软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.007098]

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

申威众核处理器访存与通信融合编译优化*

方燕飞, 李雁冰, 董恩铭, 王云飞, 刘 齐

(国家并行计算机工程技术研究中心, 北京 100190)

通信作者: 方燕飞, E-mail: flyyaj@163.com

摘 要: 申威众核片上多级存储层次是缓解众核“访存墙”的重要结构. 完全由软件管理的 SPM 结构和片上 RMA 通信机制给应用性能提升带来很多机会, 但也给应用程序开发优化与移植提出了很大挑战. 为充分挖掘片上存储层次特点提升应用程序性能, 同时减轻用户编程优化负担, 提出一种多级存储层次访存与通信融合的编译优化方法. 该方法首先设计融合编译指示, 将程序高层信息传递给编译器. 其次构建编译优化收益模型并设计启发式循环优化方案迭代求解框架, 并由编译器完成循环优化方案的求解和优化代码的变换. 通过编译生成的 DMA 和 RMA 批量数据传输操作, 将较低存储层次空间中高访问延迟的核心数据批量缓冲进低访问延迟的更高存储层次空间中. 在 3 个典型测试用例上进行优化实验测试与分析, 结果表明所提出的优化在性能上与手工优化相当, 较未优化版程序性能有显著提升.

关键词: 申威众核处理器; 多级存储层次; RMA 通信; 并行语言; 编译优化

中图法分类号: TP314

中文引用格式: 方燕飞, 李雁冰, 董恩铭, 王云飞, 刘齐. 申威众核处理器访存与通信融合编译优化. 软件学报. http://www.jos.org.cn/1000-9825/7098.htm

英文引用格式: Fang YF, Li YB, Dong EM, Wang YF, Liu Q. Memory Access and Communication Fusion Compiler Optimization for Sunway Many-core Processors. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7098.htm

Memory Access and Communication Fusion Compiler Optimization for Sunway Many-core Processors

FANG Yan-Fei, LI Yan-Bing, DONG En-Ming, WANG Yun-Fei, LIU Qi

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

Abstract: The on-chip memory hierarchy of Sunway many-core processors is an important structure to alleviate the many-core “memory access wall”. The SPM structure and on-chip RMA communication mechanism completely managed by software bring many opportunities for improving application performance but also pose great challenges for development optimization and porting of applications. To fully explore the hierarchical features of on-chip memory, improve application performance, and reduce the burden of user programming optimization, this study proposes a compiler optimization method that integrates multi-level memory access and communication. This method first designs a fusion compiler directive to transfer high-level information of the program to the compiler. Secondly, a compiler optimization revenue model is built and an iterative solution framework of a heuristic loop optimization scheme is designed. Meanwhile, the compiler completes the solution and code transformation of the loop optimization scheme. DMA and RMA batch data transmission operations are generated by compilation, batch buffer core data with high access latency from lower storage hierarchy spaces into higher storage hierarchy spaces with low access latency. Optimization experiments and analysis are conducted on three typical test cases, and the results show that the program performance optimized by this method is comparable to manual optimization, and significantly improves compared to the unoptimized version.

Key words: Sunway many-core processor; memory hierarchy; RMA communication; parallel language; compiler optimization

---

* 基金项目: 先进计算与智能工程 (国家级) 实验室基金; 国家重点研发计划重点专项 (2021YFB0301100)
本文由“编译技术与编译器设计”专题特约编辑冯晓兵研究员、郝丹教授、高耀清博士、左志强副教授推荐.
收稿时间: 2023-09-11; 修改时间: 2023-10-30; 采用时间: 2023-12-14; jos 在线出版时间: 2024-01-05

高性能计算 (HPC) 已经迈入 E 级 (E 为超级计算机运算单位, 意为每秒百亿亿次) 时代. 众核处理器 (many-core processor)[1−4]凭借超高的性能功耗比和性能面积比成为超算领域的主流处理器架构, 典型的高性能计算众核处理器有 NVIDIA、AMD 和 Intel 公司的 GPGPU、申威新一代众核处理器以及飞腾众核处理器 GX64-DSP 等. 随着工艺的不断发展, 众核处理器片上规模不断扩大, 核心数量不断增加, 众核计算能力不断提升, 但这并不意味着总是能带来众核应用程序的性能提升. 一方面, 程序员需要挖掘充分的应用并行性才能充分利用众多计算核心的并行性. 另一方面, 随着计算能力的不断提升, 众核访存能力提升却有限, 随着应用类型越来越多样化, 许多应用在众核上的计算遇到“访存墙”问题.

为应对“访存墙”挑战, 申威众核处理器设计了具有片上通信功能的从核多级存储层次结构, 片上 SRMA 采用便签式存储器 SPM (scratch pad memory) 和 Cache 的混合结构, 且两者容量灵活可配. 在片上高速网络的支持下, 片上 SPM 不仅可以配置为单个核心私有模式, 还可以配置成多核心共享等多种模式, 同时支持主存与 SPM 间的 DMA、不同核心 SPM 间的 RMA 通信等批量数据传输机制. 虽然引入片上 Cache 结构使众核编程工作在相对只有 SPM 结构时更容易, 但新一代申威众核处理器 SW26010Pro[5]只支持 1 级数据 Cache, 一旦发生 Cache 不命中, 将直接产生高延迟的主存访问. 而即使是具有较好 Cache 局部性的数组连续访问, 使用 Cache 和 SPM+DMA 编程在性能上也有较大差距, 经过在 SW26010Pro 上的实测发现, 前者性能只有后种访问方式的 80%. 此外, 片上 RMA 通信机制也是宝贵的资源, 高剑刚等人[6]通过所提出的面向申威众核处理器存储层次特点的 PLAN 并行计算模型, 证明了充分利用高速私有存储层 SPM 以及片上 RMA 通信机制可以很好地提高片上数据的利用率, 减少对主存访问带宽的需求. 为追求极致性能, 申威超算上的高性能应用事实上也大多采用 SPM 结构进行深度优化实现.

虽然在 SW26010Pro 上使用 SPM 方式编程能够给程序优化带来更大的灵活性和可控性, 应用开发人员可以充分利用运算核心间的 SPM 访问、RMA 通信等机制提升片上核心数据的重用率, 可以更好挖掘程序性能优化空间. 但与此同时, 完全由软件管理的 SPM 以及片上 RMA 通信机制也给众核应用程序优化和移植带来了很多挑战. 主要难点有两个方面: 一方面, 程序开发人员不仅要掌握应用数据访问特征, 还需要深入掌握目标处理器的存储层次特征, 才能通过算法设计与程序优化来挖掘利用硬件结构优势. 另一方面, 众核片上存储层次结构在不断发展, 不同类型的处理器, 同一类型不同代系的处理器, 其片上存储结构和容量都不尽相同, 在某一型号的众核处理器上深度优化的众核并行应用程序, 难以直接在另一款众核处理器上运行, 简单的移植改造也难以获得理想性能. 由于 SPM 完全由软件管理, 以 Cache 优化为核心的传统编译优化技术也不再适用, 并行编程语言编译需要针对众核混合片上存储层次的结构特点开展研究, 从编程、编译层面提供更多技术支持, 帮助程序开发人员减轻程序开发与优化负担.

为充分挖掘利用片上存储层次特点提升应用程序性能, 同时减轻用户编程优化负担, 本文提出了一种面向申威众核处理器的多级存储层次上访存与通信融合的访存编译优化方法, 采用编程与编译相结合的方式, 从编程语言、编译优化、运行时等多个层面开展协同设计, 其设计流程如图 1 所示, 通过对存储层次特征与应用访存模式提炼, 在语言层设计面向访存优化的编译指示语句, 引导应用程序开发人员将其掌握的程序信息进行恰当的描述, 为编译器优化提供必要的高层程序信息; 通过收益建模分析, 确定编译优化的制约关系变量及优化目标; 设计贪心求解算法完成循环变换方案的求解; 最后通过编译器代码变换生成优化代码, 达到提升应用程序访存效率的目的.

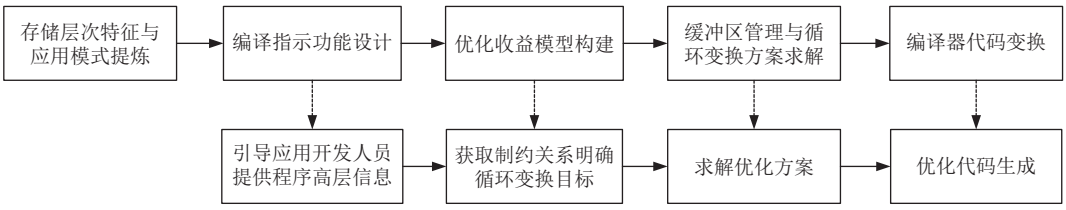


图 1 多级存储层次上访存与通信融合的编译优化方法设计流程

本文主要论述多级存储层次上访存与通信融合的编译优化方法的思路设计与具体实现. 具体内容如下: 第 2 节主要介绍优化方法所基于的硬件平台特征和编程语言特点. 第 3 节介绍融合访存优化方法设计. 第 4 节介绍基于优化收益模型的编译优化方案求解框架及编译实现. 第 5 节通过实验验证该优化方法的有效性. 第 6 节给出总

结与展望.

1 相关工作

国际上围绕 SPM 结构特点及片上数据移动优化开展了一系列研究. Venkataramani 等人针对 SPM 用于多线程应用程序时遇到的挑战, 提出了一个协调数据管理的编译时框架[7], 可以自动识别共享/私有变量, 并将这些变量及这些变量的拷贝, 一起放置到合适的片内或片外存储器中. Tao 等人提出了一个用于优化 SPM 和主存之间的多线程数据传输的编译时框架 MSDTM[8], 该框架通过应用程序分析和依赖性检查来确定数据传输操作的分配, 并通过所设计的性能模型来推导数据传输的最佳粒度. Chakraborty 等人设计了一个名为 UniSPM[9]的框架, 使用低开销的递归启发式算法来解决 NP-hard 的映射问题, 为基于 NoC 的 SPM 多核多阶段多线程应用程序提供统一的线程和数据映射框架. 李建江等人对 IBM Cell 的众核存储层次特点提出了一种基于运行支持库的 OpenMP 数组私有化的编译优化方法[10], 对可重用数据进行私有化, 充分利用有限的 SPM 资源减少 DMA 通信, 以提升程序执行效率. Yu 等人基于 SPM 提出了一种新的 GPU 资源管理方法[11], 通过将寄存器文件扩展到空闲的 SPM 资源上, 在逻辑上提供了一个更大的寄存器文件, 从而提高 GPU 的线程并行性. 上述存储层次相关优化工作针对 Cell、GPU 等商用众核架构特征开展, 而国产高性能申威众核处理器与上述硬件平台在众核架构和存储层次上都存在较大的差异, 因此需要探索针对申威众核处理器架构和存储结构特征的优化.

