软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(12):5402−5422 [doi: 10.13328/j.cnki.jos.007407] [CSTR: 32375.14.jos.007407]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# swJulia: 面向新一代神威超级计算机的 Julia 语言编译系统*

沈 莉[1], 周文浩[1], 王 飞[3], 李 斌[2], 谭 坚[1], 商红慧[1], 安 虹[1], 漆锋滨[2]

[1](中国科学技术大学, 安徽 合肥 230026)

[2](江南计算技术研究所, 江苏 无锡 214083)

[3](清华大学, 北京 100084)

通信作者: 漆锋滨, E-mail: qifb116@sina.com

**摘 要**: 随着异构融合体系结构在高性能计算领域的普及, 挖掘其潜能并探索新的应用构建策略变得至关重要. 传统的静态编译方法已无法满足复杂计算需求, 动态编程语言因其灵活性和高效性而备受瞩目. Julia 是一种现代的高性能动态编程语言, 其基于即时编译机制, 在科学计算等领域表现出色. 结合申威异构众核架构特点, 构建 ORCJIT 编译引擎并提出了动态模式下的片上存储管理方法, 并以此为基础实现针对新一代神威超级计算机的 Julia 动态语言编译器 swJulia. 其不仅继承了 Julia 编译器的灵活性, 同时还有效支持了 SACA 众核编程模型及运行时封装. 利用 swJulia 编译系统, 成功在新一代神威超级计算机上部署了 NNQS-Transformer 量子化学模拟器, 并在多个维度验证了 swJulia 的好用性和高效性. 实验结果显示, swJulia 在单线程基准测试和众核加速上性能卓越, 并能够有效支撑 NNQS-Transformer 量子化学模拟器的超大规模可扩展并行模拟.

**关键词**: Julia 编译器; 神威超级计算机; 动态链接; 即时编译; 量子化学模拟

**中图法分类号**: TP314

中文引用格式: 沈莉, 周文浩, 王飞, 李斌, 谭坚, 商红慧, 安虹, 漆锋滨. swJulia: 面向新一代神威超级计算机的Julia语言编译系统. 软件学报, 2025, 36(12): 5402–5422. http://www.jos.org.cn/1000-9825/7407.htm

英文引用格式: Shen L, Zhou WH, Wang F, Li B, Tan J, Shang HH, An H, Qi FB. swJulia: Julia Compilation System for New Generation Sunway Supercomputer. Ruan Jian Xue Bao/Journal of Software, 2025, 36(12): 5402–5422 (in Chinese). http://www.jos.org.cn/1000-9825/7407.htm

## swJulia: Julia Compilation System for New Generation Sunway Supercomputer

SHEN Li[1], ZHOU Wen-Hao[1], WANG Fei[3], LI Bin[2], TAN Jian[1], SHANG Hong-Hui[1], AN Hong[1], QI Feng-Bin[2]

[1](University of Science and Technology of China, Hefei 230026, China)

[2](Jiangnan Institute of Computing Technology, Wuxi 214083, China)

[3](Tsinghua University, Beijing 100084, China)

**Abstract**: With the increasing adoption of heterogeneous integrated architectures in high-performance computing, it has become essential to harness their potential and explore new strategies for application development. Traditional static compilation methodologies are no longer sufficient to meet the complex computational demands. Therefore, dynamic programming languages, known for their flexibility and efficiency, are gaining prominence. Julia, a modern high-performance language characterized by its JIT compilation mechanism, has demonstrated significant performance in fields such as scientific computing. Targeting the unique features of the Sunway heterogeneous many-core architecture, the ORCJIT engine is introduced, along with an on-chip storage management approach specifically designed for dynamic modes. Based on these advancements, swJulia is developed as a Julia dynamic language compiler tailored for the new generation of the Sunway supercomputer. This compiler not only inherits the flexibility of the Julia compiler but also provides robust support for the SACA many-core programming model and runtime encapsulation. By utilizing the swJulia compilation system, the deployment of the NNQS-Transformer quantum chemistry simulator on the new generation of the Sunway supercomputer is successfully achieved.

* 基金项目: 国家重点研发计划 (2023YFB3001500)
收稿时间: 2024-10-20; 修改时间: 2024-12-14; 采用时间: 2025-02-11; jos 在线出版时间: 2025-06-11
CNKI 网络首发时间: 2025-06-11

Comprehensive validation across multiple dimensions demonstrates the efficacy and efficiency of swJulia. Experimental results show exceptional performance in single-threaded benchmark tests and many-core acceleration, significantly improving ultra-large-scale parallel simulations for the NNQS-Transformer quantum chemistry simulator.

**Key words**: Julia compiler; Sunway supercomputer; dynamic linking; just-in-time compilation; quantum chemistry simulation

近年来, 随着计算机体系结构的不断演进与创新, 异构融合体系结构异军突起, 已逐渐成为高性能计算领域的主流架构. 表 1 列出了 2024 年 6 月最新发布的 Top500 排行榜[1]排名前 10 位的系统信息, 其中采用异构体系结构的系统占比达到 90%, 而在神威•太湖之光[2]基础上升级的新一代神威超级计算机也同样采用我国自主研制的异构众核处理器 SW26010Pro 构建. 异构融合体系结构通过精心整合通用处理器及专用加速计算单元, 显著增强了系统的计算能力, 并为其注入了前所未有的高效能与灵活性. 然而, 为了全面挖掘并充分利用异构系统的潜能, 研究人员必须不断地探索前沿的应用构建策略, 并将其付诸实践[3].

表 1　Top500 排行榜排名前 10 位的超级计算机的计算节点架构信息 (2024 年 6 月)

| Rank | Name | Processor | Co-processor | Heterogeneous |
|---|---|---|---|---|
| #1 | Frontier | AMD 3rd Generation EPYC 64C 2 GHz | AMD Instinct MI250X | Yes |
| #2 | Aurora | Xeon CPU Max 9470 52C 2.4 GHz | Intel Data Center GPU Max | Yes |
| #3 | Eagle | Xeon Platinum 8480C 48C 2 GHz | NVIDIA H100 | Yes |
| #4 | Fugaku | A64FX 48C 2.2 GHz | — | No |
| #5 | LUMI | AMD 3rd Generation EPYC 64C 2 GHz | AMD Instinct MI250X | Yes |
| #6 | Alps | NVIDIA Grace 72C 3.1 GHz | NVIDIA GH200 Superchip | Yes |
| #7 | Leonardo | Xeon Platinum 8358 32C 2.6 GHz | NVIDIA A100 SXM4 64 GB | Yes |
| #8 | MareNostrum 5 | Xeon Platinum 8460Y+ 32C 2.3 GHz | NVIDIA H100 64 GB | Yes |
| #9 | Summit | IBM POWER9 22C 3.07 GHz | NVIDIA Volta GV100 | Yes |
| #10 | DGX SuperPOD | Xeon Platinum 8480C 56C 3.8 GHz | NVIDIA H100 SXM5 80 GB | Yes |

在高性能计算的传统实践范畴内, 应用程序的构建往往依赖于静态链接的方式. 这种构建方法的核心在于, 它能够通过在编译阶段将所有必要的代码和数据静态地链接成一个完整的可执行文件, 从而确保系统运行时的稳定性和行为的可预测性. 然而, 这种方式的灵活性存在显著的局限性, 尤其是在面对持续升级的硬件架构和日益复杂的计算环境时, 难以迅速调整以适应新的需求. 为了克服静态链接的这一固有缺陷, 新兴的高性能计算应用开始探索并采用动态链接技术[4]. 与传统的静态链接相比, 动态链接提供了更高的灵活性和可扩展性, 展现出多方面的优势. 首先, 其高度的模块化设计使系统架构更为清晰透明, 便于开发者进行管理和维护. 其次, 动态链接引入了更为高效的模块更新和错误修补机制, 显著提高了系统的可维护性和健壮性. 最后, 通过动态加载和卸载特定模块, 该技术还能有效降低系统的内存占用, 从而提升整体资源的利用效率. 这些特点使得动态链接成为高性能计算领域应对快速变化环境的重要技术手段[5].

在众多高级编程语言中, Julia 以其基于动态链接的即时编译 (just-in-time compilation, JIT) 机制而备受瞩目[6]. Julia 的设计理念旨在实现高效执行、易用性以及与外部库的无缝集成, 这一理念使得 Julia 成为高性能计算应用的优选语言. 在科学计算、数据分析以及机器学习等领域, Julia 均展现出了卓越的性能和广阔的应用前景[7]. 新一代神威超级计算机在硬件层面进行了全面的升级, 对动态运行的支持也更为完善[8]. 这一显著进步为 Julia 等动态语言的适配提供了坚实的技术基础. 借助神威超级计算机的先进架构和强大计算能力, 我们有望充分释放 Julia 在国产超算系统上的巨大潜能, 并进一步推动相关领域的技术创新和应用拓展.

本文深入探讨了针对新一代神威超级计算机的即时编译支撑技术及动态编译优化方案, 并在此基础上成功研发了 swJulia 编译系统. 该系统是专为新一代神威超级计算机定制的 Julia 编译环境, 通过其独特的设计与优化, 显著提升了 Julia 应用的性能. 同时, 本文基于 swJulia 编译系统构建了 NNQS-Transformer 量子化学模拟器, 在新一代神威超级计算机上实现了超大规模可扩展并行模拟, 并取得了一系列国际领先的应用成果. swJulia 系统的成功研制, 不仅有效推动了相关领域的技术创新与应用拓展, 更为依托国产超级计算机的科学研究与工程实践提供了坚实的技术基础. 本文的贡献主要如下.

1) 针对 SW26010Pro 处理器架构特点, 设计了异构众核即时编译支撑方案, 包括 ORCJIT 引擎构建以及动态链接模式下的片上存储空间管理, 有效支撑了 Julia 等动态语言在国产异构众核处理器上的适配和优化.

2) 设计并实现了面向新一代神威超级计算机的动态语言编译器 swJulia, 其继承了 Julia 编译系统的灵活性, 同时支持原生众核编程、向量数据类型以及运行时接口封装等针对 SACA 编程模型的 Julia 语言扩展.

3) 基于 swJulia 成功实现了大规模并行量子化学模拟示范应用在新一代神威超级计算机上的部署, 并结合硬件架构及应用特点, 实现了多个维度的优化方法, 有效验证了 swJulia 编译系统的可用性和好用性.

本文第 1 节介绍背景及相关工作. 第 2 节介绍 swJulia 的基本结构及关键技术实现. 第 3 节以大规模并行量子化学模拟器 NNQS-Transformer 为例介绍基于 swJulia 的示范应用支撑及优化. 第 4 节通过实验评估 swJulia 的基础性能及示范应用的优化效果. 第 5 节给出总结和展望.

## 1 相关工作

### 1.1 Julia 动态编程语言

Julia 是一种现代的、开源的、高性能的科学计算和数据处理编程语言. 其由 MIT 在 2009 年发起, 旨在解决“两种语言”的问题, 即期望该语言在运行数值算法时的性能与 C++或 Fortran 一样优秀, 而在编程方面与 Python 或 Matlab 一样简单轻松[6]. 截至 2024 年 6 月, Julia 在代码托管平台 GitHub[9]上的 Stars 数量超过 44.7k, TIOBE 编程语言排行榜[10]排名第 29 位, 并且呈现稳步上升的趋势.

Julia 语言实现了静态编译语言和动态类型语言之间的平衡. 基于 LLVM 的即时编译 (JIT) 机制, Julia 提供了强大的计算性能. Julia 内置的函数式编程以及多重派发等功能, 在保证代码编写简单, 轻松易读的同时还能拥有与静态语言类似的性能[7]. 与 C 和 Fortran 等传统编程语言相比, Julia 在设计之初就考虑了并行计算相关的特性, 在编程语言层面提供了向量化、线程级并行和分布式计算的原生支持.

