transpilation among C, C++, Java, and JavaScript. Others, such as SUIF [67] and Cetus [64], use source-to-source transformations to insert parallel constructs in source code. ROSE shows AST-based transpilation is easily extensible by including 9 languages and binaries as source and target languages. Instead of using an AST, the extensibility of Tulip comes from sharing LLVM frontend compilers (C, C++, Fortran, CUDA, etc) while preserving naturalness by detransforming canonicalization passes (e.g., loop rotation) and restoring variable names during decompilation. Tulip backend can also be easily extended to include C/C++ directive-based models such as OpenACC (as demonstrated), Cilk [15], OpenCL, etc. Moreover, Tulip targets PPMs across different levels of parallel abstraction, ranging from the highly explicit CUDA to more sequential models like OpenMP and OpenACC. Given that these models lack a one-to-one mapping, a simple AST-level rewrite would not be suitable.

6.3 Direct Transpilation

Much research focuses on transpiling a PPM to various target machines. In the realm of general-purpose computing, cross-compilation from a native machine (where the

translation is performed) to a target machine is well-established. For example, compiling ARM binaries from an x86 machine using GCC tailored for ARM architectures illustrates this process. However, unlike the transpilation discussed in this work, cross-compilation from x86 to ARM still leverages the generality of the sequential programming model.

The older CUDA toolkit includes a built-in emulator for CPUs, accessible through the '-deviceemu' option, which allows CUDA code to run serially on CPUs. Similarly, in academia, works [96] have explored emulating GPU executions on the host, albeit in a serial manner. These tools are primarily designed for debugging purposes, while Tulip transpilation is aimed at high-performance code migration. MCUDA [112], GPUOcelot [37], and Polygeist [81] retarget CUDA code for multicore systems, NVIDIA, and AMD platforms, respectively. Both MCUDA and Polygeist have investigated the use of, or extensions to, loop fission to manage synchronization, achieving high-performance CUDA transpilation at the AST or IR level. Specifically, Polygeist enhances performance by fusing parallel regions to reduce the costs associated with thread spawning and by coarsening tasks per thread. Rather than targeting multiple platforms directly, Tulip generates source code in a target PPM such as OpenMP or OpenACC, which not only facilitates interaction between programmers and compilers at the target PPM level but also generates platform-independent source code that can utilize the tools of other compiler frameworks.

Orthogonal to the proposed transpilation pipeline of this work, works in parallel semantics representation [55, 48, 109, 84] can further enhance the interaction between a source PPM and a parallelizing compiler. Tapir [109] extends traditional IRs by introducing new constructs specifically designed to represent parallel operations, such as task spawning, synchronization, and communication primitives. PS-PDG [48] augments nodes and edges in a conventional PDG to capture parallel semantics from a parallel source code, creating more parallel execution plans for the parallelizing

compiler to explore.

6.4 OpenMP Offloading

Similar to prior works that port CUDA to OpenCL [108], Tulip ports CUDA to the multi-platform language OpenMP, whose GPU targets rely on offloading [5, 78, 34, 111, 33]. While some directives are shared between OpenMP for host and GPU execution (e.g., *parallel*, *for*, *barrier*), others, such as *teams* and *distribute*, are more commonly used in offloading settings due to the hierarchical nature of GPUs. Offloading also involves directives unique to device code, such as explicit data movement to and from the device (e.g., `#pragma omp target map(to/from:...)`). More recent efforts, such as OMPX [50, 54, 35], explore OpenMP extensions that resemble CUDA-like explicit parallelism and have shown success in bypassing the complexities of the OpenMP runtime. These improvements have, in some cases, exceeded existing CUDA performance.

6.5 Automatic Parallelization

Automatic parallelization, being a potential player in source-to-source transpilation, can drastically increase program performance. Recent developments in parallelization, memory analysis [7, 113], and profiling frameworks [83] all exist at the IR level. Polyhedral-based parallelizing compilers [45, 126, 19] have demonstrated tremendous speedup on scientific workloads with affine loops, with Pluto and Polly operating at the LLVM-IR level, and Polygeist [80] at the MLIR level. Perspective [6] has shown scalable speedups on irregular general-purpose workloads by reducing the cost of speculative privatization. Alternatively, the parallelizers in NOELLE [76] overcomes dependences that hinder parallelization by integrating parallelization schemes of greater applicability, such as HELIX [24] and DSWP [105]. Parallelization greatly

enhances Tulip transpilation.

Prior work has shown success in combining the automatic parallelization of sequential code with the emission of parallel code. ROSE and AutoPar-Clava [8, 25], for example, can generate OpenMP pragmas for sequential C codes within an AST-level automatic parallelization framework. Due to limited alias analysis, automatic parallelization in ROSE often requires frequent programmer assertions to denote no alias conditions. AutoPar-Clava incorporates additional DSL knowledge in its analysis. SPLENDID integrates with Polly at the IR level and automatically parallelizes C code into OpenMP. Tulip applies state-of-the-art automatic parallelization not only to sequential but also to parallel source programs, allowing collaborative enhancement of original source-level parallelism with a parallelizing compiler.

6.6 Decompilation

Existing tools [99, 52, 27] provide insights into and suggestions from the compiler to the programmer. Intel Advisor, for example, informs the programmer of memory or computation bottlenecks and insights into whether and how to offline code to GPUs. Implicit programming tools, such as COMMSET, provide programmer dependences preventing parallelization. None of these suggestion-based tools have practically reduced the work of a programmer while enabling more parallelism like SPLENDID.

Many advancements in decompilation [46, 79, 130, 129, 110, 107] have greatly improved code naturalness by reducing the usage of *goto* statements. For example, eliminating irreducible graphs [79], diamond-shaped CFGs [46], and many more transformations significantly reduce the number of *goto* statements. However, with loop rotation, loops generated by previous decompilers are often *do-while* loops. For portability, SPLENDID instead de-transforms loop rotation to produce *for*-loops.