近年来, 业界基于申威众核平台开展了许多面向 SPM 结构特征的编译优化工作. 何王全等人在面向申威众核的并行编程语言 Parallel C 的设计与实现中提出了数据自动布局与 DMA 缓冲优化[12,13], 通过编译器分析程序数据访问特征和访存收益模型, 完成小容量数组在 SPM 上的自动布局, 并针对循环中的连续访问优化, 完成自动 DMA 缓冲优化. Wu 等人针对“神威·太湖之光”超算系统上支持 DMA 的 SPM 存储结构, 提出了一种基于带宽感知的 OpenCL 程序循环平铺方法[14], 对传统的仅带宽和仅容量考虑的循环平铺方法进行了改进, 有效提升了带宽利用率和 SPM 重用. 在实现 OpenCL 编译系统在申威众核处理器上的内存模型映射过程中, 实施了针对 SPM 的访存优化, 即在 SPM 空间充足的情况下, 将主存中的数据通过 DMA 存放到 SPM 中[15]. Zhou 等人针对申威众核处理器核心的 SPM 存储结构特点, 开展了基础函数库的内存延迟优化[16], 他们提出了一种有效的自动数据转换方法和一种表查找方法来优化基础函数库的访存延迟. 上述工作充分利用了申威众核处理器 SPM+DMA 的硬件特征, 但是限于目标平台特点, 没有考虑到新一代众核处理器上私有 SPM+共享 SPM+DMA+RMA 的片上存储层次与数据移动机制特征, 无法发挥共享 SPM 及片上 RMA 通信的优势.

2 研究平台及编程语言

本文所提方法主要面向申威众核处理器进行设计, 基于新一代申威众核上的并行编程语言及编译系统 Parallel C 2.0 进行实现. 下面就相关概念和基本知识予以介绍.

2.1 新一代申威众核处理器

本文的研究工作面向国产申威众核处理器. 申威众核处理器已经经过多代发展, 下面以神威新一代超级计算机使用的最新一代申威众核处理器 SW26010Pro[17]为例, 对申威众核处理器的结构进行介绍.

SW26010Pro 是面向高性能计算领域开发的处理器, 采用片上计算阵列集群和共享存储结构相结合的异构众核体系结构. 如图 2 所示, SW26010Pro 处理器芯片主要由 6 个核组组成. 核组内采用异构众核结构, 包含 1 个管理核心和 1 个运算核心阵列. 每个运算核心阵列包含 64 个运算核心, 采用拓扑为 8×8 的阵列高速通信网络进行连接, 支持运算核心间 SPM 上的 RMA 通信.

单个核组的存储系统结构如图 3 所示, 运算核心有 256 KB 的 SRAM 作为数据缓存, SRAM 可分档配置为 SPM 和 Cache. 所有运算核心共享主存, 可以通过 ld/st 访存指令直接访问主存空间. 运算核心还可以通过 DMA 实现 SPM 和主存之间的批量数据传输, DMA 传输的效率与传输的数据量、DMA 命令数量、数据在主存中的连续性以及 DMA 传输方式等密切相关. 此外, 运算核心阵列还支持 SPM 共享和 RMA 两种机制实现阵列上 SPM 间的

数据交互. SPM 共享机制允许每个运算核心拿出部分 SPM 作为阵列共享 SPM, 共享范围和共享容量均有多种模式可选. 从核访问共享 SPM 的延迟介于访问私有 SPM 和主存之间. RMA 支持阵列上运算核心间 SPM 空间的通信, 能够实现高效的 SPM 数据传输. 运算核心访问 SPM 的延迟远低于访问主存的延迟, 但由于芯片面积及功耗的限制, SPM 的容量有限. 在实际应用中, 利用运算核心进行加速计算时, 充分利用访问时延短的 SPM 才能充分发挥 SW26010Pro 处理器的性能优势.

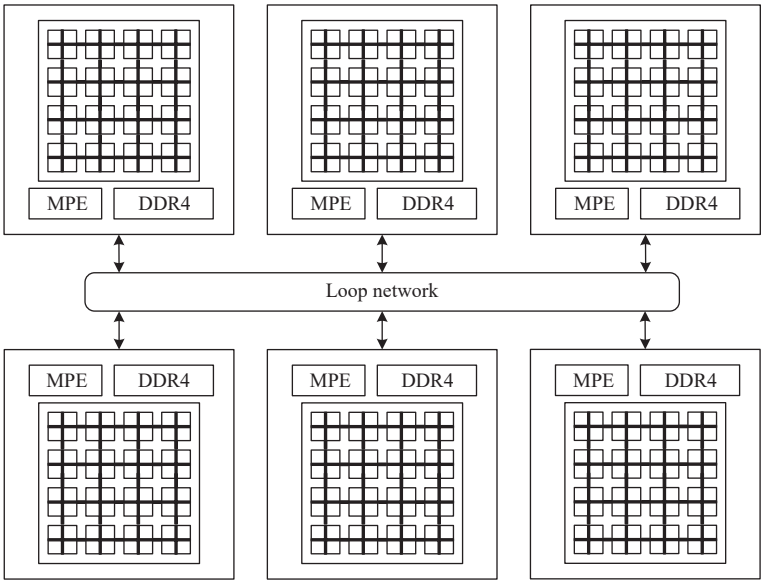


图 2　SW26010Pro 异构众核处理器逻辑示意图

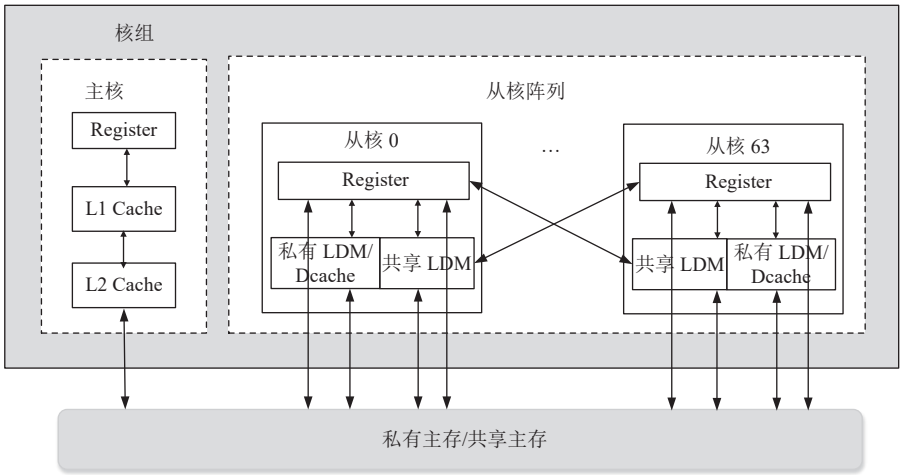


图 3　SW26010Pro 核组存储系统结构图

2.2　申威众核并行编程语言 Parallel C

Parallel C[12]是面向申威异构众核架构设计的大规模众核并行编程语言, 随着众核处理器的发展, 在兼容主体功能的基础上增加了面向新一代申威众核处理器特点的功能扩展, 目前最新版本为 2.0, 下文针对 Parallel C 2.0 功能进行介绍. Parallel C 支持异构加速编程模式, Parallel 程序分为进程程序和线程程序, 如图 4 所示, 进程程序运行在运算控制核心上, 负责数据和加速任务的分配和管理; 线程程序运行在运算核心上, 实现对核心功能模块的加速.

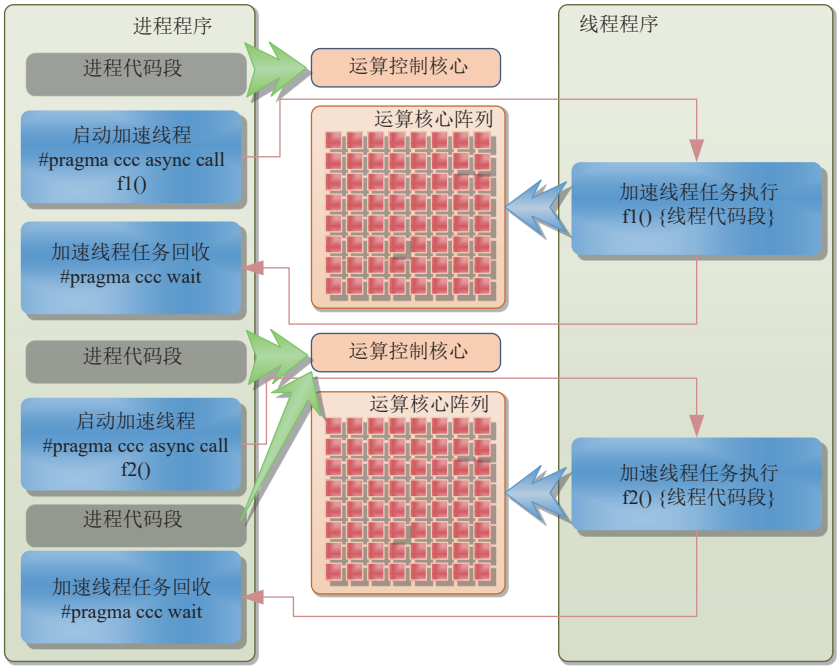


图 4 Parallel C 主从异构加速程序示意

为实现申威异构众核处理器上的异构加速并行编程, Parallel C 提供主核函数定义、从核函数调用等异构加速描述功能, Parallel C 使用编译指示的形式实现从核代码描述相关功能, 方便用户增量式的进行代码移植. Parallel C 提供的从核代码描述编译指示如表 1 所示, 包括从核函数编译指示 th_func、主核从核共用函数编译指示 com_func, 以及从核函数阻塞调用 sync_call 和非阻塞调用 async_call 两种线程函数调用编译指示.

表 1 Parallel C 编译指示列表

编译指示	书写格式	简要功能说明
th_func	#pragma ccc th_func	定义或声明线程函数
com_func	#pragma ccc com_func	定义或声明进程线程共用函数
sync_call	#pragma ccc sync_call	以阻塞调用方式启动线程入口函数
async_call	#pragma ccc async_call	以非阻塞调用方式启动线程入口函数
wait	#pragma ccc wait	等待线程入口函数返回

Parallel C 是一个两级并行的执行模型, 如图 5 所示. 第 1 级是进程并行, 进程并行是基于消息传递的分布式并行, 类似于 MPI. 第 2 级是线程并行, 每个进程可以创建多个线程, 用来加速进程任务的执行. 一个并行 C 作业包括多个进程, 每个进程又可以创建多个线程. Parallel C 的进程和线程由运行时环境根据用户指定的进程数、线程数自动创建.

