软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(5):2359−2378 [doi: 10.13328/j.cnki.jos.006896]

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

swLLVM: 面向神威新一代超级计算机的优化编译器*

沈 莉[1], 周文浩[2], 王 飞[4], 肖 谦[1], 武文浩[2], 张鲁飞[4], 安 虹[1], 漆锋滨[3]

[1](中国科学技术大学, 安徽 合肥 230026)

[2](国家并行计算机工程技术研究中心, 北京 100190)

[3](江南计算技术研究所, 江苏 无锡 214083)

[4](清华大学, 北京 100084)

通信作者: 漆锋滨, E-mail: qifb116@sina.com

摘 要: 异构众核架构具有超高的能效比, 已成为超级计算机体系结构的重要发展方向. 然而, 异构系统的复杂性给应用开发和优化提出了更高要求, 其在发展过程中面临好用性和可编程性等众多技术挑战. 我国自主研制的神威新一代超级计算机采用了国产申威异构众核处理器 SW26010Pro. 为了发挥新一代众核处理器的性能优势, 支撑新兴科学计算应用的开发和优化, 设计并实现面向 SW26010Pro 平台的优化编译器 swLLVM. 该编译器支持 Athread 和 SDAA 双模态异构编程模型, 提供多级存储层次描述及向量操作扩展, 并且针对 SW26010Pro 架构特点实现控制流向量化、基于代价的节点合并以及针对多级存储层次的编译优化. 测试结果表明, 所设计并实现的编译优化效果显著, 其中, 控制流向量化和节点合并优化的平均加速比分别为 1.23 和 1.11, 而访存相关优化最高可获得 2.49 倍的性能提升. 最后, 使用 SPEC CPU2006 标准测试集从多个维度对 swLLVM 进行了综合评估, 相较于 SWGCC 的相同优化级别, swLLVM 整型课题性能平均下降 0.12%, 浮点型课题性能平均提升 9.04%, 整体性能平均提升 5.25%, 编译速度平均提升 79.1%, 代码尺寸平均减少 1.15%.

关键词: 异构众核; 编译系统; 编程模型; 存储层次; 向量化; 节点合并; 访存优化

中图法分类号: TP314

中文引用格式: 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): 2359–2378. http://www.jos.org.cn/1000-9825/6896.htm

英文引用格式: 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, 2024, 35(5): 2359–2378 (in Chinese). http://www.jos.org.cn/1000-9825/6896.htm

swLLVM: Optimized Compiler for New Generation Sunway Supercomputer

SHEN Li[1], ZHOU Wen-Hao[2], WANG Fei[4], XIAO Qian[1], WU Wen-Hao[2], ZHANG Lu-Fei[4], AN Hong[1], QI Feng-Bin[3]

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

[2](National Research Center of Parallel Computer Engineering and Technology, Beijing 100190, China)

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

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

Abstract: The heterogeneous many-core architecture with an ultra-high energy efficiency ratio has become an important development trend of supercomputer architecture. However, the complexity of heterogeneous systems puts forward higher requirements for application development and optimization, and they face many technical challenges such as usability and programmability in the development process. The independently developed new-generation Sunway supercomputer is equipped with a homegrown heterogeneous many-core processor, SW26010Pro. To take full advantage of the performance of the new-generation many-core processors and support the development and

* 基金项目: 国家重点研发计划 (2018YFB0204200); 浙江省科技厅重大项目 (2022C01250)
收稿时间: 2022-07-07; 修改时间: 2022-09-12; 采用时间: 2023-01-05; jos 在线出版时间: 2023-06-14
CNKI 网络首发时间: 2023-06-15

optimization of emerging scientific computing applications, this study designs and implements an optimized compiler swLLVM oriented to the SW26010Pro platform. The compiler supports Athread and SDAA dual-mode heterogeneous programming models and provides multi-level storage hierarchy description and SIMD extensions for vector-like operations. In addition, it realizes control-flow vectorization, cost-based node combination, and compiler optimization for multi-level storage hierarchy according to the architecture characteristics of SW26010Pro. The experimental results show that the compiler optimization designed and implemented in this paper achieves significant performance improvements. The average speedup of control-flow vectorization and node combination and optimization is 1.23 and 1.11, respectively, and the memory access optimization achieves a maximum performance improvement of 2.49 times. Finally, a comprehensive evaluation of swLLVM is performed from multiple dimensions on the standard test set SPEC CPU2006. The results show that swLLVM reports an average increase of 9.04% in the performance of floating-point projects, 5.25% in overall performance, and 79.1% in compilation speed and an average decline of 0.12% in the performance of integer projects and 1.15% in the code size compared to SWGCC with the same optimization level.

Key words: heterogeneous many-core; compiler system; programming model; storage hierarchy; vectorization; node combination; memory access optimization

高性能计算被公认为继理论科学和实验科学之后, 人类认识世界改造世界的第 3 大科学研究方法[1], 已成为国家间科技竞争的重要体现和战略博弈的技术主战场, 对于国家安全、经济建设、社会发展都具有举足轻重的影响. 当前, 作为高性能计算“下一个明珠”的 E 级计算时代已经来临[2]. 然而, 随着系统规模的扩大以及应用复杂程度的提高, E 级系统在好用性和可编程性等众多方面均面临严峻挑战.

随着体系结构的复杂化和应用的多样化, 核数的增加已经无法继续带来性能提升. 在这样的背景下, 为了进一步增强计算能力, 硬件设计呈现出异构化的趋势[3]. 在 2017–2018 年度的 Top500 榜单[4]中, 搭载 SW26010 异构众核处理器的“神威·太湖之光”超级计算机 4 次蝉联榜首. 在 2021 年 6 月发布的 Top500 榜单中, 排名前 10 的超级计算机中有 8 台均采用异构结构. 而最新发布的神威新一代超级计算机继承和发展了“神威·太湖之光”体系架构, 基于国产申威异构众核处理器 SW26010Pro 构建. 上述数据充分说明, 异构众核架构已成为面向 E 级计算的超级计算机体系结构的重要发展方向[5].

相较于同构系统, 异构系统的复杂性给应用开发和编程编译带来了巨大的挑战[6]. 首先, 异构众核处理器编程门槛很高, 程序员需要更加了解硬件结构特点, 熟练掌握多种异构编程模型的使用. 其次, 异构系统的用户将面对处理器内、节点内、节点间多个层次的异构性, 需要挖掘程序中的可加速部分, 处理异构处理器间的数据交换和同步问题. 最后, 开发人员需要将大量成熟的开源项目部署到异构系统以提高应用开发效率, 这对异构系统编译环境的功能和性能提出了较高的要求.

伴随着体系结构的发展, 高性能计算应用的广度也前所未有地扩展. 以数据驱动为主要特征的高性能计算应用不断涌现, 应用领域遍及人工智能、量子计算、信息安全等. 这些应用又反过来对高性能计算技术, 包括矩阵并行求解、高性能大数据处理、智能芯片等产生巨大的影响, 促进了高性能计算技术的创新[7]. 新兴行业应用的快速发展对高性能计算机的编译系统提出更高的要求. 例如, 大数据应用通常使用即时编译技术进行 SQL 查询操作的性能优化[8,9], 漏洞挖掘框架主要通过 Compiler-RT 库进行代码插桩[10], 深度学习编译器往往基于模块化和可重构的编译器架构进行构建[11−13], 许多新兴应用还要求编译器针对不同硬件后端提供统一的中间表示, 从而快速地进行移植和优化[14−17]. LLVM[18]作为编译领域的后起之秀, 支持灵活高效的统一中间表示, 支持即时编译和 Compiler-RT 等多种功能扩展, 采用模块化结构设计, 能够作为独立的功能模块与应用或框架松耦合. 上述特点使得 LLVM 逐渐取代 GCC, 成为新兴领域应用的主流编译工具.

为了充分发挥神威新一代超级计算机的性能优势, 有效支撑新兴科学计算应用的开发和优化, 本文在 LLVM 开源项目的基础上, 设计和实现了 swLLVM, 针对 SW26010Pro 体系结构特点及应用需求进行了多方面的功能扩展, 并且提供丰富的优化手段. 本文的贡献主要包括如下几点.

1) 设计并实现了面向新一代神威超级计算机的优化编译器 swLLVM, 其支持多级存储层次描述, 支持向量操作扩展, 实现了对神威超级计算机基础编译功能的完全兼容.

2) 针对 SW26010Pro 双模态 PCIe 的特点, 在 swLLVM 中实现了面向 RC 模式的 Athread 编程模型以及面向

EP 模式的 SDAA 编程模型, 有效降低了用户在异构众核平台的编程难度.

3) 基于 swLLVM 设计并实现了多种优化技术, 包括控制流向量优化、基于代价的节点合并优化以及存储空间管理及访问优化, 进一步挖掘异构众核体系结构的性能优势.

本文第 1 节介绍背景及相关工作. 第 2 节介绍 swLLVM 的基础功能的设计与实现. 第 3 节介绍 swLLVM 面向新一代众核处理器的编译优化. 第 4 节通过实验评估 swLLVM 的优化效果和编译性能. 第 5 节给出总结和展望.

1 相关工作

1.1 新一代申威异构众核处理器 SW26010Pro

SW26010Pro 是我国自主研发的新一代申威异构众核处理器[19], 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). 控制核心主要功能是计算、控制、通信、I/O 等, 运算核心则主要用于加速并行计算. 核组中运算核心以 8×8 阵列方式进行排布, 运算核心之间通过阵列内网络进行互连. 核组中的任意两个运算核心之间, 可以通过远程内存访问 (remote memory access, RMA) 方式进行数据通信.

