国防科技大学学报
Journal of National University of Defense Technology
ISSN 1001-2486,CN 43-1067/T

《国防科技大学学报》网络首发论文

题目：面向多核数字信号处理器的高效 OpenCL 异构并行编程系统
作者：高琬蓉，刘锡泰，梁瀚正，张鹏，唐滔，黄春，王挺，方建滨
收稿日期：2026-01-22
网络首发日期：2026-04-10
引用格式：高琬蓉，刘锡泰，梁瀚正，张鹏，唐滔，黄春，王挺，方建滨．面向多核数字信号处理器的高效 OpenCL 异构并行编程系统[J/OL]．国防科技大学学报．https://link.cnki.net/urlid/43.1067.T.20260409.1750.002

doi: 10.11887/j.issn.1001-2486.26010045

面向多核数字信号处理器的高效 OpenCL 异构并行编程系统

高琬蓉，刘锡泰，梁瀚正，张鹏，唐滔，黄春，王挺，方建滨*

（国防科技大学 计算机学院, 湖南 长沙 410073）

摘　要：随着多核数字信号处理器（digital signal processing，DSP）在高性能计算和人工智能等领域的广泛应用，如何在其异构架构上实现高效且可移植的并行编程成为重要挑战。面向国产异构多核数字信号处理器 FT-M7032，设计并实现了高效的 OpenCL（open computing language）异构并行编程系统 MOCL4。该系统通过运行时与编译器协同优化，将 OpenCL 的单程序多数据（single program, multiple data，SPMD）执行模型高效映射到 DSP 的单指令多数据（single instruction, multiple data，SIMD）向量单元，并支持基于直接内存访问（direct memory access，DMA）的片上数据搬运与存储层次管理。实验结果表明，MOCL4 在保证 OpenCL 语义正确性的同时，显著提升了内核函数的执行性能，在 PolyBench 测试集上平均加速比达到 10.12 倍；在典型密集计算任务中性能接近手工优化水平。MOCL4 为多核 DSP 提供了一种兼具高性能与可编程性的并行编程解决方案。

关键词：多核数字信号处理器；OpenCL；编译器；向量化；DMA

中图分类号：TP314　　**文献标志码：**A

Efficient OpenCL programming system for multi-core DSPs

GAO Wanrong, LIU Xitai, LIANG Hanzheng, ZHANG Peng, TANG Tao, HUANG Chun, WANG Ting, FANG Jianbin

(College of Computer Science and Technology, National University of Defense Technology, Changsha 410073 , China)

Abstract: With the widespread application of multi-core DSP (digital signal processor) in high-performance computing and artificial intelligence, achieving efficient and portable parallel programming on these heterogeneous architectures has become a significant challenge. An efficient OpenCL-based (open computing language-based) heterogeneous parallel programming system, MOCL4, was designed and implemented for the domestically developed heterogeneous multi-core DSP platform, FT-M7032. MOCL4 collaborates runtime and compiler optimizations to efficiently map OpenCL's SPMD (single program, multiple data) execution model onto the DSP's SIMD (single instruction, multiple data) vector units, while supporting efficient DMA-based (direct memory access-based) data transfers across memory hierarchies. Experimental results show that MOCL4, while ensuring correctness of OpenCL semantics, significantly improves kernel execution performance. The average speedup on the PolyBench benchmark suite is 10.12x, and its performance on typical compute-intensive tasks is close to that of manually optimized code. MOCL4 provides a parallel programming solution for multi-core DSPs that balances high performance with programmability.

Keywords: multi-core digital signal processor; OpenCL; compiler; vectorization; DMA

收稿日期：2026-01-22

基金项目：国家重点研发计划资助项目（2023YFB3001503）

第一作者：高琬蓉（1996—），女，湖南长沙人，博士研究生，E-mail：gaowanrong@nudt.edu.cn

***通信作者：**方建滨（1984—），男，山东平度人，副教授，博士，硕士生导师，E-mail：j.fang@nudt.edu.cn

引用格式：高琬蓉, 刘锡泰, 梁瀚正, 等. 面向多核数字信号处理器的高效 OpenCL 异构并行编程系统[J]. 国防科技大学学报

Citation： Gao Wanrong, Liu Xitai, Liang Hanzheng, et al. Efficient OpenCL programming system for multi-core DSPs[J]. Journal of National University of Defense Technology

近年来，多核数字信号处理器（digital signal processor，DSP）凭借优异的能耗比和持续提升的数值计算能力，逐渐成为高性能计算和人工智能等领域的重要计算平台[1-2]。以国产异构多核DSP——FT-M7032——为例，其架构集成了软件管理的片上存储器（scratchpad memory，SPM）、高带宽直接内存访问（direct memory access，DMA）引擎以及专用单指令多数据（single instruction, multiple data，SIMD）向量计算单元，从体系结构层面为高吞吐、低功耗计算提供了有力支撑[2]。然而，该硬件架构在释放性能潜力的同时，也显著提高了程序开发的复杂度。现有高性能实现通常需要开发者基于 hthreads 编程接口或汇编级编程方式，围绕特定计算任务（如矩阵乘和模板计算）的特征手动划分并行任务，并显式管理 SPM 的使用与 DMA 数据搬运[3-7]。虽然这类方法能够较充分地利用目标平台的体系结构特征，但编程门槛高、代码平台相关性强，严重制约了应用程序的可移植性与开发效率。因此，构建一套高效且通用的并行编程系统，对于充分发挥此类多核 DSP 的硬件潜力、提升应用开发效率具有重要意义。

OpenCL（open computing language）作为一种开放、跨平台的异构并行编程标准，为多种异构计算设备提供了通用的编程抽象。开发者可在设备无关的编程框架下描述程序并行性与数据访问模式，从而在兼顾性能、可移植性与开发效率的同时，降低开发复杂度。基于上述优势，OpenCL 被视为支撑多核 DSP 应用开发的潜在理想编程模型。然而，将 OpenCL 高效映射到多核 DSP 架构面临显著挑战。一方面，OpenCL 的单程序多数据（single program, multiple data，SPMD）执行模型难以与多核 DSP 硬件的并行执行单元直接对应；另一方面，DSP 中的 SPM 与 DMA 机制也对 OpenCL 内存语义的正确实现与高效数据搬运提出了更高要求。

已有研究在中央处理器（central process unit，CPU）、嵌入式处理器及各类加速器平台上对 OpenCL 的实现与优化进行了广泛探索，其核心目标是在保持 OpenCL 语义正确性的前提下，将其并行执行与存储抽象有效映射到目标硬件体系结构之上。总体来看，这类工作主要围绕三个方面展开：其一，在执行模型层面，普遍采用工作项合并等方法，将 OpenCL 的 SPMD 执行模型重构为适合目标平台的循环执行形式，以适配较低粒度的物理并行资源[8-18]；其二，在存储层次适配层面，针对 SPM、共享内存或非一致缓存等不同体系结构特征，将 OpenCL 的全局、本地和私有内存重新映射到目标平台的物理存储层次，并结合 DMA[12-16]或运行时一致性机制[10]保证访存正确性与效率；其三，在体系结构感知优化层面，进一步结合目标平台的 SIMD[11]、粗粒度可重构架构（coarse-grained reconfigurable architecture，CGRA）[17-18] 或多簇执行结构[12]，开展显式向量化[11]、循环变换[17-18]、条件分支优化[12, 18]及各种目标相关代码生成优化。从编译实现方式上看，现有工作可分为源到源转换模式[10-15]和基于 LLVM 后端[9, 16]的实现模式两类。前者实现成本较低，但对底层代码生成控制能力有限；后者则更适合在统一中间表示层面实施细粒度分析与体系结构相关优化。总体而言，已有工作表明，高性能 OpenCL 实现依赖于执行模型重构、存储模型适配以及面向目标硬件的编译与运行时协同优化，而本文工作正是在这一思路下，面向国产多核 DSP 平台进一步探索 OpenCL 的高效实现。