Parallel C 的存储模型与申威众核处理器的存储层次一一对应, 图 6 给出了 Parallel C 进程和线程视角的存储层次示意. Parallel C 支持 6 种存储空间的描述, 具体见表 2.

Parallel C 进程可以访问的存储空间包括进程私有主存空间和进程共享主存空间, 也称为节点共享空间, 只能在一定范围实现共享, 与处理器结构相关. Parallel C 线程可以访问的存储空间包括线程私有主存空间、线程私有局存 (LDM) 空间 (对应 SPM) 和线程共享局存 (SDM) 空间 (对应共享 SPM 空间), 以及继承自父进程的线程共享主存空间. 进程私有主存空间可以被进程自己和其创建的线程访问, 数据默认位于进程私有主存空间; 节点共享空间则可以被同一节点内的所有进程和线程访问, 需要使用__node_shared 关键字显式说明. 线程共享主存空间能够被共享范围内的进程和线程访问, 只能从创建它的进程继承而来, 不能显式声明; 线程私有主存空间只能被线程自己访问, 使用关键字__thread 显式说明; 线程私有 LDM 空间也只能被线程自己访问, 该空间的变量使用存储关键

字__thread_ldm 声明, 下文称作 LDM 变量; 线程共享 SPM 空间可以被共享范围内的所有线程访问, 该空间的变量使用存储关键字__thread_sdm 声明, 下文称作 SDM 变量.

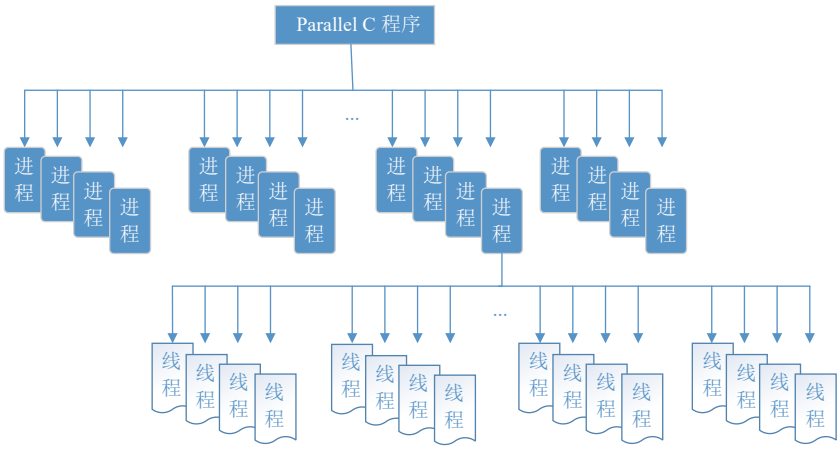


图 5 Parallel C 两级并行执行模型示意

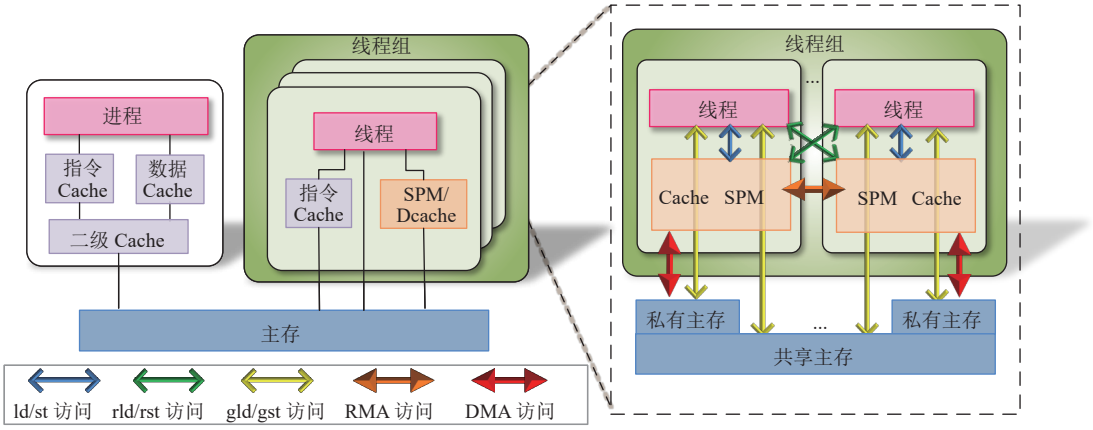


图 6 Parallel C 存储模型示意

表 2 Parallel C 存储空间类型

视角	空间类型	存储位置	关键字
进程视角	进程私有主存空间	主存	缺省
	节点共享主存空间	主存	__node_shared
线程视角	线程私有主存空间	主存	__thread
	线程共享主存空间	主存	缺省
	线程私有SPM空间	SPM	__thread_ldm
	线程共享SPM空间	SPM	__thread_sdm
	节点共享主存空间	主存	__node_shared

3 多层次存储访存融合优化设计

软件管理的私有和共享 SPM、DMA 批量数据传输和 RMA 通信机制给程序优化提供了很大的灵活性, 但程序实现过程比较复杂, 本文结合处理器存储层次结构特征和应用常见数据访问模式, 在 Parallel C 中设计了面向多级存储层次的访存通信融合的编译优化技术. 下面将具体阐述该优化技术的设计方法.

3.1 多级存储层次访存与通信融合的编译优化设计思路

申威众核处理器片上存储层次丰富, 特征明显, 合理利用好存储层次特征是一项挑战性工作. 如图 7 所示, 线程有多种方式访问各种存储空间, 不同方式下开销有较大区别, 分别有以下几种情况: 1) 通过 ld/st 访问私有 LDM 空间, 延迟只有几拍; 2) 通过 rld/rst 访问 SDM 空间, 无冲突情况下延迟数十拍; 3) 通过 gld/gst 访问主存, 无冲突情况下延迟数百拍. 此外, 线程还可将 LDM 作为缓冲, 通过 DMA 将主存数据批量传输到 LDM 空间, 或者通过 RMA 将其他核心上的 LDM 空间数据或者 SDM 空间数据批量传输到本核心的 LDM 空间. 核组 DMA 峰值带宽数十 GB/s, 每核心平均约数百 MB/s, RMA 点对点带宽数 GB/s. 在这种存储层次特征下, 应用程序开发人员必须想尽办法将程序核心数据布局进更高层次的存储空间, 对于无法完全布局进 LDM 的核心数据, 要尽可能利用上 DMA 批量数据传输和 RMA 通信机制提高访存效率. 对于普通程序员而言, 这显然是一项很有挑战的工作. 为此, 本文从并行语言和编译层面提出优化方案, 以减轻程序员的开发负担, 提高程序的访存性能.

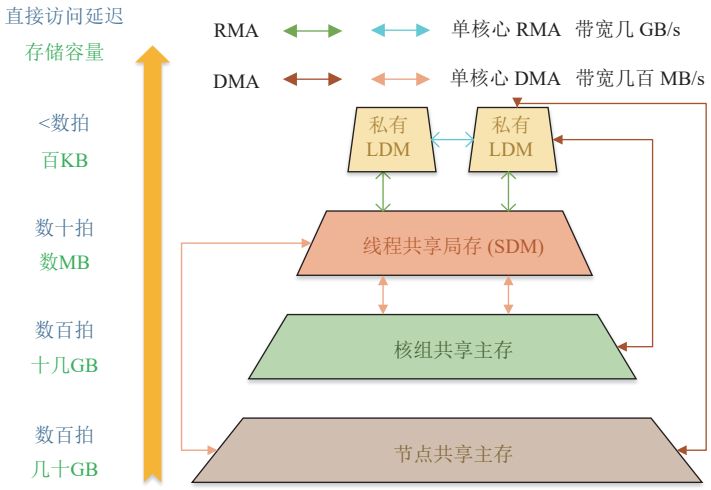


图 7 多级存储层次间数据传输优化示意图

通过对典型科学计算类应用数据访问特征的分析可以发现, 被核心循环访问的可以借助更高层次存储空间进行访存优化的数组访问基本可以分为以下 4 种: 1) 主存数组连续; 2) 主存数组局部离散访问; 3) SDM 数组连续访问; 4) SDM 数组局部离散访问. 其中主存数组可以分段 DMA 缓冲进 LDM 或 SDM, SDM 数组可以通过运行时提供的集合 RMA 消息分段缓冲进 LDM. 图 8 中给出了具有典型优化机会的程序段示例.

图 8 给出典型的可优化访存场景代码示意. 1) 在循环 j 中, 主存数组 source 为连续读访问, 主存数组 data 为局部离散访问. 优化机会: 可对循环 j 进行分块, 把 source 对应的数据分段传输进 SPM 缓冲区. 而对于 data, 可以在循环 j 外一次性传输进 data[i] 的一整维, 由于 data 的第一维需要 4 MB 空间, 所以需要用 SDM 空间作为缓冲. 2) 循环 k、l 中, 主存数组 data 为连续访问. 优化机会: 可对循环 l 进行分块, 把 data[k] 对应的数据分段传输进 SPM 缓冲区. 3) 循环 n 中, SDM 空间数组 sdm_sum 为连续读访问. 优化机会: 由 0 号线程以 RMA 方式, 将物理分布在远程 LDM 空间上的 sdm_sum 的数据缓冲进 0 号线程本地 SPM 缓冲空间, 然后再对本地缓冲数据进行累加计算.

基于上述优化机会分析, 本文设计了一种 DMA 访存和 RMA 通信融合的访存优化方法, 实现循环中多个不同地址空间中的数组的多访问方式融合访存优化.

融合访存优化方法根据被优化数组的源地址空间类型、缓冲地址空间类型可以细分为以下 3 类: 1) 主存访问 DMA 缓冲 LDM 优化 (下文简称 mem2ldm 优化): 利用 DMA 批量数据传输功能将数据缓冲进 LDM 空间; 2) SDM 连续访问同源聚合 RMA 缓冲 LDM 优化 (下文简称 sdm2ldm 优化): 将以细粒度方式连续访问或者局部离散访问的 SDM 变量的数据按照源地址中的线程号信息进行分类聚合, 利用运行时提供的集合 RMA 消息功能完成 SDM 到 LDM 的数据批量传输; 3) 主存访问 DMA 缓冲 SDM 优化 (下文简称 mem2sdm 优化): 将循环中主存空间连续访问

或者一定范围内的离散访问数据批量传输进 SDM 空间, 将原有主存高延迟访问转换为 SDM 空间的较低延迟访问.