Existing LLVM-to-C decompilers [79, 130, 129, 29] produce code that is unnat-

ural. LLVM C Backend [29], the LLVM-to-C decompiler that SPLENDID is built upon, produces assembly-like code with most branches emitted as *goto* statements. Rellic shows no indication of using variable names representative of the code semantics. The C Backend emits source file names and line numbers in its decompiled code using *#line*, a debugging directive. Prior research [32, 65] in debugging has primarily focused on validating debug information instead of using it for variable renaming. SPLENDID, however, directly generates variable names using original source variables.

More work has been devoted to binary-to-C decompilers [40, 79, 110, 26, 46], some of which are integrated into IDEs as part of the debugger (e.g., Ghidra [1], Hex-Rays Decompiler [107], and Relyze [68]). An IDE often has a graphical interface that enables some level of interaction with programmers. Ghidra, for example, allows programmers to rename variables to assist in interpreting code semantics. Likewise, rellic-xref [92], a web interface for Rellic, allows programmers to selectively run some transforms in a user-defined order. However, the kind of interaction is not comparable to the interactive development for parallelization that SPLENDID enables.

Another line of work [60, 61, 38] adopts self-supervised methods in Natural Language Processing using deep learning models. Models are trained using obfuscated source code at a similar level of abstraction to improve code naturalness. Code produced in this approach cannot guarantee correctness and thus requires a manual inspection from programmers. More recent work [63] adapts interaction with GPT to improve variable names. SPLENDID and Tulip, however, produces code that is semantically correct, portable, and with speedup identical to the underlying automatically parallelized code.

Parallelizing compilers such as QuickStep [77] and Alter [123] insert OpenMP pragmas directly into the original input C code. QuickStep cannot preserve program semantics since it trades accuracy for more parallelism. SPLENDID preserves code

semantics in decompilation. Alter requires manual annotations for parallelization and is limited to its own analysis and transformations. SPLENDID, however, is a decompiler that does not target a specific parallelizing compiler. Thus, the performance of SPLENDID-generated code will not be limited by analysis within a single parallelizing compiler.

Source-to-source compilers [56, 87, 9] were designed for code migrations due to naturalness preserved from not lowering to assembly-like IRs. Thus, source-to-source compilers are limited to transformations that do not go beyond the AST level. Some [97, 18, 126] use polyhedral transformations to parallelize code with polyhedral loops. However, parallelization near the source level is not scalable with front-end languages, breaking the ideal source-target independent IR model. Moreover, unlike LLVM IR, rarely is there a community interest in the continued development and maintenance of source-to-source compilers. SPLENDID is easily scalable to other front-end languages as it targets LLVM IR. For the same reason, SPLENDID can be easily supported and maintained within the LLVM community. Furthermore, SPLENDID produces code that is natural and easy for manual code investigation.

Chapter 7

Conclusion

7.1 Conclusion

In response to the challenges posed by increasing specialization in hardware and the risk of legacy code becoming unrunnable due to future hardware and software fragmentation, this thesis presents a transpilation pipeline that fully leverages advanced compiler optimizations, programmer interactivity, and source-level tooling.

We present Tulip, a CUDA-to-OpenMP transpilation framework that can target both CPUs and GPUs (through OpenMP offloading). Unlike previous approaches that use AST for close-to-source transpilation, our new pipeline employs IR, enabling deeper compiler analysis for automatic parallelization and various optimizations. Instead of direct transpilation across platforms, Tulip emits source code that can be compiled with different source-level toolchains to deliver peak performance tailored to each application. Furthermore, since Tulip generates natural code, programmers can engage further to optimize the code. Programmer engagement is particularly valuable when PPM changes occur, as new PPMs may contain different kinds of parallelism than what the source PPM expresses. Additionally, Tulip's components are reusable and require only lightweight communication through Tulip's API to utilize

established front ends and automatic parallelizers. Lastly, Tulip generates natural source code that enhances programmer intractability. These features allow it to outperform native CUDA compilation by 14%, achieve a 1.1x to 2.93x improvement over Polygeist—the best source-to-many-machine approach—and surpass Hipify, the leading prior source-to-source approach, by 12%.

In addition, we present SPLENDID, an OpenMP/C decompiler that produces portable and natural code. SPLENDID's naturalness is due, in part, to a novel technique that materializes variable names inferred from the original source code. SPLENDID-produced code achieves a 39x higher average BLEU score than the best prior approach. A decompiler that produces natural parallelized code can enable a more efficient collaborative parallelization effort between the compiler and the programmer. This work has shown that SPLENDID makes the work of the parallelizing compiler more available to the programmer and frees the programmer from work that can be done automatically. For 7 simple and easily parallelizable programs, in a collaboration enabled by SPLENDID, the compiler and programmer produce code that runs twice as fast as either the compiler or programmer working alone.

7.2 Future Work

The primary focus of this thesis has been on enhancing CUDA-to-OpenMP transpilation, which has notably improved code migration and enabled programs designed for NVIDIA GPUs to run on various platforms. Building on this success, future research will explore incorporating advanced parallelization schemes like PS-DSWP and HELIX into established models, to better handle irregular workloads. Additionally, efforts will focus on enhancing natural decompilation processes for more intuitive code generation. Importantly, we will also look beyond the explicit parallelism of CUDA to investigate whether CUDA, as a programming model, offers additional

value for parallelization. These inquiries are aimed at expanding the reach and efficacy of source-to-source transpilation and automatic parallelization, setting the stage for significant advancements in parallel computing.