Julia 语言能够非常方便地与其他编程语言嫁接. 目前, Julia 语言可以在不使用第三方代码的情况下, 直接调用 C/C++、Fortran、Python、R 和 Matlab 等语言, 从而获得更具可读性和高效性的计算程序. 同时, 为了有效利用 NVIDIA GPU 的强大计算能力, Julia 社区推出了 CUDA.jl[11]以实现将 CUDA 编程模型无缝集成到 Julia 的元编程系统. CUDA.jl 支持自动类型推断, 可以自动将 Julia 代码转换为 GPU 可执行的 CUDA Kernel, 并提供了一套丰富的函数库, 覆盖内存管理、同步、错误处理等各个方面[12].

自发布以来, Julia 获得了学术界和工业界的广泛关注, 并产生了大量优质的第三方库. Optim.jl[13]和 DifferentialEquations.jl[14]等科学计算库涵盖了数值分析、优化、微分方程求解等多个领域, 为研究人员提供了高效的算法实现. FinEtools.jl[15]和 FEMBase.jl[16]等有限元分析库用于求解结构力学、热传导等问题, 支持各种结构类型和加载条件下的仿真分析. CFD_Julia 是由 Pawar 等人[17]创建的流体力学模拟工具, 包含一系列经典的 CFD 算法, 并支持多种边界条件和流动模型. Flux.jl[18]是一个基于 Julia 的深度学习框架, 提供了丰富的工具和库用于构建和训练神经网络模型. Yao.jl[19]是一个开源的 Julia 语言框架, 旨在通过软件工具、量子算法设计、量子软件 2.0 和量子计算教育来赋能量子信息研究.

Julia 语言计算速度快、可读性高、开源免费, 相比 Python 等动态编程语言能够保证程序执行的高效性, 相比 C++、Fortran 等静态编程语言更加贴近自然语言的编写风格, 是开发新兴科学计算应用的极佳选择, 亟需在新一代神威超级计算机上进行部署和优化.

### 1.2 新一代申威异构众核处理器 SW26010Pro

SW26010Pro 异构众核处理器是新一代神威超级计算机的核心[20], 其基本结构如图 1 所示. SW26010Pro 片内集成 6 个核组 (core group, CG), 每个核组包含 1 个控制核心 (management process element, MPE) 和 64 个运算核心 (computing process element, CPE). MPE 主要功能是控制、通信、I/O 等, CPE 则主要用于加速并行计算. CG 中 CPE 以 8×8 阵列方式进行排布, CPE 之间通过阵列内网络进行互连. CG 中的任意两个 CPE 之间, 可以通过远程内存访问 (remote memory access, RMA) 方式进行数据通信.

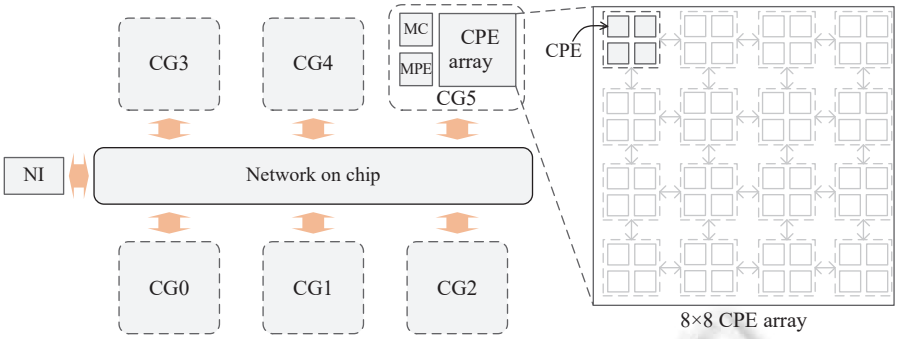



图 1　SW26010Pro 异构众核处理器基本结构

MPE 为 64 位 RISC 结构通用处理单元, 具有 32 KB L1 指令缓存、32 KB L1 数据缓存和 512 KB L2 缓存. CPE 为 64 位 RISC 结构精简处理单元, 支持 512 位 SIMD 向量扩展, 支持双精度、单精度、半精度浮点运算和整数运算. 每个 CPE 都拥有独立的指令缓存和由用户控制的片上局部数据存储器 (local data memory, LDM). CPE 可以通过访存指令或直接内存访问 (direct memory access, DMA) 方式实现 LDM 与主存之间的数据传输.

在上一代申威异构众核处理器的实现中, CPE 使用了两种程序计数器 (program counter, PC) 值, 一种为软件可见的全局 PC (global PC, GPC), 指示主存用户区程序段的 PC 值, 另一种为流水线使用的短 PC (short PC, SPC), 需要软件保证 CPE 代码段长度不超出 SPC 指示的范围, 否则将产生异常. 虽然 SPC 降低了硬件设计的复杂度, 但是导致硬件无法通过寄存器间接转移方式实现超出 SPC 指示范围的 CPE 程序段跳转, 从而限制了动态运行模式的使用. SW26010Pro 在硬件设计上进行了改进, 在 CPE 内部统一使用 GPC 方案, 使得复杂动态程序的加载和执行成为可能. 系统软件亟需结合 SW26010Pro 处理器架构特点设计动态支撑方案, 以满足 Julia 等动态编程语言的编译和运行需求.

## 2　swJulia 编译系统设计及实现

本文在新一代神威超级计算机上实现了 swJulia 动态语言编译系统, 其组成结构如后文图 2 所示, 主要包括 MPE 编译器、LLVM.jl 以及 SACA.jl 等主要模块.

下面, 我们分别对 swJulia 编译器各主要模块的工作原理及实现细节进行详细介绍.

### 2.1　MPE 编译器

MPE 编译器是 MPE 的 Julia 语言编译器, 其基于 LLVM 的 JIT 编译技术, 为 MPE 提供了对 Julia 语言语法及功能特性的完备支持. MPE 编译器支持多个层次的中间表示 (intermediate representation, IR) 降级和优化, 能够将 Julia 高级语言代码编译为高效的 LLVM IR.

MPE 编译器继承了 Julia 编译器的完整功能, 能够将符合 Julia 语法标准的高级语言降级为面向 MPE 的 Julia IR, 并对其执行类型推断、内联展开、循环融合等高级优化. 紧接着, MPE 编译器将 Julia IR 降级为面向 MPE 的 LLVM IR, 并对该 IR 执行寄存器分配、指令调度、自动向量化等低级优化. 与传统的提前编译 (ahead-of-time, AOT) 方式不同, Julia 编译器并不生成存储在磁盘上的可执行文件, 而是将 LLVM IR 通过 JIT 引擎编译并加载为内存中的可执行指令序列. 传统的 AOT 编译方式存在静态编译的局限性, 无法准确预测代码在运行时的行为, 因此不能获得极致的优化效果. 相比而言, JIT 编译方式结合了解释器和 AOT 方式的优点, 在运行时收集各种信息和指标, 既可以快速启动, 又可以生成更加高效的优化代码, 所以能在总体上达到更佳的运行效果[21].

为满足 swJulia 编译器对 JIT 编译的底层需求, 我们在为 SW26010Pro 配备的 swLLVM 编译器[22]中, 整合了对 JIT 编译的支持. 我们选择了目前 LLVM 最新版本的 ORCJIT (on-request compilation JIT) 引擎, 其支持自动内存管理、注册调试器和分析器等事件监听器, 以及并发编译等诸多新特性, 具有较高的模块化程度和可扩展性. 值

得一提的是, ORCJIT 在兼容 MCJIT (machine code JIT) 功能的同时, 还特别支持了惰性编译, 其详细执行流程可参见图 3. 操作中使用的伪指令功能如下: MV 用于将源寄存器或地址中的内容移动到目的寄存器, LD 用于从确定的内存地址中加载数据到目的寄存器, ST 用于从寄存器向确定的内存地址中存储数据, JMP 用于寄存器间接转移, CALL 用于函数调用, RET 用于函数返回. 操作中使用的寄存器功能如下: $pv 用于存储被调函数入口地址, $pc 用于存储当前的 PC 值, $ra 用于存储返回地址, $t11 为 Caller-Saved 临时寄存器, $sp 用于存储栈指针, $v0 用于存储返回值, $a0 和$a1 用于传递函数参数.

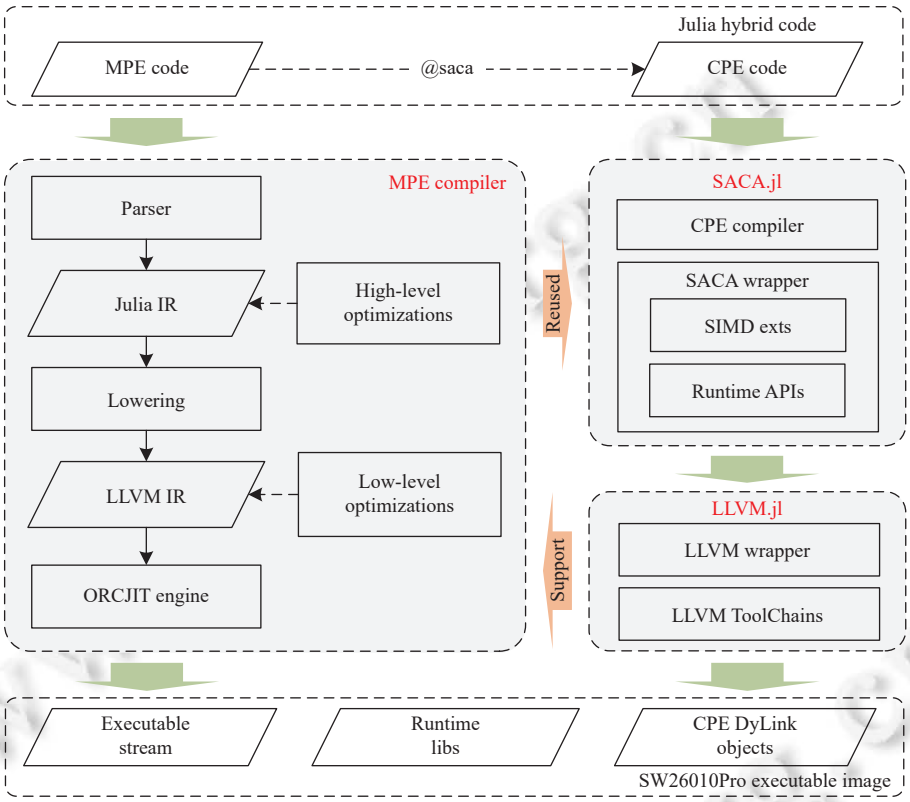



图 2 swJulia 编译系统组成结构框图

ORCJIT 提供了 LLLazyJITBuilder 类专门用于处理惰性编译, 其在初始化阶段构建 LLLazyJIT 实例并设置惰性编译的相关属性, 包括是否允许将 JIT 实例保存到缓存文件中、选择编译 module 还是 function 等. 然后, LLLazyJITBuilder 构建 Stub 管理器和 Trampoline 管理器, 并根据 IR module 实例化与符号入口一一对应的 Stub 及 Trampoline.

Stub 是一段充当占位符的汇编代码, 用于检查被调函数 Callee 是否已被 JIT 编译. 当程序尝试调用 Callee 函数时, 首先会跳转到 Callee 对应的 Stub 入口, 然后从固定内存地址 TargetAddr 中获取目标地址并执行寄存器间接转移. 当 Callee 函数首次被调用时, TargetAddr 中存储的是其对应的 Trampoline 入口地址. 而当 Callee 再次被调用时, 由于其已被 JIT 编译, TargetAddr 中的内容被更新为 Callee 的实际入口地址.

与 Stub 类似, Trampoline 也是一段动态生成的汇编代码, 其长度固定为 32 B, 用于保存 Callee 函数的返回地址并跳转到 Resolver, 进而实现对 Callee 的 JIT 编译. Trampoline 首先将$ra 中存储的 Callee 函数返回地址保存到 Caller-Saved 寄存器$t11 中, 以确保其不会被后续的函数调用所修改, 然后将更新后的 PC 值存入$ra 并跳转到 Resolver 入口.

