计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(4): 1017−1032, 2025

基于异构编程模型的共性算子移植与并行优化

马兆佳[1,2] 邵 恩[1,2,3] 狄战元[1,2] 马立贤[1,2]

[1](处理器芯片全国重点实验室(中国科学院计算技术研究所) 北京 100190)

[2](中国科学院大学 北京 100049)

[3](中科南京信息高铁研究院 南京 211100)

(mazhaojia23s@ict.ac.cn)

Porting and Parallel Optimization of Common Operators Based on Heterogeneous Programming Models

Ma Zhaojia[1,2], Shao En[1,2,3], Di Zhanyuan[1,2], and Ma Lixian[1,2]

[1] (*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)

[2] (*University of Chinese Academy of Sciences*, *Beijing* 100049)

[3] (*Nanjing Institute of InforSuperBahn*, *Nanjing* 211100)

Abstract As the fundamental computing component in constructing large-scale supercomputing systems, GPUs are undergoing architectural diversity and heterogeneity. GPU accelerators from various chip manufacturers exhibit significant variations in their architectural designs. Accelerator diversity and programming model diversity are important technical trends for building large-scale supercomputing systems. Diverse accelerators require developers to provide high-performance software for multiple hardware platforms, resulting in software duplication. To reduce the cost of duplication, the unified programming model SYCL (system-wide compute language) adapts to multiple hardware platforms, but SYCL's performance on different hardware is not as good as the native programming model of the platform, and SYCL's performance needs to be further optimized. In order to be able to apply the mature and complete CUDA (compute unified device architecture) programming ideas and high-performance programs to SYCL, it is necessary to discuss the performance of high-performance CUDA programs ported to SYCL on multiple platforms and the ideas for further optimization. Based on software-hardware co-design, we propose paraTRANS: a common operator optimization system for the code migration process of cross-heterogeneous programming model SYCL, and give the optimization methods for the migrated SYCL GEMM (general matrix multiplication) in different scenarios. We evaluate the performance of SYCL GEMM optimized by paraTRANS, which can achieve 96.95% of CUDA's FLOPS on the original NVIDIA RTX 3090, and 100.47% of CUDA's hardware peak performance percentage on AMD MI100, both close to the level before migration. This paper provides ideas for porting high-performance CUDA code to SYCL and further optimization.

收稿日期: 2023-10-31; **修回日期**: 2024-07-02

基金项目: 国家重点研发计划项目(2021YFB0300202); 国家自然科学基金项目(62102396); 北京市科技新星计划项目(Z211100002121143, 20220484217); 中国科学院青年促进会项目(2021099); CCF-蚂蚁科研基金项目(CCF-AFSGRF20230207); 天津市科技计划项目(24ZXKJGX00060); 中国科学院计算技术研究所创新课题项目(E461030)

This work was supported by the National Key Research and Development Program of China (2021YFB0300202), the National Natural Science Foundation of China (62102396), the Beijing Nova Program (Z211100002121143, 20220484217), the Youth Innovation Promotion Association of Chinese Academy of Sciences (2021099), the CCF-Ant Research Fund (CCF-AFSGRF20230207), the Tianjin Science and Technology Plan Project (24ZXKJGX00060), and the Innovation Funding of ICT, CAS (E461030).

通信作者: 邵恩(shaoen@ict.ac.cn)

Key words SYCL; cross heterogeneous architecture; code migration; GEMM; parallel optimization

摘 要 GPU 作为构造大规模超算系统的核心计算部件,向着体系结构多样化和异构化的方向发展. 来自不同芯片厂商的 GPU 加速器具有差异较大的体系结构设计. 加速器类型和编程模型多样化是构建大规模超算系统的重要技术趋势. 多样化加速器要求开发者为多种硬件平台提供高性能共性算法库软件,然而这也导致了算法库软件重复开发问题. 为降低重复开发成本,统一编程模型 SYCL(system-wide compute language)应运而生,并适配了多种硬件平台. 尽管如此,在不同硬件上,SYCL 的性能仍不及各自原生编程模型. 因此,需要进一步优化 SYCL 的性能以将目前成熟完备的 CUDA(compute unified device architecture)编程思路和高性能程序应用到 SYCL 中. 基于软硬件协同设计,提出了 paraTRANS 方法,该方法是面向跨异构编程模型 SYCL 代码移植过程中共性算子优化工具,并在不同场景下给出了对移植得到的 SYCL 的 GEMM(general matrix multiplication)进行优化的方法. 评测了 paraTRANS 优化后基于 SYCL 的 GEMM 算子在 NVIDIA RTX 3090 和 AMD MI100 上的性能情况. 结果显示,在 NVIDIA RTX 3090 上,paraTRANS 达到了 96.95% CUDA 原生算子的性能水平;在 AMD MI100 上,则接近 CUDA 在 NVIDIA RTX 3090 上硬件峰值百分比(100.47%)所表现出来的性能水平. 这些结果表明成功地将原生高性能 CUDA 算子代码移植并进一步优化至 SYCL 环境中,并为未来类似工作提供新颖且有效的优化思路.

关键词 SYCL;跨异构体系结构;代码移植;GEMM;并行优化

中图法分类号 TP312

DOI: 10.7544/issn1000-1239.202330869 **CSTR:** 32373.14.issn1000-1239.202330869

近年来,在高性能计算领域,加速器的异构性正在成为显著的发展趋势. 各类 CPU, GPU, FPGA, TPU 都成为高性能计算领域可用的选择[1-2]. 伴随着硬件加速器类型的多样化,编程模型也存在着多样化的趋势.

OpenCL(open computing language)是一个有望于实现跨异构平台的编程模型. OpenCL 的存在促进了使用 CPU, GPU 以及其他加速器进行并行编程的发展,但其易用性成为了它再进一步的阻碍. 相比于 CUDA(compute unified device architecture)编程,使用 OpenCL 进行编程更加的繁琐易错,开发效率不如使用 CUDA. 并且 OpenCL 对于 NVIDIA GPU,不能够像 CUDA 一样发挥硬件的性能,使用 OpenCL 也意味着会带来一定的性能损失[3].