7.2.1 Source Representation of Advanced Parallelization Schemes

By generating natural source code, the compiler presents its parallelization to programmers. With great readability, programmers can improve parallelization by modifying the proposed source-level representation, which the compiler can then interpret to enhance the parallelization plan. Many state-of-the-art compilers have explored parallelization schemes beyond basic DOALL parallelism with strategies such as privatization and reduction. Among these, DSWP [105] and HELIX [24] have demonstrated greater applicability than DOALL. Although these schemes have been successful in speeding up irregular workloads, they are known to be challenging to implement in practice. For instance, the original work on DSWP has shown that solving the DSWP thread partitioning problem is NP-hard. One way to achieve better performance with DSWP and HELIX is through continual enhancements to the compiler, which will result in better analysis, thereby reducing dependencies, and improved heuristics, leading to more balanced thread partitioning. Another approach is leveraging the programmer's expertise to influence compiler decisions at the source level. The following sections first describe each parallelization scheme and their commonalities. Then, Section 7.2.1 proposes a source representation common to both DSWP and HELIX.

HELIX

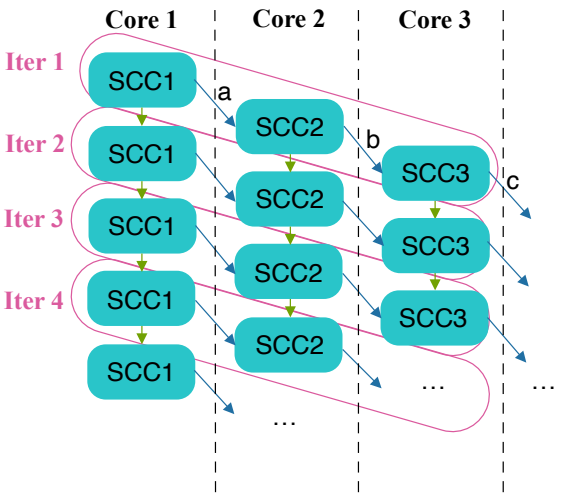
Unlike DOALL, which requires all loop-carried dependences to be disproven, HELIX uses a DOACROSS scheme that leverages inter-core communication to manage these dependences effectively. This approach allows for the overlapping of loop iterations

Source Representaion

```
#pragma ipm dswp
{
  for (auto i = 0; i < N; ++i){

    #pragma ipm stage outs(a)
    a = SCC1(a);
    #pragma ipm stage ins(a) outs(b)
    b = SCC2(a, b);
    #pragma ipm stage ins(b) outs(c)
    c = SCC3(b, c);
    //…
    #pragma ipm stage ins(g) outs(h)
    h = SCC8(g, h);
  }
}
```

Execution




```
#pragma ipm helix
{
  for (auto i = 0; i < N; ++i){

    #pragma ipm stage ins(a) outs(a)
    a = SCC1(a);
    #pragma ipm stage ins(b) outs(b)
    b = SCC2(a, b);
    #pragma ipm stage ins(c) outs(c)
    c = SCC3(b, c);
    //…
    #pragma ipm stage ins(h)
    h = SCC8(g, h);
  }
}
```

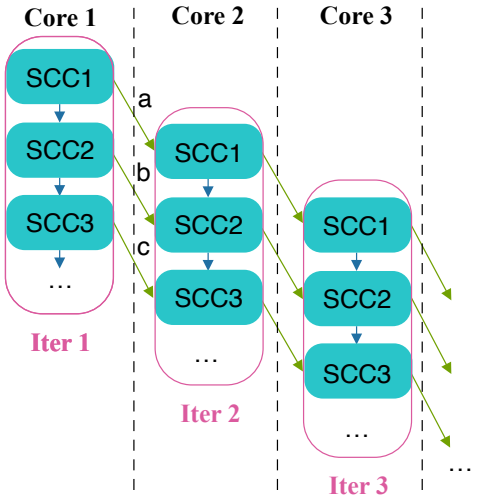


Figure 7.1: Source Representation for DSWP (top) and HELIX (bottom) parallelism based on strongly connected components (SCCs).

across multiple cores, enhancing parallelism once the initial communication setup is completed, as demonstrated in Figure 7.1.

In HELIX, a Program Dependence Graph (PDG) is used to identify Strongly Connected Components (SCCs), which represent tightly coupled operations within the loop. HELIX does not schedule SCCs of the same iteration across cores, but the inputs and outputs of SCCs need to be identified for inter-core communication. This organization necessitates structured inter-core communication to handle loop-carried dependencies appropriately, enabling parallel execution of loop iterations across cores while respecting the dependences delineated by the SCCs in the PDG.

Pipelined Parallelism

SCCs, defined by mutual dependences among their operations, can be grouped and allocated across cores. Unlike the HELIX model, which confines the execution of a loop body within a single thread to maintain thread-locality, advanced pipelined parallelism strategies like DSWP, PS-DSWP, and Pipette distribute each loop iteration across multiple cores to optimize concurrency. Specifically, DSWP is designed to retain all loop-carried dependencies within the same thread, thereby minimizing cross-core communication; however, it necessitates the management of intra-loop dependencies between cores, as demonstrated in the upper two graphs of Figure 7.1. Over the years, numerous pipelined parallelization strategies have emerged, each characterized by distinct heuristics and optimizations, as discussed by Nguyen et al.[85]. These enhancements include the integration of speculative execution techniques in speculative DSWP to reduce dependences speculatively[124], as well as the application of the DOALL strategy to stages devoid of loop-carried dependencies, further exemplified by PS-DSWP [105]. These advancements underscore the ongoing innovation in exploiting parallel execution frameworks to achieve optimal performance on multicore systems.

Motivation for an Abstraction across Parallelization Schemes

In the foundational work on Decoupled Software Pipelining (DSWP), the thread partitioning challenge is identified as NP-hard. This challenge involves strategically grouping Strongly Connected Components (SCCs) into balanced threads while minimizing communication to maximize performance. Despite the theoretical advancements, the practical adoption of these complex parallelization schemes, beyond simpler models like DOALL, remains limited in industrial-grade compilers. This hesitance reflects the significant difficulties involved in achieving profitable implementations.