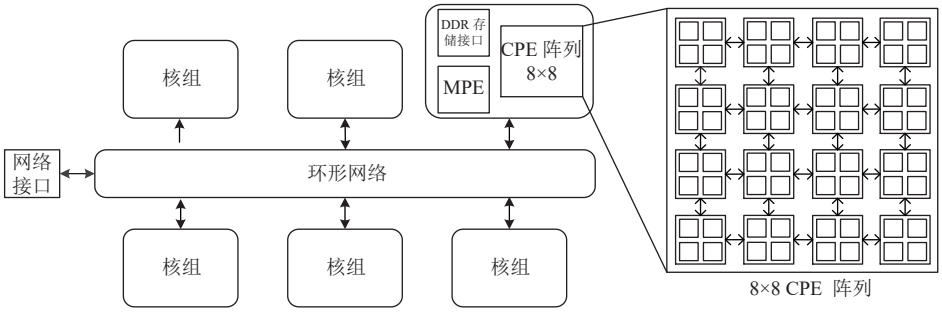


图 1 SW26010Pro 异构众核处理器基本结构

控制核心为 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. 运算核心为 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. 每个运算核心都拥有独立的指令缓存和由用户控制的局部数据存储器 (local data memory, LDM). 运算核心可以通过常规的 Load/Store 指令或直接内存访问 (direct memory access, DMA) 方式实现 LDM 与主存之间数据传输.

相比多核架构, 异构众核架构对编译器提出了更高的要求. 面对处理器内、节点内、节点间多个层次的异构性, 编译器需要充分感知硬件结构的特点, 挖掘程序中可被加速执行的部分, 处理异构处理器之间的数据交换和同步问题. 同时, 编译器还需要支持对异构特征的描述、多层次并行性的开发等功能, 以对数据局部性提供一定的控制优化能力, 最小化数据移动的开销, 最大程度的释放系统潜在的性能.

1.2 面向异构混合体系结构的优化编译器

近年来, 随着摩尔定律逐渐放缓, 处理器的主频提升已十分有限, 核内多发射、多级流水、向量宽度等提升单核性能的手段和技术也趋近发展极限. 考虑到材料、工艺、功耗、造价等多方面的挑战, 超算系统越来越多地采用异构混合体系结构, 例如 NVIDIA GPU、 AMD GPU、 Intel Xeon Phi、 PEZY[20]、申威众核处理器[21]、飞腾 Matrix 众核处理器[22]等.

为了充分发挥异构混合体系结构的性能优势, 主流处理器厂商针对其特有的编程模型均提供了底层编译系统的支持. NVCC 是针对 NVIDIA GPU 平台 CUDA 编程模型的编译器. 它将 CUDA 源程序进行预处理后分离成主机代码与设备代码, 然后分别调用不同的编译器进行编译. HCC 是 AMD 的 ROCm 平台的编译工具, 其通过单一编译环境统一支持 ISO C++11/14、C14、OpenMP 4.0, 同时适用于 AMD CPU 和 AMD GPU. DPC++编译器[23]是 Intel 的 oneAPI 工具包中的基础编译器, 其支持 SYCL 编程模型[24], 能够获得比 ICC 和 GCC 更高的性能. 飞腾

Matrix 是国防科大自主设计的异构融合加速器, 其采用 CPU+GPDSP 的异构融合架构, 兼顾高性能和高可编程性的特点. Matrix 加速器配备了一套类 CUDA 的开发环境, 其中包含分别针对 CPU 和 GPDSP 的异构编译器, 针对多核、超长指令字、向量等特点进行了一系列优化, 方便用户快速开发面向典型领域的高性能应用. 各硬件厂商针对其硬件架构和编程模型的特点对其底层编译系统进行了深度定制, 难以直接应用在新一代申威异构众核处理器平台.

SWGCC 异构众核编译系统[25]是国产高性能众核处理器 SW26010Pro 的基础编译器. 其提供高级语言到机器可执行码的高效编译转换, 提供完备的基础编译功能以及多维的编译优化视角. 然而, 大数据、人工智能、信息安全等应用领域的兴起对编译系统提出了更高的要求. 表 1 展示了多种新兴领域典型应用或框架对底层编译环境的需求情况. 可以看出, 新兴领域应用不仅要求底层编译环境支持的高效代码生成, 还依赖于即时编译、Compiler-RT 以及中间表示优化等功能. LLVM 作为编译领域的后起之秀, 支持高效的中间表示优化, 支持即时编译和 Compiler-RT 等多种功能扩展, 能够作为独立的功能模块与应用或框架松耦合. 上述特点使得 LLVM 逐渐替代 GCC, 成为新兴领域应用的主流底层编译工具.

LLVM 是由伊利诺伊大学在 2000 年发起的开源项目, 其基本结构如图 2 所示. LLVM 的设计初衷是提供一种现代的、基于静态单赋值 (static single assignment, SSA) 形式的编译策略, 能够支持多种编程语言、支持静态编译和动态编译, 提供多种优化的功能模块和插件. 开放的协议使得基于 LLVM 构建更加丰富、复杂、定制的编译系统已经成为国际上编译系统开发的主流技术路线. 尤其在异构芯片领域, 基于 LLVM 框架的编译系统已经得到了广泛的应用和部署. 目前最新版本的 NVCC、HCC 以及 DPC++均基于 LLVM 进行构建. 从版本迭代速度上, GCC 官方版本近 3 年来几个大版本发布间隔约为 12 个月, 而 LLVM 仅约为 6 个月, 其发布速度约为 GCC 的 2 倍. 在代码总量上, 目前最新版本的 LLVM-15.0.3 较 GCC-12.2.0 核心模块的总代码行数少 11.5%, LLVM 每个文件的平均代码行数比 GCC 少 53.4%. 从开发难度上, LLVM 模块组织方式更加松耦合, 且模块间的协议比较清晰, 再加上优秀的中间表示代码设计和详细的帮助文档, 使得 LLVM 对开发人员更加友善. 综上, 以 LLVM 作为基础框架开发特定芯片的编译器能够显著降低代间移植工作量, 提升开发效率.

表 1 新兴领域典型应用的编译需求

应用领域	应用名称	功能需求	编译工具
大数据	MaxCompute[8]	JIT	LLVM
	PostgreSQL[9]	JIT	LLVM
人工智能	TVM[11]	JIT/IR优化	LLVM/GCC
	Glow[12]	IR优化	LLVM/GCC
	XLA[13]	IR优化	LLVM
信息安全	AFL[26]	Compiler-RT	LLVM/GCC
	LibFuzzer[27]	Compiler-RT	LLVM/GCC
	KLEE[14]	JIT	LLVM
新型编译	MLIR[15]	IR优化	LLVM
	Julia[16]	JIT	LLVM
	Numba[17]	JIT	LLVM

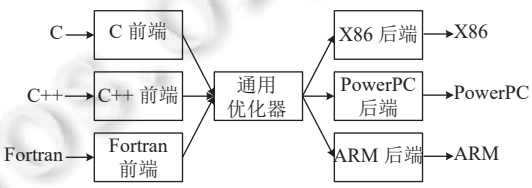


图 2 LLVM 编译器基本结构

综合考虑应用需求、实现难度、生态拓展等多方面因素, 当前亟需基于 LLVM 实现面向新一代申威异构众核处理器平台的优化编译器, 以降低异构众核架构编程难度, 提升其对新兴应用的支撑能力, 从而更好地发挥新一代异构处理器平台的多维计算能力.

2 针对新一代异构众核架构的编译支撑

为了更好地解决异构平台的编译支撑和性能优化难题, 本文设计并实现了 swLLVM 编译系统并根据 SW26010Pro

的架构特点和应用需求, 定制了多种功能拓展和性能优化. 本节在编程模型、数据映射、内建扩展等多个方面介绍 swLLVM 编译器针对新一代异构众核架构的编译支撑.

2.1 编程模型

SW26010Pro 处理器包含双模态 PCIe 模块, 支持 RC (root complex) 和 EP (end point) 两种模式. 神威新一代超级计算机中的 SW26010Pro 采用 RC 模式, 作为独立的处理器使用. 当作为加速卡设备时, SW26010Pro 通过 EP 模式与使用商用指令集的主机相连, 通过运行时库和设备驱动进行控制, 能够很好地支持深度学习和机器学习框架的部署, 例如 TensorFlow[28]、PyTorch[29]等.

为了更好地支持双模态 PCIe 功能, 减少用户的编程难度, 本文在 swLLVM 编译器中支持 RC 模式的 Athread 编程模型以及 EP 模式的 SDAA 编程模型. 本节分别介绍 swLLVM 中上述两种编程模型的实现方式.

2.1.1 RC 模式 Athread 编程模型

Athread 编程模型[25]是 SW26010Pro 处理器 RC 模式使用的加速线程编程模型, 其基于控制核心与运算核心阵列之间的主从协同工作模式, 通过显式调用的 Athread 函数管理运算核心阵列的计算和访存. Athread 包含控制核心加速线程库和运算核心加速线程库, 表 2 展示了 Athread 加速线程库的主要接口. 控制核心加速线程库主要用于控制线程的创建和回收、线程调度、中断和异常管理、异步掩码支持等操作. 运算核心加速线程库则主要用于运算核心线程的识别、发起 DMA/RMA 传输、发送中断等操作.

表 2 Athread 加速线程库的主要接口

加速线程库	Athread接口	功能描述
控制核心加速线程库	athread_init	初始化线程库
	athread_spawn	创建线程组
	athread_join	等待线程组终止
	athread_halt	关闭线程组流水线
运算核心加速线程库	athread_dma_get	DMA数据接收
	athread_dma_put	DMA数据发送
	athread_rma_get	RMA数据接收
	athread_rma_put	RMA数据发送

Athread 编程模型采用控制核心和运算核心代码分离的编程模式, 其在 swLLVM 编译器中的支持方式如后文图 3 所示. swLLVM 编译器针对控制核心和运算核心分别提供 Clang 和 Flang 前端, 以支持 C/C++/Fortran 语言到 LLVM IR 的转换. 控制核心和运算核心的 LLVM IR 是标准 LLVM IR 的派生集合, 它们继承了 LLVM IR 简洁、稳定、可扩展等优势, 并且根据 SW26010Pro 架构特点提供了相应的存储层次描述和内建扩展支持. swLLVM 中端包含一系列优化遍, 负责收集信息或进行代码变换. 除了通用的分析和优化外, swLLVM 还根据 SW26010Pro 体系结构特点定制了多种编译优化, 例如控制流向量优化、节点合并优化、存储空间管理及访问优化等. swLLVM 后端负责将优化的 LLVM IR 转换为 SW26010Pro 平台目标码. 最后, 混合链接器将多个目标文件组合在一起, 并解析其中的重定位, 形成混合链接可执行文件.