SYCL(system-wide compute language)的设计初衷可以看作是为 OpenCL 提供了更高层次的抽象. SYCL 覆盖了 OpenCL 现有的许多特性,并且因为引入更新的 C++ 特性,为开发过程降低了难度和复杂度. 在美国能源部 E 级计算机研制计划 ECP(exascale computing project)中,SYCL 被认为是实现可扩展、高性能计算的重要技术之一,被广泛使用于 ECP 的各个项目中. 如 PROTEAS-TUNE-SYCL, STRUMPACK-SuperLU, SUNDIALS, hypre, ALExa 项目的 Tasmanian 库和 Sake 的子项目 Kokkos Kernels 等,SYCL 在其中都发挥着作用[4]. 目前有多种实现支持 SYCL 标准,如 Intel DPC++, hipSYCL 等. 其中 Intel DPC++ 是目前对 SYCL 支持良好的编译器. 它支持了最新的 SYCL 2020 标准,并且开放源代码于 GitHub 的 Intel/LLVM 仓库[5]. 本文使用了 Intel DPC++作为 SYCL 的编译器实现来进行研究.

当前使用 CUDA 编写的高性能程序的体量十分庞大,生态相对完备. 在开发 SYCL 代码时,能够将已有的 CUDA 代码移植到 SYCL 提高开发效率、降低开发成本. 为了将 CUDA 程序高效地转换为其他程序,一些工具被专门用作 CUDA 到其他编程模型的移植. 例如,HIPIFY 是用来将 CUDA 源代码自动化移植为 HIP(heterogeneous-compute interface for portability) C++源代码的工具集[6], DPCT(Intel DPC++ compatibility tool)是用来将 CUDA 源代码自动化移植 SYCL 源代码的辅助工具[7].

现有移植工具的存在,提高了将已有的、成熟的 CUDA 代码移植到 SYCL 过程的效率,降低了移植过程中的难度. 但是,本文观察到优化过的程序在移植之后会出现显著的性能下降的问题. 这些因为移植造成的性能问题存在一定的普遍性,文中将其分为 2 类:一类是当 CUDA 程序移植为 SYCL 程序之后,依然运行在原硬件时存在的性能下降问题;另一类是当 CUDA 程序移植为 SYCL 程序之后运行在其他异构硬件上存在的性能下降问题. 本文以一个高度

优化的 CUDA GEMM(general matrix multiplication)为例[8]，详细描述了 CUDA GEMM 移植后性能问题的发现与优化，提出一个面向跨异构编程模型 SYCL 代码移植过程的共性算子优化工具 paraTRANS.

图 1 给出了 paraTRANS 的原理图. 如图 1 所示，paraTRANS 将 DPCT 移植得到的在不同硬件上性能不佳的 SYCL GEMM 做进一步分析、辅助优化，从而得到适应硬件平台的 SYCL GEMM. paraTRANS 主要解决了 DPCT 移植来的 SYCL GEMM 的 2 个关键问题：1)解决了在相同体系结构 GPU 上因为寄存器占用增加造成的计算资源利用率低的问题；2)解决了在不同体系结构 GPU(AMD MI100)上因为 scratch memory 访存开销造成的访存阻塞问题. 该工具通过分析和解决如上 2 个问题，使得进一步优化后的 SYCL GEMM 达到了移植前的性能水平.

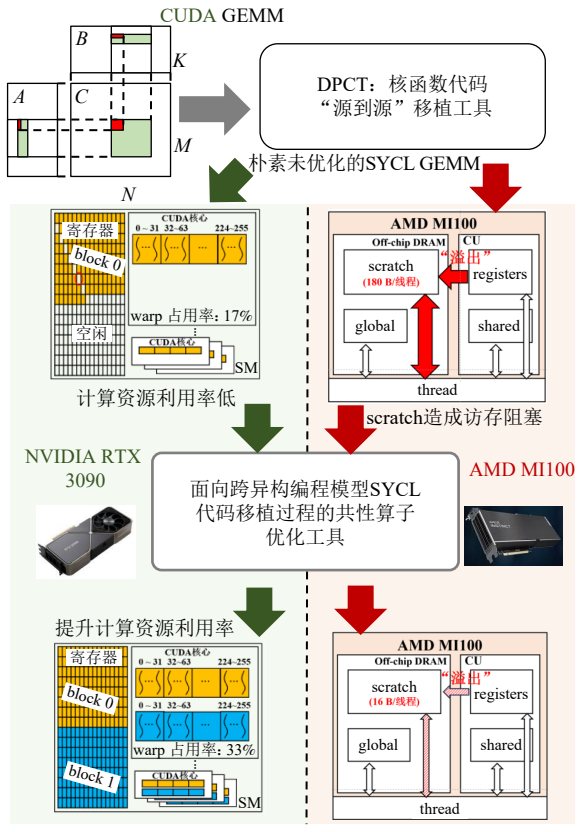


Fig. 1 Principle diagram of paraTRANS

图 1 paraTRANS 原理图

1 背景与研究动机

本节首先介绍了要移植的高度优化的 GEMM 实现，然后分析了 DPCT 移植前后的代码情况，最终总结了用 DPCT 移植后造成的性能下降情况.