Historical benchmarks, as discussed in prior research [76], indicate that the effectiveness of parallelization strategies may vary significantly depending on the nature of loop dependences. Programs with fewer loop-carried dependencies may achieve better performance enhancements from a HELIX-like schedule, which tends to reduce inter-thread communication. In contrast, those with more extensive loop-carried dependencies might benefit from pipelined parallelism approaches that place the communication of intra-loop dependencies along the critical path, thereby optimizing execution times.

To address these variations effectively, a unified framework that accommodates all parallelization schemes could be instrumental. Such a framework allows for the involvement of programmers, who can apply their domain-specific knowledge to influence thread partitioning decisions, thus narrowing the search space and enhancing profitability. Additionally, it provides a robust abstraction layer for in-depth analysis and comparison of different parallelization techniques. This flexibility could pave the way for advanced automatic parallelization methods that are adaptable across various programming contexts.

In this unified model, all parallelization strategies are conceptualized as SCC scheduling problems. The range of these strategies spans from HELIX and DOALL, which confine SCCs within the same iteration to a single core, to pipelined paral-

lelism, which distributes SCCs across different iterations but within the same core. As depicted on the left side of Figure 7.1, this framework uses directives that specify how SCCs, along with their inputs and outputs, should be managed across different parallelization schemes. In this proposed model, HELIX and DSWP share the same stages—each comprising individual SCCs—but differ in their management of inputs and outputs. By extending this model to include attributes that define whether a stage applies the DOALL approach, we can describe more complex configurations like PS-DSWP, which combines DSWP with DOALL strategies in certain code segments. This versatile scheduling could potentially offer superior performance by optimizing various portions of the loop body with different parallelization tactics, thus outperforming traditional methods.

7.2.2 Effect of CUDA Programming Across Platforms

Figure 5.3 demonstrates that in some cases, the Tulip-transpiled code outperforms native compilation across various platforms, such as Clang for OpenMP, NVCC for CUDA, and HipCC for HIP, even when automatic parallelization is disabled. Further analysis revealed, as depicted in Figure 5.4, that tiling—an optimization that improves cache locality and increases workload per thread—is automatically applied when transpiling from CUDA to OpenMP. This observation raises an intriguing question: beyond explicit parallelism, does the CUDA programming model inherently offer additional benefits? The following sections explore potential avenues to leverage these implicit advantages.

CUDA on CPU vs Sequential and OpenMP Code

When writing CUDA kernels, it is common practice to specify block sizes as multiples of the warp size (32) and to adjust grid sizes to cover the entirety of the dataset without exceeding it. These dimensions are often chosen based on general guidelines

rather than deep parallel programming knowledge. Interestingly, these choices, while enhancing GPU parallelism, inadvertently lead to a form of tiling that significantly benefits multicore CPU architectures when the CUDA code is transpiled. This unexpected outcome demonstrates the potential of CUDA as a robust programming model, even for sequential execution on CPUs.

The automatic application of tiling during the transpilation from CUDA to CPU mimics what would typically require deliberate optimization by a programmer. This automatic structuring improves cache locality and workload distribution, resulting in notable performance improvements—even when the original CUDA code does not explicitly focus on parallelism. To explore this phenomenon, a proposed experiment could compare the performance of transpiled CUDA code against traditionally optimized sequential or OpenMP code in single-threaded CPU contexts, examining metrics such as speed, cache efficiency, and hardware utilization.

Further insights are gleaned from the Parboil Benchmark Suite's Histogram benchmark, which processes the histogram of RGB colors in an image. CUDA programmers often make high-level design choices, such as dividing pixels into tiles, that are naturally conducive to parallel processing. These decisions, initially tailored for GPU efficiencies, potentially translate well to CPUs via transpilation, showcasing reduced communication and synchronization overhead compared to traditional OpenMP approaches. Expected result would suggest that CUDA's method of structuring data and tasks could inherently offer a superior framework for programming multicore CPUs, surpassing traditional models that rely heavily on explicit parallel loop constructs and synchronization mechanisms. Thus, further experiments can be developed to examine the effect of CUDA programming on algorithm-wise improvement of code robustness.

CUDA on non-Nvidia GPUs

Prior work suggests that OpenMP runtime overheads can be greatly reduced when CUDA syntax is replaced by OpenMPX, a more recent extension of OpenMP that is more explicit in defining threads and synchronization, similar to CUDA. This suggests that despite the significant engineering efforts in developing its ecosystem, CUDA heavily relies on programmer involvement in encoding explicit parallelism to achieve effective performance. Experiments can be developed to compare the performance of CUDA transpiled to OpenMPX against native CUDA performance, demonstrating that CUDA necessitates explicit parallelism for performance gains.

7.2.3 Natural Decompilation Enhanced by LLM

SPLENDID represents an initial step towards the promising future of enabling effective collaboration between programmers and compilers in parallelization tasks. While SPLENDID has significantly enhanced the naturalness of code through compiler detransformations, there is potential for further integration with large language models or other machine learning models in future work. To ensure correctness, unlike previous approaches which relied solely on machine learning models for correctness (e.g., [61, 63]), we propose a system where the compiler generates code and consults an LLM for syntax usage and transformation suggestions. However, the final decision on whether and how to apply these transformations remains with the compiler, thus ensuring correctness. For instance, the compiler could query an LLM like GPT for meaningful variable names, replacing the heuristic-based variable name generation used in SPLENDID, but it is the compiler's responsibility to apply these names to guarantee correctness. Similarly, the compiler can consult GPT for suggestions on intuitive control flows, loop structures, and comments to describe code segments. Additionally, the compiler can leverage machine learning models to narrow down the search and optimization space by adopting transformations and detransformations

suggested by GPT.