针对 FT-M7032 系列多核 DSP，已有研究从不同技术路线探索了 OpenCL 的实现与优化。一类工作以 DSP 核心为主要并行执行单元，通过工作项合并适配 OpenCL 执行模型，在核间并行的基础上实现对 OpenCL 语义的通用支持，即 MOCL3[2]。后续研究进一步针对内核函数中的同步开销与显式数据传输进行优化，以缓解性能瓶颈[16]。另一类工作则将 OpenCL 工作项直接映射到核内的向量计算单元，并围绕向量阵列的微结构特性进行指令级调度与内核特定优化[19]。前者对核内向量单元的利用较为有限，而后者难以形成通用且可扩展的 OpenCL 执行模型。

针对上述挑战，面向 FT-M7032 多核 DSP 平台，设计并实现了一套高效的 OpenCL 异构并行编程系统 MOCL4。与 MOCL3 类似，MOCL4 基于 PoCL[9]实现框架并针对 DSP 体系结构进行了扩展。在继承 MOCL3 并行映射策略、保持 OpenCL 语义正确性的前提下，MOCL4 进一步通过自动向量化充分利用 DSP 的向量计算单元。同时，相较于 MOCL3 依赖 LLVM 中间表示（intermediate representation，IR）到 C 代码的转换进行编译，MOCL4 直接在 LLVM IR 层实施体系结构相关优化，从而能够更加灵活地进行向量化与访存优化。

MOCL4 的实现从运行时支持与内核编译器两方面进行协同优化。在运行时层面，MOCL4

以工作组为基本调度单位，提供符合 OpenCL 规范的执行控制与资源管理机制，并针对多核 DSP 的执行特性优化内核映射与启动开销。在编译器层面，基于 LLVM 构建面向 DSP 的内核编译流程，通过内核函数向量化策略以及 DMA 传输优化，实现 OpenCL 内核在多核 DSP 上的高效执行。实验结果表明，MOCL4 在保证 OpenCL 语义正确性的同时能够显著提升内核函数执行性能。

1 OpenCL 与 FT-M7032 架构

1.1 国际异构并行编程标准 OpenCL

OpenCL 是一种面向异构计算系统的开放式、跨平台并行编程框架，为 CPU、图形处理器（graphics processing unit，GPU）、DSP 及现场可编程门阵列（field-programmable gate array，FPGA）等多种计算设备提供统一的编程模型与执行环境。其采用基于 C 语言的扩展语法描述并行计算内核，通过统一的平台模型、执行模型与内存模型，在保证可移植性的同时降低异构系统编程复杂度。

在平台与执行模型方面，OpenCL 将计算系统抽象为由主机与一个或多个计算设备协同工作的体系结构。主机负责程序控制与资源管理，计算设备用于执行并行计算任务；设备内部由多个计算单元及处理单元构成层次化并行结构。内核函数采用最多三维的索引空间（NDRange）组织执行，每个工作项对应一个内核实例，多个工作项进一步组成工作组，从而支持分层并行执行与组内同步。在内存模型方面，OpenCL 定义了全局内存、局部内存与私有内存等分层存储空间，分别对应不同层级的执行作用域。主机内存与设备端内存在逻辑上相互独立，二者之间的数据交互通过显式的数据拷贝或映射机制完成。

1.2 FT-M7032 处理器架构

FT-M7032 是国防科技大学面向 E 级计算自主研发的一款异构多核数字信号处理器，其设计面向高性能计算、信号处理等计算密集型场景，旨在实现高能效与高性能的协同。如图 1 所示，该芯片由一个多核 CPU 与四个通用 DSP 簇（GPDSP 簇）构成。多核 CPU 基于 ARMv8 架构设计，集成了 16 个处理器核心，主要负责操作系统运行、任务调度、外设管理及多芯片间通信等系统级控制与管理功能。计算任务主要由 GPDSP 簇承担，每个簇包含 8 个 DSP 核心及 6 MB 容量的全局共享内存（global shared memory, GSM），各 DSP 核心通过片上互连网络实现高效数据交互与协同计算。CPU 与所有 GPDSP 簇共享统一的内存地址空间，但每个 GPDSP 簇仅能访问其对应的独立 DDR 内存通道。

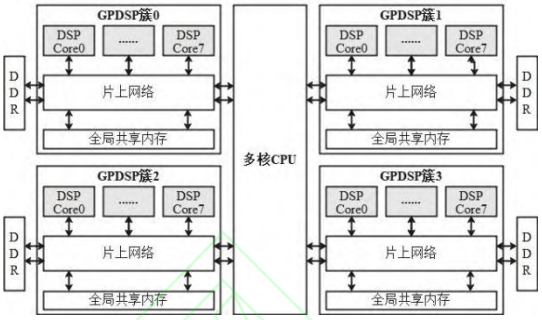

图 1 FT-M7032 芯片的整体架构

Fig.1 Architecture of FT-M7032 multi-core DSP

如图 2 所示，每个 DSP 核心采用基于超长指令字的标量-向量协同执行架构，集成了标量处理单元（scalar processing unit, SPU）、向量处理单元（vector processing unit, VPU）、指令调度单元（instruction fetch unit, IFU）及 DMA 引擎等功能模块。SPU 包括指令流控制器、一级指令缓存（L1I）及标量处理部件（scalar processing element, SPE），并配备 64 KB 的标量存储器（scalar memory, SM），用于程序流控制与标量运算。VPU 作为核心计算单元，由 16 个以 SIMD 方式协同工作的向量处理部件（vector processing elements, VPEs）及 768 KB 的向量存储器（vector memory, AM）组成。每个 VPE 集成 3 个浮点乘累加（floating point multiply accumulator，FMAC）单元和 1 个位操作（bit processing，BP）单元以及 2 个 Store/Load 部件来实现运算和访存。每个 VPE 包含 64 个 64 位寄存器，整体 VPU 支持 1024 位（16×64）的向量宽度，适用于各类数据密集型并行计算。

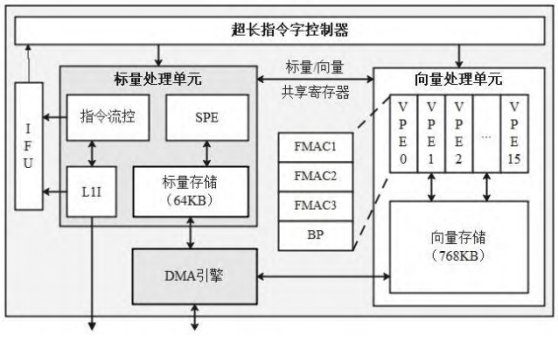

图 2 DSP 核心微架构

Fig.2 Microarchitecture of FT-M7032 DSP core

如图 3 所示，FT-M7032 具备多层次存储体

系，涵盖 DDR、簇内 GSM 及各核心内部的 SM 与 AM，访存带宽逐级递增。其中 GSM、SM 与 AM 均属于软件管理的片上存储器，具备高效低耗、灵活管理的特点。其中，向量寄存器仅能访问 AM 空间，而标量寄存器可访问除 AM 外的其他存储空间。VPU 与 SPU 在架构上相互独立，二者仅能通过标量/向量共享寄存器实现数据交互，支持标量广播、向量提取等操作。DMA 引擎负责在不同存储层次之间进行高效数据传输，提升整体数据吞吐率。

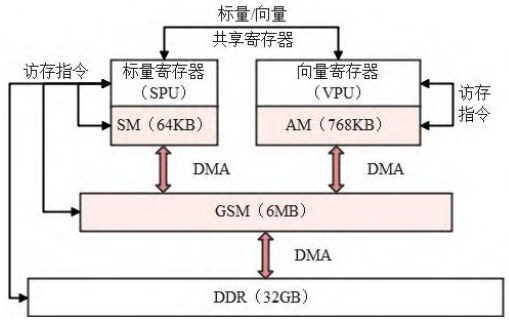

图 3　FT-M7032 芯片的存储层次

Fig.3 Memory hierarchy in FT-M7032

2　MOCL4 运行时支持

MOCL4 总体架构如图 4 所示，从整体上看由运行时系统与内核编译器两大部分组成。其中，运行时系统主要由 CPU 上的主机端负责 OpenCL 运行时接口的处理，并通过 hthreads 接口调用多核 DSP 设备端完成内存分配与释放、内核执行等操作。内核编译器则负责将 OpenCL 内核代码转换为 DSP 可执行文件，并在编译阶段完成与体系结构相关的关键优化。