上述 3 种优化方式的数据移动路径如图 7 所示, 根据各级存储空间的访问延迟可以看出, 优化后的数据访问延迟将从数百拍提升至数十拍甚至数拍, 在单次传输的数据量远大于细粒度访问的数据宽度情况下, 批量传输的数据访问效率远高于单个细粒度的数据访问, 所以整体效率可以获得明显提升.

```
#include <parallel.h>
#define N 1L<<30
#define M 1<<20
//data和source为主存数组
int data[N][M];
int source[N][N];
__thread_sdm int sum[64];//sum为SDM数组
#pragma  ccc   th_func
void kernel()
{
        …
    // 每个线程不同任务块
    for(i=mybs;i<mybe;i++)
    {
       for(j=0;j<N;j++)
       {
    // 局部离散写访问主存数组data
    // 连续读访问数组Source
            pose = func(source[i][j]);
            data[i][pose] =  ...;
       }// 循环j
       m_sync(CCC_ARRAY);//核组同步
    }// 循环i
```

```
// 连续读访问主存数组data
    for(k=0;k<N;k++)
    for(l=0;l<M;l++)
    {
        mysum=data[k][l]+...;
    }
    sum[myid] = mysum;
    m_sync(CCC_ARRAY);//核组同步
    if(myid==0)
    {
    // 连续读访问SDM数组sum
        for(n=1;n<64;n++)
            sum[0]+=sum[n];
     }
       …
}
```

图 8　可优化的访存场景示意图

3.2　多级存储层次访存融合优化编译指示设计

在编译器访存融合优化方法进行实现的过程中, 为保证程序的正确性, 必须开展精确的数据相关性分析, 而依赖编译器的自动分析有一定的局限性, 对于核心循环有函数调用、指针别名等情况下, 编译时难以获取准确的数据相关性信息, 从而导致编译器放弃一些潜在的优化机会, 比如在图 8 中, 函数 fun 中是否有对全局数组 source 的其他别名访问, 编译器如果分析不清楚, 就会放弃对 source 的访存优化. 编译指示是一种将程序设计人员掌握的程序信息或者优化意图传递给编译器的通用方式, 结合了人的经验和编译器的自动分析变换能力, 是应对复杂编程优化问题的一种常用方式. 为了使面向多级存储层次的访存融合优化方法能够对应用程序具有更好的适应性, 挖掘更多优化机会, 本文设计了相应的优化编译指示, 采用半自动的方式对程序性能进行优化实现.

从存储空间类型的角度, 本文设计了 3 种缓冲优化模式, 分别对应第 3.1 节中的 3 种优化, 即 mem2ldm、sdm2ldm 和 mem2sdm. 另外, 从缓冲区数量的角度, 又可以将上述每种优化区分为单缓冲优化和双缓冲优化. Parallel C 设计的缓冲优化编译指示具体说明如下.

编译指示格式如下:

```
#pragma ccc sbuf/dbuf mem2ldm/sdm2ldm/mem2sdm (变量列表) \
ldm_addr/sdm_addr (缓冲区地址) ldm_size/sdm_size (整数表达式)
```

该编译指示用在循环结构之前. 指导语句及各子句的含义见表 3.

为引导用户写出正确的编译指示, 以确保编译器可以开展高效正确的优化, 需做以下定义.

1) 优化维的定义: 编译器在数组访问方式分析时, 对每个数组从最高维到低维依次分析各维的访问下标, 将分析时遇到的第 1 个下标是非循环不变量的维定义为优化维.

2) 离散访问数组的定义: 优化维的下标不是循环变量的放射下标.

3) 连续访问数组的定义: 优化维的下标是循环变量的放射下标, 形如: $i+c$, 其中 i 为被指示循环的索引变量, c 为常量.

4) 连续访问数组的元素宽度定义: 优化维以下的所有更低维的数据总长量.

5) 局部离散访问数组的元素宽度定义: 优化维及以下的所有更低维的数据总长量.

6) 编译指示使用合法性定义: 编译指示所作用的循环中, 若存在指针, 被优化指示的数组与指针访问不存在读写相关或者写写相关.

表 3　多级存储层次访问融合优化编译指示说明

字句	含义
sbuf	指示使用单缓冲区进行优化
dbuf	指示使用双缓冲区进行优化
mem2ldm	指示变量列表中是循环中可以使用DMA缓冲优化的连续访问或局部离散访问的主存变量
sdm2ldm	指示变量列表中是循环中可以使用RMA缓冲优化的连续访问或局部离散访问的ldm共享变量
mem2sdm	指示变量列表中是循环中可以使用DMA缓冲优化的连续访问或局部离散访问的主存变量
ldm_addr	指示缓冲区地址
sdm_addr	指示共享数据缓冲区地址
ldm_size	指示用于访存优化的ldm缓冲空间的大小 (以字节为单位)
sdm_size	指示用于访存优化的sdm缓冲空间的大小 (以字节为单位)

程序员在使用制导语句优化一个数组时, 该数组在循环中的所有访问必须是满足以上定义的局部离散访问数组或者连续访问数组, 才允许被编译指示. 而且在编译器优化时, 如果新定义的数组元素宽度超过用户给出的缓冲区容量, 直接放弃对该数组的优化. 其他代码变换合法性由编译系统保证.

3.3 编译指示使用示例

多级存储层次访存通信融合编译优化的编译指示的设计思路是在用户指导下, 编译器将多次对局部离散的主存或 SDM 访问集中为一次或者几次高效的批量数据传输, 利用 DMA 和 RMA 机制的高带宽提升访存性能. 程序设计人员只需要将可以优化的变量以及可用的缓冲区信息写入编译指示, 编译器自动完成 DMA 和 RMA 操作的生成和相应的程序变换. 图 9 展示了使用编译指示对第 3.1 节中的示例代码进行优化指示的编程示例. 编译器将根据指示信息, 对相应循环中的数组访问 source、data 和 sdm_sum 进行缓冲优化.

```
#include <parallel.h>
#define N 1L<<30
#define M 1<<20
//data和source为主存数组
int data[N]  [M];
int source[N]  [N];
__thread_sdm intsum[64];//sum为SDM数组
__thread_ldm int ldm_buf[1024];
__thread_sdm int sdm_buf[M];
#pragma ccc th_func
void kernel()
{
        …
    //每个线程不同任务块
    for(i=mybs;i<mybe;i++)
    {
        #pragma ccc sbuf mem2ldm(source)
ldm_addr(ldm_buf) ldm_size(4096)
        #pragma ccc sbuf mem2sdm(data)
sdm_addr (sdm_buf) sdm_size (1<<24)
        for(j=0;j<N;j++)
        {
            pose = func(source[i][j]);
            data[i][pose] =  ...;
        }//循环j
        m_sync(CCC_ARRAY);//核组同步
    }//循环i

    //连续读访问主存数组data
        for(k=0;k<N;k++)
        {
#pragma ccc sbuf mem2ldm (data)
ldm_addr(ldm_buf) ldm_size (4096)
            for(l=0;l<M;l++)
            {
                mysum=data[k][l]+...;
            }
        }
        sum[myid] = mysum;
        m_sync(CCC_ARRAY);//核组同步
        if(myid==0)
        {
        // 连续读访问SDM数组sum
#pragma ccc sbuf sdm2ldm(sum)
ldm_addr(ldm_buf) ldm_size(256)
            for(n=1;n<64;n++)
                sum[0]+=sum[n];
        }
            …
}
```

图 9　编译指示优化示例

4 基于优化收益模型的编译器分析与实现

本文设计的编译指示的典型应用背景是: 用户掌握基本的存储层次特征, 对于小容量核心数据, 用户已经用相关关键字变量进行声明, 布局进最适合的存储层次中, 或者基于前期已有编译优化[12], 可以由编译器一次性布局进最适合的存储层次中. 因此, 需要用本文第 3 节中设计的优化编译指示的典型场景是: 循环中有大数组变量的访问, 这些数组不能一次性布局进 LDM 或 SDM 空间, 但是具有较好的空间局部性, 可以通过循环变换, 分段缓冲进 LDM 或者 SDM 空间, 增加重用性, 减少访问开销.

对于主存局部离散访问数组的优化, 主要是由编译器根据指示信息完成相应空间上的一整块数据传输和循环中相应数组访问的直接变换, 而无需考虑对循环的分块以及对数组访问的映射. 因此, 下文重点将放在对连续访问数组的优化建模的讨论上. 另外, 双缓冲与单缓冲的建模原理相同, 可以在单缓冲模型基础上稍作调整即可得出双缓冲优化方案, 为方便描述, 下文重点讨论单缓冲编译指示下的连续访问数组的优化过程.

4.1 优化收益模型设计与制约关系分析

当编译器根据指示信息开展循环优化时, 所采取的循环优化方案必须解决以下 3 个关键问题.

1) 对循环进行合理分块. 待优化的数组都是难以一次性布局进更高层存储层次的, 必须对循环进行合理分块, 才能分段传输到缓冲区. 如何选取循环分块长度, 使得缓冲区空间需求不超过可用空间, 又能使缓冲开销尽量小, 是在循环分块方案求解中要解决的问题.

2) 对数组访问进行合理分区 (以下简称数组分区为 region). 当被编译指示的循环中有多个变量或者同一变量的多个不同引用需要进行访问优化时, 不仅需要对缓冲区进行合理划分以缓冲不同的数组, 同一数组的多个不同访问也需要进行合理的聚类. 对不同的引用如何进行聚类, 以及如何划分缓冲区, 有多种可能的方案. 需要一种能够高效快速找到优势聚类方案的方法. 聚类过程还需要考虑依赖距离, 以确保代码变换的合法性问题. 如图 10 所示, 循环中共有 4 个访问 a[i]、a[i+1]、a[i–1] 和 a[i+128], 编译器需要结合依赖距离考虑 4 个不同访问的分区聚类方法和循环分块长度.

```
#pragma ccc sbuf mem2ldm(a) ldm_addr(buf) ldm_size(1024)
for(i = 1; i <N; i++)
{
    a[i]+= (a[i+1] + a[i-1]);
    a[i+128] += (a[i+1] -a[i-1]);
    …//其他代码
}
```

图 10 需要考虑分区聚类的数组访问代码示意

3) 任何一个数组访问在被实施优化后所取得的性能提升不能为负. 如果因为不合理的分块或者分区导致某些数组访问优化引入的开销超过延迟降低带来的收益, 则应该放弃这样的优化方案. 如图 9 所示, 最后一个循环中对 sum 的优化, 需要结合 sum 的物理地址分布情况以及 RMA 的启动开销和 sdm 访问延迟综合判断是否要对 sum 数组访问实施优化.

本文首先构建优化收益与开销模型, 寻找时间开销与空间开销的制约关系变量以及优化收益目标函数. 然后基于开销与空间开销的制约关系设计了二分搜索算法的循环分块和 region 求解方法. 最后, 基于收益模型设计了启发式循环优化方案迭代求解框架. 下文先介绍优化收益与开销模型.