1.1 基于 CUDA 编程模型的 GEMM 算子

本文研究高性能的 CUDA 代码在移植前后的性能差异，因此选择一个高度优化的 CUDA GEMM 实现作为移植对象. 该 CUDA GEMM 通过多层的矩阵分块、共享内存、预取、循环展开、解决访存冲突等方法进行了优化. 图 2 给出了 CUDA GEMM 的基本逻辑与访存情况，该实现将会先后把输入矩阵 ***A***, ***B*** 的部分数据从全局内存写入共享内存，然后再从共享内存写入寄存器进行计算. 共享内存和寄存器的写入都使用了预取策略，将本轮计算与下一轮数据读取重叠，提升计算单元利用率.

```
1: Init (sa, sb, Av, Bv);
2: for (k=1; k<K; k+=Kb) {
3:    vload(prcf_Av, A, Mb, Kb, k,⋯) ; // 预取: pref_Av <- A
4:    vload(pref_Bv, B, Kb, Nb, k,⋯) ; // 预取: pref_Bv <- B
5:    for (inner k=0; inner k<Kb; inner k++) {
6:       vload(Av[(inner_k+1)&1], sa, Mt=8, Kt, inner_k+1,⋯); // 取下轮数据
7:       vload(Bv[(inner_k+1)&1], sb, Kt=8, Nt, inner_k+1,⋯); // Av <- sa
8:       Cres += Av[inner_k] × Bv[inner_k];                    // Bv <- sb
9: }                                                           // 做本轮计算
10:   __syncthreads() ;
11:   sa = prcf_Av; // sa <- prcf_Av(已完成预取)
12:   sb = pref_Bv; // sb <- pref_Bv(已完成预取)
13:   __syncthreads() ;
14:   vload(Av[0], sa, Mt=8, Kt, 0,⋯) ;
15:   vload(Bv[0], sb, Kt=8, Nl, 0,⋯) ;
16: }
17: vload(Cv, C, Mt=8, Nt=8) ;
18: Cres = alpha × Cres + bota × Cv;

19: dim3 blockDim(256) ;
20: dim3 gridDim(CEIL_DTV(M, 128) , CEIL_DIV(N, 128)) ;
21: sgcmm<<<gridDim, blockDim>>> (⋯) ;
```

Fig. 2 Native GEMM operator example code based on CUDA programming model

图 2 基于 CUDA 编程模型的原生 GEMM 算子实例代码

1.2 针对 CUDA GEMM 的 DPCT 移植案例分析

Intel oneAPI toolkit 提供了 DPCT 用来辅助进行 CUDA 到 SYCL 的代码移植过程. 通常来说，90%~95%的 CUDA 代码能够通过 DPCT 工具转换为 SYCL 代码. 不能够通过工具进行自动转换的部分，DPCT 会以“WARNING”的形式给出提示，然后由人工进行调整[7].

使用 DPCT 能够实现 CUDA GEMM 到 SYCL GEMM 的自动化移植. 如图 3 所示，移植后的 SYCL GEMM 在编程模型相关的关键字上出现了改变，但是算法并没有发生改变. 移植前后的 CUDA GEMM 与 SYCL GEMM 按照同样的逻辑执行计算.

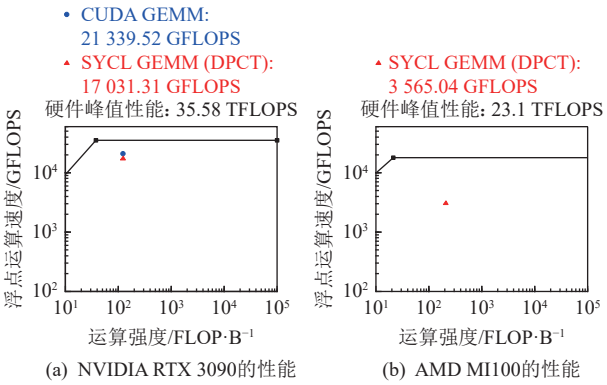
1.3 针对移植造成性能下降的评价

本文对同样逻辑、算法的 CUDA GEMM 与 SYCL GEMM 的性能进行了测试. 首先分析了 CUDA GEMM 与 SYCL GEMM 计算、访存的瓶颈. 如图 4 所示，本文首先选取了大小为 $M=N=K=4\,864$ 下的 GEMM 测

```
1: Init (sa, sb, Av, Bv);
2: for (k=1; k<K; k+=Kb) {
3:     vload(prcf_Av, A, Mb, Kb, k,⋯) ; // 预取: pref_Av ⩽ A
4:     vload(pref_Bv, B, Kb, Nb, k,⋯) ; // 预取: pref_Bv ⩽ B
5:     for (inner k=0; inner k<Kb; inner k++) {
6:         vload(Av[(inner_k+1)&1], sa, Mt=8, Kt, inner_k+1,⋯); // 取下轮数据
7:         vload(Bv[(inner_k+1)&1], sb, Kt=8, Nt, inner_k+1,⋯); // Av ⩽ sa
8:         Cres += Av[inner_k] × Bv[inner_k];                    // Bv ⩽ sb
9: }                                                             // 做本轮计算
10:    item_ct1.barrier() ;
11:    sa = prcf_Av; // sa <- prcf_Av(已完成预取)
12:    sb = pref_Bv; // sb <- pref_Bv(已完成预取)
13:    item_ct1.barrier() ;
14:    vload(Av[0], sa, Mt=8, Kt=1, 0,⋯) ;
15:    vload(Bv[0], sb, Kt=8, Nl=1, 0,⋯) ;
16: }
17: vload(Cv, C, Mt=8, Kt=8) ;
18: Cres = alpha × Cres + bota × Cv;
19: rang<3> blockDim(1, 1, 256) ;
20: rang<3> gridDim(1, CEIL_DTV(N, 128) , CEIL_DIV(M, 128)) ;
21: get_defau1t_queue() . submit([&](handler&cgh)) {
22: cgh.parallel_for(nd_range<3>(gridDim × blockDim, blockDim),
23:                  [=] (nd_item<3> item_ctl) {sgemm(⋯) ; }) ; }) ;
```

Fig. 3 Naive unoptimized SYCL GEMM code transplanted by DPCT

图 3 经 DPCT 移植的朴素未优化的 SYCL GEMM 代码



(a) NVIDIA RTX 3090的性能 (b) AMD MI100的性能

Fig. 4 Roofline model of naive unoptimized GEMM operator transplanted by DPCT

图 4 经 DPCT 移植的朴素未优化 GEMM 算子的 Roofline 模型

试了其 Roofline 情况. 由图 4 可知，该 GEMM 的性能瓶颈为硬件计算能力. 在图 4(a)中，用 DPCT 移植得到的 SYCL GEMM 表现与 CUDA GEMM 相似，但出现了性能下降，存在进一步优化的空间. 而图 4(b)中的 SYCL GEMM 则远离硬件计算能力的上限，有显著的优化空间.