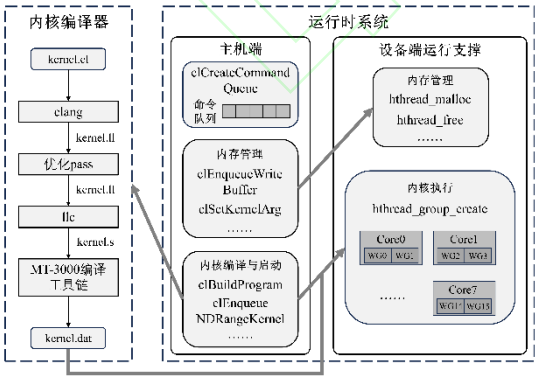

图 4　MOCL4 系统架构

Fig.4　Overall architecture of MOCL4

运行时系统部分如图 4 右侧所示，主要负责 OpenCL 执行语义的管理与内核执行控制。在主机端，MOCL4 运行时实现了 OpenCL 规范定义的编程接口，包括平台与设备管理、命令队列创建、内存对象操作以及内核构建与启动等功能。主机端运行时将 OpenCL 主机程序中的执行请求组织为命令队列，并维护命令之间的顺序与依赖关系。针对该队列中的执行请求，运行时通过 hthreads 接口[2]对多核 DSP 设备端资源进行统一管理与控制，从而完成设备端内存分配以及内核执行的触发。内核函数以工作组为基本执行单位映射至不同 DSP 核心执行。相较于 MOCL3[2]，MOCL4 在运行时系统层面对 OpenCL 语义支持进行了完善，并在内核映射策略与启动开销等方面进行了针对性的优化。

在内核编译方面，与 MOCL3 不同，MOCL4 不再将 LLVM IR 转换为 C 代码，而是直接基于 LLVM 编译框架生成面向 DSP 的汇编代码。随后，生成的汇编文件通过 FT-M7032 的原生编译工具链完成编译与链接，最终形成可在多核 DSP 上执行的内核二进制文件（kernel.dat）。这一编译路径的调整使得 MOCL4 能够在 IR 层更直接地实施体系结构相关优化，例如内核向量化以及基于 DMA 的数据传输优化。

2.1　OpenCL 模型映射

为确保 OpenCL 程序在多核 DSP 平台上的高效执行，MOCL4 对 OpenCL 模型进行了面向 DSP 体系结构的定制化映射。这一映射是整个运行时系统设计的基石。表 1 详细展示了 OpenCL 模型中的主要抽象概念与 FT-M7032 硬件实体的对应关系。

表 1　OpenCL 模型与 FT-M7032 硬件的映射

Tab.1 Mapping between OpenCL and FT-M7032 hardware

OpenCL 抽象	FT-M7032
主机	多核 CPU
主机内存	DDR
计算设备	GPDSP 簇
全局/常量内存	DDR
计算单元	DSP 核心
本地内存	标量：SM 向量：AM
处理单元	虚拟处理单元
私有内存	寄存器

在 MOCL4 的架构中，多核 CPU 承担主机角色，负责运行主机端程序并实现 OpenCL 规范所定义的管理与调度接口。每个 GPDSP 簇被映射为一个独立的计算设备。一个计算设备包含 8 个

计算单元，即 DSP 核心，用于执行 OpenCL 内核函数。由于 DSP 核心数量有限，MOCL4 采用了工作项合并技术来模拟大量处理单元。具体而言，工作组内的所有工作项不再并发执行，而是被合并成一个循环，以工作项索引为迭代变量进行顺序执行。计算任务以工作组为基本单位分配到各个 DSP 核心。

根据 OpenCL 规范，全局内存必须能够被所有工作组中的所有工作项访问。因此，MOCL4 将全局内存映射到 DDR 上。由于 FT-M7032 的每个 GPDSP 簇只能访问其对应的 DDR，而无法直接访问其他簇的 DDR，且主机端程序遵循标准 C 语言语义进行内存分配，无法在分配时指定目标簇。因此，在实现主机与设备间的内存对象交互（如显式复制或映射）时，运行时系统仍需在设备端显式分配内存空间并进行数据拷贝，而无法直接操作由主机端分配的内存。在每个 DSP 核心内部，其 SM 和 AM 被用作本地内存，分别存储工作组内共享的标量与向量数据。而工作项私有变量则被分配到核心的寄存器中。

2.2 内核调度与执行

在 FT-M7032 多核 DSP 平台上，MOCL4 运行时以工作组作为内核调度与执行的基本单位，采用静态工作组分配策略。在内核启动阶段，运行时根据 NDRange 定义的工作组组织方式及可用 DSP 核心数量，将三维工作组索引空间线性化并进行均衡划分，将连续的工作组区间静态映射至各 DSP 核心。在内核执行过程中，工作组始终固定在同一 DSP 核心上运行，不发生跨核心迁移，从而避免跨核心同步与状态维护开销。该调度方式无需运行时负载反馈或动态调度机制，具有实现简单、调度行为确定的特点，在工作组执行代价相对均衡的场景下能够有效平衡各 DSP 核心负载，符合多核 DSP 平台以粗粒度任务执行为主的体系结构特征。

基于上述执行映射方式，OpenCL 中的工作组屏障（barrier）不再需要通过硬件同步，而是由内核编译器在编译阶段通过软件方式实现。MOCL4 在内核编译过程中采用了基于延续的同步（continuation-based synchronization，CBS）[20-21]实现方法。该方法以 barrier 为边界将内核划分为多个顺序执行的子区域，并为每个子区域生成对应的工作项循环。内核执行时，按照 barrier 顺序依次执行这些工作项循环，从而保证所有工作项在进入下一执行阶段之前均已完成 barrier 之前的代码段，满足 OpenCL 对工作组内同步与内存一致性的语义要求。

针对 CBS 实现中跨 barrier 私有变量需要通过上下文数组保存与恢复而引入的额外访存开销，MOCL4 进一步采用了基于重计算的变量扩展优化策略[16]。该优化对可由统一变量或工作项索引推导的跨 barrier 私有变量，不再生成上下文数组访问，而是在使用点通过等价计算进行重构，以计算替代访存，从而降低 barrier 相关的内存访问开销并提升同步效率。

2.3 启动开销优化

即时编译（just-in-time, JIT）技术是现代运行时系统的核心组件，它通过在程序执行期间将字节码或中间表示动态编译为优化的本地机器码，有效平衡了跨平台可移植性与执行效率之间的矛盾。因此，MOCL4 也采用了该机制用于内核程序的生成和执行。

然而，JIT 技术会由于以下原因引入额外开销：（1）JIT 编译过程本身需要消耗可观的 CPU 计算与内存资源，在执行复杂的多阶段优化流水线时可能导致应用程序线程的显著停顿。（2）在异构系统中，编译生成的内核代码必须经由高速外设组件互连（peripheral component interconnect express，PCIe）等总线加载到加速设备上才能执行，产生了额外的数据传输开销。

在上述的两种开销中，前者可在内核函数反复执行的过程中被均摊，而后者则在每次内核函数启动时都会产生。根据实际测量，未优化的情况下每次内核函数的加载开销可以达到 12 毫秒，这会严重妨碍由小规模内核函数构成的程序的性能。

为此，MOCL4 提出了一种启动开销优化方法。该方法规避重复加载同一内核函数，同时将一段时间内可能连续执行的多个内核函数统一链接为一个内核二进制进行聚合加载，以减少程序运行时加载内核函数的次数。具体而言，该方法包括以下几个步骤：（1）内核函数通过 JIT 编译生成中间代码。（2）在运行时维护的数据结构中查询该内核函数是否已经存在于设备当前已加载的内核镜像文件中。（3）若存在，则跳过内核函数加载过程，仅在主机端记录相关元数据。（4）若不存在，则将该中间代码与当前设备已加载的内核函数进行链接，生成更新后的内核二进制函数，重新加载到设备上。