在模型构建过程中, 本文对 DMA/RMA 启动开销、传输带宽等系统性能数据进行参数化描述, 构建访存和片上通信融合的收益模型及循环优化方案求解算法.

设待优化循环中共有 N 个待优化的数组 (经过前述规则做了转化), 序号 0, 1,…, N–1; 不妨设 $N=N1+N2+N3$, 其中 $N1$ 个数组属于 mem2ldm, $N2$ 个数组属于 mem2sdm, $N3$ 个属于 sdm2ldm, 每个数组在循环中有若干个访问. 在进行计算循环分块大小时, 为了充分利用缓冲区空间, 需要对同一数组的访问进行聚类, 同一类的称作一个 region, 一个 region 对应一个 DMA 或 RMA 缓冲, 一个 region 需要的缓冲区长度设为 region_space.

根据转换规则, 对于同一 region 中的数组访问, 优化维索引表达式有如下形式: $i+S_k$, 其中 i 为循环变量, S_k 为

常量. 不妨设 region 中有 n 个访问, 且有 $S_1 < S_2 < \ldots < S_n$. 并设该数组的元素大小为 *region_size* (根据转换规则转换后的元素大小), 循环分块长度为 *block_len*, 则一个 region 的空间需求 *region_space* 的计算如公式 (1) 所示:

$$region\_space = (block\_len + S_n - S_1) \times region\_size \tag{1}$$

循环优化总的 DMA 或 RMA 缓冲区开销计算如公式 (2) 和公式 (3) 所示:

$$ldm\_space = \sum_{i=1}^{M_1+M_3} region\_space_i \tag{2}$$

$$sdm\_space = \sum_{i=1}^{M_2} region\_space_i \tag{3}$$

缓冲区需求必须不能超过编译指示中给出的缓冲区大小, 因此, 循环分块大小 *block_len*、region 数与缓冲区大小的约束关系如公式 (4) 和公式 (5) 所示:

$$ldm\_space = \sum_{i=1}^{M_1+M_3} region\_space_i \leqslant ldm\_size \tag{4}$$

$$sdm\_space = \sum_{i=1}^{M_2} region\_space_i \leqslant sdm\_size \tag{5}$$

公式 (1)–公式 (5) 的相关变量含义见表 4. 从公式 (1)–公式 (5) 可以看出, 循环分块长度 *block_len* 受缓冲区空间大小的制约.

表 4　变量一览表 1

参数	含义
block_len	循环分块长度, 即分块内的迭代数
M_1	$N1$个数组 (mem2ldm)的访问region数
M_2	$N2$个数组 (mem2sdm)的访问region数
M_3	$N3$个数组 (sdm2ldm)的访问region数
$region\_size_i$	第i个region的数组元素长度
$region\_space_i$	第i个region需要的缓冲区空间大小
ldm_space	LDM缓冲空间需求总和
sdm_space	SDM缓冲空间需求总和
ldm_size	编译指示中给出的LDM缓冲区空间大小
sdm_size	编译指示中给出的SDM缓冲区空间大小

每个 region 的优化时间收益计算计算如公式 (6) 所示:

$$TR_{\text{region}} = ITERS \times Delay_{\text{opt}} - \left[T0 + Block\_num \times \frac{region\_size}{B \times \mu} \right] \tag{6}$$

其中, $Delay_{\text{opt}}$ 值根据 region 的数组优化类型取 $Delay_{\text{mem}} - Delay_{\text{sdm}}$ 或者 $Delay_{\text{sdm}} - Delay_{\text{ldm}}$, B 根据缓冲数据传输方式取 *B_DMA* 或者 *B_RMA*. 其他参数见表 4 和表 5.

每个 region 的收益开销比如公式 (7) 所示:

$$P_{\text{region}} = \frac{ITERS \times Delay_{\text{opt}} - \left[T0 + Block\_num \times \dfrac{region\_size}{B \times \mu} \right]}{region\_space} \tag{7}$$

每个循环分块的数据传输时间开销计算如公式 (8) 所示:

$$T_{\text{trans}} = (M_1 + M_2) \times T0_{\text{dma}} + M_3 \times T0_{\text{rma}} + \sum_{i=1}^{M_1+M_2} \left(\frac{region\_space_i}{B\_DMA \times \mu} \right) + \sum_{i=1}^{M_3} \left(\frac{region\_space_i}{B\_RMA \times \mu} \right) \tag{8}$$

循环优化后的访问总开销如公式 (9) 所示:

$$T_{\text{opt}} = Block\_num \times T_{\text{trans}} + ITERS \times [(Na1 + Na3) \times Delay_{\text{ldm}} + Na2 \times Delay_{\text{sdm}}] + T_{\text{other}} \tag{9}$$

相对的, 循环优化前的访问总开销如公式 (10) 所示:

$$T_{\text{noopt}} = ITERS \times [(Na1 + Na2) \times Delay_{\text{mem}} + Na3 \times Delay_{\text{sdm}} + T_{\text{other}}] \tag{10}$$

因此, 若用节省的时间开销表示优化收益, 则优化收益计算如公式 (11) 所示:

$$TR_{\text{loop}} = ITERS \times \left[Na1 \times (Delay_{\text{mem}} - Delay_{\text{ldm}}) + Na2 \times (Delay_{\text{mem}} - Delya_{\text{sdm}}) + Na3 \times (Delay_{\text{sdm}} - Delay_{\text{ldm}}) - \frac{ITERS}{block\_len} \times T_{\text{trans}} \right] \tag{11}$$

公式 (6)–公式 (11) 中相关变量含义见表 5.

从循环优化前后的时间开销公式 (9)、公式 (10) 和循环收益公式 (11) 可以看出, 优化前后的性能变化由两部分因素构成, 一部分是数据存储层次提升后带来的访存延迟的下降, 这部分性能变化是固定的, 是对性能提升正向的贡献. 另外一部分是新增的优化传输开销, 需要尽量减小这部分开销. 结合公式 (8) 和公式 (9) 可以看出, 优化传输开销 T_{trans} 与循环分块数成正比, 与分块长度 *block_len* 是成反比的, 进一步结合公式 (4) 和公式 (5) 中缓冲区空间需求制约关系可以得出, 在满足缓冲区需求的前提下 *block_len* 越大越好. 具体的循环分块求解算法在第 4.2 节中进行阐述.

表 5 变量一览表 2

参数	含义	SW26010Pro上的参数值
$Na1$	循环中mem2ldm的数组的访问个数	—
$Na2$	循环中mem2sdm的数组的访问个数	—
$Na3$	循环中sdm2ldm的数组的访问个数	—
Block_num	循环分块数, *Block_num= ITERS/block_len*	—
$T0_{\text{dma}}$	主存到LDM或SDM的DMA启动开销	DMA启动开销为200拍
$T0_{\text{rma}}$	LDM或SDM到LDM的RMA启动开销	RMA启动开销为50拍
B_RMA	DMA带宽	DMA带宽为平均单核心640 MB/s
B_DMA	RMA带宽	RMA带宽为点对点4 GB/s
μ	带宽利用率	当DMA/RMA的数据长度超过128 B, 取μ=1, 否则取μ=数据长度/128
$Delay_{\text{mem}}$	主存空间访问延迟	LDM访问延迟6拍
$Delay_{\text{sdm}}$	SDM空间访问延迟	SDM无冲突访问延迟约50拍
$Delay_{\text{ldm}}$	LDM空间访问延迟	LDM无冲突访问延迟6拍
T_{other}	循环中其他非优化数组访问的开销, 在实际评估时这部分开销不需要参与计算	—
ITERS	循环迭代总次数	—

4.2 基于二分搜索的循环分块和 region 求解方法

通过第 4.1 节中构建的收益开销评估模型分析得出, 需要求解一个循环分块方案, 使得循环分块大小 *block_len* 尽量大, 以降低 DMA/RMA 启动开销, 提升批量数据传输效率, 同时要满足所有 region 的缓冲区空间需求不超过用户给出的缓冲区大小. 对于数组访问较多, 循环较大的情况下, 可能有很多种分块方式和数组分区方案, 如何在较短时间内寻找最佳的循环分块和分区方案, 是编译器实现过程中需要解决的一个关键问题.

本文设计了基于二分搜索的求解算法 (见算法 1), 可以在 log$N$ 的时间复杂度下求解得到最优的循环分块长度 *block_len* 及 region 划分方案. 算法主要思想为在当前搜索的循环分块长度下, 按照 region 更新算法 (见算法 update_region) 更新 region 的划分, 并在空间制约关系下进行二分搜索, 直到求得满足空间需求的最大 *block_len* 时停止搜索. 算法复杂度为 log$N$, 具体算法流程描述见算法 1.

算法 1. 基于二分搜索的循环分块和 region 求解算法.

输入: 当前 *block_len* 和 region 信息;

输出: 缓冲区空间需求能被满足的最大 *block_len* 和相应的 region 信息.

1. cur_region=region.
2. *max_block_len*=0; *step_len*= *block_len*/2.
3. 调用算法 2 更新 cur_region, 转 Step 4.
4. 根据公式 (1) 和公式 (3) 计算 cur_region 的 Space.
5. 根据公式 (2) 判断 Space 是否小于 *buf_size*, 否则转 Step 9.
6. 判断 *block_len* 是否大于 *max_block_len*, 否则结束.
7. 如果 has_newregion_flag=1, 则 cur_region=region.
8. *max_block_len*=*block_len*, *block_len*=*block_len*+*step_len*, *step_len*=*step_len*/2, 转 Step 3.
9. 如果 has_newregion_flag=1, 则 region=cur_region.
10. *block_len*=*block_len*–*step_len*, *step_len*=*step_len*/2, 转 Step 3.

算法 2. update_region.

输入: *block_len* 和 region 信息;
输出: region 信息和 region 是否被更新的标志 has_newregion_flag.

1. 初始化 has_newregion_flag=0.
2. 遍历所有 region 直到所有 region 都被处理过.
2.1. 依次遍历当前 region 的所有访问偏移 S_i, 若存在 i, 使得 S_i+1–S_0>*block_len*, 新增一个 region, 将 S_i+1 到 S_n 对应的数组访问划分进新 region, 并记 has_newregion_flag=1; 否则, continue.
3. 返回 has_newregion_flag.

以图 11 中的代码作为优化方案求解示例, 该例中共有 a 数组的 4 个访问 a[i]、a[i+1]、a[i+2] 和 a[i+128], 编译器分析将得到数组 a 的 4 个访问的 offset 分别为 0、1、2、128, 数组 b 的 offset 为 0.

```
#pragma ccc sbuf mem2ldm(a,b) ldm_addr(buf) ldm_size(1024)
  for(i = 0; i < 10240; i++)
  {
    a[i] = (a[i+1] + a[i+2]) / 2;
    x = a[i] / a[i+128]+b[i];
    …//其他代码
  }
```

图 11　编译指示代码段示例

实际求解过程如下.