本文进一步测量了移植带来的性能下降情况. 图 5 给出了 DPCT 移植的 SYCL GEMM 在 NVIDIA RTX 3090 与 AMD MI100 上峰值性能百分比的下降. 在 NVIDIA RTX 3090 上进行测试时，发现移植后的 SYCL GEMM 的硬件峰值性能百分比是移植前 CUDA GEMM 的 78.7%；而在 AMD MI100 上进行测试时，发现移植后的 SYCL GEMM 能够达到的硬件峰值性能百分比是移植前 CUDA GEMM 的 25.4%. 这说明代码移植后，同样的算法与逻辑以及编程模型与硬件的改变带来了明显的性能问题，高性能的代码从 CUDA 到 SYCL 不能保证性能的可移植性.

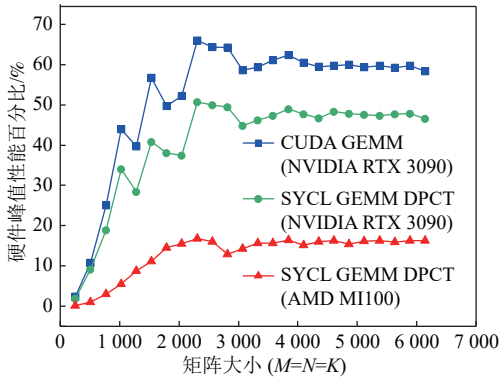


Fig. 5 Degradation of peak performance percentage of SYCL GEMM transplanted by DPCT on NVIDIA RTX 3090 and AMD MI100

图 5 DPCT 移植的 SYCL GEMM 在 NVIDIA RTX 3090 和 AMD MI100 上峰值性能百分比的下降

2 整体系统设计

当移植得到的程序运行在 NVIDIA 与 AMD 的 GPU 上时，关于如何取得最佳性能存在不同的挑战. 为了解决移植后不同硬件的性能下降问题，本文提出了面向跨异构编程模型 SYCL 代码移植过程的共性算子优化工具 paraTRANS. 其实现思路如图 6 所示，

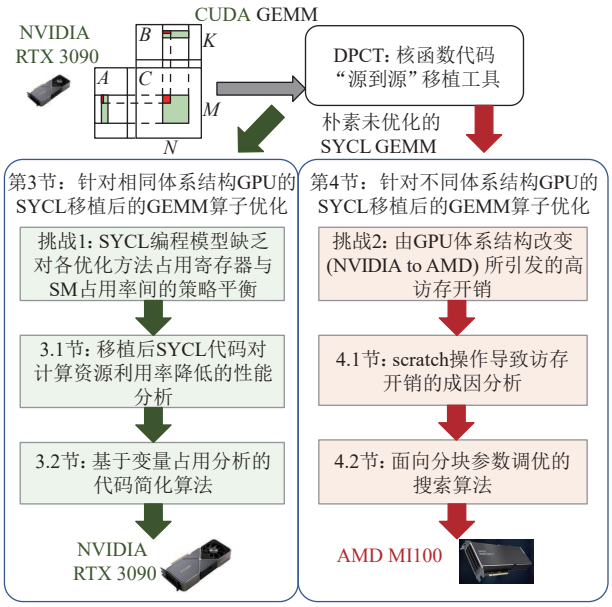


Fig. 6 Research content overview of paraTRANS

图 6 paraTRANS 研究内容总览

该工具分别针对移植后 SYCL GEMM 运行在相同与不同体系结构 GPU 下的性能瓶颈定位与进一步优化.

第 3 节针对相同体系结构 GPU 的 SYCL 移植后的 GEMM 算子进行了优化. 第 3 节定位了 SYCL 编程模型缺乏对各种优化方法的寄存器占用与程序的 SM (streaming multiprocessors)占用率之间的策略平衡. 首先分析了性能瓶颈, 然后采取基于变量占用分析的代码化简算法, 减少少量寄存器以恢复 SM 占用率, 最终在 NVIDIA RTX 3090 上取得接近移植前的 CUDA GEMM 水平.

第 4 节针对不同体系结构 GPU 的 SYCL 移植后 GEMM 算子进行了优化, 并定位了 SYCL GEMM 在 GPU 体系结构改变时所引发的高额的访存开销问题. 该节分析了访存开销由 scratch memory 的大量访问造成, 然后通过面向分块参数调优的搜索算法, 缩小了线程分块, 最终在 AMD MI100 上取得移植前 CUDA GEMM 的水平.

3 针对相同体系结构 GPU 的 SYCL 移植后的 GEMM 算子优化

本节介绍了在相同体系结构 GPU 下 DPCT 产生代码的性能瓶颈, 以及 paraTRANS 如何对 SYCL GEMM 进行优化, 最终达到接近移植前的水平.

3.1 移植后 SYCL 代码对计算资源利用率降低的性能分析

实验测试发现, 在同体系结构 GPU 上, 移植后的 SYCL GEMM 若要获取与移植前同样水平的性能, 存在如下挑战: SYCL 编程模型缺乏对优化算法的寄存器占用数与 warp 占用率的策略平衡. 移植后的 SYCL GEMM 会因为相比于 CUDA 的少量寄存器的额外开销, 而造成 SYCL GEMM 的 warp 占用率的显著下降. warp 占用率下降将会影响程序的 SM 利用率, 从而降低计算资源利用率.

从图 4(a)和图 5 可知, 在 NVIDIA RTX 3090 上, DPCT 移植的 SYCL GEMM 浮点运算速度相比 CUDA GEMM 出现下降. 本文测量了一组输入大小不同的 GEMM 移植前后的性能情况, 移植后的浮点运算速度约是移植前的 78.7%. 可见直接移植得到的 SYCL GEMM 性能不佳, 存在一定的优化空间.