后文图 4 给出了使用 Athread 编程模型的程序示例, 它能够直观地与 Athread 加速线程编程模型相对应. 在 RC 模式运行过程中, 控制核心程序使用 athread_init 接口进行线程的初始化操作, 使用任何加速线程库接口前都需要调用该初始化接口. 控制核心通过 athread_spawn 接口创建线程组并传递运行参数, 其指定的线程任务被提交到运算核心阵列执行. 运算核心启动对应的加速线程任务, 运行结束后通知控制核心. 控制核心通过 athread_join 接口等待运算核心的加速线程任务结束, 最后使用 athread_halt 接口关闭运算核心流水线.

2.1.2 EP 模式 SDAA 编程模型

SW26010Pro 支持作为独立的加速卡设备使用, 通过 EP 模式的 PCIe 接口与商用指令集的主机相连. 由于通过纯硬件方式支持双模态 PCIe 的复杂度较高, SW26010Pro 采用精简的硬件设计方案, 通过软硬件协同的方式支

持完备的上下文加载、启动、同步、转发等操作. 图 5 展示了 EP 模式下主机端与 SW26010Pro 设备端的软硬件交互机制. 部署在主机端的加速卡设备驱动程序能够处理与加速卡设备接口的低级操作, 包括数据移动、上下文管理、执行和同步等, 为上层软件提供了 SW26010Pro 的清晰抽象.

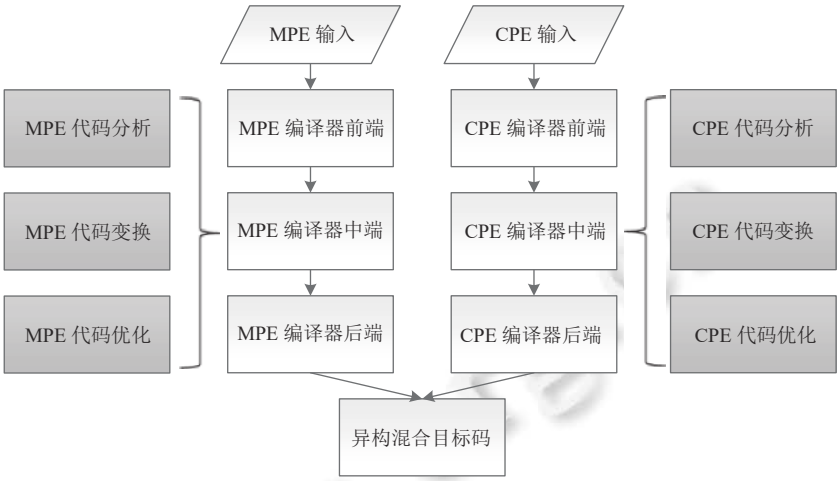


图 3 Athread 编程模型的实现方式

```
// master.c
#include <athread.h>
#include “util.h”
arg_t arg;
extern SLAVE_FUN (add)(arg_t *)
intmain () {
   arg.a=2;   arg.b=7;
   athread_init ();
   athread_spawn (add, &arg);
   athread_join ();
   athread_halt ();
   printf ("2+7=%d\n", arg.c);
   return 0;
}
```

```
// slave.c
#include <slave.h>
#include “util.h”

void slave_add (arg_t*arg) {
   arg->c=arg->a+arg->b;
}
```

```
// util.h
typedef struct arg_t
{
   int a, b, c;
};
```

图 4 Athread 编程模型示例

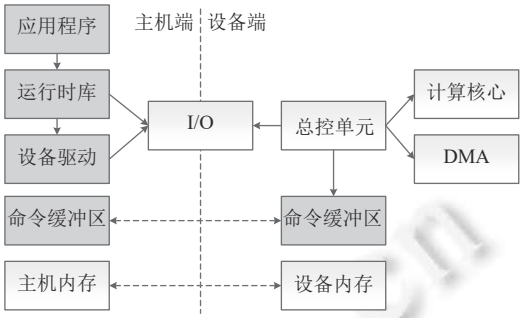


图 5 主机端与设备端的交互机制

在使用 EP 模式时, SW26010Pro 加速卡拥有独立于主机内存的设备内存, 以提升设备端的访存性能, 但是用户需要显式控制主机和设备间的内存拷贝. 同时, 主机端与设备端指令集体系结构的巨大差异使得 SW26010Pro 加速卡设备的可编程性面临严峻挑战. 针对上述问题, 本文引入 SDAA (software defined accelerator architecture) 编程模型, 以处理不同内存层次结构和指令集体系结构的计算域之间的组合和交互.

SDAA 编程模型底层基于 SDAA 运行时库, 其与 Athread 加速线程库功能类似, 用于实现 EP 模式下计算任务通用的、低延迟的、无特殊硬件支持的调度和管理. SDAA 运行时库的核心 API 接口如表 3 所示, 主要包括设备管理、内存管理和运行控制 3 类. 设备管理接口用于获取当前可用的加速卡设备, 并配置当前使用的设备. 内存管理接口用于在主机或设备内存上为用户程序申请和释放所需的内存空间, 并实现主机与设备内存间的数据拷贝. 运行控制接口用于注册 kernel 函数、配置 kernel 参数以及将包含 kernel 的代码段加载到 SW26010Pro 设备内存后启动执行.

为了简化基于 SDAA 运行时库的加速卡程序构建, 本文在 swLLVM 中实现了 SDAA 编程模型. 图 6 展示了 swLLVM 中 SDAA 编程模型的支持方式. 与 CUDA 编程模型类似, SDAA 编程模型支持统一源码编程方式, 编译器前端通过识别__global__和__kernel__等关键字, 对主机端代码和设备端代码分别进行预处理. swLLVM 设备端编译器在中端兼容 Athread 编程模型的存储层次描述和内建功能扩展, 并且在后端复用了 RC 模式下的架构无关

及架构相关的代码分析、变换及优化方法. 优化后的设备端代码被链接为动态库, 最终以 FatBinary 的形式打包到主机端的可执行目标码中.

表 3 SDAA 运行时库的主要接口

类型	SDAA接口	功能描述
设备管理	sdaaGetDeviceCount	获取可用设备数量
	sdaaSetDevice	设置需要使用的设备号
	sdaaGetDevice	获取正在使用的设备号
内存管理	sdaaMalloc	在设备端分配内存
	sdaaFree	释放设备端已分配内存
	sdaaMallocHost	在主机端分配内存
	sdaaFreeHost	释放主机端已分配内存
	sdaaMemcpy	在主机和设备间传输数据
	sdaaMemset	将设备内存初始化特定值
运行控制	sdaaConfigureCall	配置kernel需要的资源
	sdaaSetupArgument	配置kernel运行参数
	sdaaLaunch	启动kernel函数执行

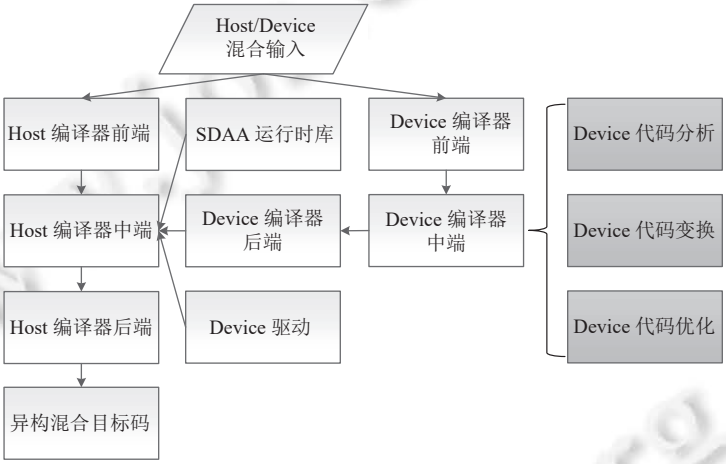


图 6 SDAA 编程模型的实现方式

后文图 7 展示了 SDAA 编程模型的程序示例. 与 SDAA 运行时库接口相对应, EP 模式下程序运行过程也可以概括为 3 个主要操作: ① 设备管理接口选择一个可用设备以执行 kernel; ② 内存管理接口在主机和设备内存上分配计算所需的内存空间, 并且将输入数据从主机复制到设备内存; ③ 运行控制接口启动 kernel 在设备端执行, 并且在执行完毕后将输出数据从设备复制回主机内存, 并返回给用户程序.

2.2 存储层次描述

SW26010Pro 处理器的主存地址空间采用虚地址编址, 分连续段空间和交叉段空间. 连续段是每个核组的私有空间, 在核组内连续编址, 控制核心和运算核心均可直接访问本核组的连续段空间. 连续段又可以进一步划分成私有连续段和共享连续段, 前者为本核组的每个运算核心的私有变量空间, 后者为本核组的所有运算核心和控制核心共享. 交叉段是同一芯片内核组间的共享空间, 控制核心和运算核心均能够访问全芯片的该分布共享区域.

同时, 每个运算核心拥有一块高速的本地局部数据存储空间 LDM. LDM 空间可分为私有 LDM 空间和共享 LDM 空间, 前者是本运算核心快速访问的本地 LDM 空间, 后者是核组内所有运算核心共享访问的 LDM 空间. LDM 空间访存速度快, 但要获得高性能, 用户需要显式管理 LDM 的使用.