Step 1. 初始化分块大小 *block_len*=1024/8=128.

Step 2. 初始化 region1={a[i]、a[i+1]、a[i+2]、a[i+128]}; region2={b[i]}.

Step 3. 计算 Space: (128–0+128)×4+128×4=1536.

Step 4. Space=1536>1024.

Step 5. 调用算法 2 返回 0, region 不变.

Step 6. *block_len*=128/2=64.

Step 7. 计算 Space: (128–0+64)×4+64×4=1280.

Step 8. Space=1280>1024.

Step 9. 调用算法 3 返回 1, region 变为: region1={a[i]、a[i+1]、a[i+2]}, region2={b[i] }, region3={a[i+128]}.

Step 10. 计算 Space: (2–0+64)×4+64×4+64×4=776.

Step 11. Space=776<1024, 终止.

基于上述计算过程可得到编译优化参数:

block_len=64, 3 个 region: region1={a[i]、a[i+1]、a[i+2]}, region2={b[i]}, region3={a[i+128]}.

4.3 启发式循环优化方案迭代求解框架

从 region 收益公式 (6) 和循环收益公式 (11) 可以看出, 当 *TR* 大于 0 时, 优化有正向的收益, 而当 *TR* 小于 0 时, 优化收益是负的, 即优化产生的缓冲数据传输开销超过了数据访问节省的时间. 这种情况出现的可能原因两个, 一是用户给出的缓冲区空间过小, 还有一种可能是循环总迭代数太小, 使得最终求出的最大 *block_len* 过于小, 从而使得某些 region 的每次 DMA 或者 RMA 传输的数据量不足以填满带宽. 极端的如 *block_len*=1 时, 对于 region 的元素宽度为 4 的数组访问, 每次传输 4 字节, 由于固定的 DMA 启动开销相对较大, DMA 总开销远超过直接主存访问的开销.

因此, 在利用第 4.2 节中介绍的二分搜索算法求得 *block_len* 和 region 方案后, 需要将具体优化方案下的 region 数据进一步代入第 4.1 节中的收益公式进行评估, 如果收益值不为正, 则需要舍弃一个或更多个 region 的优化, 以确保剩下的数组 region 优化可以获得正向收益. 当发生负向收益时, 选择放弃哪个 region 的优化, 以及最终需要选择放弃多少个 region, 是一个关键问题. 本文利用第 4.1 节构建的时间收益与空间开销比模型 P_{region} (见公式 (7)) 衡量一个 region 的优先级, 当发生负向收益时, 优先级最低的 region (即收益开销比值 P_{region} 最小的 region) 首先被放弃.

本文设计了一个基于收益模型的启发式迭代求解框架. 为尽量满足编译指示意图, 框架在保证收益为正的情况下尽可能多地实现对用户编译指示中所指出的数组的优化. 算法框架如图 12 所示, 主要思想是先将循环中被编译指示的所有数组的访问信息作为启发式初值信息, 调用第 4.2 节中的搜索算法求解当前条件下的最优循环分块方案. 然后根据优化收益函数评估求得的当前最优循环分块方案下的每个 region 的收益和总体收益, 当发生收益为负的情况, 则舍弃最低优先级 region 的信息, 然后根据剩下的 region 信息重新输入算法 1 进行优化方案求解, 以此迭代直至剩下的所有 region 收益为正. 从算法流程可以看出, 整个框架调用二分搜索算法的次数最差情况下是被指示的待优化数组的访问总个数. 而二分搜索算法复杂度为 log*N* (*N* 为循环长度), 因此, 整个搜索算法的时间复杂度是多项式时间的, 经实际测试, 编译阶段的搜索时间开销可忽略.

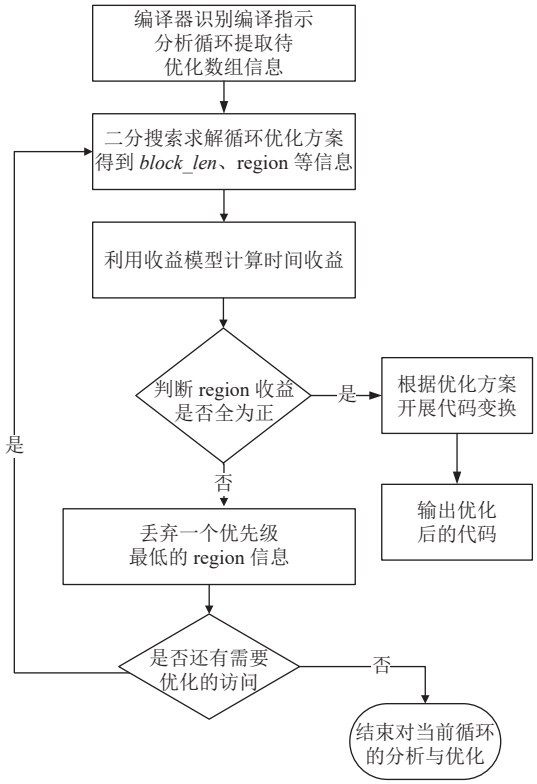


图 12 基于收益模型的启发式迭代求解算法框架

框架第一次调用算法 1 时的 region 初始划方法为: 不同的数组首先划分为不同的 region, 对于同 region 内的多个数组访问, 若优化维以上的更高维的表达式不同, 则进一步划分为多个不同 region, 若相同, 则归入同一 region.

框架初始化参数取值如下:

$$\begin{cases} block\_len = \dfrac{\min(\text{循环总迭代数} \times \sum region\_size, buf\_size)}{\sum region\_size} \\ max\_block\_len = 0 \\ step\_len = \dfrac{block\_len}{2} \end{cases}.$$

求解算法中当循环总迭代数不是常量时置为无穷大, 编译器代码变换时会增加相应的判断代码避免循环分块超过循环迭代总量.

4.4 编译器代码变换

在根据第 4.3 节中的算法计算出数组访问聚类分区方案以及循环分块大小后, 需要编译器对代码进行变换, 主要是完成循环分块, 生成 DMA、RMA 操作及配套指令, 修改循环体中的访存操作等, 以及当有多次相邻 DMA、RMA 操作时, 对回答字测试过程进行优化, 将多次相邻的回答字测试进行合并.

GCC 中循环结构的通用表示及代码变换过程如图 13 所示. 图 13(a) 中所示为 GCC 中循环结构的通用表示. 具体变换步骤如下.

```
for (V = N1; V cond N2; V += STEP)
    BODY;
```

(a) GCC for 循环中间表示

```
trip = 0;
L0:
  s0 = trip * CHUNK;
  e0 = min(s0 + CHUNK, n);
  if (s0 < n) goto L1; else goto L4;
L1:
  V = s0 * STEP + N1;
  e = e0 * STEP + N1;
L2:
  BODY;
  V += STEP;
  if (V cond e) goto L2; else goto L3;
L3:
  trip += 1;
  goto L0;
L4:
```

(b) 循环分块后的中间表示示意图

```
buffer = ldm_malloc();
  trip = 0;
L0:
  s0 = trip * CHUNK;
  e0 = min(s0 + CHUNK, n);
  if (s0 < n) goto L1; else goto L4;
  dma/rma_get(data[trip], buffer, reply);
  wait(reply);
L1:
  V = s0 * STEP + N1;
  e = e0 * STEP + N1;
L2:
  BODY;
  Use(buffer);
  V += STEP;
  if (V cond e) goto L2; else goto L3;
L3:
  dma/rma_put(data[trip], buffer, reply);
  wait(reply);
  trip += 1;
  goto L0;
L4:
```

(c) 插入DMA/RMA 操作及修改循环体后的中间表示示意图

图 13 代码变换过程示意图

Step 1. 进行循环分块变换, 生成的 GCC 中间表示如图 13(b) 所示, CHUNK 为 4.3 中求解出的分块大小 *block_len*.

Step 2. 遍历 region, 对每一个 region 生成一次 DMA 或 RMA 操作. 具体生成 get 还是 put 操作, 需要根据变量数据流分析结果和访问特征确定, 对于只读变量生成 get 操作, 对于向下暴露变量, 生成 put 操作. 先读后写变量, 既要在循环前生成 get 操作, 又要在循环后生成 put 操作.

Step 3. 修改循环体, 将原数组访问修改成对相应缓冲区数据的访问, 生成的 GCC 中间表示如图 13(c) 所示.

Step 4. 对边界条件进行处理. 循环迭代次数不能整除循环分块大小的情况: 对于 dma/rma _get 操作, 最后一个循环块中多余 get 的数据不会影响程序结果; 对于 dma/rma_put 操作, 最后一个循环块中多余 put 的数据可能会冲

掉主存中的有用数据, 需要根据实际循环大小修正数据量.

Step 5. 进行回答字测试优化, 多个相邻的 DMA 操作, 其回答字测试可以合并, 减少开销.

5 实验验证

本节对设计实现的访存优化编译技术进行实验验证和分析. 测试平台为基于 SW26010Pro 构建的计算系统, 安装部署了本文设计实现的融合优化编译指示的编程语言环境. 在实际应用中, 也可将 3 种优化类型的编译指示同时作用在同一个循环的多个数组访问中, 产生叠加的优化效果. 在本实验中, 为分别展示 3 种优化类型的编译指示的效果, 本文选取了 3 个使用不同优化模式的测试用例分别进行了实验测试与分析.