Resolver 是一段较为复杂的汇编代码, 其主要功能是调用 Reentry 函数完成对 Callee 的 JIT 编译, 并且管理

Prologue 和 Epilogue 以实现 Caller-Saved 寄存器的保留及恢复. Resolver 的执行过程主要包括 3 个阶段: 首先, 其开辟一块栈空间用于保存当前上下文的 Caller-Saved 寄存器, 并且为 Reentry 函数准备参数. 在 ORCJIT 具体实现中, Reentry 根据 Trampoline 来匹配被 JIT 编译的目标函数, 因此需要通过$ra-32 计算出 Callee 的 Trampoline 入口地址. 然后, 其调用 Reentry 实施 Callee 函数的 JIT 编译, 并使用 Callee 函数的实际加载地址更新 TargetAddr 中的内容. 最后, 其从栈上恢复被保留的 Caller-Saved 寄存器, 从$t11 中恢复被保留的 Callee 函数返回地址, 并且通过寄存器间接转移跳转到 Callee 函数的实际入口地址开始执行.

如图 3 所示, 当 Callee 函数被首次调用时, ORCJIT 的执行路径依次经过 1-2-3-4-5-6-7-8. 而在后续调用 Callee 函数时, 无需再重复复杂的 Stub 解析与函数编译流程, 执行路径简化为 1-2′. 通过提供面向 SW26010Pro 的 ORCJIT 以及惰性编译等功能, 本文有效支撑了 Julia 在 MPE 上的动态运行, 并且为充分利用硬件特性开展针对性的编译优化奠定了基础.

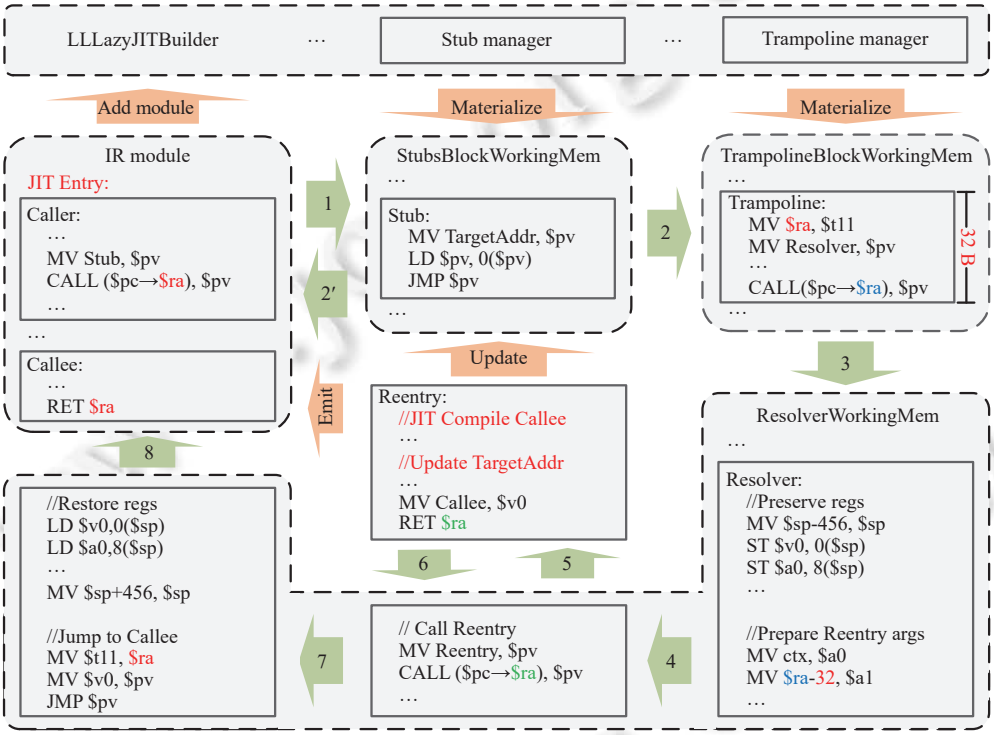



图 3 SW26010Pro 后端中 ORCJIT 惰性编译流程

### 2.2 LLVM.jl

LLVM.jl[12]是一个在 Julia 环境中为集成 LLVM 工具而设计的第三方库. 该库通过为 LLVM API 提供高级封装, 充分利用 Julia 强大的外部函数接口 (foreign function interface, FFI), 实现与底层运行时和加速计算库的高效交互. 借助 LLVM.jl, Julia 开发者能够更加便捷地利用 LLVM 的先进功能, 例如目标代码优化和平台代码生成, 进而提升 Julia 代码的性能及其跨平台兼容性.

在 SW26010Pro 平台上, 我们基于 LLVM.jl 提供了面向 SW26010Pro 的优化编译器 swLLVM 及二进制工具链的交互接口封装, 同时还针对 Julia 的动态特性增加了相应的编译优化支持. 如第 1.2 节所述, 由于 SW26010Pro 的 CPE 全面支持 GPC 方案, 使得动态库间的从核函数跳转成为可能. 动态程序访存行为复杂性的增加对片上存储空间 LDM 的有效管理提出了更高的要求.

如图 4 所示, 在静态链接模式下, swLLVM 编译器采用!ldmhigh/!ldmlow 重定位对 LDM 变量进行相对偏移寻址, 以规避基于全局偏移表 (global offset table, GOT) 间接寻址的!literal 重定位所引入的额外访存开销[22]. 其中, !literal 重定位需要 ldl 和 stw 两次访存, 而!ldmhigh/!ldmlow 重定位仅需要 stw 一次访存.

```
#   !literal example                    #   !ldmhigh/!ldmlow example
ldi     $t1, 1($zero)                   ldi     $t1, 1($zero)
ldl     $t0, a($gp)    !literal         ldih    $t0, a($zero)    !ldmhigh
stw     $t1, 0($t0)                     stw     $t1, a($t0)      !ldmlow
```

图 4 !literal 与!ldmhigh/!ldmlow 重定位的汇编示例

若动态链接模式继续沿用该实现方案, 那么每个动态库的.ldm 段均从相同物理地址开始编址. 这种做法会引发一个问题, 即不同动态库中的 LDM 变量可能产生地址冲突, 具体情形可参见图 5. 在该示例中, 由于 libB.so 中的 Callee 函数可能会使用到 libA.so 中静态声明的 LDM 数组 a[64], LDM 变量地址冲突可能会导致程序运行结果出现错误.

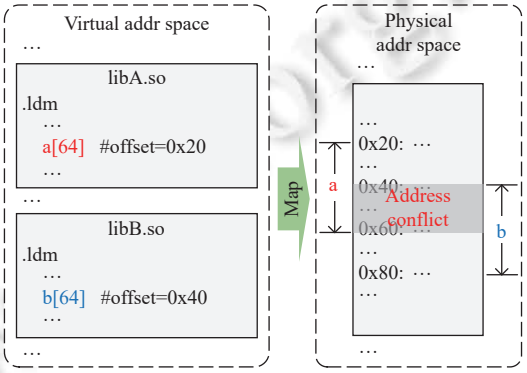



图 5 动态链接模式下 LDM 变量地址冲突的示例

为了有效满足 Julia 等动态应用对片上存储空间的访问需求, 并尽可能避免额外的性能开销, 我们在 swLLVM 中设计并实现了一种针对动态链接模式的片上存储空间优化访问方法. 对于在动态库中静态声明的 LDM 变量, swLLVM 编译器继续使用!ldmhigh/!ldmlow 重定位进行相对偏移寻址, 以确保片上存储空间的访问效率. 同时, 为了让动态链接器能够准确识别和处理上述重定位, 我们在编译器和二进制工具链中新增了两种重定位类型 (R_SW_64_LDMHI 和 R_SW_64_LDMLO), 并且在构建动态库的重定位表时详细登记相关的重定位信息. 图 6 清晰地展示了图 5 中的 libA.so 动态库在实施上述访存优化后的重定位表内容.

```
$ readelf  -r libA.so
Relocation section '.rela.dyn' at offset 0x10ee8 contains 2 entries:
    Offset          Info           Type            Sym. Value        Sym. Name + Addend
0000000014a8  00330000002a  R_SW_64_LDMHI   0000000010000000  a + 0
0000000014b0  00330000002b  R_SW_64_LDMLO   0000000010000000  a + 0
…
```

图 6 LDM 访存优化后的动态库重定位表信息

在动态库的加载过程中, 动态链接器会执行!ldmhigh/!ldmlow 重定位的修正操作, 具体流程参见算法 1. 首先, 动态链接器利用操作系统提供的 mmap() 接口为当前动态库的.ldm 段分配物理地址, 记为 ldm_pstart. 然后, 动态链接器遍历当前动态库的重定位表 relaTable, 特别关注 R_SW_64_LDMHI 和 R_SW_64_LDMLO 两类重定位信息. 对于上述重定位对应的 LDM 变量, 根据其所在.ldm 段的起始物理地址 ldm_pstart 以及该变量在.ldm 段内的相对偏移量 offset, 动态链接器能够精确计算出该变量的实际物理地址, 记为 ldm_paddr. 最后, 针对不同的重定位类型, 动态链接器根据 ldm_paddr 计算和修正相应汇编指令的偏移字段.

**算法 1**. !ldmhigh/!ldmlow 重定位修正算法.

输入: 动态库的重定位表 relaTable 以及.ldm 段 LDMSec;
输出: 重定位修正后的动态库代码段 text.

```
1. ldm_pstart = mmap(LDMSec.getSize(), FLAG_LDM)
2. FOR rela∈relaTable DO
3.       IF rela.getType()∈{R_SW_64_LDMHI, R_SW_64_LDMLO} THEN
4.            offset = rela.getSym().getVAddr() – LDMSec.getVAddr()
5.            ldm_paddr = ldm_pstart + offset
6.            IF rela.getType() == R_SW_64_LDMLO THEN
7.                 disp = ldm_paddr & 0xFFFF
8.            ELSE
9.                 carry = (ldm_paddr >> 15) & 0x1
10.                disp = ((ldm_paddr >> 16) + carry) & 0xFFFF
11.           END IF
12.           rela.getInstr(text).fixOffset(disp)
13.      END IF
14. END FOR
```

该方案通过改进 swLLVM 编译器及其配套的二进制工具链, 既规避了 LDM 变量潜在的地址冲突问题, 保证了 LLVM IR 向 CPE 汇编指令降级的正确性, 又实现了对片上存储空间的高效访问, 为 swJulia 的动态运行提供了稳定的底层支撑.

### 2.3 SACA.jl

SACA (Sunway accelerate computing architecture) 是面向申威众核架构的并行加速计算编程模型, 其优化整合了基础编译及运行时等编程接口, 为科学计算、人工智能等领域应用提供统一的编程和优化支持[23]. SACA.jl 是为了在 Julia 语言中支持 SACA 编程模型而开发的第三方库. 其为 Julia 开发者提供了一种便捷和高效的方式来利用 SW26010Pro 进行计算加速, 无需用户深入了解 SACA 的底层细节, 降低了学习和开发成本.

SACA.jl 实际上由两个主要组件组成: CPE 编译器负责将 Julia 源代码转换为 CPE 上可执行的机器代码, SACA 封装则提供了在 CPE 上执行常用操作所需的类型和接口. 本文基于 Julia 社区已有的基础设施, 在不改动 Julia 编译器底层实现的前提下, 通过 SACA.jl 实现了 Julia 语言对 SACA 编程模型的支持. SACA.jl 支持 Julia 语言的一个子集, 但是已经证明能够满足 SW26010Pro 应用开发的现实需求.

#### 2.3.1 CPE 编译器

图 7 展示了在 C 语言中使用 SACA API 进行向量加法的示例, 包括线程初始化、线程组创建、线程等待及线程回收等. 其中, athread_init 接口用于进行线程的初始化操作, 使用任何 SACA 接口前都需要调用该初始化接口. athread_spawn 接口用于创建线程组并传递运行参数, 其指定的线程任务将被提交到 CPE 阵列执行. athread_join 接口用于等待 CPE 阵列的加速线程任务结束, 而 athread_halt 接口则用于关闭 CPE 流水线.