为了支持 SW26010Pro 主存及 LDM 空间的多级存储层次描述, swLLVM 编译器前端提供了 C 语言和 Fortran 语言的扩展关键字, 如表 4 所示.

```
// add.sdaa
#include “sdaa.h”
#include “sdaa_runtime.h”
__global__ void add (void *args) {
  int **data=(int**)((uint64_t*) args);
  int *a=(int*) data [0];
  int *b=(int*) data [1];
  int *c=(int*) data [2];
  c [0]=a [0]+b [0];
  return;
}
void main () {
  sdaaSetDevice (0);                                                    ①
  int *a=(int *) malloc (sizeof (int));
  int *b=(int *) malloc (sizeof (int));
  int *c=(int *) malloc (sizeof (int));
  a [0]=2;
  b [0]=7;
  int *dev_a, *dev_b, *dev_c;                                           ②
  sdaaMalloc ((void**) (&dev_c), N * sizeof (int));
  sdaaMalloc ((void**) (&dev_a), N * sizeof (int));
  sdaaMalloc ((void**) (&dev_b), N * sizeof (int));
  sdaaMemcpy (dev_a, a, sizeof (int), sdaaMemcpyHostToDevice);
  sdaaMemcpy (dev_b, b, sizeof (int), sdaaMemcpyHostToDevice);
  add<<<1>>>(dev_a, dev_b, dev_c);
  sdaaMemcpy (c, dev_c, sizeof (int), sdaaMemcpyDeviceToHost);          ③
  printf ("2+7=%d\n", c);
  return 0;
}
```

图 7　SDAA 编程模型示例

表 4　主存及 LDM 空间扩展关键字

关键字类型	C关键字	Fortran关键字	功能描述
主存空间扩展关键字	__thread	common /private_*/	私有连续段空间
	N/A	common /*/	共享连续段空间
	__cross	common /cross_*/	交叉段空间
LDM空间扩展关键字	__ldm	common /local_*/	LDM私有空间
	__ldm_share	common /slocal_*/	LDM共享空间

为了提高 LDM 空间利用率, 增强异构众核程序设计的灵活性, swLLVM 编译器还实现了 LDM 空间重用机制. 在编译器前端, swLLVM 提供了__ldm_kernel(kname) 关键字, 其中, kname 表示 kernel 名称, 不同的 kernel 名称修饰的__ldm_kernel 变量将共享同一块 LDM 空间. 在编译器后端, 链接器收集全局信息, 根据段属性找出符合重用条件的 LDM 数据段, 为标识相同的段分配相同的起始地址, 并以满足重用条件的最大的 LDM 数据段的大小作为该重用段的大小.

2.3　向量扩展操作

单指令流多数据流 (single instruction multiple data, SIMD) 技术又称为向量技术, 是提高处理器性能的有效手段[30,31]. SW26010Pro 处理器实现了控制核心和运算核心宽度非一致的 SIMD 扩展指令集, 其中控制核心向量宽度为 256 位, 运算核心向量宽度为 512 位. swLLVM 编译器在数据类型、运算符和内建函数 3 个方面分别进行了向量扩展支持. 本节以运算核心的 C 语言向量扩展支持为例进行简要介绍.

swLLVM 运算核心编译器在标准 C 语言的基础上扩展了 7 种数据类型, 包括 intv16, uintv16, int512, uint512, floatv8, doublev8 以及 halfv32. 表 5 给出了各种扩展数据类型的含义以及分量的表示范围.

为了减小用户使用向量数据类型的编程难度, swLLVM 对高级语言中的运算符进行了重载, 表 6 列出了各个运算符适用的 SIMD 扩展数据类型. 对于无法使用运算符进行描述的复杂操作, 例如插入、提取、拷贝、选择、混洗等, swLLVM 均扩充了相应的内建函数或宏定义接口供编程时调用. 用户只需要引用<simd.h>头文件就可以方便地使用这些扩展数据类型和内建函数.

表 5 运算核心 SIMD 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 ~ 6.5504E4

表 6 扩展数据类型支持的运算符及内建函数操作

运算符和访存相关内建函数	操作	适用的扩展数据类型
运算符	+, –	(u)intv16, (u)int512, floatv8, doublev8, halfv32
	*	floatv8, doublev8, halfv32
	<<, >>	intv16, uintv16
	==, <=, <	(u)intv16, floatv8, doublev8, halfv32
访存相关内建函数	simd_load	(u)intv16, (u)int512, floatv8, doublev8, halfv32
	simd_store	(u)intv16, (u)int512, floatv8, doublev8, halfv32
	simd_loade	halfv32
	simd_loadu	(u)intv16, floatv8, doublev8
	simd_storeu	(u)intv16, floatv8, doublev8

为了实现 SIMD 扩展数据类型与标量数据类型之间的转换, swLLVM 运算核心编译器提供了 5 个访存相关的内建函数接口, 如表 6 所示. 其中, simd_load 用于将 512 位长度的标量类型的数据从向量对界的内存地址加载到 SIMD 扩展类型变量中, simd_store 用于将 SIMD 扩展类型变量中的数据存储到 512 位长度的向量对界的内存中, simd_loade 则用于将标量类型的数据从内存加载到 SIMD 扩展类型变量的低位并扩展到高位. 同时, swLLVM 编译器还提供了 simd_loadu 和 simd_storeu 接口, 以支持不对界内存地址的标量数据与 SIMD 扩展类型的映射.

3 面向新一代众核处理器的编译优化

生成新一代异构众核处理器平台上运行高效的可执行码是 swLLVM 编译器的重要目标. 本节介绍 4 种基于 swLLVM 编译器实现的编译优化技术, 包括控制流向量优化、基于代价的节点合并优化以及针对存储空间的管理和访问优化. 第 4 节将详细展示每种优化在 SW26010Pro 平台的实测效果.

3.1 控制流向量优化

高性能计算应用的典型特征之一是核心代码中拥有大量的嵌套循环语句. 如何选择并实施循环变换以获得最优的程序性能是学术界公认的难题[32]. 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[30,31].

向量优化的硬件基础是 SIMD 扩展部件, 其硬件结构决定了更加适合处理计算密集、相关性少的指令序列. 控制相关是向量优化中需要考虑的一种主要依赖形式. 在控制相关中, 一条语句的执行受控于另一条语句, 这为数据流分析带来了很多的不确定性和复杂性. 因此, 为了充分挖掘程序中的数据并行性, 需要尽可能消除控制相关.

控制相关会导致复杂的数据重组, 导致指令代价大量消耗在存储指令上. 针对上述问题, 本文在 swLLVM 中引入向量条件选择操作 VSELECT, 利用 VSELECT 将控制相关转换为数据相关, 然后通过 SW26010Pro 的 SIMD 扩展指令集中的向量条件选择指令完成向量化操作. VSELECT 操作的基本格式为: vdst = VSELECT(vsrc1, vsrc2, vmask), 其中, vsrc1 和 vsrc2 是两个源操作数向量, vmask 为掩码向量. 当 vmask 的某个分量取值为 0 时, 将对应位置的 vsrc1 的分量赋值给 vdst 的分量, 否则, 将对应位置的 vsrc2 的分量赋值给 vdst 的分量.

控制流向量优化中使用的 VSELECT 转换算法如算法 1 所示. 其中, Body 表示程序最内层循环中的 IF 结构包含的基本块集合, Instr 表示 Body 中的指令, Header 表示 Instr 所属基本块的分支条件块, 而 Cmp 表示 Header 块的分支条件. 在进行向量化决策时, 该算法遍历程序最内层循环, 使用 Def-Use 关系获取控制流的 IF 结构, 并且将其转换为向量条件选择指令, 从而将控制相关转化为数据相关, 实现向量化代码的生成. VSELECT 转换后改变了原有的语句结构, 因此需更新存储指令代价以获取最佳的向量化决策.

算法 1. VSELECT 转换算法.

输入: 程序最内层循环中的 IF 结构;
输出: IF 结构转换成的 VSELECT 向量操作.

```
1. Body = getBody(IF)
2. FOR Instr ∈ Body DO
3.     IF Instr ∈ StoreInstr THEN
4.         Header = Instr→getHeader()
5.         Cmp = Header→getCmp()
6.         mask = Cmp.getMask()
7.         IF mask THEN
8.             LoadI = CreateLoad();
9.             SelectI = CreateSelect(mask, LoadI);
10.            StoreI = CreateStore(SelectI);
11.        END IF
12.        Header→replace(Instr, StoreI);
13.        CostNewSI += CostLoad + CostSel;
14.    END IF
15. END FOR
```

3.2 节点合并优化

合并优化是当前编译领域的研究热点. 例如, 加速线性代数 (accelerated linear algebra, XLA) 编译器通过将计算图中的多个算子合并为一个高效算子, 以提高执行速度、改善内存使用、减少对自定义操作的依赖[13]. 从 LLVM IR 到机器指令 (machine instruction, MI) 的过程, 也是从平台无关的节点向平台相关的节点的降级过程. 在该过程中, LLVM IR 会转换成有向无环图 (direct acyclic graph, DAG), 并利用图的匹配算法完成降级处理. 每个 DAG 图对应一个基本块, 其中节点代表指令, 节点之间的边代表指令之间的数据依赖关系. 编译器根据拓扑排序遍历 DAG 图中每一个节点, 分析优化可行性并执行节点合并操作. 算法 2 展示了节点合并优化的主要流程.

算法 2. 节点合并算法.

输入: 待合并节点 Node, 合并模式 (Op1, Op2)→Op;
输出: 合并后的节点 combineNode.

```
1. costBefore = cost(Node)
2. costAfter = 0
3. IF Node.getOpcode() == Op1 THEN
4.     Node1 = Node.getOperand()
5.     IF Node1.getOpcode() == Op2 THEN
```

```
6.          fNode = DAG.getNode(Op, Node, Node1)
7.          IF hasOneUse(Node1) THEN
8.              costBefore += cost(Node1)
9.          END IF
10.         costAfter = cost(fNode)
11.         IF costBefore > costAfter THEN
12.             combineNode = fNode
13.         ELSE
14.             combineNode = Node
15.         END IF
16.     END IF
17. END IF
```

以图 8 中的浮点乘加操作为例, 待合并节点为 ADD, 合并模式为 (MUL, ADD)→FMA. 首先, 算法初始化合并前后的指令代价, 然后判断待合并节点及其叶节点是否符合合并模式. 如果匹配成功, 则根据待合并节点 ADD 及其叶子节点 MUL 创建合并节点 FMA. HasOneUse 方法用来判断节点是否只有一个使用者, 如果只有 ADD 节点依赖 MUL 节点的结果, 则需要将 MUL 节点的指令代价计入合并前的指令代价. 最后, 算法比较合并前后的指令代价, 若合并后的代价小于合并前的代价, 则使用合并节点 FMA 替换待合并节点 ADD.