与未优化方法相比，该方法仅在内核函数首

次编译时引入额外的时间开销。当程序包含多个内核函数时，需要将新生成的中间代码与当前设备已加载的内核函数进行链接以生成更新后的内核二进制。而在后续每次内核启动时，仅需执行一次主机端数据结构查询，以判断目标内核函数是否已存在，其开销可以忽略不计。作为回报，在内核函数被多次启动的情况下，仅需一次加载内核函数镜像，从而减少重复加载带来的开销。同时，该方法可能带来的额外内存占用主要体现在设备侧代码镜像的驻留上。对于单一内核函数程序，优化前后设备内存占用基本不变；对于多内核函数程序，设备侧需保存聚合后的多内核镜像，但其规模仅为 KB 量级，相对于目标 DSP 的 16 GB 设备内存可以忽略不计。

3 OpenCL 内核编译器设计

为使 OpenCL 内核在多核 DSP 上获得高效执行，内核编译器不仅需要正确实现 OpenCL 并行语义，更必须充分契合 DSP 的体系结构特征。FT-M7032 在单核内部集成专用 SIMD 向量单元，采用多级异构 SPM 存储层次，并支持通过 DMA 引擎实现高带宽数据搬运。此类硬件架构具有较强的数据级并行计算能力与可控的数据缓存能力，但其性能潜力在很大程度上依赖于编译器在并行映射、向量化、数据布局等方面的系统化优化。

针对 OpenCL 内核函数的向量化实现，目前主要存在两种策略：一是依赖编译器对内核内部循环进行自动向量化，其向量化对象由编译器的分析结果决定；二是从工作项并行性出发，将整个内核视为 SPMD 计算实例的集合，通过全函数向量化方式将多个工作项直接映射到 SIMD 向量通道执行[20-24]。其中，前者的自动向量化技术主要面向传统标量程序，其效果严重依赖于对循环结构和访存模式的分析，且无法有效识别以 SPMD 形式显式表达的跨工作项并行语义。基于上述认识，并考虑到向量化层次与粒度选择对程序性能的直接影响[25]，MOCL4 选择在工作项层次对内核函数进行向量化。该方法充分利用了工作项之间天然的数据并行性，同时避免了对复杂循环结构与依赖关系的间接推断。

围绕向量化设计，内核编译器进一步引入一系列面向 DSP 硬件的协同优化机制。首先，编译器实施了面向硬件的 IR 规范化处理，使向量化后的中间表示在类型、控制流与访存形式上更贴合目标 DSP 的执行模型与指令约束，从而降低后端代码生成的复杂度。其次，编译器设计了标量与向量统一的 DMA 数据搬运机制，在确保向量运算正确性的同时，实现高带宽访存。

3.1 内核函数向量化

内核编译器采用全函数向量化方法，以内核函数整体为对象进行向量化处理，借助掩码传播与谓词化执行机制，将原有的标量控制流转换为向量化控制流。该方法能够直接刻画工作项并行性与 SIMD 向量通道之间的对应关系，实现从 OpenCL 并行语义到向量单元的有效映射。

在对内核函数进行函数级向量化时，关键在于确定向量因子以及进行向量一致性分析。向量因子指每次向量化执行中并行处理的工作项数量，该值在整个内核函数范围内保持一致。编译器结合向量单元的恒定位宽以及内核函数的全局参数的数据类型位宽，确定统一的向量因子。

向量一致性分析用于识别静态单赋值（static single assignment，SSA）定义的值在不同工作项中是否保持一致或呈现规则变化，进而为向量化转换提供依据。根据分析结果，值可以分为三类：统一的、连续的以及变化的，其中连续是变化的一种特例。在 OpenCL 的执行语义下，所有工作项在内核函数入口处共享相同的函数参数与常量值，因此这些值在分析初始阶段被视为统一的。工作项索引是并行差异的主要来源，其值随工作项实例不同而变化。由于工作项可能具备多个维度，需进一步明确其在向量通道间的变化规律。编译器从多个工作项维度中选定一个作为向量化维度，仅在该维度上定义工作项索引是连续变化的，其余维度的工作项索引则是统一的。为了提升向量化执行时的内存访问效率，优先选择能够生成更多连续或统一访存模式的工作项维度进行向量化。

具体而言，编译器针对每个候选工作项维度分别执行一次向量一致性分析，并统计访存操作中连续访存、统一访存以及变化访存三类模式的数量，分别记为 N_c、N_u 和 N_v。其中，连续访存可以直接映射为向量连续访存，统一访存可通过标量广播实现，而变化访存则往往需要额外的数据重组，因此其执行开销相对较高。基于这一观察，编译器采用如下启发式评分函数对各维度进行评估：

$$s = 1.2 \times N_c + 0.8 \times N_u - 1.8 \times N_v$$

其中 s 为评分，最终选择评分最高的维度作为向量化维度。当多个维度评分相同时，默认选择第 0 维作为向量化维度；若最高评分为负，则认为向量化难以带来性能收益，从而放弃该内核的向量化转换。当前评分权重基于经验设定，后续将根据目标平台的实际内存访问特性进一步进行调整。

在向量化阶段，编译器仅对内核函数的计算结构进行向量化处理，并显式保留原有的 barrier 同步点，而不在该阶段引入同步语义的实现。为支持工作组 barrier 的正确执行，编译器需在向量化后的中间表示上再一次执行一致性分析。该分析用于判定在 barrier 前后是否需要为每个工作项分别保存和恢复其私有变量值。与第一次分析时不同，这一阶段不再区分向量化维度，而是将所有维度的工作项都设置为连续变化的。

3.2 面向硬件的 IR 规范化

在完成内核函数的全函数向量化之后，编译器构建的向量化执行路径主要以保持向量语义正确性为目标，其类型与谓词表示仍具有较强的体系结构无关性，因此 IR 中可能同时存在多种位宽的向量类型及<N×i1>形式的布尔谓词。这些表示在 LLVM IR 语义上是合法的，但并不一定与目标 DSP 的指令约束直接对应。若不加处理地进入指令选择阶段，后端可能无法为部分向量操作找到满足类型与操作数约束的目标指令，只能通过插入额外的类型转换、向量拆分，甚至放弃向量化路径来规避该问题，从而影响代码生成的稳定性。因此，内核编译器在全函数向量化之后引入一次面向硬件的 IR 规范化过程，在进入指令选择之前消除中间表示与目标 DSP 指令集之间的结构性不匹配问题。该过程主要从向量数据类型、向量谓词表示以及条件访存三个方面对向量化后的 IR 进行规范化处理。

首先，在向量数据类型方面，目标 DSP 的向量部件采用固定宽度的 SIMD 向量寄存器（1024 位），对应 16 个 64 位向量通道。而全函数向量化后的 IR 中可能同时存在<16×i32>、<16×f32>等不同元素位宽的向量类型，这将影响后端在寄存器分配与指令选择阶段的正确性和复杂度。为此，编译器在不改变向量通道数的前提下，对向量元素位宽进行统一规范化处理，将 32 位整数与浮点向量元素统一拓宽为 64 位表示，使向量数据能够直接映射到目标 DSP 的物理向量寄存器，从而避免由于元素位宽不一致导致的指令匹配困难。

其次，在向量谓词表示方面，LLVM IR 中的向量比较指令通常产生<N×i1>类型的布尔向量，而目标 DSP 的条件执行机制依赖条件寄存器控制各 VPE 的执行与否。条件寄存器由 16 个 64 位分量构成，并不支持布尔向量形式。基于这一差异，编译器在 IR 规范化阶段将所有向量谓词统一转换为<16×i64>类型的谓词掩码表示，并以该表示作为条件执行与掩码传播的统一形式。在此基础上，原有依赖布尔向量的条件选择操作被改写为基于谓词掩码的向量条件移动，从而使条件执行语义在 IR 层面与 DSP 的条件指令支持保持一致。