本节进一步测量了移植前后 GEMM 在 NVIDIA RTX 3090 上的 SM 占用情况. 数据项 SM busy 表示在一个 SM 上实际达到的指令吞吐占理论上能够达到的指令吞吐的百分比, 因此该项数据越高, 在计算任务相同时, SM 利用率越高, 计算效率越高. 图 7 给出了移植前的 CUDA GEMM 与 DPCT 移植得到的 SYCL GEMM 的 SM busy 情况. 从图 7 中可知, SYCL GEMM 能够达到的 SM busy 平均是 CUDA GEMM 的 74.97%, 移植造成了 SM 利用率的下降.

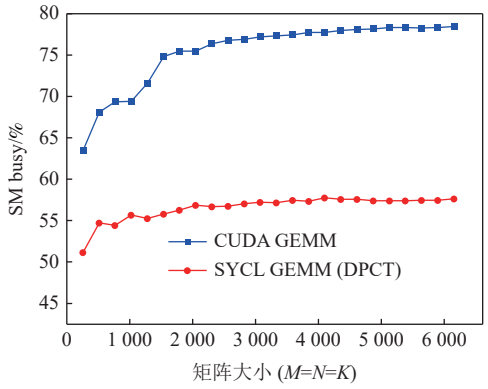


Fig. 7 Difference in SM busy of GEMM on NVIDIA RTX 3090 before and after transplantating from CUDA to SYCL

图 7 从 CUDA 到 SYCL 移植前后 GEMM 在 NVIDIA RTX 3090 上 SM busy 的差异

SM 上能够发射的指令数量受到理论上一个 SM 能够驻存的线程块数量的影响, 该指标通常用 warp 占用率描述. warp 占用率表示一个 SM 上能够驻留的 warp 数量占理论最多能够启动 warp 数量的百分比. 而占用率受到核函数启动配置, 如寄存器、共享内存等的影响, SM 上有限的资源将会限制线程块启动数量. 本节测量了移植前后 GEMM 的启动配置, 计算了各自的 warp 占用率. 实验发现: CUDA GEMM 每个线程占用 128 个寄存器, 理论 warp 占用率达到 33.3%; SYCL GEMM 每个线程占用 129 个寄存器, 理论 warp 占用率只有 16.7%; 在移植后因为寄存器占用增加, warp 占用率降低了一半.

下面以 GEMM 大小为 M=N=K=5 120 时为例, 说明寄存器占用增加情况与造成 warp 占用率降低情况. 在 1.2 节中已经得知 DPCT 移植前后 GEMM 算法不变, 但是编程模型对应的关键字改变. 在图 8 中能够看到 SYCL 与 CUDA 代码的后续编译流也存在差异. NVCC(nvidia CUDA compiler)中的 CICC 部件负责处理 CUDA 核函数代码, 而 DPC++中的不同的 LLVM(low level virtual machine) Pass 负责处理 SYCL 核函数代码. 不同关键字对应的不同函数调用与后续不同的编译流程使得相同算法下的 SYCL GEMM 与 CUDA GEMM 生成了不同的 PTX 指令, 从而导致在 NVIDIA GPU 上最终执行需要的寄存器数目不同.

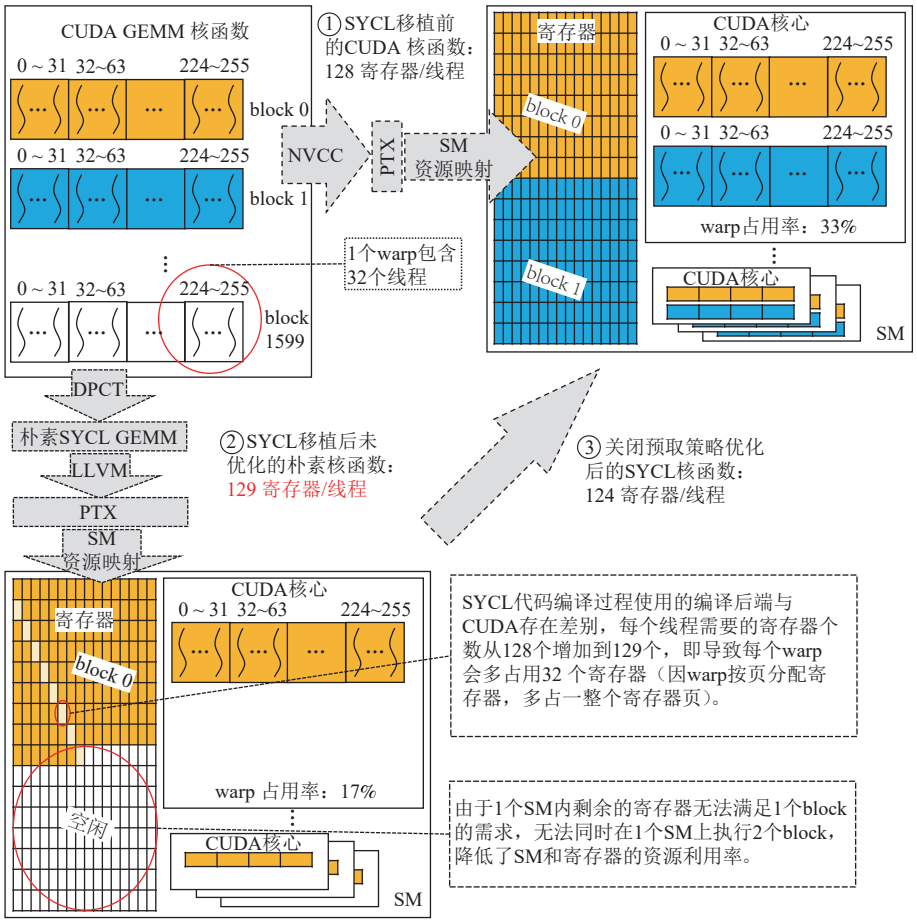


Fig. 8 Register allocation differences and resource idle due to SYCL and CUDA compilation differences

图 8 SYCL 与 CUDA 的编译差异导致寄存器分配差异与资源空闲

因此在图 8 中的①与②尽管源自相同的算法实现，但是最终的寄存器占用数不同. NVCC 生成了占用寄存器数目更少的最终指令.