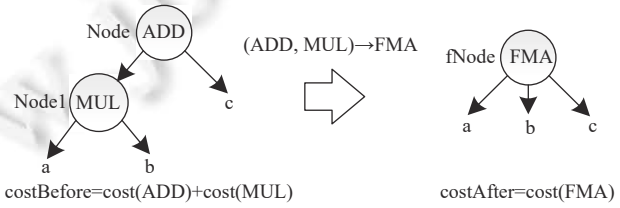


图 8 节点合并示意图

节点合并的思想与窥孔优化 (peephole optimization)[33]类似, 是一种局部的优化方法. 与激进的窥孔优化不同, 基于代价的节点合并通过对优化前后的节点进行代价评估, 根据代价的大小决定是否进行合并, 从而避免过度优化. 表 7 给出了 swLLVM 中节点合并模式示例, 其中, 对于 (RECIPROCAL, MUL)→DIV 模式, 合并前的总代价为 15, 小于合并后代价 19, 因此不执行合并操作. 而对于 (SIN, COS)→SINCOS 模式, 由于 SW26010Pro 后端不支持 SINCOS 操作, 因此合并后代价为∞, 也无法执行合并操作. 同时, 对于上述代价较大的 DAG 节点, swLLVM 会在节点合并阶段将其拆分为代价较小的 DAG 节点的组合, 以达到更高的性能.

3.3 LDM 空间管理

SW26010Pro 采用控制核心加运算核心阵列的异构混合体系结构. 运算核心为 64 位 RISC 结构精简处理单元, 包含一块由软件管理的 256 KB 的片上存储空间 LDM, 其容量有限, 属于处理器的珍稀资源. 因此, 如何通过软件的方式用好 LDM 空间, 成为 swLLVM 编译器访存优化面临的首要问题.

用户主要通过静态变量以及动态 malloc 的方式使用 LDM 空间.其中, 静态变量声明简单, 使用高效, 但需要用户对 LDM 空间进行显式管理. 为了方便用户实时检测 LDM 空间使用量, 本文在 swLLVM 中提供了 ldmreport 工具. 图 9 展示了 ldmreport 工具的使用示例, 该工具扫描 ELF 格式目标码中的所有 LDM 变量, 输出每个变量占用的 LDM 空间大小, 并且统计 LDM 空间整体占用情况. 如果总的 LDM 空间占用超过 128 KB, ldmreport 将会输出提示信息, 以指导用户手动优化 LDM 空间的使用. 如果 LDM 空间占用超过 256 KB, ldmreport 将会输出错误报警.

表 7 节点合并模式示例

合并前		合并后	
DAG	代价	DAG	代价
(MUL, ADD)	14	FMA	7
(NEG, MUL, SUB, NEG)	28	FMA	7
(MUL, NEG, SUB, NEG)	28	FMA	7
(SLL, ADD)	2	SADD	1
(ADD, LOAD)	>101	LOAD	>100
(ADD, STORE)	>101	STORE	>100
(LOAD, EXT)	>105	EXTLOAD	>100
(TRUNCATE, STORE)	>105	TRUNCSTORE	>100
(OR, NOT)	2	ORNOT	1
(SELECT, SETCC)	8	SELECTCC	5
(RECIPROCAL, MUL)	15	DIV	19
(SIN, COS)	20	SINCOS	∞

```
// ldmreport_test.c
#include <slave.h>
__ldm int a[10];
__ldm long b;
__ldm_share float c[10];
__ldm_share double d;
int main {
   // Operation of a, b, c, d
   …
   return 0;
}
```

(a) C 语言程序示例

```
$ clang  -mslave  -c ldmreport_test.c    -o ldmreport_test.o
$ ldmreport ldmreport_test.o
==============    LDM INFO begin    ==============
LDM privatevariable list is:
indx: 0      size: 40      name: a
indx: 1      size: 8       name: b
================================================
LDM sharedvariable list is:
indx: 0      size: 40      name: c
indx: 1      size: 8       name: d
====    ALL LDM SIZE: 48B(private),48B(shared)]      ====
```

(b) Idmreport 工具检测结果

图 9 ldmreport 工具使用示例

堆和栈是两类重要的动态存储分配方式. 其中, 栈空间以栈帧为分配单位, 用于存放函数执行过程中所需的参数、局部变量和中间临时变量等数据, 是异构众核程序执行过程中频繁访问的热点区域之一. 为了优化访存速度, 编译器将运算核心的栈空间置于 LDM 上, 此时的 LDM 空间划分如图 10 所示. 由于 LDM 空间容量受限, 而栈空间在程序运行过程动态增长, 为了避免 LDM 空间栈溢出导致程序执行崩溃, 需要编译器提供 LDM 空间的栈保护功能.

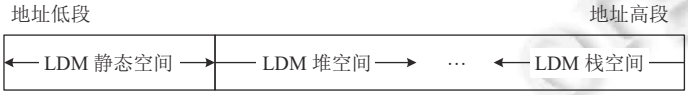


图 10 运算核心 LDM 空间排布

swLLVM 编译器的 Clang 前端拥有静态分析的功能插件, 能够在编译时静态分析运算核心函数的栈空间使用情况, 为用户优化提供参考. 但是, 考虑到函数调用关系的复杂性以及栈空间分配的动态性, 静态分析方法存在很多无法识别的情况, 例如函数指针、嵌套函数调用以及其他运行时才能得知的动态栈深度调整. 为了辅助用户进行程序检查及错误定位, 本文在 swLLVM 编译器中基于 Compiler-RT 编译运行时库实现了运算核心 LDM 空间栈溢出检查功能. 通过在编译时添加-fcheck-ldm-stack 选项, swLLVM 在每个运算核心函数的入口处插桩栈空间溢出检查函数. 在程序运行时, 首先检查栈空间是否溢出到 LDM 堆空间, 若没有则继续检查是否溢出到 LDM 静态空间. 当出现栈空间溢出时, 程序在函数执行前跳转到异常处理函数, 并打印出错的 PC 地址, 用户通过 addr2line 工具就可以方便地进行错误定位.

3.4 主存访问优化

LDM 空间访问速度快, 但是容量较小, 仅适用于程序核心段的热点函数优化, 使用局限性较大. 随着高性能计

算应用逻辑趋于复杂, 代码量逐渐增加, 非核心段程序性能优化的重要性开始凸显[34]. 由于片上存储空间容量受限, 绝大多数异构众核应用的非核心段数据无法放入 LDM 空间, 运算核心访存开销较大. 因此, 如何优化主存空间的访问成为 swLLVM 编译器的另一个重要目标.

SW26010Pro 的每个核组配备 16 GB 的主存空间, 由核组内的控制核心和运算核心所共享. 通过分析高性能计算应用的访存行为可以发现, 运算核心访问主存主要存在两种情况: 获取变量数据以及获取变量/函数地址. 本文在 swLLVM 中分别针对上述情况进行了相应的优化.

ELF 文件以段的形式对数据进行存储和管理. 其中, rodata 段又称为只读数据段, 用于存放只读数据, 例如程序中 const 修饰的只读变量和字符串常量等. 由于只读数据存放在内存中, 运算核心每次访问都需要 Load 操作, 开销较大. 考虑到 rodata 段的只读属性, 编译器通过立即数的形式将其与指令共同编译在代码段, 从而减少内存访问次数. 图 11 给出了访问 rodata 段的简单示例以及优化前后的汇编指令.

```
// imm_opt.c
const unsigned long a =
        0x1111222233334444;
int main () {
   …
   b=a;
   …
}
```

(a) C 语言程序示例

```
// ori.s
…
ldih $t0,CPI0_0($gp)!gprelhigh
ldl   $t1,CPI0_0($t0)!gprellow
…
.section   .rodata
CPI0_0:
   .8byte  1229801703532086340
```

(b) 优化前汇编代码

```
// opt.s
…
ldih $t0,0x1111($zero)
ldi $t0,0x2222($t0)
sll $t0,32,$t0
ldih $t0,0x3333($t0)
ldi $t1,0x4444($t0)
…
```

(c) 优化后汇编代码

图 11 rodata 段数据访存优化示例

优化前, 只读变量 a 存放在 rodata 段中, 程序执行过程中首先根据 GOT 段基址 $gp 计算出变量 a 的内存地址, 然后通过 ldl 指令从内存中取出变量 a 的值. 优化后, 程序通过 4 条装入立即数指令和一条移位指令完成 64 位常量数据的拼接, 虽然指令数量增加, 但是避免了长延迟的访存操作, 整体性能获得提升. rodata 段数据访存优化不仅适用于 int、long、float、double 等标量数据类型, 还适用于 intv16、floatv8 等 SIMD 扩展数据类型. 对于各分量完全相同的 SIMD 扩展数据类型, 编译器首先构造出一个标量类型分量数据, 然后通过向量元素拷贝指令将其扩展到所有分量, 从而避免长延迟的向量访存操作.

SW26010Pro 运算核心获取全局变量及函数的地址主要通过 literal 重定位实现. literal 重定位是一种 GOT 段基址相关的重定位, 其基本原理如图 12 所示. literal 重定位包含两次访存操作, 首先通过相对偏移从 GOT 段获取变量的内存地址, 然后根据地址从内存中获取变量的值. 由于 GOT 段位于主存空间, 第 1 条 Load 指令的访存开销无法被忽略.