最后，在条件访存方面，目标 DSP 的向量存取指令不支持直接的条件执行，但可以预先通过向量长度寄存器决定实际参与执行的向量通道。为此，编译器在 IR 规范化阶段对向量化后的掩码访存指令进行统一处理。对于加载类操作，由于加载本身不产生可观测副作用，其结果是否生效由后续条件写回决定，因此掩码加载可在 IR 层面统一映射为普通向量加载。而对于存储类操作，则需要在实际写回指令执行前后，依据谓词掩码显式配置并恢复向量长度寄存器，从而确保仅对满足条件的向量通道执行写回操作。

此外，内核编译器还针对一些与目标 DSP 指令约束相关的中间表示进行了必要的规范化处理，具体实现不在此进行赘述。图 5 结合一个示例 IR 片段进一步说明上述规范化过程。该示例仅保留与规范化相关的关键指令，包括通过 insertelement 与 shufflevector 广播构造的小位宽向量、由向量比较生成的谓词，以及两个典型的条件访存操作。通过对比规范化前后的 IR 可以看到，原始的<16×i32>广播向量被统一拓宽为<16×i64>，由比较指令产生的<16×i1>谓词被转换为<16×i64>掩码，以该谓词为操作数的访存指令也进一步被重写，从而使生成的中间表示更符合目标 DSP 的后端约束。

```
; 广播构造 <16 x i32> 向量
%v0 = insertelement <16 x i32> poison, i32 %x, i32 0
%vx = shufflevector <16 x i32> %v0, <16 x i32> poison, <16 x i32> zeroinitializer
; 加法生成索引向量
%idx = add <16 x i32> %vx, <i32 0, i32 1, ... , i32 15>
; 比较生成 <16 x i1> 谓词
%mask = icmp slt <16 x i32> %idx, %bound
; 条件访存
%val = call <16 x double> @llvm.masked.load(<16 x double>* %ptr1, i32 8,
                                    <16 x i1> %mask, <16 x double> undef)
call void @llvm.masked.store(<16 x double> %res, <16 x double>* %ptr2,
                                    i32 8, <16 x i1> %mask)
```

(a)原始向量化 IR

(a)Original vectorized IR

```
; 拓宽为 <16 x i64>
%v0.w = insertelement <16 x i64> poison, i64 %x.w, i32 0
%vx.w = shufflevector <16 x i64> %v0.w, <16 x i64> poison, <16 x i32> zeroinitializer
%idx.w = add <16 x i64> %vx.w, <i64 0, i64 1, ... , i64 15>
; 调用DSP内建函数直接生成 <16 x i64> 掩码
%mask.w = call <16 x i64> @vec_lt( <16 x i64> %idx.w, <16 x i64> %bound.w)
; masked load 改写为普通向量加载
%val = load <16 x double>, <16 x double>* %ptr1, align 8
; masked store 改写为基于配置VLR的条件存储
call void @configVlr(<16 x i64> %mask.w)
store <16 x double> %res, <16 x double>* %ptr2, align 8
call void @recoverVlr()
```

(b)规范化后的向量化 IR

(b)Normalized vectorized IR

图 5 规范化示例

Fig.5 Normalization example

3.3 DMA 数据传输优化

尽管向量计算路径已在指令层面就绪，但对向量单元的有效利用仍受限于目标多核 DSP 存储体系结构对数据访问的严格约束。在 FT-M7032 多核 DSP 中，向量寄存器仅能访问 AM 空间，而 DDR 中的数据必须通过 DMA 搬运至片上存储后才能参与向量访存与计算。同时，尽管标量寄存器可以直接访问 DDR，但在访存密集的计算阶段，直接进行标量 Store/Load 操作仍受限于 DDR 带宽。因而，仅完成计算结构的向量化并不足以充分发挥硬件性能，仍需在编译阶段显式构造 DMA 数据传输，使计算与数据搬运在片上存储层次上协同进行。

图 6 以通用矩阵乘法（general matrix multiply，GEMM）为例展示了向量化与 DMA 协同优化的整体过程。图 6(a)给出了一个以工作项循环形式组织的原始 GEMM 内核函数，其中循环 j 与 i 分别对应第 0 维与第 1 维工作项，lz_j 与 lz_i 表示对应维度上的工作组大小。GEMM 的计算规模为 N，循环 k 沿矩阵维度展开标量乘加运算。在该形式下，所有数组访问均直接面向 DDR 存储空间。

```
1  for (i=0; i < lz_i; i++){                          //1维工作项循环
2    for (j=0; j <  lz_j; j++) {                      //0维工作项循环
3    c[i * N + j] *= beta;
4      for(k=0; k < N; k++) {
5        c[i * N + j] += alpha * a[i * N + k] * b[k * N +j];
6  }}}
```

(a)原始 GEMM 内核

(a)Original GEMM kernel

```
1  sm_a[N];
2  am_b[N*lz_j];                                      //double16
3  am_c[lz_i*lz_j];                                   //double16
4  dma(&b[0], N, lz_j, N-lz_j, am_b, N, lz_j, 0);
5  dma(&c[0], lz_i, lz_j, lz_i -lz_j, am_c, lz_i, lz_j, 0);
6  for (i=0; i < lz_i; i++){                          //1维工作项循环
7    for (j=0; j <  lz_j/16; j++) {                   //0维工作项循环, 向量化
8      am_c[i * (lz_j/16) + j] *= vbroadcast(beta);
9      dma(&a[i*N], 1, N, 0, sm_a, 1, N, 0);
10     for(k=0; k < N; k++) {
11       am_c[i * (lz_j/16) + j] += alpha * sm_a[k] * am_b[k * (N/16) +j];
12 }}}
13 dma(am_c, lz_i, lz_j, 0, &c[0], lz_i, lz_j, lz_i -lz_j);
```

(b) 优化后的 GEMM 内核

(b) Optimized GEMM kernel

图 6 向量化与 DMA 协同优化的 GEMM 内核

Fig.6 GEMM kernel with vectorization and DMA optimization

图 6(b)展示了向量化与 DMA 优化后的内核函数。编译器选择第 0 维工作项进行向量化，将原先的标量循环 j 转化为向量循环，对应向量寄存器中 16 个通道的并行执行。在此基础上，编译器结合向量化后的访存模式与循环嵌套结构，自动构造 DMA 数据传输，在计算阶段开始前将向量与标量计算所需的数据块搬运至 AM 或 SM 等片上存储空间，从而为后续的向量运算提供高带宽的数据访问支持。

DMA 传输的本质在于将循环迭代过程中产生的多次离散访存请求合并为批量数据搬运。在以工作项循环形式组织的向量化内核中，循环结构通常呈现嵌套形式，不同循环层次分别引入连续或规则跨步的地址变化，使得访存模式可抽象为由多个循环维度共同描述的规则访问结构。若仅依据单一循环层次构造一维 DMA 传输，不仅会限制单次传输的数据规模，降低有效带宽利用率，还会由于频繁启动 DMA 而引入显著的调度与同步开销。为此，编译器在构造 DMA 传输时将嵌套循环结构作为重要的分析依据，通过同时考虑多个循环层次对访存地址的影响，将访存模式映射为一维或二维 DMA 请求。在该映射过程中，连续访存被抽象为 DMA 传输中的一行，而规则跨步则对应行间步长，两个循环层次的迭代范围共同决定了二维 DMA 覆盖的数据区域。以

图 6 中的矩阵乘为例，矩阵 B 与 C 在向量化维度上呈现连续访问，而在另一维度上具有固定步长。编译器据此生成覆盖多个工作项与循环迭代的二维 DMA 请求，在计算阶段开始前一次性将所需数据子块搬运至片上存储，并在计算完成后按相同的二维结构写回 DDR。通过这种方式，原本分散在循环体内的多次访存操作被合并为少量高效的 DMA 数据传输，从而显著提高带宽利用率。

除向量访存外，编译器还对内核中的残留标量访存进行分析与优化。与向量访存不同，标量访存的 DMA 优化并非功能正确性所必需，而是以性能收益为导向。考虑到 SM 空间容量有限（64KB），若对标量访存直接采用二维 DMA 合并，单次传输所需的片上存储空间可能超过可用容量，并引入额外的分块与调度复杂度。基于上述权衡，选择对标量访存采用一维 DMA 合并策略，以在传输开销与片上存储资源占用之间取得平衡。