SACA.jl 通过 CPE 编译器支持对 Julia 函数的编译, 从而实现与 SACA C 类似的程序语义. 图 8 展示了一个基于 SACA.jl 实现的 Julia 语言向量加法程序示例, 该程序的功能与图 7 中展示的 C 语言版本完全相同. 代码的第 2–7 行定义了一个名为 vadd 的 Kernel 函数, 该函数的风格与普通 Julia 函数相似, 但在计算迭代索引变量 $i$ 时特别采用了 SACA.jl 的内建函数_MYID(), 以获取当前 CPE 的逻辑编号. 与 CUDA 或 OpenCL 等异构编程模型相比, SACA.jl 的一个显著优势在于, 它无需用户对 Kernel 进行显式的注释或封装, 从而显著提升了代码的可重用性. 在代码的第 14 行, 程序通过@saca (config…) function(args…) 接口调用了 Kernel 函数. 这里的 config 元组用于指定

生成配置, 其功能与 SACA C 中的运行时配置参数类似. 在本示例中, spawn_mode 参数用于设定当前 Kernel 的运行模式, 它提供了两个选项: SINGLE_CG 和 ALL_CGS, 分别代表单核组模式 (启用 64 个 CPE) 和全片共享模式 (启用 384 个 CPE). 另外, flush_cache 参数则用于决定在当前 Kernel 执行结束后, 是否需要对从核的 Cache 进行刷新. 特别是当 CPE 的片上存储空间部分被设定为软件管理的 LDCache 时, 必须开启 flush_cache 功能, 以确保 Cache 中的最新数据能够及时写回到主存储器中.

```c
/* MPE SACA C Code */
#include <athread.h>
struct ARGS {
  float *a; float *b; float *c;
};
extern void SLAVE_FUN(vadd)(struct ARGS*);
int main() {
  float a[512], b[512], c[512];
  for(int i = 0; i < 512; i++) {
    a[i] = rand(); b[i] = rand();
  }
  struct ARGS args;
  args.a = a; args.b = b; args.c = c;
  athread_init();
  athread_spawn(vadd, &args);
  athread_join();
  athread_halt();
  return 0;
}
```

(a) MPE SACA C 代码

```c
/* CPE SACA C Code */
#include <slave.h>
struct ARGS {
  float *a; float *b; float *c;
};
void vadd(struct ARGS *args) {
  float *a = args->a, *b = args->b, *c = args->c;
  for (int i = _MYID*8; i < (_MYID+1)*8; i++)
    c[i] = a[i] + b[i];
  return;
}
```

(b) CPE SACA C 代码

图 7 使用 SACA C API 实现的向量加法示例

```
using SACA
function vadd(a::Ptr{Float32}, b::Ptr{Float32}, c::Ptr{Float32})
  for i in (_MYID()*8)+1:(_MYID()+1)*8
    c[i] = a[i] + b[i]
  end
  return
end

len = 512
a = rand(Float32, len)
b = rand(Float32, len)
c = similar(a)

@saca spawn_mode = SINGLE_CG flush_cache = ON vadd(a, b, c)
```

图 8 基于 SACA.jl 实现的 Julia 语言向量加法程序示例

在当前版本的 SACA 实现中, athread_spawn 接口采用轻量化实现方式, 通过类型为 void *的指针传递 Kernel 参数以优化线程组创建速度. 当参数有多个时, 需要用户在 athread_spawn 前将参数显式组织成结构体. 为了简化用户使用 SACA.jl 编程的复杂度, 我们提供了 pack_arguments 方法用于打包 Kernel 参数. 当 Kernel 函数拥有多个参数时, @saca 宏会自动将其展开为图 9 中第 4–8 行所示的调用形式, 并通过 CPE 编译器在 Kernel 入口处插入对参数的解析代码.

```
handle = dlopen(libkernel, RTLD_NOW)
kernel = dlsym(handle, "kernel_entry")
ccall((:athread_init, libkernel), Cvoid, ())
pack_arguments(a, b, c) do params
    ccall((:athread_spawn, libkernel), Cvoid,
          (Ptr{Cvoid}, Ptr{Ptr{Cvoid}}), kernel, params)
    ccall((:athread_join, libkernel), Cvoid, ())
end
dlclose(handle)
```

图 9 SACA.jl 对 Kernel 参数 pack 的代码示例

当被@saca 修饰的 Kernel 首次调用时, 由于元编程特性, swJulia 会使用 SACA.jl 中的 CPE 编译器, 将 Kernel 函数编译和链接为 CPE 动态目标码. 当该 Kernel 再次被调用时, 由于动态目标码已被加载到内存, 因此无需再次执行编译和链接操作, 此时的 Kernel 启动时间与 SACA C 实现的静态 Kernel 调用基本相当. 由于 CPE 代码通常与 MPE 代码紧耦合, 并且可能依赖于 MPE 代码中的数据类型和函数, 如果 MPE 代码中的数据类型或函数发生变化, 那么相关的 CPE 代码也需要被重新编译以确保程序的正确性.

与 MPE 编译器支持完备的 Julia 语言语法和特性不同, CPE 编译器仅支持 Julia 语言的部分核心功能, 包括基本数据类型、数组操作、函数定义、元组和命名元组、结构体、原子操作等. 而对于协程、任务、通道等复杂并发编程特性, 以及多重分派、抽象类型等高级类型系统特性, 在 CPE 编译器中的使用则会受到限制. 为了确保最佳性能和兼容性, 用户在编写 SACA Kernel 时应尽量避免复杂的控制流和过多的分支, 同时尽量减少高级数据结构和算法的使用.

### 2.3.2 SACA 封装

作为当前编译领域主流的优化方法, 基于向量化的程序变换已得到学术界的广泛研究[24,25]. SACA.jl 针对 SW26010Pro 的 SIMD 指令集特点, 提供了完备的向量类型扩展. 图 8 所示的向量加法示例中, 每个 CPE 循环执行 8 次 Float32 类型的标量加法操作, SACA.jl 能够对其进行向量优化, 优化后的 Julia 代码如图 10 所示.

```
1  # Defined in SACA.jl
2  floatv8 = NTuple{8, VecElement{Float32}}
3  # Lowered by CPE Compiler
4  function vadd(a::Vector{floatv8},
5                b::Vector{floatv8},
6                c::Vector{floatv8})
7    i = _MYID()
8    ai = unsafe_load(a, i)
9    bi = unsafe_load(b, i)
10   ci =
11    (VecElement(ai[1].value+bi[1].value),
12     VecElement(ai[2].value+bi[2].value),
13     VecElement(ai[3].value+bi[3].value),
14     VecElement(ai[4].value+bi[4].value),
15     VecElement(ai[5].value+bi[5].value),
16     VecElement(ai[6].value+bi[6].value),
17     VecElement(ai[7].value+bi[7].value),
18     VecElement(ai[8].value+bi[8].value))
19   unsafe_store!(c, ci, i)
20   return
21 end
```

图 10 CPE 编译器实施 SIMD 优化后的 Julia 代码

图 10 中, floatv8 是 SACA.jl 的内建数据类型, 表示分量个数为 8 的 Float32 向量. SACA.jl 支持的 CPE 向量扩展数据类型在表 2 中进行了详细说明, 其命名方式与 SACA C 保持一致. unsafe_load 及 unsafe_store!接口使用了元编程技术, 通过@generate 生成针对特定地址空间的 LLVM IR. 这些 IR 会在编译时注入到生成的代码中, 从而在运行时实现优化的内存访问操作. 最后, CPE 编译器将向量加法操作进行逐个元素展开, 如图 10 中第 11–18 行所示, 以生成 swLLVM 编译器 CPE 后端可识别的向量化的中间表示.

表 2 SACA.jl 支持的 CPE 向量扩展数据类型

| 类型 | 含义 | 分量表示范围 |
|---|---|---|
| intv16 | 16×32位有符号整型 | –2E31~2E31–1 |
| uintv16 | 16×32位无符号整型 | 0~2E32–1 |
| int512 | 512位有符号长整型 | — |
| uint512 | 512位无符号长整型 | — |
| floatv8 | 8×32位单精度浮点 | 1.175E–38~3.402E38 |
| doublev8 | 8×64位双精度浮点 | 2.22E–308~1.79E308 |
| halfv32 | 32×16位半精度浮点 | 6.104E–5~65 504 |

同时, 我们在 SACA.jl 中还实现了针对存储管理、异步传输、线程索引及阵列同步等功能的运行时 API 封装, 其与 SACA C API 的对应关系如表 3 所示. 在 SACA C 中, 考虑到用户编程的便捷性和高效性, 运行时 API 的实现方式较为灵活. 例如, 存储空间管理既支持通过__ldm/__cross 等关键字静态声明, 也支持通过 ldm_malloc 等接口动态分配, 而同步等对性能要求较高的接口则采用嵌汇编方式实现.

表 3 Julia 运行时 API 封装及与 SACA C API 的对应关系

| 类型 | Julia API | SACA C API |
|---|---|---|
| 存储管理 | SACAShareLDMArray(::Type{T}, dims::Tuple) | __ldm_share |
| | SACACrossArray(::Type{T}, dims::Tuple) | __cross |
| | SACAStaticLDMArray(::Type{T}, dims::Tuple) | __ldm |
| | SACADynamicLDMArray(::Type{T}, dims::Tuple) | void* ldm_malloc(size_t) |
| 异步传输 | SACADMAGet(mem_addr::Ptr{Cvoid}, ldm_addr:: Ptr{Cvoid}, len::Integer) | void athread_dma_get(void *mem_addr, void *ldm_addr, int len) |
| | SACADMAPut(mem_addr::Ptr{Cvoid}, ldm_addr:: Ptr{Cvoid}, len::Integer) | void athread_dma_put(void *mem_addr, void *ldm_addr, int len) |
| 线程索引 | _PEN() | _PEN |
| | _CGN() | _CGN |
| | _ROW() | _ROW |
| | _COL() | _COL |
| | _MYID() | _MYID |
| 阵列同步 | sync_row(mask::Integer=0xff) | CRTS_ssync_row() |
| | sync_col(mask::Integer=0xff) | CRTS_ssync_col() |
| | sync_array(mask::Integer=0xff) | CRTS_ssync_array() |

在 SACA.jl 的运行时 API 封装实现中, 我们沿袭了 SACA C 的设计理念, 针对不同接口的功能特性采取了多样化的实现策略. 图 11 展示了 SACA.jl 运行时 API 封装的 3 个典型范例. 对于诸如 SACAStaticLDMArray 等静态存储空间管理接口, 我们巧妙地运用了@generated 宏, 使得在编译阶段便能根据函数参数类型动态生成专属代码. 这种方法相较于传统的函数定义方式, 展现了一种更为灵活且高效的多重分派机制. 在处理诸如同步等实现简洁且需频繁调用的接口时, 我们借助 LLVM.jl 提供的@asmcall 宏进行定义. 通过@asmcall, Julia 代码能够直接生成底层的汇编指令, 从而显著减少因多层函数调用而产生的额外运行开销. 对于其他常规的运行时接口, 例如 SACADMAGet, 我们采用 ccall 方法直接生成 LLVM IR 层的外部函数调用, 在保证代码简洁高效的同时确保与底层系统的顺畅交互. 上述多样化的实现方案不仅有效继承了 SACA C 运行时接口的灵活性, 更在最大程度上减少了对 Julia 编译器的底层修改需求, 从而确保了整个系统的稳定性和高效性.

```
# Defined with @generated
@inline function SACAStaticLDMArray(::Type{T},
  dims::Tuple) where {T}
    N = length(dims)
    len = prod(dims)
    # @generated function emit_ldm(...)
    ptr = emit_ldm(T, Val(len))
    SACADeviceArray{T,N,AS.LDM}(ptr, dims)
end
# Defined with ccall
@inline function SACADMAPut(::Type{T},
  mem_addr::SACADeviceArray{T, N, AS0},
  ldm_addr::SACADeviceArray{T, N, AS1},
  len::Int64) where {T, N, AS0, AS1}
    return ccall("extern athread_dma_put", llvmcall,
      Cvoid, (LLVMPtr{T, AS0}, LLVMPtr{T, AS1}, Int64,),
      pointer(mem_addr), pointer(ldm_addr), len)
end
# Define with asmcall
@inline sync_array(mask::Integer=0xff) =
  @asmcall("synr \$0;sync \$0;", "r", true, Cvoid,
  Tuple{UInt32}, convert(UInt32, mask))
```

图 11 Julia 运行时 API 封装的实现方式示例

## 3 示范应用: 大规模并行量子化学模拟

在微观世界中, 描述粒子状态和运动的规律已不再遵循牛顿第二定律, 而是转向更为复杂的量子力学方程. 特别是对于分子、原子等化学体系, 其状态和演化过程严格遵循薛定谔方程. 该方程可简化为:

$$\hat{H}|\Psi\rangle = E|\Psi\rangle \tag{1}$$

其中, $\hat{H}$ 的具体形式通常包含对坐标 $x, y, z$ 的二阶导数算符等, 并可被解析地表达出来. 在此方程中, $|\Psi\rangle$和 $E$ 是待求解的未知量. $|\Psi\rangle$代表波函数, 是一个以所有电子和原子核的坐标为自变量的高维函数, 而 $E$ 则代表该化学体系的能量, 是一个具体的数值. 薛定谔方程提供了探索分子结构和光谱特性的理论框架, 在理解化学反应机制、开发新型材料以及优化药物分子设计等领域具有不可替代的重要性. 总体来看, 这是一个二阶偏微分方程. 然而, 当化学体系的规模超过一个电子和一个质子时, 我们就无法再获得解析解, 而只能通过数值方法进行求解.