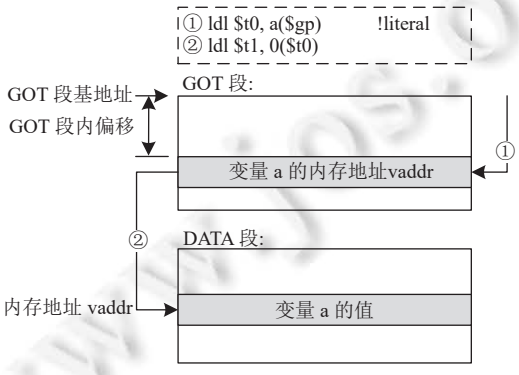


图 12 literal 重定位基本原理

为了减少主存访问次数, 编译器将获取地址的 literal 重定位优化为 gprel 重定位, 通过 GOT 段基址及其与变量地址的相对偏移直接计算出变量内存地址, 从而减少一次访存操作. 优化后的地址获取方式如图 13 所示. 考虑

到 gprel 重定位中立即数宽度的限制, 该优化仅适用于目标地址与 GOT 段基址偏移较小的情况. 绝大部分静态链接程序都满足该要求, 对于动态链接程序, 需要手动调整链接方式, 尽可能减小目标地址与 GOT 段基址的相对距离, 以适应该优化方式.

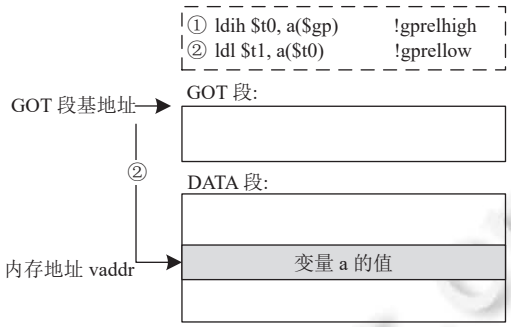


图 13 gprel 重定位基本原理

4 实验与分析

4.1 实验平台与测试用例

本文使用神威新一代超级计算机的计算节点作为实验平台, 通过交叉编译的方式, 在前端机完成编译并通过作业管理系统提交可执行文件到计算节点上运行. 本文使用 SPEC CPU2006 和 NPB3.0 两种主流的开源测试集对 swLLVM 进行评测, 各测试集的组成和功能简要介绍如下.

● SPEC CPU2006 是 SPEC 标准性能评测组织开发的用于评价 CPU 性能的一套基准程序. 该测试集包含 12 道整型测试用例以及 17 道浮点测试用例, 用于比较不同类型 CPU 的整型和浮点运算性能, 是当前编译领域权威的基准测试程序. 虽然 SPEC 组织已经推出了新版本 CPU2017 测试集, 但考虑到针对 CPU2006 已经积累的丰富的分析和优化经验, 其仍是进行第 1 轮编译性能分析的最佳选择.

● NPB (the NAS parallel benchmarks) 测试集是美国宇航局支持开发的并行测试程序. 该测试集包含 8 道课题, 测试范围从整数排序到复杂的数值计算. 为了评估编程语言对优化效果的影响, 本文分别使用官方 NPB3.0 测试集[35]以及开源的 NPB3.0-C 测试集[36]进行测试. 其中, NPB3.0 测试集的主要编程语言为 Fortran, 而 NPB3.0-C 测试集全部课题均由 C 语言实现.

4.2 优化效果评估

本文在 swLLVM 中实现了控制流向量优化, 以充分挖掘程序中的数据并行性. 向量化循环数是评价向量发掘效果的重要指标, 本文分别统计了 swLLVM 和 ICC18.0 编译器在 SPEC CPU2006 各课题上的向量化循环数, 如图 14 中折线图所示. 由于 SW26010Pro 处理器暂不支持 ICC 编译器, 因此 ICC 编译器性能数据为针对 Intel Xeon 处理器平台的统计结果. swLLVM 向量优化的平均向量化循环数为 520, 达到 ICC18.0 的 94.36%. 同时, 图 14 中以柱状图的形式展示了 swLLVM 向量优化的性能加速比. 如图 14 所示, SPEC CPU2006 测试集中的超过 65% 的课题均获得的性能提升, 平均加速比为 1.23. 其中, 462.libquantum 课题中执行时间占比 97% 以上的 3 个核心段均为包含 IF 嵌套的循环结构, 满足控制流向量优化的条件. swLLVM 通过 VSELECT 转换算法, 将循环中 IF 结构的控制相关转换为数据相关, 并基于向量条件选择指令实现向量优化, 在该课题上获得了高达 2.9 倍的性能提升.

借助 swLLVM 指令降级过程的 DAG 抽象表示, 本文实现了基于代价的节点合并优化. 图 15 展示了节点合并优化相较-O3 基准优化的加速比. 对于所有测试用例, 节点合并优化均取得了正向加速, 平均加速比为 1.11. 其中, 436.cactusADM 课题优化效果最明显, 加速比达到 1.36. 通过对该课题进行性能分析可以发现, 96.18% 的执行时间集中在函数 bench_staggeredleapfrog2_. 该函数包含多个嵌套循环, 循环内部使用的大量的临时变量, 总空间超过 477 MB. 节点合并能够尽可能减少冗余的临时变量, 优化寄存器溢出的问题. 同时, 该函数的核心计算部分包含

多个浮点乘法和浮点加法操作, 通过节点合并优化, 能够将其转换为 SW26010Pro 平台的特定加速指令, 从而提高程序的执行效率.

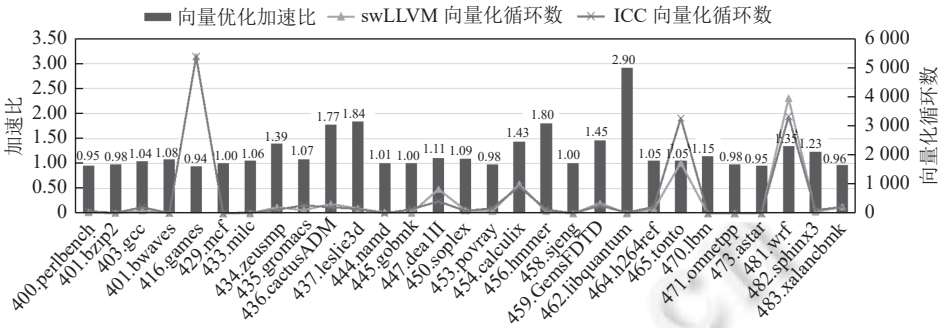


图 14 SPEC CPU2006 测试集上控制流向量优化效果

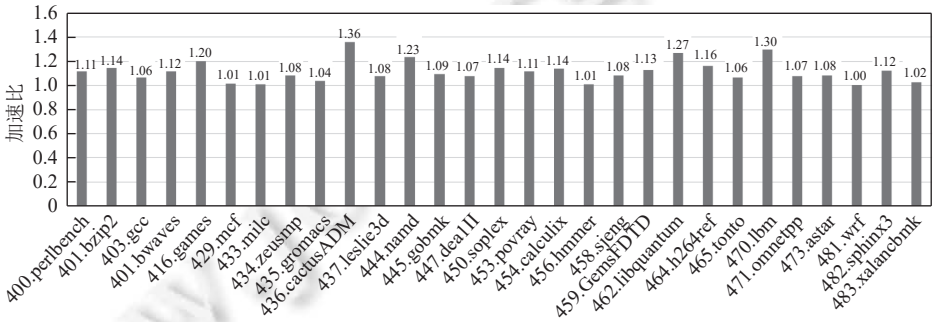


图 15 SPEC CPU2006 测试集上基于代价的节点合并优化效果

栈空间是异构众核程序执行过程中频繁访问的热点区域之一, 本文将运算核心的栈空间存储在 LDM 上以有效提升栈变量的访问速度. 图 16 给出了 swLLVM 编译器在 NPB 测试集上栈空间访问优化效果. 整体上, NPB3.0-C 测试集的优化效果优于 NPB3.0, 平均加速比达到 1.31. 其中, FT-C 的优化效果最为明显, 加速比达到 2.49. 较高的栈空间优化加速比表明程序存在大量的栈变量访问, 而较大的栈深度会导致较高的栈空间溢出风险. 针对该类应用, 用户可以在编译时添加-fcheck-ldm-stack 选项, 使用 swLLVM 的 LDM 空间栈溢出检查功能, 从而更加方便地进行程序检查及错误定位.

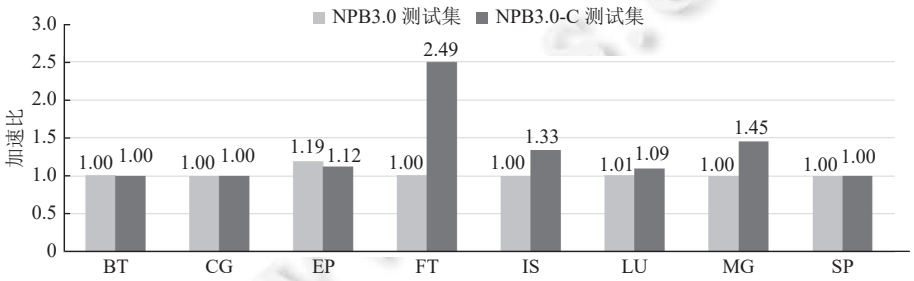


图 16 NPB3.0 测试集上栈空间访问优化效果

考虑到 LDM 空间容量受限, 对于无法放入 LDM 空间的非核心段数据, 主存空间的访问成为异构众核编译优化的主要性能瓶颈. 本文通过立即数的形式将只读数据与指令共同编译在代码段, 从而减少内存访问次数. 图 17 展示了 swLLVM 只读数据段访问优化的效果, 所有课题的平均加速比为 1.25. 其中 EP、IS、LU 三道课题优化效

果较为明显, 特别是 EP-C 课题, 加速比达到 2.34. 通过分析 ELF 文件信息可以发现, 优化前运算核心代码段大小为 26.38 KB, 优化后为 26.92 KB, 立即数加载及移位指令导致的代码段膨胀仅为 2%.