4 实验效果与性能分析

4.1 实验环境与设置

实验在异构多核 DSP 芯片 FT-M7032 上进行，用于评估 MOCL4 的正确性与性能优化效果。对于以 GEMM 为代表的密集计算内核，进一步与基于 hthreads 接口的手工优化实现进行对比分析，其中 hthreads 实现与 MOCL4 内核编译器采用相同的 LLVM 后端和底层编译工具链，以避免编译器差异对性能结果产生影响。正确性验证采用 PoCL 提供的标准 OpenCL 测试用例集，对运行时与编译器语义实现进行系统测试。性能评估主要采用 PolyBench 基准测试集[26]，该测试集涵盖多种典型数值计算与科学计算内核，能够较为全面地反映内核向量化、DMA 数据搬运及片上存储利用等方面的优化效果。所有性能测试结果均取 20 次独立运行的平均值，以减小系统噪声与偶发抖动的影响。

4.2 MOCL4 正确性验证

PoCL 测试用例集涵盖了 OpenCL 运行时（Runtime）、计算内核（Kernel）以及工作组调度（Workgroup）等多个关键维度，能够有效评估编译器的正确性。

实验结果如表 2 所示，在测试的 234 个用例中，MOCL4 成功通过了 221 个，整体通过率达到了 95%。未通过的 13 个测试用例主要集中在图像与采样器操作相关测试，受限于目标硬件架构特性，此类功能目前不在支持范围内。

表 2 MOCL4 正确性测试结果

Tab.2 Correctness test results of MOCL4

测试类别	用例数量	通过数	通过率
Runtime	31	30	97%
Kernel	77	67	87%
Workgroup	31	31	100%
Regression	95	93	98%
Total	234	221	95%

4.3 启动开销优化效果评估

为评估内核函数启动开销优化的效果，从 PolyBench 基准测试集中选取了 5 个具有代表性的程序进行测试，具体如表 3 所示。表中“内核函数数目”表示程序中包含的不同内核函数数量，“内核函数启动次数”表示每次程序运行过程中向设备端提交的内核函数总次数。表 3 给出了不同测试程序在启用启动开销优化前后的运行时加速比。在该测试中，运行时包含首次执行时的内核编译与加载开销，反映的是启动优化在程序重复运行场景下的整体收益。

表 3 启动开销优化测试程序

Tab.3 Benchmark programs for startup overhead optimization

程序名称	内核函数数目	内核函数启动次数	加速比
2DCONV	1	1	1.04
ATAX	2	2	1.10
CORR	4	4	1.003
FDTD2D	3	1500	1.29
SYR2K	1	1	1.002

启动优化在所选程序上平均能够获得约 1.08 倍的性能提升。其中，FDTD2D、ATAX 和 CORR 均包含多个内核函数，因此在首次编译执行时都需要将多个内核统一链接后加载到设备端。不过，这一额外开销较低，且容易在程序重复运行过程中被摊销。三者优化效果的差异主要来源于内核启动次数的不同。FDTD2D 的每个内核函数在一次程序执行中都会被重复启动 500 次，因此相较于未经优化的方法，避免重复加载内核镜像所带来的累计收益最大，从而获得最高 1.29 倍的加速比。相比之下，ATAX 和 CORR 的每个内核函数在一次程序执行中仅启动 1 次，其收益主要来源于每次程序执行中只需加载一次聚合后的内核镜

像，因此整体优化效果弱于 FDTD2D。在这两者中，ATAX 的计算负载较低，启动开销在总执行时间中的占比更高，因此仍可获得约 1.10 倍的性能提升；而对于 CORR 这类计算和访存开销占主导地位的程序，优化收益则不明显。

对于 2DCONV 和 SYR2K 这类仅包含单一内核函数且每次程序运行仅启动一次内核的程序，该方法在首次编译执行时不会引入额外的多内核链接开销，仅增加一次主机端数据结构查询，其开销可以忽略不计。此类程序在实验中的性能收益主要来源于测试时对同一程序的重复执行，因为设备端已加载的内核镜像在执行过程中未被释放，后续运行仍可避免重复加载开销。

4.4 内核编译器性能评估

4.4.1 MOCL4 总体性能

为评估 MOCL4 相较于现有 OpenCL 实现的改进效果，本文在 PolyBench 基准测试集上对 MOCL4 与 MOCL3 进行了对比实验。实验选择 MOCL3 为基线，对比的是未进行内核编译优化的 MOCL4 实现，因此该对比主要反映两种系统在编译流程上的差异。实验结果如图 7 所示，其中 AVG 表示所有测试程序的平均加速比。从整体结果来看，MOCL4 在大多数测试程序上均取得了一定程度的性能提升，平均加速比为 1.20 倍，最高可达 1.50 倍。MOCL3 通过 C 语言的间接编译方式会将较为精确的中间表示重新映射为较高层次的 C 语言结构，部分与优化相关的信息（如数据类型、别名关系等）难以完全保留，从而在一定程度上影响最终生成代码的执行效率。相较而言，MOCL4 直接基于 LLVM 中间表示生成目标代码的编译链能够带来更稳定的性能表现，并为后续实施体系结构相关编译优化提供了更加灵活的基础。

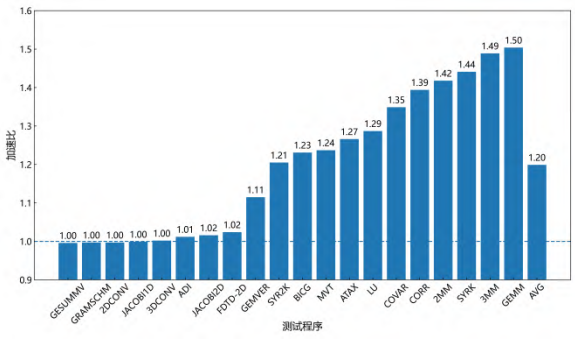

图 7　MOCL4 相对于 MOCL3 的性能加速比

Fig.7 The speedups of MOCL4 over MOCL3

进一步，为评估 MOCL4 的内核编译优化策略，实验以前述未进行内核编译优化的 MOCL4 为基线，对比启用不同优化配置后的性能加速比。实验结果如图 8 所示。其中，VEC-AM 表示启用内核向量化并通过 DMA 引擎将数据传输到 AM 以支撑向量计算的数据访问；+SM 表示在 VEC-AM 的基础上，进一步引入 SM，通过 DMA 对标量访存进行合并，从而减少计算阶段对 DDR 的直接访问，AVG 则表示开启全部优化后，所有测试程序加速比的平均值。

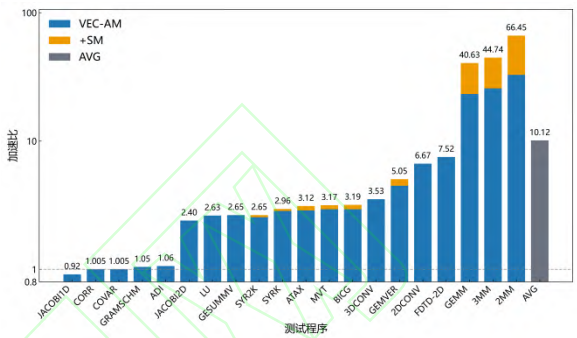

图 8　MOCL4 内核编译优化的性能加速比

Fig.8 The speedups of MOCL4 kernel compiler optimizations

从整体结果来看，优化编译器在绝大多数 PolyBench 程序上均取得了不同程度的性能提升，平均加速比达到 10.12 倍。其中，GEMM、2MM 与 3MM 等典型密集线性代数程序的加速效果最为显著，在引入 SM 后分别达到 40.6、44.7 与 66.5 倍。这类程序具有多重嵌套循环、规则且连续的内存访问模式以及较高的数据复用度。一方面内核向量化充分利用了向量单元算力，另一方面将数据搬运至 AM 和 SM 有效提高了访存带宽，从而在整体上获得数量级的性能提升。