求解薛定谔方程实质上等同于求解特征值方程 $HC=CE$, 其中 $H$ 是一个维度为 $2^N \times 2^N$ 的矩阵, 其特征值和特征向量分别为 $E$ 和 $C$. 依据量子力学的变分原理, 求解 $E$ 和 $C$ 的问题可以转化为优化问题, 即:

$$E(\theta) = \min_{\theta}\left[(C(\theta))^{\mathrm{T}} HC(\theta)\right] \tag{2}$$

这个待优化的函数 $(C(\theta))^{\mathrm{T}} HC(\theta)$ 可以通过量子计算机或量子模拟器进行计算. 在此过程中, $C(\theta)$ 被编码到量子线路中, 所得数值随后被提供给经典计算机, 进而使用经典算法对参数 $\theta$ 进行局部最小值的优化. 然而, 传统的量子化学方法, 如 Hartree-Fock 理论或密度泛函理论, 虽然能够提供精确的结果, 但在处理大分子系统时面临严重的可扩展性和效率问题.

神经网络量子态 (neural network quantum state, NNQS) 方法是一种利用神经网络参数化方法描述量子系统的新兴技术[26,27]. 这种方法能够捕获指数级大规模编码希尔伯特空间内的非琐碎相关性, 为量子系统的模拟提供了有效手段. NNQS 方法不仅具有强大的表示能力以捕获复杂的量子相关性, 而且可以利用神经网络的自学习特性, 通过训练持续优化波函数的表示, 从而提升模拟的准确性. 同时, 在评估 NNQS 定义的概率分布或生成样本时, 往往需要进行大量采样. 这些采样任务是高度独立的, 非常适合在大规模的分布式环境中并行执行. 基于 Transformer 的 NNQS 方法是一种创新的量子计算方法. 该方法通过引入 Transformer 神经网络作为波函数拟设, 从而实现了对量子多体波函数的高效表征. 其基本原理是利用 Transformer 网络的强大表达能力来模拟量子多体系统的基态波函数, 并通过变分蒙特卡洛算法优化这些参数以逼近真实的量子态. 在该方法中, 波函数的振幅部分由 Transformer 网络处理, 而相位部分则由多层感知机来表征, 这种分离的处理方式使模型能更有效地捕捉波函数的复杂结构.

作为量子计算领域的主流编程语言, Julia 不仅具备快速的原型设计能力, 还拥有丰富的科学计算库和基准测试工具. 同时, Julia 提供的跨语言调用机制允许无缝集成第三方库, 能够显著增强应用的功能性和适应性. 基于 Julia 编程语言, 我们在新一代神威超级计算机上设计了可扩展的量子模拟器 NNQS-Transformer[28], 以实现对大分子化学体系能量的模拟求解. 其基本结构如后文图 12 所示. 除了第 2 节中介绍的 swJulia 编译系统外, NNQS-Transformer 还包括深度学习框架 SWPyTorch、计算加速库、第三方 Julia 库以及底层文件系统等主要模块. 该模拟器采用多种方式灵活使用众核计算资源, 并结合高效的策略以在高性能计算机上实现大规模分布式并行. 其不仅能够克服传统量子化学方法的局限性, 还为大规模量子化学计算提供了一种新颖且强大的工具, 有望在量子化学和其他相关领域发挥重要作用.

### 3.1 灵活的众核加速方式

基于 Julia 灵活高效的跨语言调用机制, NNQS-Transformer 得以支持多样化的众核加速方式. 首先, 通过 PyCall.jl 的 pyimport 方法, 我们可以实现对 SWPyTorch 框架的引用. SWPyTorch 是专为新一代神威超级计算机硬件架构及其网络特性适配和优化的深度学习框架[29]. 该框架在不改变上层 API 的前提下, 通过整合众核算子库 swDNN 及 swTENSOR, 实现了对 Transformer 等深度神经网络的并行加速计算, 其调用路径如图 12 中的①号箭头所展示.

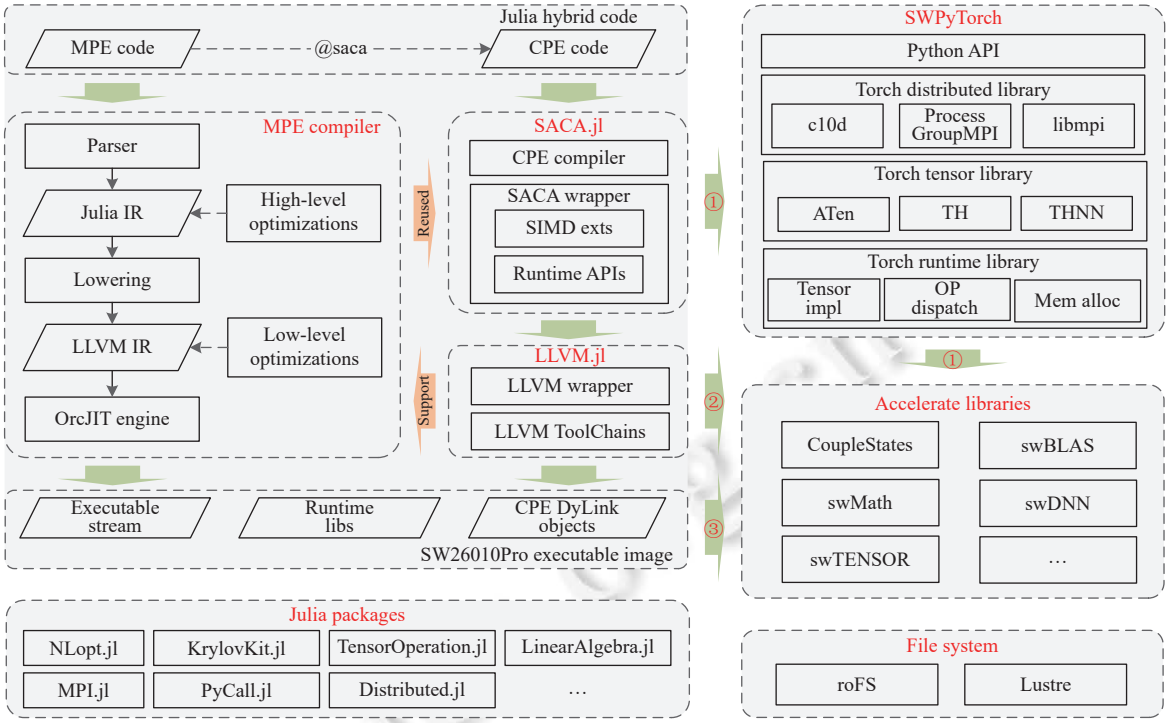



图 12 基于 Julia 的可扩展量子化学模拟器 NNQS-Transformer 组成结构

其次, Julia 通过其基础库中的 ccall 方法, 能够实现对 C/C++动态库中众核函数的直接调用, 诸如 swBLAS、swMath 等. 此方法不仅充分利用了 C 语言众核加速库的丰富资源和卓越性能, 同时保留了 Julia 编程的便捷性与灵活性. 这一调用路径在图 12 中以②号箭头标明.

最后, 对于其他支持众核优化的关键函数, 我们借助 SACA.jl 提供的@saca 宏, 在 Julia 中直接进行 SACA 众核编程. 这种方式可以灵活控制多核组计算资源和内存空间, 从而满足 NNQS-Transformer 对片内大共享空间、多级细粒度并行等高级功能的需求. 相应的调用路径在图 12 中以③号箭头清晰指示.

### 3.2 完备的第三方库适配

结合 NNQS-Transformer 的实际需求, 我们在新一代神威超级计算机上成功移植了多个 Julia 第三方库, 功能涵盖数值计算、分布式通信及外部函数调用等多个领域. 同时, 我们对相关的底层支撑环境进行了专门的适配和优化. 相关第三方库及其底层支撑环境如表 4 所示.

表 4 NNQS-Transformer 依赖的第三方 Julia 库及其底层支撑环境

| 类别 | Julia库 | 底层支撑环境 |
|---|---|---|
| 数值计算 | NLopt.jl | libnlopt.so |
| | KrylovKit.jl | — |
| | TensorOperations.jl | — |
| | LinearAlgebra.jl | libswblas.so, libswlapack.so |
| 分布式通信 | Distributed.jl | — |
| | MPI.jl | libmpi.so |
| 外部调用 | PyCall.jl | swPython解释器 |

在数值计算方面, 我们引入了 NLopt.jl[30], 其底层基于 libnlopt.so 实现, 能够应对从简单的数值最小化到复杂的优化约束问题. 此外, KrylovKit.jl[31], 一个纯 Julia 语言实现的开源项目, 也被纳入我们的库体系中. 它提供了多

种基于 Krylov 空间的算法, 能有效解决线性系统问题、计算特征值、奇异值, 以及处理函数应用于线性映射的问题. 同样, TensorOperations.jl[32], 这一旨在加速并简化 Julia 语言中高维数组操作的开源包, 也是我们的重要工具. 而 LinearAlgebra.jl 则对 swblas 及 swlapack 等众核线性代数库中的常用接口提供了高效封装, 能够轻松应对矩阵计算、高斯消元、LU 分解等多种线性代数问题.

在分布式通信方面, 我们采用了 Distributed.jl[33]和 MPI.jl[34]两种方式. Distributed.jl 允许我们在独立的内存空间中创建新的进程, 其主从分布式架构使得小规模并行化变得简洁方便. 而 MPI.jl 则是消息传递接口 (message passing interface, MPI) 协议的 Julia 封装, 并且针对 SW26010Pro 进行了高度优化. 它灵活支持各种 MPI 实现, 并且提供与 C 函数名称几乎相同的接口, 使得通过 Julia 使用 MPI 变得简单易行.

为了充分利用 Julia 的高性能和 Python 的丰富生态, 我们还引入了 PyCall.jl[35]这一核心包. 它允许用户无缝地在 Julia 中调用外部 Python 函数. PyCall.jl 底层依赖于新一代神威超级计算机上的 swPython 解释器, 该解释器为申威处理器提供了包括 AI 框架、数值计算、分布式通信、高性能应用工具等在内的上百个第三方库, 从而满足了人工智能和科学计算的高性能需求. 值得一提的是, PyCall.jl 通过使用 Cython 编译器生成的 C API 直接与 swPython 的动态类型系统交互, 确保了调用 Python 函数时的低延迟和高效率.

### 3.3 高效的文件系统支撑

新一代神威超级计算机的计算节点不配备本地存储, 而是依赖基于 Lustre 的全局文件系统[36]提供存储服务. 当 NNQS-Transformer 执行时, 每个计算节点需要加载大约 500 个动态库和预编译文件, 总数据量高达约 1 GB. 在大规模并行文件读写的应用场景下, Lustre 的元数据服务器可能成为性能瓶颈.