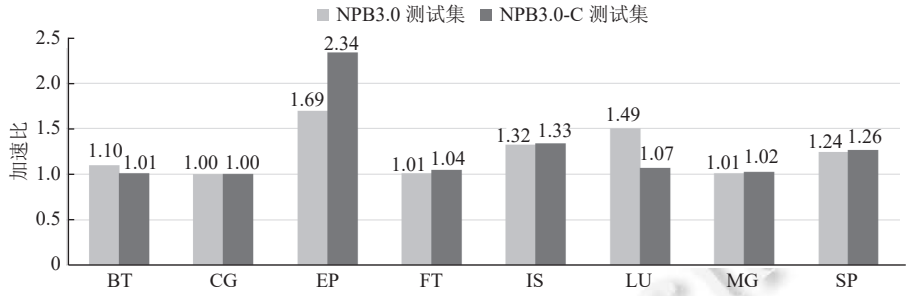


图 17 NPB3.0 测试集上只读数据访问优化效果

针对 literal 重定位引入的 GOT 段主存访问的问题, 本文在运算核心程序编译阶段将满足条件的 literal 重定位优化为 gprel 重定位, 从而避免获取变量/函数地址的主存访问. 该优化在 NPB3.0 测试集上的效果如图 18 所示. 对于不同 NPB 测试集中的相同课题, 全局变量访存优化的效果基本一致, 整体平均加速比为 1.06. 其中, IS-C 课题优化效果最为明显, 加速比达到 1.33.

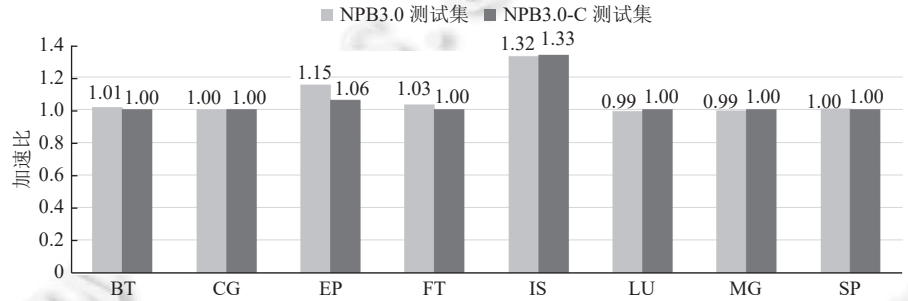


图 18 NPB3.0 测试集上地址访问优化效果

4.3 编译性能对比

基于本文提出的多种针对申威架构的编译优化技术, 结合 LLVM 丰富的架构无关优化方法, 本节从代码质量、编译速度和代码尺寸 3 个维度, 对 swLLVM 和 SWGCC 的编译性能进行综合对比.

为了评估代码质量, 本文使用 swLLVM 和 SWGCC 分别编译 SPEC CPU2006 测试集中的每个基准测试课题, 统计生成的可执行代码的运行时间, 并计算 swLLVM 相对 SWGCC 的加速比. 相对加速比的计算公式为:

$$\text{相对加速比}=\left(\frac{\text{运行时间}_{\text{SWGCC}}}{\text{运行时间}_{\text{swLLVM}}}-1\right)\times 100\%.$$

如图 19 所示, 在-O3 优化级别下, swLLVM 在 SPEC CPU2006 测试集上的相对加速比介于 118.2% (410.bwares 课题) 与−34.1% (465.tonto 课题) 之间, 平均相对加速比为 5.25%. 其中, swLLVM 在整型测试集 SPECint 上的性能与 SWGCC 相当, 平均相对加速比为–0.12%, 而在浮点型测试集 SPECfp 上性能较优, 平均相对加速比为 9.04%. 对于 410.bwaves 课题, 其核心段存在大量的离散访存及浮点除法操作, swLLVM 编译器针对数据和指令预取优化效果较好, 并且能够将浮点除法优化为浮点乘倒数操作, 因此性能提升较为明显. 而对于 465.tonto 课题, 由于其频繁调用数学库函数, 运算过程中会产生非规格化数, 非规格化数的运算要求系统进入模拟状态, 较为耗时, SWGCC 针对该问题提供了-ftz 选项, 在保证精度的前提下将非规格化数归零, 因此优化效果较 swLLVM 更优. 编程语言对 swLLVM 的编译优化效果也有一定的影响, 图 19 中用不同颜色的柱状图对 C/C++/Fortran 课题进行了区分. 如图所示, swLLVM 在 Fortran 课题上的性能较优, 平均加速比为 19.03%, 而在 C/C++课题上性能略低, 平均加速比分别为−1.25% 和−3.31%.

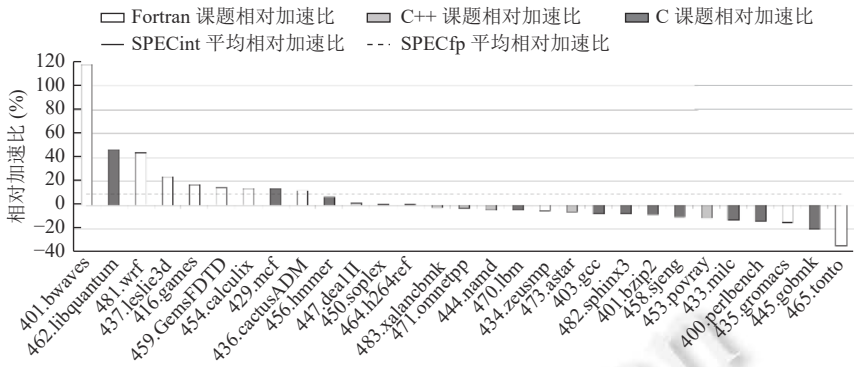


图 19 SPEC CPU2006 测试集上代码性能的相对加速比

随着软件规模和复杂度的增加, 编译构建时间占软件总开发时间的比例越来越高, 已成为衡量编译器性能的重要指标. 图 20 中以柱状图的形式对比了 swLLVM 和 SWGCC 在 SPEC CPU2006 测试集上的串行编译时间, 优化级别均设置为-O3. 如图 20 所示, swLLVM 的编译速度较 SWGCC 平均提升 79.1%, 尤其是对于 483.xalancbmk 课题, swLLVM 仅需 149.69 s, 较 SWGCC 提升至 5.6 倍. swLLVM 较短的编译时间能够有效降低软件测试和运行维护的难度, 提升异构众核平台上的软件开发效率.

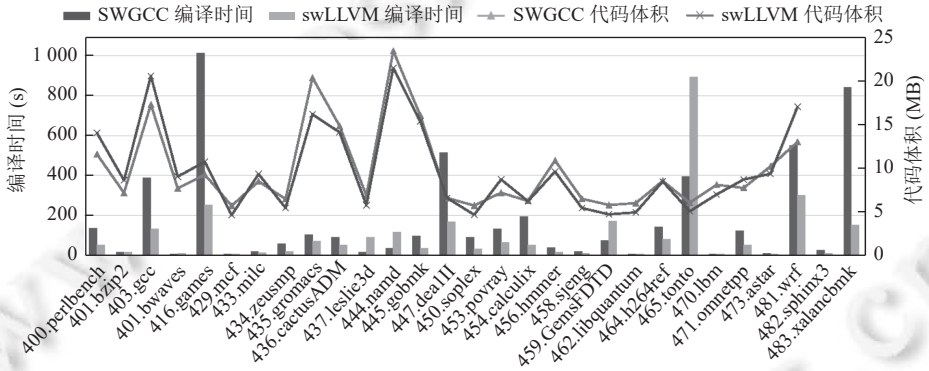


图 20 SPEC CPU2006 测试集上的编译时间和代码尺寸

编译优化通常需要在代码尺寸和运行性能之间进行权衡. 例如, 循环展开优化能够有效增加指令调度的机会, 减少分支指令的开销, 但是也会导致代码段膨胀, 影响指令 Cache 的局部性. 图 20 以折线图的形式比较了 SPEC CPU2006 测试集上 swLLVM 与 SWGCC 编译生成的静态链接可执行代码的尺寸, 优化级别均设置为-O3. 如图 20 所示, swLLVM 的可执行代码尺寸介于 21.32 MB (483.xalancbmk 课题) 和 4.56 MB (483.xalancbmk 课题) 之间, 平均代码尺寸为 9.35 MB, 较 SWGCC 减少 1.15%.

SWGCC 编译器作为神威系列超级计算机的基础支持软件, 历经“神威·蓝光”“神威·太湖之光”以及神威新一代超级计算机系统的锤炼, 开发时间较长, 积累了众多的优化技术, 支撑了一大批重大应用成果. swLLVM 编译器作为异构众核编译领域的后起之秀, 经过较短时间的开发和优化, 已在代码质量、编译速度和代码尺寸等指标上达到甚至超越了 SWGCC, 实验数据表明, swLLVM 是一款具有发展潜力的优化编译器, 未来将在异构众核编译领域扮演更加重要的角色.

5 结束语

为了充分发挥新一代申威异构众核处理器的体系结构优势, 有效支撑新兴科学计算应用的开发和优化, 本文在 LLVM 开源项目的基础上, 设计和实现了 swLLVM 优化编译器. swLLVM 采用模块化结构设计, 支持 C/C++/

Fortran 等高级语言, 支持 Athread 和 SDAA 双模态异构编程模型, 提供多级存储层次描述, 支持 SIMD 操作扩展, 同时针对 SW26010Pro 架构特点实现了控制流向量优化、基于代价的节点合并优化以及针对存储空间的管理及访问优化, 能够生成高效的异构众核可执行代码. 本文使用 SPEC CPU2006 以及 NPB3.0 标准测试集, 在神威新一代超级计算机上对 swLLVM 的优化效果及编译性能进行了详细评测. 实验结果显示, swLLVM 能够充分发挥 SW26010Pro 异构众核处理器体系结构、指令集和存储层次优势, 在代码质量、编译速度和代码尺寸等方面均取得了较好的优化效果.

目前我们已经基于 swLLVM 适配了多个领域的新兴应用, 包括阿里大数据计算服务 MaxCompute、深度学习编译器 TVM、漏洞挖掘框架 AFL、符号执行虚拟机 KLEE、新兴语言编译器 Numba 及 Julia 等. 在后续工作中, 我们将针对 SW26010Pro 的架构特点设计并实现新的优化方法, 进一步提升 swLLVM 的优化效果. 同时, 还需要结合用户需求继续开展新兴领域应用支撑, 进一步基于 swLLVM 拓展申威异构众核架构的应用潜力.