在 SYRK、SYR2K、ATAX、GESUMMV、MVT 与 BICG 等程序中，优化加速比主要集中在 2.6～3.2 倍，明显低于 GEMM 类程序。其主要原因在于，这类程序在向量化维度上普遍存在跨步访存模式。尽管其整体循环结构规则，适合进行内核向量化，但跨步访存导致向量通道内的访存难以完全合并为连续的数据块。对于跨步访存，目前的 MOCL4 实现无法构造高效的 DMA 传输，只能通过标量/向量共享寄存器搬运 DDR 数据创建向量，限制了访存带宽，从而限制了整体加速上限。针对上述跨步访存带来的 DMA 效率下降问题，未来可考虑从两个方向进行改进。一方面，对于固定小步长的跨步访问（如步长为 2 或 4），可利用 DSP 提供的模 2/模 4 读写指令直接实现跨步访问。另一方面，对于同一访存语句在向量循

环层呈现跨步访问、而在与其嵌套的另一循环维度上具有连续访存特征的情况，可在该连续维度构造多次 DDR 连续的 DMA 传输，而在 AM 中保持跨步布局。由于系统带宽瓶颈主要位于 DDR 侧，该策略有助于提升 DMA 传输阶段的带宽利用率，从而在一定程度上缓解跨步访存带来的性能影响。

对于少数性能提升不明显的程序，其主要原因在于程序结构本身限制了内核向量化的适用性。部分程序包含多个内核函数，但并非所有内核函数都能够进行有效的向量化，典型如 CORR 与 COVAR，其关键计算内核 corr_kernel 无法进行内核向量化，导致程序整体几乎未获得性能提升。此外，JACOBI1D 等程序的内核算术强度过低，向量化引入的额外开销难以被有效摊销，在该类程序中启用内核向量化反而会导致性能下降。

综合实验结果可以看出，MOCL4 所采用的内核编译优化策略能够根据程序的结构特性有效发挥作用。对于具有较高计算强度、规则且连续的向量访存模式以及显著数据复用特性的内核，向量化与 DMA 的协同优化可以稳定地带来显著性能提升；而对于计算粒度较小或存在跨步访存特征的内核，优化收益则受限于其访存模式与计算特性。

4.4.2 GEMM 性能分析

为进一步评估 MOCL4 在密集计算内核上的优化效果，本小节以 GEMM 为例，对比 MOCL4 实现与基于 hthreads 接口的手工优化实现的性能表现。所有实验结果中的运行时间均已扣除启动开销，仅统计内核执行阶段的纯计算时间，以保证不同实现之间的可比性。图 9 以未优化的 hthreads 实现（即 native）为性能基线，给出了矩阵规模 N=2048 时，不同实现方式下 GEMM 内核的性能对比结果，包括基于 hthreads 接口的多种手工优化实现，以及不同优化配置下的 MOCL4 实现。hthreads 实现采用“X-Y-Z”的方式进行标记，其中 X 表示 hthreads 线程组中每个线程的计算任务划分方式，2D 表示线程在二维空间上协同计算矩阵块，该方式与 MOCL4 中基于二维 NDRange 的内核调度方式一致；Y 表示在单个内核函数内部对矩阵乘嵌套循环进行分块时所选取的分块维度；Z 表示使用的片上存储类型。

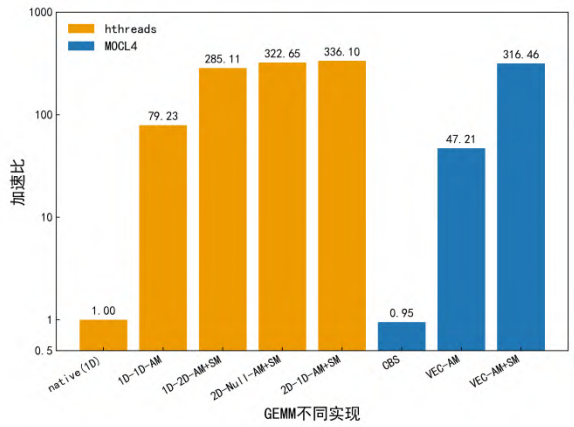

图 9 GEMM 内核在不同 hthreads 手工优化与 MOCL4 实现下的性能对比（矩阵规模为 2048）

Fig.9 Performance comparison between hthreads and MOCL4 for GEMM (N=2048)

从图中可以看出，未进行内核编译优化的 MOCL4 实现（CBS）性能与 hthreads 的 native 实现处于同一量级，表明在缺乏向量化与片上数据复用机制时，OpenCL 抽象本身并不能带来显著性能优势。基于 hthreads 接口的手工优化实现通过显式控制任务划分方式、DMA 数据搬运以及 AM/SM 片上存储的使用，能够在 native 实现的基础上获得数量级的性能提升。其中采用二维任务划分、引入循环分块并结合 AM 与 SM 的 DMA 搬运策略（2D-1D-AM+SM）在该规模下达到了性能上界。在 MOCL4 实现中，引入内核向量化并结合 AM 进行数据搬运后（VEC-AM），GEMM 性能得到显著提升；进一步在此基础上引入 SM 以增强片上数据复用（VEC-AM+SM）后，加速比达到 336.1 倍。总体而言，MOCL4 优化实现已经可以达到与 hthreads 性能上界相同的量级，仅存在小幅差距。

值得注意的是，hthreads 中的 2D-Null-AM+SM 与 MOCL4 的最优实现（即 MOCL4-VEC-AM+SM）是完全等价的，二者均采用一致的二维任务划分方式，并通过 DMA 将矩阵数据搬运至 AM 与 SM 中进行复用。但在实际测试中，二者的实际性能却呈现出一定差异，其主要源于 MOCL4 在运行时引入的额外抽象层开销，且呈现出显著的规模依赖性。

如表 4 所示，在矩阵规模较小时，MOCL4 抽象层开销在总执行时间中占比显著，致使 MOCL4 最优实现的加速比落后超过 40%。随着矩阵规模的增大，该开销被大幅增长的计算量有效摊销，对整体性能的影响已趋于微弱，MOCL4 的最优实现相对于 2D-Null-AM+SM 的性能比例

逐步提高，甚至可接近 100%。然而，与引入了循环分块策略的更高阶优化版本（2D-1D-AM+SM）相比，即便在大规模下，MOCL4 的最优实现仍与之存在持续的性能差距。这进一步表明，在较大计算规模下，精细的分块策略对于提升数据复用率、缓解片外访存压力的作用变得更为关键，而当前 MOCL4 在自动化构造此类优化策略方面仍存在局限。这一差距可通过调整 OpenCL 工作组大小来缩小，但此优化非本工作重点，故不做深入探讨。

表 4 不同矩阵规模下 MOCL4 最优实现相对于 hthreads 优化实现的性能比例

Tab.4 Relative performance of optimized MOCL4 versus optimized hthreads implementations across matrix sizes

矩阵规模	2D-Null-AM+SM	2D-1D-AM+SM
512	56.64%	58.48%
1024	89.71%	87.71%
1536	97.21%	91.91%
2048	98.08%	94.16%
3072	99.74%	95.00%
4096	99.97%	95.31%

为进一步验证 MOCL4 在更大规模问题下的可扩展性，本文对矩阵规模从 512 到 16384 的 GEMM 进行了测试，并采用归一化运行时间对结果进行分析。即以矩阵规模为 512 时的运行时间为基准，对各规模下的运行时间进行归一化处理。如图 10 所示，随着矩阵规模持续增大，矩阵乘运行时间基本呈现稳定的立方增长趋势，与理论计算复杂度 $O(N^3)$ 基本一致，没有发生明显的性能退化。实验结果表明，MOCL4 所采用的向量化与 DMA 协同优化策略在大规模问题下仍具有良好的稳定性与可扩展性，能够较好地维持计算与访存资源的利用效率。