Appendix A

Implementation Details

A.1 BLEU For Formal Languages

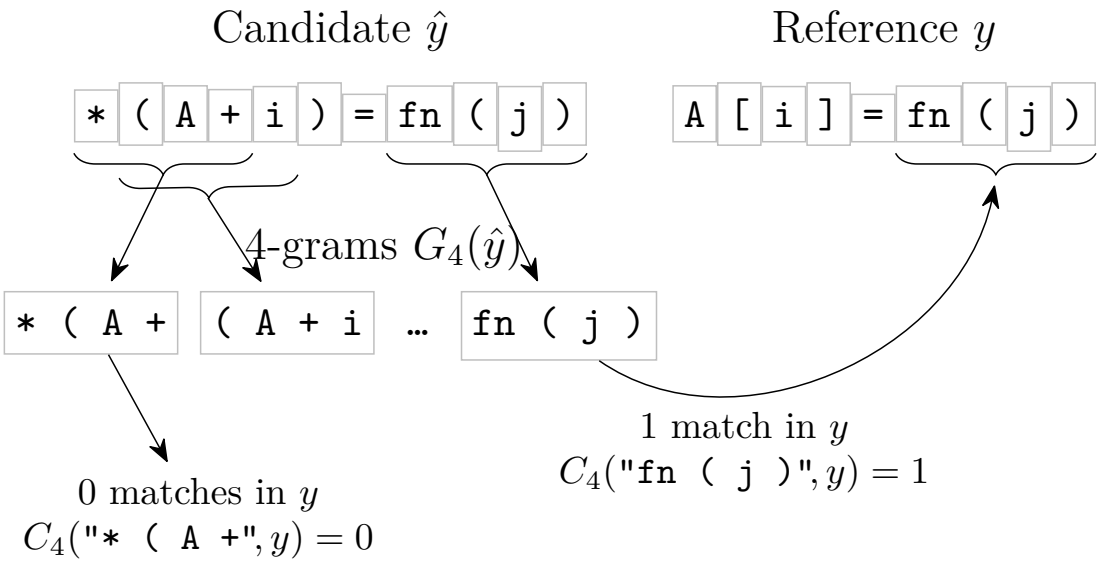


Figure A.1: BLUE score calculation

Figure A.1 illustrates the calculation of the BLEU score. The underlying idea is to build a set of all sub-sequences of length n of a candidate phrase, called n-grams, and see whether they also occur in a reference phrase. In the case of formal languages, a phrase is a sequence of tokens as detected by the language lexer. The BLEU score

Reference Program

```
for(i=1; i<N-1; i++)
  B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
```

Candidate Programs

(a) Obfuscated Variable Names
BLEU Score: 0.3730

```
for(var0 = 1; var0 < N - 1; var0++)
  var1[var0] = (var2[var0-1] + var2[var0] + var2[var0+1]) / 3;
```

(b) Unnatural Control Flow
BLEU Score: 0.5928

```
if (N - 1 > 0) {
  i = 1;
  do {
    i += 1;
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  } while (i < N - 1);
}
```

(c) No Explicit Parallelism
BLEU Score: 0.3600

```
__kmpc_fork_call(param1, param2, param3, kmp_int32
4, forked_function, param5, A, B, &lb, &ub);

void forked_function(Type1 arg1, Type2 arg2,
double *A, double *B, int *lb, int *ub){
 __kmpc_for_static_init_8(arg1, arg2, 33,
                          lb, ub, 1, 1);
 for (i=*lb; i<*ub; i++)
   B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
 __kmpc_for_static_fini(arg1, arg2);
}
```

Figure A.2: A hand-crafted example of BLEU scores reflecting each area of unnaturalness in Section Section 2.1.3.

is a percentage of matched n-grams relative to the theoretical maximum number of matches (i.e., if the candidate and reference are identical):

$$\frac{\text{Number of matches}}{\text{Theoretical max number of matches}} = \frac{\sum_{s \in G_n(\hat{y})} C(s, y)}{\sum_{s \in G_n(\hat{y})} C(s, \hat{y})} \tag{A.1}$$

A candidate n-gram can occur more times in the reference that in itself; to ensure that the score is in the range $[0, 1]$, the number number of matches that are counted is bounded:

$$\frac{\sum_{s \in G_n(\hat{y})} \min(C(s, \hat{y}), C(s, y))}{\sum_{s \in G_n(\hat{y})} C(s, \hat{y})} \tag{A.2}$$

The final BLEU-4 score is the geometric mean of the n-gram scores of $n = 1, \ldots, 4$. If the candidate phrase is very short, then the denominator will be small and fewer matches be needed to reach a high BLEU score. Therefore, when the candidate phrase is shorter than the reference, an additional brevity penalty is applied[1]. Typically, the final score is presented as a percentage, i.e. multiplied by 100.

The BLEU score for natural languages also allows comparison multiple reference phrases, in which case for each n-gram, the reference phrase with the most matches is used.

This work measures code naturalness using the BLEU-4 score since it is also used in other literature [38, 61] to evaluate formal language naturalness. As shown in Figure A.2, unnatural variable names, control flow, and parallelism representation all degrade the BLEU score from 1 (identical to the reference program). While program (a) has a higher 1-gram score with better word-by-word matching, program (b) contains at least an identical loop body to the reference code, resulting in higher 2-gram

[1]In contrast, the CodeBLEU score is biased towards longer candidates, but does not apply a "verbosity" penalty.

to 4-gram scores. Thus, programs (b) have shown higher BLEU-4 scores. This means that variable renaming described in Section Section 4.1.3] has a significant influence on improving the BLEU score. To show that BLEU scores still reflect improvement in naturalness by constructing natural control flow and explicit parallelism, we thus reported the BLEU score of SPLENDID with variable renaming turned off, namely SPLENDID v1 and Portable SPLENDID, as shown in Figure 5.7. Overall, the BLEU score of Rellic-produced code in Figure 4.1 is 0.0035, and SPLENDID-produced code instead scores 0.2932[2].

[2]This section was largely contributed by Michael Kruse and all the coauthors of SPLENDID.