寄存器是所有驻留在 SM 的线程共享的有限资源. 当一个线程块驻留在 SM 上，需要为线程块中的每个线程分配寄存器，因为寄存器有限，所以某一时刻能够驻留在 SM 上的线程块数量会受到限制. 在图 8 中，移植得到的 SYCL GEMM 的每个线程多占用 1 个寄存器，在 1 个 SM 上为 1 个 block 分配资源时将会多占用 8 个寄存器页. 此时 1 个 SM 上的寄存器资源允许同时启动的 block 从 2 个变为 1 个，活跃的 warp 的数量从 16 个下降到 8 个，所以 warp 占用率下降到原来的一半，程序并行性下降进而导致计算资源利用率下降.

3.2 基于变量占用分析的代码简化算法

3.1 节分析了寄存器占用增加以及 warp 占用率下降，进而带来 SM 利用率下降，导致程序性能下降的问题. 本节针对此问题，给出解决方法. 算法 1 给出在面临寄存器占用增加造成 warp 占用率下降时的解决方案.

算法 1 的行②~⑪给出了算法适用的条件，即满足移植后寄存器数量增加且 warp 占用率下降时，此算法须进行代码的简化. 行⑫*scanVariables* 函数将会扫描整个核函数中的变量开销，记录变量名与大小到 *variableMap* 中. 行⑬将会根据当前的变量开销，选择可能的代码化简方案存储到 *list* 变量中. 行⑭~⑯对于生成代码化简方案将会依次进行测试，并将测试的结果记录到 *res* 变量中. 在完成测试之后，选择最佳的代码简化方案作为最终的简化算法.

算法 1 的关键是选择合适的方式简化 DPCT 移植得到的 SYCL 代码，从而达到减少寄存器占用的目的，进而提升程序 warp 占用率，使得程序能够实现更高的 SM 利用率，以达到更高的性能.

算法 1. 基于变量占用分析的代码简化算法.

输入：CUDA 源程序 *src_cu*，SYCL 源程序 *src_sycl*；

输出：最佳的简化方案 *result*.

① Procedure *codeSimplification*

② *occupancyCuda* ← *testOccupancy*(*src_cu*);

③ *occupancySycl* ← *testOccupancy*(*src_sycl*);
④ if *occupancyCuda* ≤ *occupancySycl* then
⑤ break;
⑥ end if
⑦ *regCuda* ← *getRegNum*(*src_cu*);
⑧ *regSycl* ← *getRegNum*(*src_sycl*);
⑨ if *regCuda* ≥ *regSycl* then
⑩ break;
⑪ end if
⑫ *variableMap* ← *scanVariables*(*src_sycl*);
⑬ *list* ← *simplificationStrategy*(*variableMap*);
⑭ for *strategy* in *list* do
⑮ *res.append*(*strategy.test*());
⑯ end for
⑰ *result* ← *res.findBest*();
⑱ end procedure

3.3 预取策略与资源利用率的权衡案例分析

3.2 节给出了分析变量占用来简化代码以降低寄存器占用的算法，本节给出一个可行的通过放弃预取操作简化代码的案例. 首先，根据算法 1 首先判断寄存器与 warp 占用率是否满足适用条件. 图 9 给出了移植前后 CUDA, SYCL 的代码寄存器占用与 warp 占用率，可见满足适用条件：在移植之后，寄存器从 128 提升到 129, warp 占用率从 33.3% 下降到 16.7%. 然后算法 1 会扫描分析 DPCT 移植后 SYCL 程序的变量占用情况.

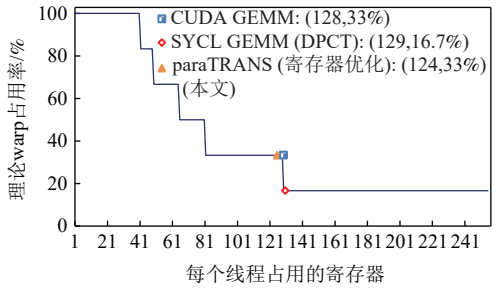


Fig. 9 warp occupancy varies with register usage
图 9 warp 占用率随寄存器占用的变化

算法最终确定的简化算法是去掉移植后 SYCL GEMM 的预取. 如图 10 可见，在去掉预取之后，原本所需要的 Av_1, Bv_1 将不再需要，即去掉预取之后无需双倍的变量开销从共享内存中预取下一轮计算的数据.

最终图 9 给出去掉预取之后的 SYCL GEMM(图 9 中 paraTRANS 寄存器优化)的寄存器占用回落到每线程 124 个, warp 占用率此时已恢复到移植前的 CUDA

```
1: for (k=0; k<K; k+=Kb) {
2:     vload(sa, A, Mb, Kb, k,⋯); //无预取
3:     vload(sb, B, Kb, Nb, k,⋯); //无预取
4:     item_ct1.barrier();
5:     for (inner_k=0; inner_k<Kb; inner_k++) {
6:         vload(Av, sa, Mt=8, Kt, inner_k,⋯); //无预取
7:         vload(Bv, sb, Nt=8, Kt, inner_k,⋯); //无预取
8:         Cres+=Av × Bv;
9:     }
10:    item_ct1.barrier();
11: }
12: vload(Cv, C, Mt=8, Nt=8);
13: Cres=alpha × Cres+ beta × Cv;
14: range<3> blockDim(1, 1, 256);
15: range<3> gridDim(1, CEIL_DIV(N, 128), CEIL_DIV(M, 128));
16: get_default_queue(). submit([&](handler&cgh) {
17: cgh.parallel_for(nd_range <3> (gridDim × blockDim, blockDim),
18:                  [=](nd_item <3> item_ct1) {sgemm(⋯); }); });
```

所需变量减少：
Av0, ~~*Av1*~~, *Bv0*, ~~*Bv1*~~
pref_Av, pref_Bv
Cres, Cv

Fig. 10 SYCL GEMM pseudocode without prefetching
图 10 关闭预取后的 SYCL GEMM 实例代码

GEMM 水平. 需要强调的是，该问题的解决关键是减少寄存器的占用，而非预取对 GEMM 性能有害. 这里选择去掉预取的方式来减少寄存器占用，是因为更高的 warp 占用率带来的性能提升超过预取带来的性能提升.