为了解决这个问题, 新一代神威超级计算机同时配备了只读文件系统 (read-only file system, ROFS). ROFS 在多个 SSD 存储服务节点上创建虚拟块设备, 并将应用的动态库和预编译文件等复制到这些虚拟设备中. 随后, 这些虚拟块设备通过网络以只读模式输出到计算节点. 当程序执行时, 应用会从这些远程虚拟块设备中加载所需的可执行程序、动态库和预编译文件.

值得一提的是, Julia 编译器支持“方法年龄”的概念, 这使得动态方法能够重新定义. 每当源代码片段被编辑时, 其“年龄”就会发生变化, 进而触发预编译机制以生成新的预编译文件. 在将数据复制到虚拟块设备之前, 我们首先在基于 Lustre 的可读写块设备中完成预编译操作. 随后, 将 ROFS 的虚拟块设备挂载到相同路径, 从而有效避免了在 ROFS 中执行写操作的问题.

在新一代神威超级计算机上, 我们配置了 600 个数据服务器作为虚拟块的 Target 端, 同时约有 9 万个计算节点作为 Host 端. 这些 Host 端通过哈希方式映射到这 600 个 Target 端上. 根据实际测试数据, 采用 ROFS 后, 整机规模的动态库和预编译文件加载时间从原本的 22 min 显著缩短至约 3 min.

## 4 实验与分析

### 4.1 实验环境与测试用例

本文以新一代神威超级计算机为实验平台, 借助前端机的作业管理系统, 将可执行文件提交至 SW26010Pro 计算节点运行. 实验中采用的 swJulia 编译器基于 Julia 1.8.1 开源版本进行移植, 并依托于 swLLVM 18.0.1 的 ORCJIT 引擎构建而成. 同时, SW26010Pro 的原生编译器采用 swGCC 7.1.0 版本, 其底层汇编器、链接器等二进制工具链版本为 2.24, GLIBC 基础库的版本为 2.23.

Micro-Benchmarks[37]是一套用于测试 Julia 编译器性能的基准测试集, 它涵盖了多种常见的代码模式, 如函数调用、字符串解析、排序、数值循环、随机数生成、递归和数组操作等. 在该测试集中, 测试课题被设计为使用不同编程语言实现相同的算法模式, 并在相同的软件环境和硬件条件下执行, 以确保测试的公平性. 本文使用 Micro-Benchmarks 对比了 Julia 语言相对于其他编程语言的性能, 并将测试结果以 C 语言实现为基准进行归一化处理, 以验证 swJulia 编译器在解决“两种语言”问题上的有效性.

本文采用 Rodinia[38]测试集以全面评估 swJulia 在众核加速及运行时开销等方面的表现. Rodinia 是针对并行

和异构计算平台设计的标准化测试集, 其中包含多个计算密集型和内存密集型的应用程序, 常被用于评估 GPU、多核 CPU 及其他异构计算设备的性能. Rodinia 中的测试用例均来源于实际问题, 因此能够有效代表现实中可能遇到的计算挑战. 通过移植 Rodinia v3.1 版本中的 Julia 用例[39]至 SACA.jl, 我们能够系统地评估 swJulia 在新一代神威超级计算机上的编程成熟度, 并深入探究 SACA.jl 在生成高效众核代码方面的能力. 这一研究路径不仅有助于量化 swJulia 的性能表现, 更能为未来的优化工作提供依据.

针对示范应用, 本文选取了 4 个典型分子系统作为测试用例, 以全面评估 NNQS-Transformer 在新一代神威超级计算机上的模拟效果. 具体而言, 包括 $C_2$、LiCl、$C_2H_4O$ 这 3 个小分子系统, 以及 Fenton 反应的 $Fe(H_2O)_5H_2O_2$ 大分子系统 (包含 52 个量子位数和 46 个电子). 通过这些具有挑战性的测试用例, 我们能够有效地验证 swJulia 在支撑大规模量子化学模拟时的性能潜力及可扩展性.

### 4.2 swJulia 编译器性能测试

为了全面展示 swJulia 编译系统的能力, 我们进行了 3 个方面的评估. 首先, 我们在神威平台上使用 Micro-Benchmarks 对比了 Julia 与其他主流编程语言的性能, 以验证 MPE 编译器的高效性. 其次, 我们对比并分析了 Rodinia 中单线程 Julia 测试课题在 x86 和神威平台上的性能, 以评估使用 swJulia 的运行时开销. 最后, 我们使用 SACA.jl 对单线程 Julia 测试课题实施了初步的众核优化, 并且在 SW26010Pro 上测试并统计了性能加速比以及代码修改行数, 以展示 SACA.jl 在众核编程上的便捷性.

#### 4.2.1 MPE 编译器性能测试

图 13 展示了 Julia 与其他主流编程语言针对常见操作的性能对比, 其中, Fortran、Python 和 Julia 以 C 为基准的平均相对执行时间分别为 1.50、77.71 以及 2.12. C 和 Fortran 均属于静态编程语言, 需要经过 swGCC 编译器将源代码转换为机器码. swGCC 提供了丰富的 AOT 编译优化, 能够根据程序特点进行针对性的代码变换及数据排布, 因此整体性能相对较优. Python 是典型的解释型语言, 其每次执行前都需要通过解释器将源代码转换为字节码. 同时, Python 解释器在执行每条语句时都需要进行类型检查, 以动态确定对象的类型. 上述转换和检查等操作会引入额外的开销, 导致程序启动和执行时间的增加, 因此在执行效率上通常不如编译型语言. Julia 结合了编译型和解释型语言的优点, 既通过多重分派机制支持动态的泛型编程, 又采用 JIT 机制根据实际参数类型实施专门优化, 因此在执行效率上接近 C 语言.

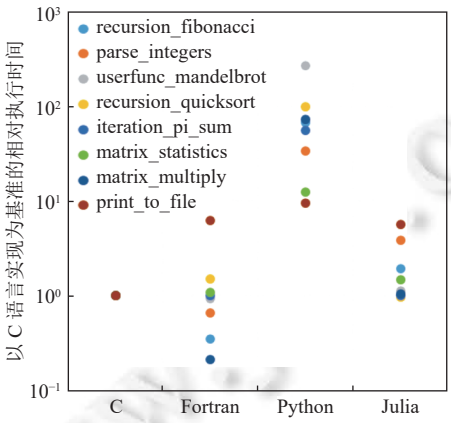



图 13 Julia 与其他语言针对常见操作的性能对比

Julia 的 JIT 机制会在函数首次执行时进行编译, 主要流程包括词法分析、语法分析以及类型推断等, 并根据推断得到的类型信息生成特例化的代码. 同时, 在函数运行过程中, Julia 的垃圾回收 (garbage collection, GC) 机制会自动清理不再使用的内存. 在首次运行时, 由于需要初始化内存并跟踪内存回收, 因此垃圾回收的开销相对较高. 考虑到不同处理器在指令集、硬件结构以及峰值性能等方面的差异性, 仅比较 Julia 程序在不同系统上的执行

时间难以全面反映 JIT 编译的性能. 本文通过对 Julia 函数首次执行时的 JIT 编译和垃圾回收时间占比等指标进行分析, 能够有效屏蔽硬件差异性等影响因素, 从而较为准确地评估 MPE 编译器的性能.

Julia 在标准库中提供了@time 宏, 用于测量代码块的执行时间, 并统计编译和垃圾回收的时间占比. 本文使用@time 分别在 x86 和神威平台上对单线程 Julia 测试课题的编译和垃圾回收时间占比进行了统计, 结果如图 14 和图 15 所示. 其中, x86 表示新一代神威超级计算机的前端机, 其配备了 24 核心的 Intel Xeon E5-2440 处理器 (2.40 GHz), SW 则表示新一代神威超级计算机的 SW26010Pro 处理器. 由于 particlefilter、pathfinder 及 lud 这 3 道课题的执行时间较短, @time 无法准确统计编译和垃圾回收的时间占比, 因此未展示相关数据.

编译时间是评估 JIT 编译器效率的关键指标. 较短的编译时间意味着在交互式运行环境中, 开发者可以快速看到代码更改的效果, 而在具有确定性调用行为的脚本运行方式下, 则可以减少程序的启动时间. 如图 14 所示, SW 的平均编译时间占总时间的比例为 27.20%, 略高于 x86 的 22.02%. 其中, 除 nw 外的大部分 Julia 基准测试课题, SW 的编译时间占比与 x86 相当. 由于 nw 课题核心段存在大量循环及条件判断, 在 JIT 编译阶段引入了复杂的控制流分析及优化操作, 导致代码生成时产生大量用于分支跳转的控制基本块, 相较于计算密集型课题的 JIT 编译过程, 更加依赖离散内存访问的性能. 考虑到 SW26010Pro 采用精简设计的异构融合架构, 其 MPE 的 Cache 容量远低于 Intel Xeon E5-2440, 导致 SW 的编译时间占比进一步提高.

Julia 的垃圾收集器采用了分代垃圾收集算法, 即根据对象的存活时间将其分为不同的代. 新创建的对象通常被分配到第一代, 而经过多次扫描后仍然存活的对象则会被移至下一代, 这种方式有助于提高垃圾收集的效率, 但也可能对程序性能产生一定影响. 首先, 垃圾回收过程需要占用 CPU 时间来扫描和回收内存中的无用对象, 在高负载情况下, 这可能会降低应用程序的吞吐量. 其次, 为了进行垃圾回收, 系统可能需要暂停应用程序的执行. 这种暂停虽然短暂, 但可能影响用户体验, 尤其是在对实时性要求较高的应用中. 最后, 垃圾回收器在工作时可能需要额外的内存空间, 从而增加了内存的占用. 如图 15 所示, SW 的平均垃圾回收时间占比为 6.07%, 略低于 x86 的 8.49%. 其中, nn、bfs 和 hotspot 这 3 道课题在其核心段内部声明了多个数组等大型数据结构的局部变量, 当离开其作用域时, 垃圾回收器会更加积极地回收它们, 以释放更多内存. 因此, 上述 3 道课题的 GC 时间占比相对略高.

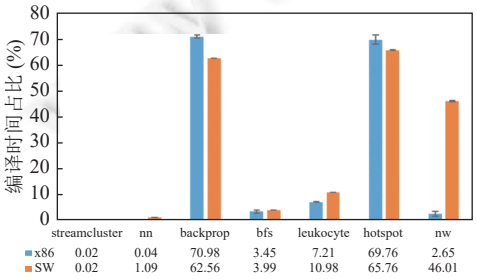



图 14 Julia 基准测试课题中的编译时间占比

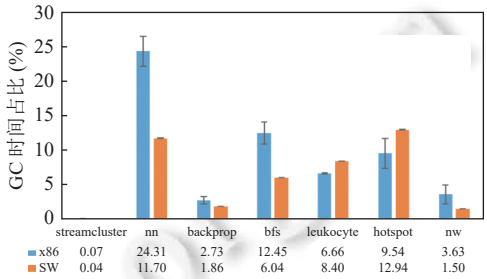



图 15 Julia 基准测试课题中的垃圾回收时间占比

通过上述分析, 在排除硬件影响因素的前提下, swJulia 的 MPE 编译器在 JIT 编译和垃圾回收方面的性能与 x86 平台的开源 Julia 实现基本相当, 因此能够有效满足 Julia 应用程序对实时性和性能的要求.

#### 4.2.2 SACA.jl 性能测试

在执行静态编译的 SACA C 代码时, 启动 Kernel 的运行时成本完全由加速线程库和底层硬件决定. 而在使用 SACA.jl 的情况下, 如第 2.2 节所述, 启动 Kernel 需要执行一系列的任务以实现高度的动态性. 为了评估运行时开销, 我们使用 SACA API 执行一个空的 Kernel, 并且通过嵌汇编的方式测量其节拍数以估算运行时间, 结果如图 16 所示.