References:

[1] Li GG, Gui YD, Liu X. The briefly view on functions and status of high performance computation. Computer Applications and Software, 2006, 23(9): 3–4, 18 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-386X.2006.09.002]

[2] Liu S, Lu K, Guo Y, Liu Z, Chen HY, Lei YW, Sun HY, Yang QM, Chen XW, Chen SG, Liu BW, Lu JZ. A self-designed heterogeneous accelerator for exascale high performance computing. Journal of Computer Research and Development, 2021, 58(6): 1234–1237 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2021.20210189]

[3] Wu MC, Huang L, Liu Y, He XB, Feng XB. An OpenCL compiler for the homegrown heterogeneous many-core processor on the Sunway TaihuLight supercomputer. Chinese Journal of Computers, 2018, 41(10): 2236–2250 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2018.02236]

[4] Top500 list. 2022. https://www.top500.org

[5] He WQ, Liu Y, Fang YF, Wei D, Qi FB. Design and implementation of Parallel C programming language for domestic heterogeneous many-core systems. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 764–785. (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5197.htm [doi: 10.13328/j.cnki.jos.005197]

[6] Qian DP, Wang R. Key issues in exascale computing. SCIENTIA SINICA Informationis, 2020, 50(9): 1303–1326 (in Chinese with English abstract). [doi: 10.1360/SSI-2020-0099]

[7] Feng SZ, Li GG, Li XL, Qi FM, Huang D, Wan Y, Wu JC. Development strategy of emerging applications of HPC. Bulletin of Chinese Academy of Sciences, 2019, 34(6): 640–647 (in Chinese with English abstract). [doi: 10.16418/j.issn.1000-3045.2019.06.005]

[8] Yang X, Zhang S. A high performance distributed cloud computing frame for geophysical applications. In: Proc. of the 2016 Workshop: Workshop High Performance Computing. Beijing: Society of Exploration Geophysicists, 2016. 7–8. [doi: 10.1190/hpc2016-003]

[9] German DM. A study of the contributors of PostgreSQL. In: Proc. of the 2006 Int'l Workshop on Mining Software Repositories. Shanghai: ACM, 2006. 163–164. [doi: 10.1145/1137983.1138022]

[10] Chen YL, Jiang Y, Ma FC, Liang J, Wang MZ, Zhou CJ, Jiao X, Su Z. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In: Proc. of the 28th USENIX Security Symp. Santa Clara: USENIX, 2019. 1967–1983. [doi: 10.5555/3361338.3361474]

[11] Chen TQ, Moreau T, Jiang ZH, Zheng LM, Yan E, Cowan M, Shen HC, Wang LY, Hu YW, Ceze L, Guestrin C, Krishnamurthy A. TVM: An automated end-to-end optimizing compiler for deep learning. In: Proc. of the 13th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX, 2018. 579–594. [doi: 10.5555/3291168.3291211]

[12] Pytorch Glow. 2022. https://github.com/pytorch/glow

[13] Li MZ, Liu Y, Liu XY, Sun QX, You X, Yang HL, Luan ZZ, Gan L, Yang GW, Qian DP. The deep learning compiler: A comprehensive survey. IEEE Trans. on Parallel and Distributed Systems, 2021, 32(3): 708–727. [doi: 10.1109/TPDS.2020.3030548]

[14] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation. San Diego: USENIX, 2008. 209–224. [doi: 10.5555/1855741.1855756]

[15] Lattner C, Amini M, Bondhugula U, Cohen A, Davis A, Pienaar J, Riddle R, Shpeisman T, Vasilache N, Zinenko O. MLIR: Scaling compiler infrastructure for domain specific computation. In: Proc. of the 2021 IEEE/ACM Int'l Symp. on Code Generation and Optimization. Seoul: IEEE, 2021. 2–14. [doi: 10.1109/CGO51591.2021.9370308]

[16] 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]

[17] Lam SK, Pitrou A, Seibert S. Numba: A LLVM-based Python JIT compiler. In: Proc. of the 2nd Workshop on the LLVM Compiler Infrastructure in HPC. Austin: ACM, 2015. 7. [doi: 10.1145/2833157.2833162]

[18] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the 2004 IEEE/ACM Int'l Symp. on Code Generation and Optimization. San Jose: IEEE, 2004. 75–86. [doi: 10.1109/CGO.2004.1281665]

[19] 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, Networking, Storage and Analysis. Saint Louis: ACM, 2021. 3. [doi: 10.1145/3458817.3487399]

[20] PEZY. PEZY-SC2 Processor and Module. 2022. http://pezy.jp/products/pezy-sc2module-processor

[21] 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]

[22] Liao XK, Xiao LQ, Yang CQ, Lu YT. MilkyWay-2 supercomputer: System and application. Frontiers of Computer Science, 2014, 8(3): 345–356. [doi: 10.1007/s11704-014-3501-3]

[23] Reinders J, Ashbaugh B, Brodman J, Kinsner M, Pennycook J, Tian XM. Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL. Berkeley: Apress, 2021. [doi: 10.1007/978-1-4842-5574-2]

[24] Deakin T, Mclntosh-Smith S. Evaluating the performance of HPC-style SYCL applications. In: Proc. of the 2020 Int'l Workshop on OpenCL. Munich: ACM, 2020. 12. [doi: 10.1145/3388333.3388643]

[25] Yang C, Xue W, Fu HH, You HT, Wang XL, Ao YL, Liu FF, Gan L, Xu P, Wang LN, Yang GW, Zheng WM. 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In: Proc. of the 2016 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Salt Lake City: IEEE, 2016. 57–68. [doi: 10.1109/SC.2016.5]

[26] American Fuzzy Lop. 2022. http://lcamtuf.coredump.cx/afl

[27] LibFuzzer. 2022. https://github.com/Dor1s/libfuzzer-workshop

[28] Abadi M, Barham P, Chen JM, Chen ZF, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng XQ. TensorFlow: A system for large-scale machine learning. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation. Savannah: USENIX, 2016. 265–283. [doi: 10.5555/3026877.3026899]

[29] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin ZM, Gimelshein N, Antiga L, Desmaison A, Köpf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai JJ, Chintala S. PyTorch: An imperative style, high-performance deep learning library. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Vancouver: NIPS, 2019. 721. [doi: 10.5555/3454287.3455008]

[30] Nuzman D, Zaks A. Outer-loop vectorization: Revisited for short SIMD architectures. In: Proc. of the 17th Int'l Conf. on Parallel Architectures and Compilation Techniques. Toronto: ACM, 2008. 2–11. [doi: 10.1145/1454115.1454119]

[31] 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]

[32] Doerfert J, Grosser T, Hack S. Optimistic loop optimization. In: Proc. of the 2017 IEEE/ACM Int'l Symp. on Code Generation and Optimization. Austin: IEEE, 2017. 292–304. [doi: 10.1109/CGO.2017.7863748]

[33] McKeeman WM. Peephole optimization. Communications of the ACM, 1965, 8(7): 443–444. [doi: 10.1145/364995.365000]

[34] Hu W, Liu GM, Li Q, Jiang YH, Cai GL. Storage wall for exascale supercomputing. Frontiers of Information Technology & Electronic Engineering, 2016, 17(11): 1154–1175. [doi: 10.1631/FITEE.1601336]

[35] NPB 3.0. 2022. https://www.nas.nasa.gov/software/npb.html

[36] NAS-C-OpenMP3.0. 2022. http://benchmark-subsetting.github.io/cNPB

附中文参考文献:

[1] 李根国, 桂亚东, 刘欣. 浅谈高性能计算的地位及应用. 计算机应用与软件, 2006, 23(9): 3–4, 18. [doi: 10.3969/j.issn.1000-386X.2006.09.002]

[2] 刘胜, 卢凯, 郭阳, 刘仲, 陈海燕, 雷元武, 孙海燕, 杨乾明, 陈小文, 陈胜刚, 刘必慰, 鲁建壮. 一种自主设计的面向E级高性能计算的异构融合加速器. 计算机研究与发展, 2021, 58(6): 1234–1237. [doi: 10.7544/issn1000-1239.2021.20210189]

[3] 伍明川, 黄磊, 刘颖, 何先波, 冯晓兵. 面向神威·太湖之光的国产异构众核处理器OpenCL编译系统. 计算机学报, 2018, 41(10):

2236–2250. [doi: 10.11897/SP.J.1016.2018.02236]
[5] 何王全, 刘勇, 方燕飞, 魏迪, 漆锋滨. 面向国产异构众核系统的Parallel C语言设计与实现. 软件学报, 2017, 28(4): 764–785. http://www.jos.org.cn/1000-9825/5197.htm [doi: 10.13328/j.cnki.jos.005197]
[6] 钱德沛, 王锐. E级计算的几个问题. 中国科学: 信息科学, 2020, 50(9): 1303–1326. [doi: 10.1360/SSI-2020-0099]
[7] 冯圣中, 李根国, 栗学磊, 齐富民, 黄典, 万艺, 吴金成. 新兴高性能计算行业应用及发展战略. 中国科学院院刊, 2019, 34(6): 640–647. [doi: 10.16418/j.issn.1000-3045.2019.06.005]

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

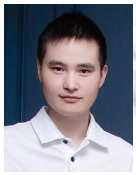
武文浩(1992－), 男, 工程师, 主要研究领域为异构众核编译系统.

周文浩(1991－), 男, 工程师, 主要研究领域为异构众核编译系统.

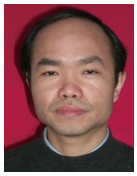
张鲁飞(1986－), 男, 博士, 工程师, 主要研究领域为人工智能, 操作系统.

王飞(1981－), 男, 博士生, 副研究员, 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, CCF 高级会员, 主要研究领域为并行计算系统, 众核芯片架构.



肖谦(1988－), 男, 博士生, 工程师, 主要研究领域为编译优化, 数据流计算, AI 框架.



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