Bibliography

[1] National Security Agency. Ghidra. `https://ghidra-sre.org/`, 2019.

[2] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. Using machine translation for converting python 2 to python 3 code. Technical report, PeerJ PrePrints, 2015.

[3] Paul F. Albrecht, Philip E. Garrison, Susan L. Graham, Robert H. Hyerle, Patricia Ip, and Bernd Krieg-Brückner. Source-to-source translation: Ada to pascal and pascal to ada. *Proceedings of the ACM-SIGPLAN symposium on Ada programming language*, 1980.

[4] AMD. Hipify. `https://github.com/ROCm/HIPIFY`, 2024.

[5] SF Antao, A Bataev, and AC Jacob. Offloading support for openmp in clang and llvm. In *Third Workshop on the LLVM Compiler Infrastructure in HPC*. IEEE, 2016.

[6] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I August. Perspective: A sensible approach to speculative automatic parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 351–367, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Greg Chan, Simone Campanoni, and David I August. Scaf: A speculation-aware collaborative dependence analysis framework. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 638–654, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Hamid Arabnejad, João Bispo, Jorge G. Barbosa, and João M.P. Cardoso. Autopar-clava: An automatic parallelization source-to-source tool for c code applications. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '18, page 13–19, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Hamid Arabnejad, João Bispo, João MP Cardoso, and Jorge G Barbosa. Source-to-source compilation targeting openmp-based automatic parallelization of c applications. *The Journal of Supercomputing*, 76(9):6753–6785, 2020.

[10] Jeremy Ashkenas. Coffeescript. `https://github.com/jashkenas/coffeescript`, 2009.

[11] David I. August and Matthew J. Bridges. The velocity compiler: extracting efficient multicore execution from legacy sequential codes, 2008.

[12] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms®: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, page 625–634, USA, 2004. IEEE Computer Society.

[13] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[14] Guy E Blelloch and John Greiner. A provable time and space efficient implementation of nesl. *ACM SIGPLAN Notices*, 31(6):213–225, 1996.

[15] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, page 207–216, New York, NY, USA, 1995. Association for Computing Machinery.

[16] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1995.

[17] Michael Bolin. *Closure: The definitive guide: Google tools to add power to your JavaScript.* " O'Reilly Media, Inc.", 2010.

[18] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (CC)*, page 132–146, Berlin, Heidelberg, 2008. Springer.

[19] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 101–113, New York, NY, USA, 2008. Association for Computing Machinery.

[20] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[21] M. Bravenboer and E. Visser. The stratego xt transformation system. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[22] Matthew J. Bridges. *The VELOCITY Compiler: Extracting Efficient Multicore Execution from Legacy Sequential Codes*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, Nov 2008.

[23] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using Semantics-Preserving structural analysis and iterative Control-Flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 353–368, Washington, D.C., August 2013. USENIX Association.

[24] S. Campanoni, T. M. Jones, G. Holloway, G. Y. Wei, and D. Brooks. HELIX: Making the extraction of thread-level parallelism mainstream. *IEEE Micro*, 32(4):8–18, July 2012.

[25] João M.P. Cardoso, Tiago Carvalho, José G.F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz, and Zlatko Petrov. Lara: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*, AOSD '12, page 179–190, New York, NY, USA, 2012. Association for Computing Machinery.

[26] Gengbiao Chen, Zhuo Wang, Ruoyu Zhang, Kan Zhou, Shiqiu Huang, Kangqi Ni, Zhengwei Qi, Kai Chen, and Haibing Guan. A refined decompiler to generate c code with high readability. In *2010 17th Working Conference on Reverse Engineering*, pages 150–154, Beverly, MA, USA, 2010. Institute of Electrical and Electronics Engineers (IEEE).

[27] Clang. Expressive diagnostics. `https://clang.llvm.org/diagnostics.html`, 2023.

[28] LLVM Community. Introduction and development of the emitc dialect in mlir. `https://reviews.llvm.org/D76571`. Discussion on the development and features of the EmitC dialect, accessed: [insert date here].

[29] Julia Computing. Llvm cbackend. `https://github.com/JuliaComputingOSS/llvm-cbe`, 2022.

[30] Enrico A Deiana, Vincent St-Amour, Peter A Dinda, Nikos Hardavellas, and Simone Campanoni. Unconventional parallelization of nondeterministic applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 432–447, 2018.

[31] Zachary DeVito, Niels Joubert, Francisco Palacios, Sean Oakley, Margarita Medina, Mario Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[32] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. Who's debugging the debuggers? exposing debug information bugs in optimized binaries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 1034–1045, New York, NY, USA, 2021. Association for Computing Machinery.

[33] JM Diaz, S Pophale, K Friedline, and O Hernandez. Evaluating support for openmp offload features. In *Proceedings of the Second Annual OpenMP Users Conference*. ACM, 2018.

[34] R Dietrich, F Schmitt, A Grund, and D Schmidl. Performance measurement for the openmp 4.0 offloading model. In *Euro-Par 2014: Parallel Processing Workshops*. Springer, 2014.

[35] Johannes Doerfert, Atemn Patel, Joseph Huber, Shilei Tian, Jose M Monsalve Diaz, Barbara Chapman, and Giorgis Georgakoudis. Co-designing an openmp gpu runtime and optimizations for near-zero overhead execution. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 504–514, 2022.

[36] Rob Farber. CLACC: An Open Source OpenACC Compiler and Source Code Translation Project. `https://www.exascaleproject.org/highlight/clacc-an-open-source-openacc-compiler-and-source-code-translation-project/`, 2024. Accessed: April 19, 2024.

[37] Naila Farooqui, Andrew Kerr, Gregory Diamos, Sudhakar Yalamanchili, and Karsten Schwan. A framework for dynamically instrumenting gpu compute applications within gpu ocelot. page 9, 03 2011.

[38] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages, 2020.

[39] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, jul 1987.