5.1 编译指示使用示例及代码量对比分析

本节通过代码量对比示例直观展示使用本文设计的编译指示进行优化编程的效果. 其中图 14 为基于编译指示的单、双缓冲程序代码示意, 图 15、图 16 为手工实现的相同优化效果的程序代码示意. 通过对比代码量可以看出使用本文设计的编译指示进行编程优化, 大幅减小了编程工作量, 实现的程序代码更简洁.

```
/*单缓冲优化编译指示编程示例*/
#define WGSIZE 4096
float LDM_Tbuf = (float *)ldm_malloc(5*1024)
#pragma ccc sbuf mem2ldm(poses) \
                    ldm_addr(LDM_Tbuf) ldm_size(5120)
for(int ix=0; ix<WGSIZE; ix++)
{
        //Compute transformation matrix
        const float sx = sinf(poses[0][CCC_TID*WGSIZE][ix]);
        const float cx = cosf(poses[0][CCC_TID*WGSIZE][ix]);
        const float sy = sinf(poses[1][CCC_TID*WGSIZE][ix]);
        const float cy = cosf(poses[1][CCC_TID*WGSIZE][ix]);
        const float sz = sinf(poses[2][CCC_TID*WGSIZE][ix]);
        const float cz = cosf(poses[2][CCC_TID*WGSIZE][ix]);
        …//其他与优化无关代码
        Transform[0][3][ix] = poses[3][CCC_TID*WGSIZE][ix];
        …//其他与优化无关代码
        Transform[1][3][ix] = poses[4][CCC_TID*WGSIZE][ix];
        …//其他与优化无关代码
        Transform[2][3][ix] = poses[5][CCC_TID*WGSIZE][ix];
        …//其他与优化无关代码
}
```

```
/*双缓冲优化编译指示编程示例*/
#define WGSIZE 4096
float LDM_Tbuf = (float *)ldm_malloc(    10*1024)
#pragma ccc    dbuf mem2ldm(poses) \
                    ldm_addr(LDM_Tbuf) ldm_size(   10240 )
for(int ix=0; ix<WGSIZE; ix++)
{
        //Compute transformation matrix
        const float sx = sinf(poses[0][CCC_TID*WGSIZE][ix]);
        const float cx = cosf(poses[0][CCC_TID*WGSIZE][ix]);
        const float sy = sinf(poses[1][CCC_TID*WGSIZE][ix]);
        const float cy = cosf(poses[1][CCC_TID*WGSIZE][ix]);
        const float sz = sinf(poses[2][CCC_TID*WGSIZE][ix]);
        const float cz = cosf(poses[2][CCC_TID*WGSIZE][ix]);
        …//其他与优化无关代码
        Transform[0][3][ix] = poses[3][CCC_TID*WGSIZE][ix];
        …//其他与优化无关代码
        Transform[1][3][ix] = poses[4][CCC_TID*WGSIZE][ix];
        …//其他与优化无关代码
        Transform[2][3][ix] = poses[5][CCC_TID*WGSIZE][ix];
        …//其他与优化无关代码
}
```

图 14 基于编译指示的单、双缓冲优化程序代码示意

```
//手工单缓冲优化编程示例
#define WGSIZE 4096
#define BLOCK 256
float LDM_Tbuf=(float*)ldm_malloc(5*1024)
float *ldm_poses[5];
for (int k=0;k<5;k++)
{
      ldm_poses[k]=LDM_Tbuf[BLOCK*k];
}
for(int bix=0;bix<WGSIZE;bix+=BLOCK)
{
      for(int k=0;k<5;k++)
      {
dma_get(ldm_poses[k],&poses[k][CCC_TID*WGSIZE][bix
], sizeof(float)*BLOCK,MEM_TO_LDM);
      }
```

```
      for(int ix=0; ix<BLOCK;ix++)
      {
        //Compute transformation matrix
          const float sx = sinf(ldm_poses[0][ix]);
          const float cx = cosf(ldm_poses[0][ix]);
          const float sy = sinf(ldm_poses[1][ix]);
          const float cy = cosf(ldm_poses[1][ix]);
          const float sz = sinf(ldm_poses[2][ix]);
          const float cz = cosf(ldm_poses[2][ix]);
          …//其他与优化无关代码
          Transform[0][3][ix] =ldm_poses[3][ix];
          …//其他与优化无关代码
          Transform[1][3][ix] =ldm_poses[4][ix];
          …//其他与优化无关代码
          Transform[2][3][ix] =ldm_poses[5][ix];
          …//其他与优化无关代码
    }//ix
}//bix
```

图 15 手工实现的单缓冲优化程序代码示意

```
/*手工双缓冲示优化编程示例*/

#define WGSIZE 4096
#define BLOCK 256
float LDM_Tbuf=(float*)ldm_malloc(10*1024)
float *ldm_poses[10];
for(int k=0;k<5;k++)
{
    ldm_poses[k]=LDM_Tbuf[BLOCK*k];
    ldm_poses[k+5]=LDM_Tbuf[BLOCK*(k+5)];
}
for(int k=0;k<5;k++)
{
dma_get(ldm_poses[k],&poses[k][CCC_TID*WGSIZE][0],s
izeof(float)*BLOCK,MEM_TO_LDM);
}
intbflag=0;
for(intbix=0;bix<WGSIZE;bix+=BLOCK)
{
  bflag++;
  int off0=(bflag^1)*5;
  int off1=((bflag-1)^1)*5;
```

```
if(bix+BLOCK<WGSIZE)
 for(int k=0;k<5;k++)
   dma_get(ldm_poses[k+off1],&poses[k][CCC_TID*WGSIZE]
 [bix+BLOCK],sizeof(float)*BLOCK,MEM_TO_LDM);
for(int ix=0; ix<BLOCK; ix++)
{
    //Compute transformation matrix
      const float sx = sinf(ldm_poses[0+off0][ix]);
      const float cx = cosf(ldm_poses[0+off0][ix]);
      const float sy = sinf(ldm_poses[1+off0][ix]);
      const float cy = cosf(ldm_poses[1+off0][ix]);
      const float sz = sinf(ldm_poses[2+off0][ix]);
      const float cz = cosf(ldm_poses[2+off0][ix]);
      …//其他与优化无关代码
      Transform[0][3][ix] = ldm_poses[3+off0][ix];
      …//其他与优化无关代码
      Transform[1][3][ix] = ldm_poses[4+off0][ix];
      …//其他与优化无关代码
      Transform[2][3][ix] = ldm_poses[5+off0][ix];
      …//其他与优化无关代码
  }//ix
}//bix
```

图 16　手工实现的双缓冲优化程序代码示意

5.2　mem2ldm 优化编译指示测试分析

mem2ldm 优化编译指示的测试程序为第 5.1 节中使用的迷你分子动力学程序 MiniBude.

本文对该程序的直接访问主存、基于编译指示的编译器自动单缓冲优化、手工实现的单缓冲优化、基于编译指示的编译器自动双缓冲优化、手工实现的双缓冲优化等 5 个版本的程序在数组容量 4 KB、8 KB、12 KB、16 KB 这 4 个规模下进行了测试, 测试结果如图 17 所示, 图中展示各种规模下的程序运行时间对比.

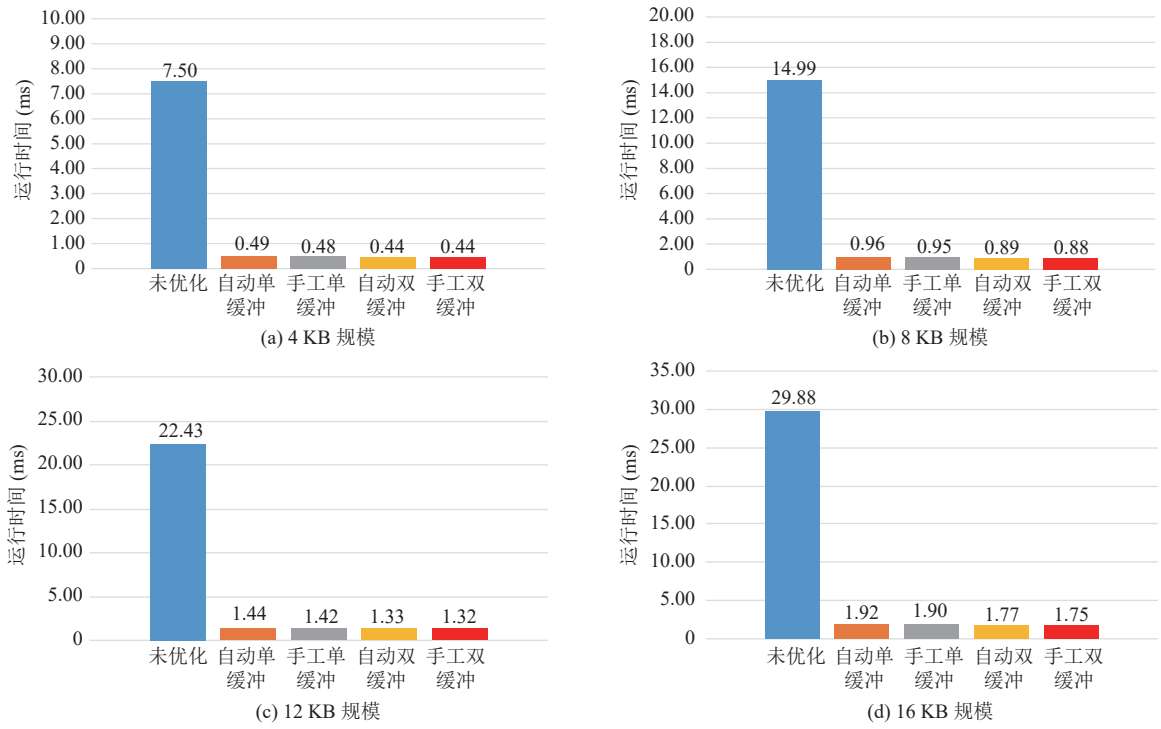


图 17　mem2ldm 优化效果对比图

相对于访主存版本, 4 个规模下, 自动单缓冲优化、手工单缓冲优化、自动双缓冲优化、手工双缓冲优化的平均加速比分别为 15.28、15.71、16.21 和 17.03. 对比自动单缓冲优化和手工单缓冲优化版本, 在 4 个规模下, 自动单缓冲优化平均性能达到手工单缓冲优化版本的 97.3%; 对比自动双缓冲优化和手工双缓冲优化版本, 在 4 个规模下, 自动双缓冲优化平均性能达到手工双缓冲优化版本的 95.2%. 双缓冲版本较单缓冲版本没有明显的性能提升的原因是该程序计算部分比例明显高于访存部分, 实际使用中双缓冲在程序计算和访存占比相当的情况下才会获得较好的效果.

测试结果说明, 本文实现的优化方法能够获得很好的优化效果. 经过分析, 对于该测试程序, 自动优化的性能略低于手工优化的性能的原因主要是两方面, 一是自动变换要统一考虑循环上下是变量, 以及循环不能被分块整除等情况, 生成的分块代码比手工代码更加复杂; 二是计算分块大小时, 为了统一考虑数组可能的多个访问, 数据按向量宽度对界等情况, 预留了小部分缓冲空间, 导致自动计算得到的分块大小略小于手工计算结果.

5.3 sdm2ldm 优化编译指示测试分析

sdm2ldm 优化编译指示的测试程序使用了根据图形处理程序核心计算编写的测试程序. 核心计算过程需要使用 sin 和 cos 两个三角函数的值, 由于其取值范围是确定的, 可将其提前计算, 计算过程用查表代替. 但表空间的规模随问题规模增大, 测试过程对能够进行 sdm2ldm 优化的规模区间进行了测试分析. 测试结果如图 18 所示, 图中展示了该测试程序在 4 个不同数据规模下的程序运行时间, 单位为 ms.