4 针对不同体系结构 GPU 的 SYCL 移植后 GEMM 算子优化

本节介绍 DPCT 移植得到的 SYCL GEMM 在体系结构不同的 AMD MI100 上的性能瓶颈，以及 paraTRANS 如何减小 scratch memory 的访存开销，提升移植后 SYCL GEMM 的性能水平.

4.1 scratch 操作导致访存开销的成因分析

我们发现，在不同体系结构 GPU 上(如 AMD GPU)，移植后的 SYCL GEMM 若要获取与移植前同样水平的性能，存在如下挑战：SYCL 代码移植过程中，由 GPU 体系结构改变(由 NVIDIA 到 AMD GPU)所引发的高访存开销. SYCL GEMM 运行在 AMD MI100 上时，源自 CUDA GEMM 的分块策略会造成严重的 scratch memory 开销. 这是因为 DPC++在 AMD MI100 上产生了 warp 占用率更高的代码，但寄存器资源不足，从而占用额外的慢速的 scratch memory. 因此体系结构改变之后，原先一个线程的计算负载将不适用新的硬件.

在将 GEMM 移植到 AMD MI100 上之后，因为运行硬件性能不同，无法直接比较移植前后的浮点运算速度以说明运行前后的性能损失. 所以通过式(1)计算 GEMM 能够达到的硬件峰值性能百分比来衡量

移植后的核函数对于硬件的利用率情况.

$$Ratio = \frac{\text{程序达到的FP32性能}}{\text{硬件的FP32峰值性能}}. \quad (1)$$

CUDA GEMM 与 SYCL GEMM 分别在 NVIDIA Geforce RTX 3090 和 AMD Instinct MI100 上进行测试. 移植后程序的最大峰值性能从 64.4% 下降到 16.37%, 说明高度优化的 CUDA GEMM 移植到 SYCL GEMM 之后运行在其他硬件上的运行性能不佳, 存在显著的优化空间.

对于移植之后运行在 AMD MI100 上的 SYCL GEMM, 本节通过 ROC Profiler 来做进一步分析. 首先观察访存计算的时间占比情况. 如图 11 所示, 移植后的 SYCL GEMM 访存单元时间占比总是最长, 而 VALU 时间占比很短. 因访存而阻塞的时间平均值达到 21.32%, 在大多数矩阵大小下均长于计算时间.

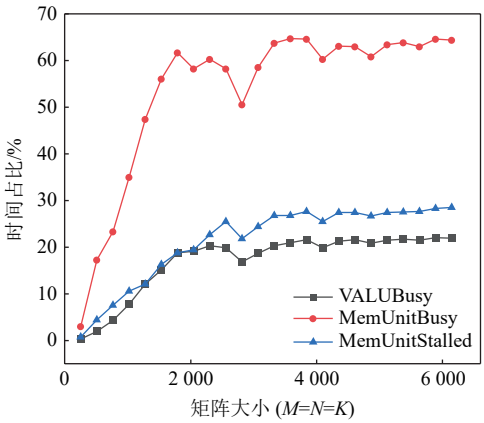


Fig. 11 Time proportion of memory access calculation of unoptimized SYCL GEMM operator on AMD MI100 after transplantation

图 11 AMD MI100 上移植后未优化的 SYCL GEMM 算子的访存计算的时间占比

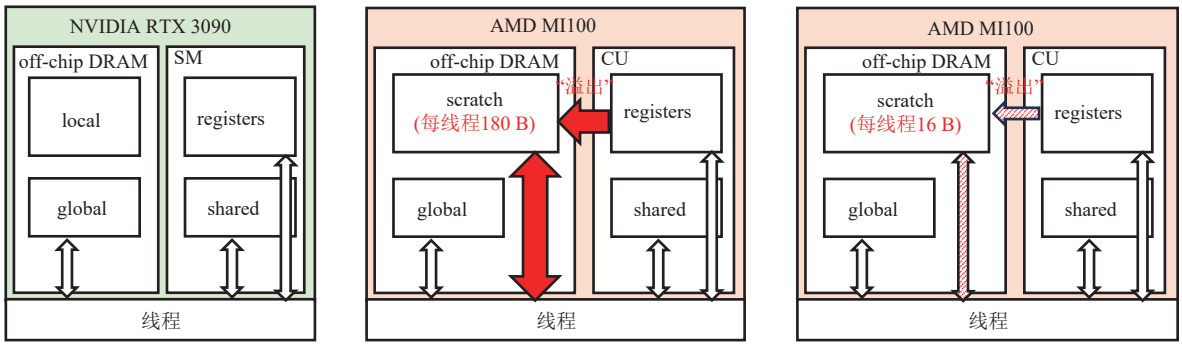
因为 CUDA GEMM 通过使用共享内存, 每个线程块首先读矩阵 ***A***, ***B*** 到快速共享内存, 后续对 ***A***, ***B*** 的频繁访问在共享内存中进行. 极大降低了访问 ***A***, ***B*** 的开销. 但是移植后却出现了高昂的访存开销, 因此考虑移植后核函数是否有其他反常的访存现象. 本节通过 ROC Profiler 对 SYCL GEMM 在 AMD MI100 上进行了测试, 发现移植后 SYCL GEMM 出现了大量的 scratch memory 的占用.

下面以在 MI100 上的 CUDA GEMM 与在 3090 上的 SYCL GEMM 为例, 介绍 scratch memory 开销如何造成了严重的访存延迟.

如图 12 所示, scratch memory 和 global memory 均位于 AMD GPU 的 HBM2 上. 对于访问同一规模数据, 访问片下 HBM2(high bandwidth memory 2)的延迟远高于访问片上的寄存器速度.