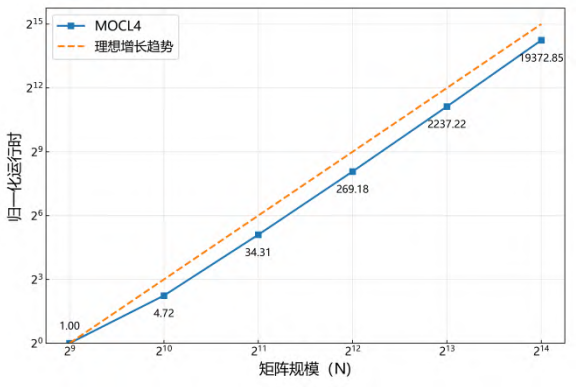

图 10 不同矩阵规模下 GEMM 归一化运行时间

Fig.10 Normalized runtime of GEMM across matrix sizes

总的来说，在 GEMM 这类密集计算内核中，MOCL4 通过编译层的内核向量化与 DMA 协同优化，能够将性能提升至接近原生手工优化实现的水平，且在大规模问题下仍表现出良好的稳定性与可扩展性。这一结果验证了 MOCL4 在规则算子上的性能潜力，也为后续进一步引入更高层次的自动分块与数据布局优化提供了方向。

5 结论

本文面向国产异构多核 DSP 平台 FT-M7032，设计并实现了高效的 OpenCL 异构并行编程系统 MOCL4。该系统通过运行时与编译器的协同优化，实现了 OpenCL 执行模型向 DSP 硬件的高效映射，支持内核向量化、DMA 数据搬运与片上存储层次管理，在保证 OpenCL 语义正确性的前提下显著提升了执行性能。在 PolyBench 测试集上，MOCL4 相较于 MOCL3 实现平均获得约 1.20 倍的性能提升。在启用内核编译优化后，MOCL4 进一步实现了 10.12 倍的平均加速比，其中 3MM 程序的加速比可达 66.45 倍。在 GEMM 等典型密集计算内核上，其性能已接近手工优化实现，并具备良好的可扩展性，表明该系统在性能与可编程性之间取得了良好平衡。

未来工作将重点探索自动化分块策略与面向不规则负载的智能调度机制，以进一步提升系统的通用性与性能上限。

参考文献（References）

[1]LI W J, FAN Z, LIU T, et al. DFU-E: A dataflow architecture for edge DSP and AI applications[J]. IEEE Transactions on Parallel and Distributed Systems, 2025, 36(6): 1100-1114.

[2]FANG J, ZHANG P, HUANG C, et al. Programming bare-metal accelerators with heterogeneous threading models: A case study of Matrix-3000[J]. Frontiers of Information Technology & Electronic Engineering, 2023, 24(4): 509–520.

[3]QI X, FANG J, ZHANG P, et al. Constraint-driven auto-tuning of GEMM-like operators for MT-3000 many-core processor[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE/ACM, 2025: 185-199.

[4]ZHU F, QI X, ZHANG P, et al. Optimizing stencil computation on multi-core DSPs[C]//Proceedings of the International Conference on Parallel Processing. ACM/IEEE, 2024: 679-690.

[5]YU K, QI X, ZHANG P, et al. Optimizing general matrix multiplications on modern multi-core DSPs[C]//Proceedings of the IEEE International Parallel and Distributed Processing Symposium. IEEE, 2024: 964–975.

[6]裴向东,王庆林,廖林玉,等.多核数字信号处理器并行矩阵转置算法优化[J].国防科技大学学报,2023,45(1):57-66.
PEI X D, WANG Q L, LIAO L Y, et al. Optimizing parallel matrix transpose algorithm on multi-core digital signal processors[J]. Journal of National University of Defense Technology,2023,45(1):57-66. (in Chinese)

[7]王庆林,裴向东,廖林玉,等.多核数字信号处理器矩阵乘卷积算法性能评测[J].国防科技大学学报,2023,45(1):86-94.
WANG Q L, PEI X D, LIAO L Y, et al. Evaluating matrix multiplication-based convolution algorithm on multi-core digital signal processors[J]. Journal of National University of Defense Technology,2023,45(1):86-94. (in Chinese)

[8]HAN R, ZHAO J, KIM H, et al. Unleashing CPU potential for executing GPU programs through compiler/runtime optimizations[C]//Proceedings of the IEEE/ACM International Symposium on Microarchitecture. 2024: 186–197.

[9]JÄÄSKELÄINEN P, SÁNCHEZ DE LA LAMA C, SCHNETTER E, et al. pocl: A performance-portable OpenCL implementation[J]. International Journal of Parallel Programming, 2015, 43(5): 752-785.

[10]LEE J, KIM J, KIM J, et al. An OpenCL framework for homogeneous manycores with no hardware cache coherence[C]//Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. IEEE/ACM, 2011: 56-67.

[11]JO G, JEON W J, JUNG W, et al. OpenCL framework for ARM processors with NEON support[C]//Proceedings of the Workshop on Programming Models for SIMD/Vector Processing. ACM, 2014: 33-40.

[12]KIM J, PARK S, SEO S, et al. Enabling an OpenCL compiler for embedded multicore DSP systems[C]//Proceedings of the International Conference on Parallel Processing Workshops. IEEE, 2012: 545-552.

[13]伍明川,黄磊,刘颖,等.面向神威·太湖之光的国产异构众核处理器 OpenCL 编译系统[J].计算机学报,2018,41(10):2236-2250.
WU M C, HUANG L, LIU Y, et al. An OpenCL compiler for the homegrown heterogeneous many-core processors on the Sunway TaihuLight Supercomputer[J]. Chinese Journal of Computers, 2018, 41(10): 2236-2250. (in Chinese)

[14]LEE J, KIM J, SEO S, et al. An OpenCL framework for heterogeneous multicores with local memory[C]//Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. IEEE/ACM, 2010: 193-204.

[15]WU M, LIU Y, CUI H, et al. Bandwidth-aware loop tiling for DMA-supported scratchpad memory[C]//Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. IEEE/ACM, 2020: 97-109.

[16]GAO W, FANG J, ZHANG P, et al. Optimizing OpenCL barrier synchronization and memory efficiency on multi-core DSPs[J]. ACM Transactions on Architecture and Code Optimization, 2025, 22(4): Article 120: 1-26.

[17]KIM H S, AHN M, STRATTON J A, HWU W M W. Design evaluation of OpenCL compiler framework for coarse-grained reconfigurable arrays[C]//Proceedings of 2012 International Conference on Field-Programmable Technology (FPT). IEEE, 2012: 313 - 320.

[18]NAH J, LEE J, KIM H, LEE J, HWANG S J, YOO D, LEE J. An OpenCL optimizing compiler for reconfigurable processors[C]//Proceedings of 2013 International Conference on Field-Programmable Technology (FPT). IEEE, 2013: 184 - 191.

[19]ZHAO X, WEN M, CHEN Z, et al. Automatic mapping and code optimization for OpenCL kernels on FT-Matrix architecture[C]//Proceedings of the ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems. ACM, 2021: 37-41.

[20]KARRENBERG R, HACK S. Improving performance of OpenCL on CPUs[C]//Proceedings of the International Conference on Compiler Construction. Springer, 2012: 1-20.

[21]MEYER J, ALPAY A, HACK S, et al. Implementation techniques for SPMD kernels on CPUs[C]//Proceedings of the ACM International Workshop on OpenCL. ACM, 2023: 1-12.

[22]KARRENBERG R, HACK S. Whole-function vectorization[C]//Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization. IEEE/ACM, 2011: 141-150.

[23]GHIGLIO P, DOLINSKY U, GOLI M, et al. Improving performance of SYCL applications on CPU architectures using LLVM-directed compilation flow[C]//Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores. ACM, 2022: 1-10.

[24]KANDIAH V, LUSTIG D, VILLA O, et al. Parsimony: Enabling SIMD/Vector Programming in Standard Compiler Flows[C]//Proceedings of the 21st ACM/IEEE International Symposium on Code

Generation and Optimization. IEEE/ACM, 2023: 186-198.

[25]BÜTTNER M, ALT C, KENTER T, et al. Analyzing performance portability for a SYCL implementation of the 2D shallow water equations[J]. The Journal of Supercomputing, 2025, 81(6): 772.

[26]GRAUER-GRAY S, XU L, SEARLES R, et al. Auto-tuning a high-level language targeted to GPU codes[C]//Proceedings of the Innovative Parallel Computing. IEEE, 2012: 1-10.