[40] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. Smartdec: Approaching c++ decompilation. In *2011 18th Working Conference on Reverse Engineering*, pages 347–356, Limerick, Ireland, 2011. IEEE.

[41] Haxe Foundation. Haxe. `https://github.com/HaxeFoundation/haxe`, 2005.

[42] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. Springer, Berlin, Heidelberg.

[43] GNU. GNU libgomp. `https://gcc.gnu.org/onlinedocs/libgomp/`, 2022.

[44] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 innovative parallel computing (InPar)*, pages 1–10, San Jose, CA, USA, 2012. IEEE.

[45] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[46] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A comb for decompiled c code. In *Asia Conference on Computer and Communications Security (ASIA CCS'20)*, page 637–651, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 251–264, Freiburg, Germany, 2007. Association for Computing Machinery.

[48] Brian Homerding, Atmn Patel, Enrico Armenio Deiana, Yian Su, Zujun Tan, Ziyang Xu, Bhargav Reddy Godala, David I. August, and Simone Campanoni. The parallel semantics program dependence graph, 2024.

[49] Richard D Hornung and Jeffrey A Keasler. The raja portability layer: overview and status. 2014.

[50] Joseph Huber, Melanie Cornelius, Giorgis Georgakoudis, Shilei Tian, Jose M Monsalve Diaz, Kuter Dinel, Barbara Chapman, and Johannes Doerfert. Efficient execution of openmp on gpus. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 41–52, 2022.

[51] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H Kelm, Isaac Gelado, Sam S Stone, Robert E Kidd, Sara S Baghsorkhi, Aqeel A Mahesri, Stephanie C Tsao, et al. Implicitly parallel programming models for thousand-core microprocessors. In *Proceedings of the 44th annual Design Automation Conference*, pages 754–759, San Diego, CA, USA, 2007. IEEE.

[52] Intel. Intel® advisor user guide. `https://www.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top.html`, 2022.

[53] I. R. Ivanov, O. Zinenko, J. Domke, T. Endo, and W. S. Moses. Retargeting and respecializing gpu workloads for performance portability. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 119–132, Los Alamitos, CA, USA, mar 2024. IEEE Computer Society.

[54] Alister Johnson, Camille Coti, Allen D. Malony, and Johannes Doerfert. Martini: The little match and replace tool for automatic application rewriting with code examples. page 19–34, Berlin, Heidelberg, 2022. Springer-Verlag.

[55] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. Inspire: The insieme parallel intermediate representation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 7–17, 2013.

[56] Lester Kalms, Tim Hebbeler, and Diana Göhringer. Automatic opencl code generation from llvm-ir using polyhedral optimization. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '18, page 45–50, New York, NY, USA, 2018. Association for Computing Machinery.

[57] Khronos Group. *SYCL Specification*, 2020. Version 2020.

[58] Khronos Group. *OpenCL Specification*, 3.0 edition, 2022.

[59] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions*, pages 177–180, Prague, Czech Republic, 2007. Association for Computational Linguistics.

[60] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages, 2020.

[61] Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. *Advances in Neural Information Processing Systems*, 34:1–18, 2021.

[62] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[63] Celine Lee, Abdulrahman Mahmoud, Michal Kurek, Simone Campanoni, David Brooks, Stephen Chong, Gu-Yeon Wei, and Alexander M Rush. Guess & sketch: Language model guided transpilation. *arXiv preprint arXiv:2309.14396*, 2023.

[64] Sang-Ik Lee, Troy A Johnson, and Rudolf Eigenmann. Cetus–an extensible compiler infrastructure for source-to-source transformation. In *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers 16*, pages 539–553. Springer, 2004.

[65] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. Debug information validation for optimized code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1052–1065, New York, NY, USA, 2020. Association for Computing Machinery.

[66] Chunhua Liao, Daniel J Quinlan, Thomas Panas, and Bronis R De Supinski. A rose-based openmp 3.0 research compiler supporting multiple runtime libraries. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More: 6th Internationan Workshop on OpenMP, IWOMP 2010, Tsukuba, Japan, June 14-16, 2010 Proceedings 6*, pages 15–28. Springer, 2010.

[67] Shih-Wei Liao, Amer Diwan, Robert P Bosch Jr, Anwar Ghuloum, and Monica S Lam. Suif explorer: An interactive and interprocedural parallelizer. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 37–48, 1999.

[68] Relyze Software Limited. Relyze. `https://www.relyze.com/`, 2022.

[69] LLVM. Llvm loop terminology (and canonical forms), 2023.

[70] LLVM/OpenMP. LLVM/OpenMP 15.0.0git documentation. `https://openmp.llvm.org/design/Runtimes.html`, 2023.

[71] David B. Loveman. Program improvement by source-to-source transformation. *J. ACM*, 24(1):121–145, jan 1977.

[72] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664:1–14, 2021.

[73] Roberto Lublinerman, Swarat Chaudhuri, and Pavol Cerny. Parallel programming with object assemblies. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 61–80, New York, NY, USA, 2009. ACM.

[74] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[75] Norio Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[76] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, Brian Homerding, et al. Noelle offers empowering llvm extensions. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 179–192, Seoul, Korea, 2022. IEEE, IEEE.

[77] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):1–26, 2013.

[78] A Mishra, L Li, M Kong, and H Finkel. Benchmarking and evaluating unified memory for openmp gpu offloading. In *Proceedings of the Fourth Workshop on Accelerator Programming Using Directives*. ACM, 2017.

[79] Simon Moll. Ast - extractor for llvm (axtor). `https://github.com/cdl-saarland/axtor`, 2017.

[80] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising c to polyhedral mlir. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–59, 2021.

[81] William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. High-performance gpu-to-cpu transpilation and optimization via high-level parallel constructs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 119–134, New York, NY, USA, 2023. Association for Computing Machinery.