可以看出, 不论采用何种编译和运行方式, Kernel 的运行时开销均会随着从核数量的增加而近似线性上升. 当从核数量较多时, 访存压力增加所导致的性能开销就会逐渐凸显. 其中, 使用静态编译的 SACA C 性能最优, 64 从核情况下的开销仅为 24.74 μs. 动态编译的 SACA C 在从核数量较少时, 开销与静态编译接近, 但是随着从核数量

的增加, 差距逐渐增大, 这主要是由于动态编译所依赖的地址位置无关要求的限制, 许多访存优化方法无法有效应用, 例如!gprelhigh/!gprellow 重定位等. SACA.jl 以动态编译为基础, 并且引入了包括 Kernel 函数实例化、方法年龄匹配、参数 Pack/Unpack 等一系列运行时操作. 上述操作的开销与从核数量的相关性较小, 因此, 图 16 中的 SACA.jl 曲线在动态编译的 SACA C 基础上整体上移, 平均增幅约为 27.67 μs. 考虑到真实 Kernel 的计算时间都在百万 μs 量级以上, 因此上述运行时开销对 Julia 程序整体执行时间的影响可以忽略不计.

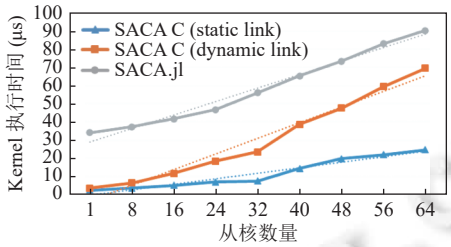



图 16 Kernel 启动的运行时开销随从核数量的变化

为了评估使用 Julia 编写的 SACA Kernel 的原始性能, 我们将 Rodinia 基准测试集中的多个单线程 Julia 测试课题移植到了 SACA.jl 上, 并使用 SACA 封装进行了初步的众核优化. 图 17 展示了使用 SACA.jl 实现的 Kernel 函数相较于 MPE 单线程版本的性能加速比. 对于拥有多个 Kernel 的测试课题, 其总执行时间为多个 Kernel 的平均执行时间之和. 如图 17 所示, 所有基准测试课题经过 SACA.jl 优化后均获得了性能提升, 平均加速比为 31.46. 其中, streamcluster 课题获得了最高的 97.92 的超线性加速比, 这一方面是由于其 Kernel 函数的数据依赖关系简单、可并行性较好, 另一方面得益于 CPE 的 LDCache 以及 LDM Stack 等优化机制的使用. backprop 和 lud 两道课题的优化效果并不理想, 主要是由于其 Kernel 函数的实现复杂且计算密度较低, 特别是对于 backprop 课题, 如图 14 和图 15 所示, 其 MPE 单线程版本在神威上的编译及垃圾回收的时间占比高达 64.42%, 因此难以充分发挥众核的计算潜力.

基于 Julia 的元编程机制以及高层 API 封装, SACA.jl 在提升 Julia 程序性能同时并不会引入复杂的代码修改工作. 图 18 展示了使用 SACA.jl 优化后的 Julia 基准测试课题的代码修改行数, 并使用不同颜色对修改的范围进行了区分. 其中, “@saca”表示使用@saca 宏将 Kernel 函数加载至从核阵列运行, “Loop tiling”表示将 Kernel 函数的核心计算部分映射到不同从核线程, “Runtime API”表示使用第 2.3.2 节介绍的 SACA 运行时 API, 例如通过 SACAShareLDMArray 申请从核阵列的 LDM 共享空间等, “Import SACA”表示对 SACA.jl 的引用. 如图 18 所示, 所有 Julia 测试课题的平均代码修改行数为 7, 特别地, streamcluster 课题在仅修改 3 行代码的情况下获得了 97.92 倍的性能提升. 仅从代码修改行数上分析, 使用 SACA.jl 与 CUDA.jl 在并行应用的开发难度上并没有较大差别. 然而, 由于 SACA.jl 基于 SACA 编程模型设计, 并且进行了针对性的优化, 因此在国产众核平台上能够显著提升性能, 实现更优的运行效果.

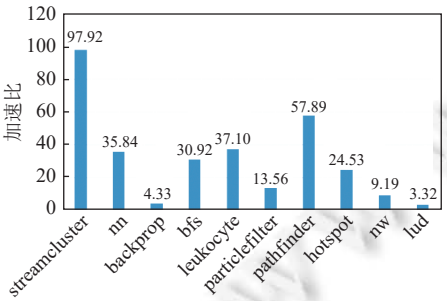



图 17 使用 SACA.jl 优化后的 Kernel 性能加速比

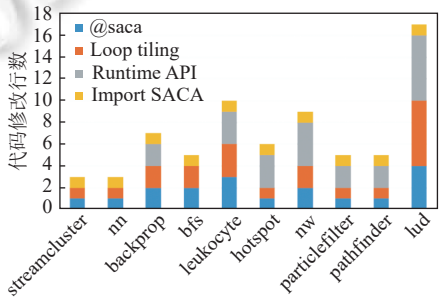



图 18 使用 SACA.jl 优化后的代码修改行数

综上所述, 借助于 SACA.jl, 我们能够在尽可能接近原始 SACA C 性能的同时, 享受到 Julia 语言带来的编程便利性, 这使得使用 Julia 在神威新一代超级计算机上进行众核编程成为一种既便捷又高效的选择.

### 4.3 示范应用优化效果测试

本文基于 swJulia 实现了对量子化学模拟器 NNQS-Transformer 的支撑和优化. 一方面, 我们借助 swJulia 灵活的众核加速方式, 对热点函数进行了充分的众核优化, 以提升模拟器在单 CPU 上的计算效率. 另一方面, 我们使用 MPI.jl 实现了 NNQS-Transformer 在新一代神威超级计算机上的大规模可扩展并行模拟, 以充分继承原生 MPI 库针对神威硬件特性的定制优化.

在 SACA 编程模型中, 我们设计了一种全局共享执行模式, 从全芯片的角度实现众核加速编程. 为了最大化资源利用率, 我们让每个 CPU 执行单个进程, 并赋予其对所有芯片资源的独占访问权限. 这包括整个 96 GB 的全局共享内存空间以及处理器阵列中所有 384 个 CPE 的集体计算能力. 通过最小化每个 CPU 内部的进程间通信, 这种方法提高了整体处理器效率, 使用户能够充分利用硬件的潜力. 图 19 展示了在全片共享模式下, NNQS-Transformer 核心算子相较于 MPE 单线程版本的性能优化效果, 3 种典型分子系统的平均性能加速比为 35.52. 其中, $C_2$ 的加速比仅为 6.81, 远低于其他两个系统. 为了深入了解限制性能的因素, 我们使用 Roofline 模型进行了分析. 从图 20 中可以看出, 代表 $C_2$ 的数据标记点位于 Roofline 曲线的下方. 在这个区域中, 增加处理器的计算能力并不会显著提高性能, 因为内存带宽已经成为瓶颈. 在后续的应用优化中, 除了提升访存效率外, 我们还需要进一步利用底层硬件的特点, 有效地排布热点数据以保证片上存储空间的利用率, 合理地安排 CPE 之间的通信模式以降低同步开销, 通过 LDM 双缓冲等机制隐藏 DMA 异步操作的延迟, 从而最大限度地释放 SW26010Pro 处理器异构众核架构的计算潜能.

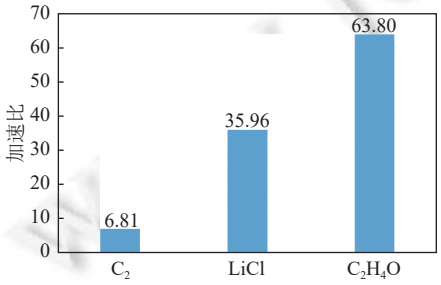



图 19 NNQS-Transformer 核心算子的优化效果

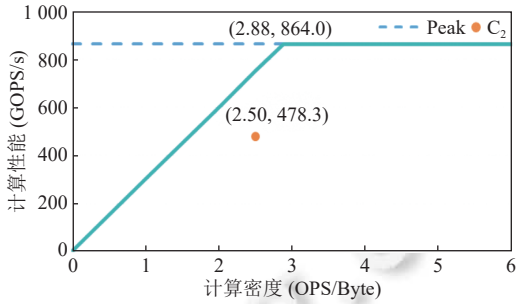



图 20 $C_2$ 体系的核心算子的 Roofline 模型

NNQS-Transformer 采用多层次的 MPI 任务分发及数据并行化的分布式计算策略, 以实现在新一代神威超级计算机上的大规模可扩展量子化学模拟. 以图 21 所示的 $Fe(H_2O)_5H_2O_2$ 大分子系统为例, NNQS-Transformer 采用批量自回归采样方法高效地生成大量独特样本. 这些样本随后被划分至多个 MPI 组, 每个 MPI 组内包含若干个工作进程, 负责独立进行局部能量评估. 由于各 MPI 组间无需频繁通信, 减少了全局同步的需求, 因此显著提升了并行度. 各 MPI 组定期汇总梯度信息并通过 Allreduce 操作将结果发送至主进程. 主进程根据汇总的梯度更新模型参数, 并将最新参数广播至所有工作进程. 这一过程不断重复, 直至达到预定精度或最大迭代次数, 从而实现对整机规模计算资源的有效利用.

为了充分展示 swJulia 对分布式计算的支持和优化效果, 我们在新一代神威超级计算机上进行了可扩展性测试, 最大并行规模扩展到约 10 万个节点. 图 22 给出了强可扩展性测试结果. 对于 $Fe(H_2O)_5H_2O_2$ 体系, 随着使用的节点数从 1.4 万增加到 4.4 万, 并行效率均高于 70%, 而当节点数增加到 9.7 万时, 效率降低到 46.6%. 并行效率的降低源于每个 MPI 组中生成的唯一样本数量的不平衡, 这可能是由于物理约束导致的样本随机剪枝所产生的影响. 我们还测试了 NNQS-Transformer 的弱可扩展性, 结果如图 23 所示. 对于 $Fe(H_2O)_5H_2O_2$ 体系, 在确保每个 MPI 组处理大致均匀的唯一样本子集的前提下, 随着节点数从 0.1 万增加到 4.8 万, 并行效率均高于 92%, 而当节

点数增加到 9.7 万时, 效率降低到 66.2%. 并行效率下降的主要原因是不同 MPI 组之间的随机种子变化, 这导致某些种子生成特殊的唯一采样路径, 从而在采样过程中造成工作负载分布的不均衡.

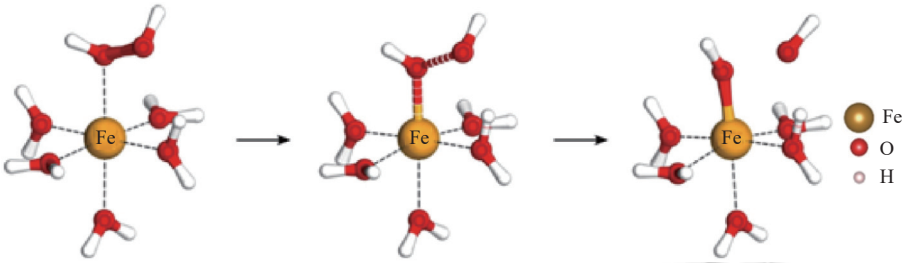



图 21 Fenton 反应的 $Fe(H_2O)_5H_2O_2$ 大分子系统的化学结构

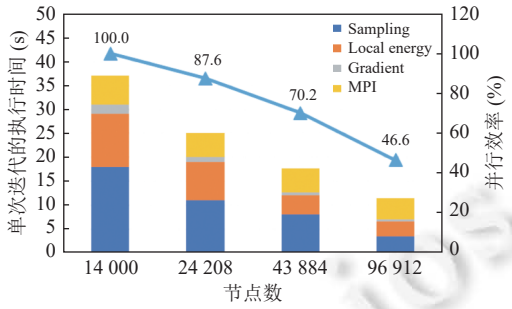



图 22 NNQS-Transformer 的强可扩展性测试结果

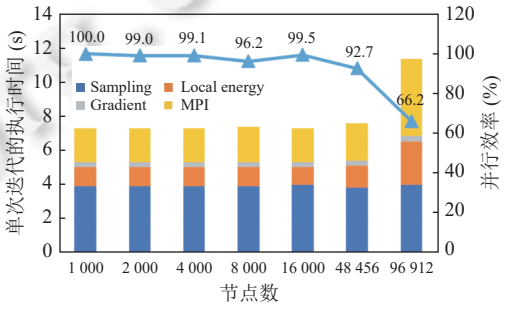



图 23 NNQS-Transformer 的弱可扩展性测试结果

实验结果表明, swJulia 的 MPI.jl 实现了对新一代神威超级计算机消息传递接口的高效封装, 为 NNQS-Transformer 在分布式环境中的高效执行提供了有力支撑. 同时, 得益于 Julia 在分布式计算和并行处理方面的强大支持, NNQS-Transformer 能够采用专门设计的多层并行化策略, 如 MPI 进程分组以及数据驱动的并行化方法等, 上述策略显著提升了 NNQS-Transformer 在大规模量子化学模拟中的计算效率和可扩展性.