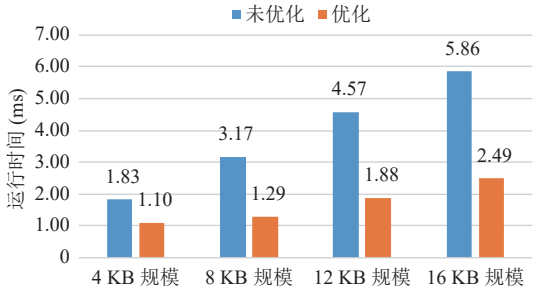


图 18 sdm2ldm 优化效果对比图

相对于未优化版本, 优化版本在 4 个规模下平均加速比达到 2.23. 由于计算过程中的核心数据占用了较多的 ldm 空间, 该程序只使用了单缓冲优化. 如果规模继续增大, 则无法将 sin 和 cos 的表放到 sdm 空间, 此时可将表布局在主存空间, 使用 mem2ldm 进行优化. sdm2ldm 与 mem2ldm 的区别是 sdm2ldm 可以利用片上的通信网络带宽, 提出 sdm2ldm 优化的目的是将对访存带宽的需求转化为片上通信带宽, 更好地利用处理器资源.

5.4 mem2sdm 优化编译指示测试分析

mem2sdm 优化编译指示的测试程序是根据口令恢复类应用常用的 bitmap 模式编写的测试程序. 这类程序中, 对 bitmap 的访问是离散访问, bitmap 越大越能获得更高的结果比对成功率, 因此适合将 bitmap 放入 sdm 空间.

测试结果如图 19 所示, 在 6 个测试规模下, 相对于访主存版本, 自动缓冲优化和手工缓冲优化平均得到 1.34 和 1.35 的加速比. 自动缓冲优化的性能和手工缓冲优化很接近, 稍有差异的原因是自动变换的代码增加了针对缓冲空间大小的多版本代码, 比手工代码复杂. 在该程序变换过程中, 因对 bitmap 的访问时随机访问, 在进行访问分析时, 如果能将 bitmap 全部装入缓冲区, 则进行 mem2sdm 优化, 否则放弃优化, 该测试程序也无需进行双缓冲优化.

6 总结与展望

申威众核处理器提供的 SPM 结构、DMA 传输机制和片上高速 RMA 通信机制等存储结构特点给应用程序访存性能优化带来了很多机会, 同时也对程序开发人员和编译优化提出了很大挑战. 本文提出了一种基于编译指示的多级存储层次上访存与通信融合的编译优化方法, 通过编译指示的功能设计引导用户程序给编译器传递程序

高层信息, 再由编译器通过优化建模、循环优化方案求解和代码变换完成精准高效的核心循环数据访问优化, 提升应用程序访存性能的同时减轻了用户编程负担.

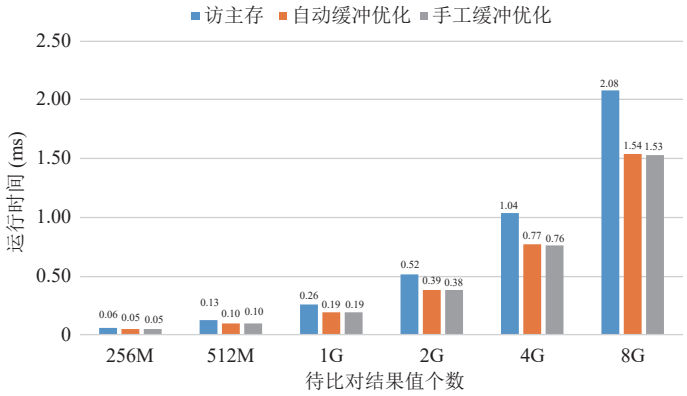


图 19 mem2sdm 优化效果对比图

本文在目前的设计中, 要求编程人员在编译指示中给出优化缓冲区地址和长度等信息, 后续将研究在这部分信息允许缺省的情况下, 编译器结合运行时接口自动查询当前可用的缓冲区大小并进行自动申请与释放的管理, 一方面有机会挖掘更多的 LDM 空间作为优化缓冲, 另一方面可进一步简化编译指示的使用要求, 提升好编程性.

References:

[1] TOP500 List. 2021. https://www.top500.org/lists/top500/2021/11/

[2] Banakar R, Steinke S, Lee BS, Balakrishnan M, Marwedel P. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In: Proc. of the 10th Int'l Symp. on Hardware/Software Codesign. Estes Park: ACM, 2002. 73–78. [doi: 10.1145/774789.774805]

[3] Sato M, Ishikawa Y, Tomita H, Kodama Y, Odajima T, Tsuji M, Yashiro H, Aoki M, Shida N, Miyoshi I, Hirai K, Furuya A, Asato A, Morita K, Shimizu T. Co-design for A64FX manycore processor and "Fugaku". In: Proc. of the 2020 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Atlanta: IEEE, 2020. 1–15. [doi: 10.1109/SC41405.2020.00051]

[4] Wen H, Zhang W. Reducing cache leakage energy for hybrid SPM-cache architectures. In: Proc. of the 2014 Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES). New Delhi: ACM, 2014. 21. [doi: 10.1145/2656106.2656124]

[5] Fang YF, Liu Q, Dong EM, Li YB, Guo F, Wang D, He WQ, Qi FB. Research on manycore on-chip storage hierarchy for exascale supercomputer systems. Computer Engineering, 2023, 49(12): 10–24 (in Chinese with English abstract). [doi: 10.19678/j.issn.1000-3428.0066548]

[6] Gao JG, Liu X, Li F, Liu Y, Peng DJ, Chen X, Chen DX. Research on parallel computing model for sunway many-core supercomputing system. Chinese Journal of Computers, 2023, 46(7): 1339–1349 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2023.01339]

[7] Venkataramani V, Chan MC, Mitra T. Scratchpad-memory management for multi-threaded applications on many-core architectures. ACM Trans. on Embedded Computing Systems, 2019, 18(1): 10. [doi: 10.1145/3301308]

[8] Tao XH, Pang JM, Xu JL, Zhu Y. Compiler-directed scratchpad memory data transfer optimization for multithreaded applications on a heterogeneous many-core architecture. The Journal of Supercomputing, 2021, 77(12): 14502–14524. [doi: 10.1007/s11227-021-03853-x]

[9] Chakraborty P, Panda PR, Sen S. Partitioning and data mapping in reconfigurable cache and scratchpad memory-based architectures. ACM Trans. on Design Automation of Electronic Systems, 2016, 22(1): 12. [doi: 10.1145/2934680]

[10] Li JJ, Liu ZZ, Wang J. Optimizing OpenMP by array privatization on the multi-core platform of IBM cell. Journal of Computer Research and Development, 2010, 47(8): 1434–1441 (in Chinese with English abstract).

[11] Yu C, Bai YB, Sun QX, Yang HL. Improving thread-level parallelism in GPUs through expanding register file to scratchpad memory. ACM Trans. on Architecture and Code Optimization, 2018, 15(4): 48. [doi: 10.1145/3280849]

[12] 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]

[13] Liu Y, Liu L, He WQ. A static data placement optimisation model oriented towards multi-core hierarchical accessible resources. Computer Applications and Software, 2011, 28(7): 53–56 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-386X.2011.07.016]

[14] Wu MC, Liu Y, Cui HM, Wei QF, Li QF, Li LM, Lv F, Xue JL, Feng XB. Bandwidth-aware loop tiling for DMA-supported scratchpad memory. In: Proc. of the 2020 ACM Int'l Conf. on Parallel Architectures and Compilation Techniques. New York: ACM, 2020. 97–109. [doi: 10.1145/3410463.3414637]

[15] Wu MC, Liu Y, Li LM, Feng XB. An inter-CG collaborative OpenCL compilation method on the Sunway TaihuLight supercomputer. Chinese High Technology Letters, 2022, 32(9): 927–936 (in Chinese with English abstract). [doi: 10.3772/j.issn.1002-0470.2022.09.006]

[16] Zhou B, Huang YZ, Xu JC, Guo SZ, Qi HY. Memory latency optimizations for the elementary functions on the Sunway architecture. The Journal of Supercomputing, 2019, 75(7): 3917–3944. [doi: 10.1007/s11227-018-02741-1]

[17] Jiang YQ. Research on parallel optimization of transformer model based on the new generation of Sunway many-core processors [MS. Thesis]. Shanghai: East China Normal University, 2022 (in Chinese with English abstract). [doi: 10.27149/d.cnki.ghdsu.2022.002946]

附中文参考文献:

[5] 方燕飞, 刘齐, 董恩铭, 李雁冰, 过锋, 王谛, 何王全, 漆锋滨. 面向 E 级超算系统的众核片上存储层次研究. 计算机工程, 2023, 49(12): 10–24. [doi: 10.19678/j.issn.1000-3428.0066548]

[6] 高剑刚, 刘鑫, 李芳, 刘勇, 彭达佳, 陈鑫, 陈德训. 面向神威众核超算系统的并行计算模型研究. 计算机学报, 2023, 46(7): 1339–1349. [doi: 10.11897/SP.J.1016.2023.01339]

[10] 李建江, 刘珍珍, 王珏. 基于 IBM Cell 多核平台的 OpenMP 数组私有化技术研究. 计算机研究与发展, 2010, 47(8): 1434–1441.

[12] 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]

[13] 刘勇, 刘丽, 何王全. 面向众核多级访存资源的静态数据布局优化模型. 计算机应用与软件, 2011, 28(7): 53–56. [doi: 10.3969/j.issn.1000-386X.2011.07.016]

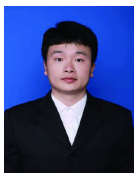
[15] 伍明川, 刘颖, 李立民, 冯晓兵. 面向神威·太湖之光的多核组协同的 OpenCL 编译方法. 高技术通讯, 2022, 32(9): 927–936. [doi: 10.3772/j.issn.1002-0470.2022.09.006]

[17] 姜云桥. 基于新一代申威众核处理器的 Transformer 模型并行优化的研究 [硕士学位论文]. 上海: 华东师范大学, 2022. [doi: 10.27149/d.cnki.ghdsu.2022.002946]

方燕飞(1980－), 女, 高级工程师, 主要研究领域为并行语言, 编译优化.

王云飞(1995－), 男, 硕士, 主要研究领域为软件工程.

李雁冰(1989－), 男, 博士, 助理研究员, 主要研究领域为并行编译.



刘齐(1992－), 男, 助理研究员, 主要研究领域为并行语言, 编译优化.

董恩铭(1988－), 男, 博士, 助理研究员, 主要研究领域为高性能计算软件.