[82] Vincent D Moynihan and Peter JL Wallis. The design and implementation of a high-level language converter. *Software: Practice and Experience*, 21(4):391–400, 1991.

[83] Author(s) Name(s). Prompt: A framework for streamlined development of fast memory profilers. *arXiv preprint arXiv:231103263X*, 2023.

[84] V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. A transformation framework for optimizing task-parallel programs. *ACM Trans. Program. Lang. Syst.*, 35(1), apr 2013.

[85] Quan M. Nguyen and Daniel Sanchez. Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 596–608. IEEE, 2020.

[86] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[87] Gabriel Noaje, Christophe Jaillet, and Michaël Krajecki. Source-to-source code translator: Openmp c to cuda. In *2011 IEEE International Conference on High Performance Computing and Communications*, pages 512–519, Banff, AB, Canada, 2011. IEEE.

[88] Cedric Nugteren and Henk 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*, GPGPU-5, page 1–10, New York, NY, USA, 2012. Association for Computing Machinery.

[89] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[90] NVIDIA Corporation. *Thrust: Parallel Algorithms Library*, 2023.

[91] N. Nystrom, A. Aiken, and M. Odersky. Polyglot: An extensible compiler framework for java. In *Proceedings of the 2003 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2003.

[92] Trail of Bits Inc. rellic-xref. `https://github.com/lifting-bits/rellic/tree/master/tools/xref`, 2022.

[93] OpenMP Architecture Review Board. OpenMP Application Program Interface, October 2007.

[94] OpenACC-Standard Organization. *OpenACC Programming and Best Practices Guide*. OpenACC-Standard Organization, Online, 2022. Accessed: 2023-04-22.

[95] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[96] Atmn Patel, Shilei Tian, Johannes Doerfert, and Barbara Chapman. A virtual gpu as developer-friendly openmp offload target. In *LLPP '21: The First Workshop on LLVM in Parallel Processing*, pages 1–7. ACM, 2021.

[97] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Conference on Supercomputing (SC'10)*, pages 1–11, New Orleans, LA, November 2010. IEEE Computer Society Press.

[98] Louis-Noël Pouchet. Polybench/c. `http://web.cs.ucla.edu/~pouchet/software/polybench/`, 2021.

[99] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 1–11, New York, NY, USA, 2011. Association for Computing Machinery.

[100] Flang Project. Flang: a Fortran Compiler Targeting LLVM. `https://github.com/flang-compiler/flang`, 2019.

[101] LLVM Project. LLVM Language Reference Manual. `https://llvm.org/docs/LangRef.html`, 2022.

[102] Dan Quinlan, Shmuel Ur, and Richard Vuduc. An extensible open-source compiler infrastructure for testing. In Shmuel Ur, Eyal Bin, and Yaron Wolfsthal, editors, *Hardware and Software, Verification and Testing*, pages 116–133, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[103] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[104] Arun Raman, Jae W. Lee, and David I. August. From sequential programming to flexible parallel execution. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, page 37–40, New York, NY, USA, 2012. Association for Computing Machinery.

[105] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, page 114–123, New York, NY, USA, 2008. Association for Computing Machinery.

[106] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297:1–8, 2020.

[107] Hex-Rays SA. Hex-Rays Decompiler - User Manual. https://www.hex-rays.com/products/decompiler/manual/, 2021.

[108] Paul Sathre, Mark Gardner, and Wu-chun Feng. On the portability of cpu-accelerated applications via automated source-to-source translation. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia '19, page 1–8, New York, NY, USA, 2019. Association for Computing Machinery.

[109] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. *SIGPLAN Not.*, 52(8):249–265, jan 2017.

[110] Snowman decompiler. https://github.com/yegord/snowman, 2021.

[111] L Sommer, J Korinth, and A Koch. Openmp device offloading to fpga accelerators. In *IEEE 28th International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 2017.

[112] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, pages 16–30, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[113] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[114] Marek Surovič and Francesco Bertolaccini. Rellic.

[115] Steven Swanson, Andrew Schwerin, Michela Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan Eggers. Sequoia: Programming the memory hierarchy. *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[116] Armstrong A Takang, Penny A Grubb, and Robert D Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

[117] Zujun Tan, Yebin Chon, Michael Kruse, Johannes Doerfert, Ziyang Xu, Brian Homerding, Simone Campanoni, and David I. August. Splendid: Supporting parallel llvm-ir enhanced natural decompilation for interactive development. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 679–693, New York, NY, USA, 2023. Association for Computing Machinery.

[118] Babel Team. Babel. https://github.com/babel/babel, 2014.

[119] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196, Berlin, Heidelberg, 2002. Springer, Springer.

[120] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, 2002.

[121] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. Does BLEU score work for code migration? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, page 165–176, Montreal, Quebec, Canada, may 2019. IEEE Press.

[122] Christian R Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S Hollman, Dan Ibanez, et al. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817, 2021.

[123] Abhishek Udupa, Kaushik Rajan, and William Thies. Alter: exploiting breakable dependences for parallelization. *ACM SIGPLAN Notices*, 47:480, 08 2012.

[124] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2007.

[125] Hans Vandierendonck, Simon Rul, and Koen De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 389–400, New York, NY, USA, 2010. ACM.

[126] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4), jan 2013.

[127] Christoph Von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 79–89, New York, NY, USA, 2007. Association for Computing Machinery.

[128] Zhiming Wang, Yury Plyakhin, Chenwei Sun, Ziran Zhang, Zhiwei Jiang, Andy Huang, and Hao Wang. A source-to-source cuda to sycl code migration tool: Intel® dpc++ compatibility tool. In *International Workshop on OpenCL*, IWOCL'22, New York, NY, USA, 2022. Association for Computing Machinery.

[129] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177, Los Alamitos, CA, USA, 2016. IEEE, IEEE Computer Society.

[130] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS Symposium 2015*, NDSS '15, pages 1–15, San Diego, CA, USA, 2015. Internet Society.