## 5 结束语

本文通过对新一代神威超级计算机即时编译需求的深入研究, 创新设计并实现了动态语言编译器 swJulia. 该编译器显著提高了 Julia 在新一代神威超级计算机上的执行效率和灵活性. 针对 SW26010Pro 处理器的硬件特点, 我们成功构建了 ORCJIT 引擎, 并提出了动态模式下的片上存储空间管理方法. 这些技术突破为 Julia 等动态语言在国产众核处理器上的流畅运行奠定了坚实的技术基础. 更进一步, 我们通过支持 Julia 的原生众核编程和 SACA 运行时封装, 展示了 swJulia 编译系统的强大功能. 值得一提的是, 我们基于 swJulia 成功在新一代神威超级计算机上支撑并优化了 NNQS-Transformer 量子化学模拟器, 实现了前所未有的超大规模可扩展并行模拟, 并取得了多项国际领先的应用成果.

未来, 我们将持续推进 swJulia 的研发工作, 旨在进一步提升 Julia 在新一代神威超级计算机上的性能表现. 首先, 我们将加强针对特定领域应用的定制优化, 通过精准对接应用需求和硬件特性, 设计更加高效的众核编程和加速方案. 其次, 我们将扩大对第三方库的适配和优化范围, 以丰富 swJulia 的软件生态, 为用户提供更加全面的编程支持. 最后, 我们将探索应用更先进的编译优化技术, 例如利用机器学习进行代码优化、开发更高效的并行处理策略等, 从而不断提升编译系统的智能水平和自动化程度. 同时, 我们也将致力于推动 swJulia 在更广泛领域的应用, 为基于国产超级计算机的科研与工程实践提供强有力的技术支持.

**References:**

[1] Top500 the list. 2024. https://www.top500.org

[2] Fu HH, Liao JF, Yang JZ, Wang LN, Song ZY, Huang XM, Yang C, Xue W, Liu FF, Qiao FL, Zhao W, Yin XQ, Hou CF, Zhang CL, Ge W, Zhang J, Wang YG, Zhou CB, Yang GW. The Sunway TaihuLight supercomputer: System and applications. Science China Information Sciences, 2016, 59(7): 072001. [doi: 10.1007/s11432-016-5588-7]

[3] Darema F. New software technologies for the development and runtime support of complex applications. The Int'l Journal of High Performance Computing Applications, 1999, 13(3): 180–190. [doi: 10.1177/109434209901300302]

[4] Laros III JH, Kelly SM, Levenhagen MJ, Pedretti K. Investigating methods of supporting dynamically linked executables on high performance computing platforms. Technical Report, SAND2009-5515. Albuquerque: Sandia National Laboratories (SNL), 2009. [doi: 10.2172/1035350]

[5] Beazley DM, Ward BD, Cooke IR. The inside story on shared libraries and dynamic loading. Computing in Science & Engineering, 2001, 3(5): 90–97. [doi: 10.1109/5992.947112]

[6] Bezanson J, Karpinski S, Shah V, Edelman A. Julia: A fast dynamic language for technical computing. arXiv:1209.5145, 2012.

[7] Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: A fresh approach to numerical computing. SIAM Review, 2017, 59(1): 65–98. [doi: 10.1137/141000671]

[8] Gao JG, Zheng F, Qi FB, Ding YJ, Li HL, Lu HS, He WQ, Wei HM, Jin LF, Liu X, Gong DY, Wang F, Zheng Y, Sun HH, Zhou Z, Liu Y, You HT. Sunway supercomputer architecture towards exascale computing: Analysis and practice. Science China Information Sciences, 2021, 64(4): 141101. [doi: 10.1007/s11432-020-3104-7]

[9] The Julia language. 2024. https://github.com/JuliaLang/julia

[10] Jansen P. TIOBE index for April 2025. 2024. https://www.tiobe.com/tiobe-index/

[11] CUDA.jl. 2024. https://github.com/JuliaGPU/CUDA.jl

[12] Besard T, Foket C, De Sutter B. Effective extensible programming: Unleashing Julia on GPUs. IEEE Trans. on Parallel and Distributed Systems, 2019, 30(4): 827–841. [doi: 10.1109/TPDS.2018.2872064]

[13] Rackauckas C, Nie Q. DifferentialEquations.jl—A performant and feature-rich ecosystem for solving differential equations in Julia. Journal of Open Research Software, 2017, 5(1): 15. [doi: 10.5334/jors.151]

[14] Mogensen PK, Riseth AN. Optim: A mathematical optimization package for Julia. Journal of Open Research Software, 2018, 3(24): 615. [doi: 10.21105/joss.00615]

[15] FinEtools: Finite element tools in Julia. 2024. https://github.com/PetrKryslUCSD/FinEtools.jl

[16] Aho J, Vuotikka AJ, Frondelius T. Introduction to JuliaFEM, an open source FEM solver. Rakenteiden Mekaniikka, 2019, 52(3): 148–159. [doi: 10.23998/rm.75103]

[17] Pawar S, San O. CFD Julia: A learning module structuring an introductory course on computational fluid dynamics. Fluids, 2019, 4(3): 159. [doi: 10.3390/fluids4030159]

[18] Innes M. Flux: Elegant machine learning with Julia. Journal of Open Source Software, 2018, 3(25): 602. [doi: 10.21105/joss.00602]

[19] Luo XZ, Liu JG, Zhang P, Wang L. Yao.jl: Extensible, efficient framework for quantum algorithm design. Quantum, 2020, 4: 341. [doi: 10.22331/q-2020-10-11-341]

[20] Liu Y, Liu X, Li F, Fu HH, Yang YL, Song JW, Zhao PP, Wang Z, Peng DJ, Chen HR, Guo C, Huang HL, Wu WZ, Chen DX. Closing the "quantum supremacy" gap: Achieving real-time simulation of a random quantum circuit using a new Sunway supercomputer. In: Proc. of the 2021 Int'l Conf. for High Performance Computing, Network, Storage and Analysis. Saint Louis: IEEE, 2021. 1–12. [doi: 10.1145/3458817.3487399]

[21] Kulkarni PA. JIT compilation policy for modern machines. In: Proc. of the 2011 ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications. Portland: ACM, 2011. 773–788. [doi; 10.1145/2048066.2048126]

[22] Shen L, Zhou WH, Wang F, Xiao Q, Wu WH, Zhang LF, An H, Qi FB. swLLVM: Optimized compiler for new generation Sunway supercomputer. Ruan Jian Xue Bao/Journal of Software, 2023, 35(5): 2359–2378 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6896.htm [doi: 10.13328/j.cnki.jos.006896]

[23] Wu W, Qian H, Zhu Q, Wang J, Fan XJ. Research on full-chip programming for Sunway heterogeneous many-core processor. In: Proc. of the 3rd World Symp. on Software Engineering. Xiamen: ACM, 2021. 174–179. [doi: 10.1145/3488838.3488868]

[24] Nuzman D, Zaks A. Outer-loop vectorization-revisited for short SIMD architectures. In: Proc. of the 2008 Int'l Conf. on Parallel Architectures and Compilation Techniques. Toronto: IEEE, 2008. 2–11.

[25] Porpodas V. Supergraph-SLP auto-vectorization. In: Proc. of the 26th Int'l Conf. on Parallel Architectures and Compilation Techniques. Portland: IEEE, 2017. 330–342. [doi: 10.1109/PACT.2017.21]

[26] Torlai G, Mazzola G, Carrasquilla J, Troyer M, Melko R, Carleo G. Neural-network quantum state tomography. Nature Physics, 2018, 14(5): 447–450. [doi: 10.1038/s41567-018-0048-5]

[27] Shang HH, Shen L, Fan Y, Xu ZQ, Guo C, Liu J, Zhou WH, Ma H, Lin RF, Yang YL, Li F, Wang ZY, Zhang YQ, Li ZY. Large-scale simulation of quantum computational chemistry on a new Sunway supercomputer. In: Proc. of the 2022 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Dallas: IEEE, 2022. 1–14. [doi: 10.1109/SC41404.2022.00019]

[28] Wu YJ, Guo C, Fan Y, Zhou PY, Shang HH. NNQS-Transformer: An efficient and scalable neural network quantum states approach for ab initio quantum chemistry. In: Proc. of the 2023 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Denver: ACM, 2023. 42. [doi: 10.1145/3581784.3607061]

[29] Liu S, Gao J, Liu X, Huang ZQ, Zheng TY. Establishing high performance AI ecosystem on Sunway platform. CCF Trans. on High Performance Computing, 2021, 3(3): 224–241. [doi: 10.1007/s42514-021-00072-x]

[30] NLopt.jl. 2024. https://github.com/JuliaOpt/NLopt.jl

[31] KrylovKit.jl. 2024. https://github.com/Jutho/KrylovKit.jl

[32] TensorOperations.jl. 2024. https://github.com/Jutho/TensorOperations.jl

[33] Distributed. 2024. https://github.com/JuliaLang/Distributed.jl

[34] Byrne S, Wilcox LC, Churavy V. MPI.jl: Julia bindings for the message passing interface. Proc. of JuliaCon, 2021, 1(1): 68.

[35] PyCall. 2024. https://github.com/JuliaPy/PyCall.jl

[36] Schwan P. Lustre: Building a file system for 1000-node clusters. In: Proc. of the 2003 Linux Symp. 2003. 380–386.

[37] Micro-Benchmarks. 2024. https://github.com/JuliaLang/Microbenchmarks

[38] Che S, Boyer M, Meng JY, Tarjan D, Sheaffer JW, Lee SH. Rodinia: A benchmark suite for heterogeneous computing. In: Proc. of the 2009 IEEE Int'l Symp. on Workload Characterization (IISWC). Austin: IEEE, 2009. 44–54. [doi: 10.1109/IISWC.2009.5306797]

[39] Rodinia, Single thread Julia. 2024. https://github.com/JuliaParallel/rodinia/julia_st

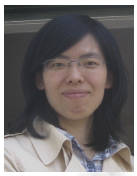
## 附中文参考文献:

[22] 沈莉, 周文浩, 王飞, 肖谦, 武文浩, 张鲁飞, 安虹, 漆锋滨. swLLVM: 面向神威新一代超级计算机的优化编译器. 软件学报, 2024, 35(5): 2359–2379. http://www.jos.org.cn/1000-9825/6896.htm [doi: 10.13328/j.cnki.jos.006896]

**沈莉**(1981－), 女, 博士生, 主要研究领域为编译系统, 编译优化.

**谭坚**(1985－), 男, 博士生, 主要研究领域为高性能计算, 人工智能处理器性能优化.

**周文浩**(1991－), 男, 博士生, 主要研究领域为异构众核编程模型, 编译系统.



**商红慧**(1984－), 女, 博士, 教授, 主要研究领域为科学软件开发, 高性能计算.

**王飞**(1981－), 男, 博士生, 主要研究领域为编译优化, 众核编程环境.

**安虹**(1963－), 女, 博士, 教授, 主要研究领域为并行计算系统, 众核芯片架构.

**李斌**(1991－), 男, 工程师, 主要研究领域为异构众核编译系统.

**漆锋滨**(1966－), 男, 博士, 正高级工程师, CCF 会士, 主要研究领域为高性能计算体系结构, 编译优化.