在移植前的 NVIDIA GPU 上, 一个线程需要 32 b 寄存器 128 个(512 B), CUDA GEMM 即需要 100% 的寄存器文件访问. 而在移植后的 AMD GPU 上, 一个线程需要 32 b 向量寄存器 64 个(256 B)、一个 scratch memory(180 B). 在不考虑其他因素时, SYCL GEMM 将会进行 58.71% 的寄存器文件访问和 41.28% 的 scratch memory 访问. 由此可见, 移植的 SYCL GEMM 在 AMD GPU 上增加了大量慢速的 scratch 访问, 进而造成了严重的访存阻塞.

造成该问题的根本原因是硬件体系结构改变之后, DPC++倾向在 AMD 上启动 warp 占用率更高的核函数, 需要启动更多线程. 移植后 SYCL GEMM 的 warp 占用率在 AMD GPU 上达到 40%, 而移植前的 CUDA GEMM 只有 33.3%. 更高的 warp 占用率, 需要更多的寄存器资源. 而在 AMD Instinct MI100 与 NVIDA RTX 3090 上 FP32 单元能够访问的寄存器文件均为 65 536 个 32 b 寄存器. 此时寄存器资源不变、单个线



(a) 移植前CUDA GEMM不需要访问local memory

(b) 直接移植的SYCL GEMM在AMD MI100运行时每个线程需要额外访问180 B的scratch memory

(c) 优化后的SYCL GEMM在AMD MI100运行时，scratch memory访问降低到16 B

Fig. 12 Memory access bottleneck caused by the change of GPU's memory access architecture

图 12 GPU 访存体系结构改变所导致的访存瓶颈

程计算负载不变时，寄存器资源出现不足，数据需要写入到 scratch memory 来维持高 warp 占用率. 因此在不改变编译器的情况下，解决此问题需要大幅度减少单个线程寄存器的占用量.

4.2 面向分块参数调优的搜索算法

4.1 节分析了额外的 scratch memory 开销造成访存阻塞，访存时间占比远高于计算时间，最终导致程序性能下降. 本节针对此问题给出解决方法，算法 2 为面临此问题时的算法流程描述.

算法 2 的行②~⑥给出算法适用的条件，即需要满足移植后 CUDA 程序访问 scratch memory 的量高于移植前 CUDA 程序访问 local memory 的量. 算法 2 的行⑦对 DPCT 移植得到的 SYCL 程序进行模板化，该过程将 GEMM 的 3 层分块抽取到模板参数以生成模板文件. 算法 2 行⑧将根据用户的配置生成期望的参数搜索空间，以确定最佳分块的取值范围. 行⑨将结合生成的参数空间与模板文件搜索性能最佳的分块结果.

算法 2 的关键是通过调整分块的大小来调整单个线程需要计算的任务，从而避免单个线程计算过多元素而编译器生成高 warp 占用率程序造成的寄存器资源紧张，需要交换到访问速度更慢的 scratch memory 的问题.

算法 2. 面向分块参数调优的搜索算法.

输入：CUDA 程序 *src_cu*，SYCL 程序 *src_sycl*；

输出：最佳分块策略 *result*.

① Procedure *searchTileSize*

② *localCuda* ← *getLocal*(*src_cuda*);

③ *localSycl* ← *getScratch*(*src_sycl*);

④ if *localCuda* ⩾ *scratchSycl* then

⑤ break;

⑥ end if

⑦ *templateFile* ← *genTemplate*(*dpctSyclFile*);

⑧ *searchsSpace* ← *genSpace*(*configurations*);

⑨ *result*←*searchBestTileSize*(*templateFile*, *searchSpace*);

⑩ end procedure

4.3 通过缩小分块降低 scratch 操作引发的访存开销案例分析

4.2 节给出了针对分块参数进行搜索的算法，本节给出基于对分块参数进行搜索来确定移植后 SYCL GEMM 新分块的研究案例. 首先，根据算法 2 判断移植前后 scratch memory 访问情况. 由图 12 可知，移植前 GEMM 不存在 local memory 的开销，移植后每个线程存在 180 B 的开销，因此适用于此算法. 为了减少 scratch memory 的使用，我们选择了更小的分块参数搜索范围，最终确定的分块情况如表 1 所示.

Table 1 SYCL GEMM Code Optimized by Reducing Thread Tile Size

表 1 减少线程分块大小优化 SYCL GEMM 代码

分块配置	*Mb*	*Nb*	*Kb*	*Mw*	*Nw*	*Mt*	*Nt*
原分块	128	128	8	32	64	8	8
新分块	128	128	16	16	64	4	8

在分块情况改变之后，图 13 给出了缩小分块后代码层面的改变. 由图 13 可知，调整分块之后，SYCL GEMM 算法未改变，单个线程的计算量减半，启动线程数量加倍. 图 14 进一步展示与图 13 对应的更改分块前后的变量开销情况. 当不考虑后续编译优化时，理论上，计算该分块需要从共享内存中的矩阵 ***A*** 块读取 *Mt*×1 的子块到寄存器，从共享内存中的矩阵 ***B*** 块读取 *Nt*×1 的子块到寄存器. 读取矩阵 ***C*** 对应的 *Mt*×*Nt* 的子块到寄存器，计算的结果需要 *Mt*×*Nt* 的寄存器存储. 最终需要的寄存器数量为 *Mt*+*Nt*+2×*Mt*×*Nt*. 代入前后的 *Mt* 与 *Nt* 大小，可得到理论上优化后节省了每线程 272 B. 因此，优化后因采用更小的线程分块，scratch memory 的占用降低到了每线程 16 B，并且访存阻塞时间下降显著，从平均 21.32% 下降到了平均 0.39% .

算法 2 自动优化移植后的分块策略，降低了对硬件寄存器的使用. 因为优化后分块变小，所以变量占用与寄存器开销降低. 最终，scratch memory 开销降低，解决了访存阻塞的问题.

```
1: Init(sa, sb, Av, Bv);
2: for (k=1; k<K; k+=Kb) {
3:   vload(pref_Av, A, Mb, Kb, k,⋯) ;
4:   vload(pref_Bv, B, Kb, Nb, k,⋯) ;
5:   for (inner_k=0; inner_k<Kb; inner_k++) {
6:     vload(Av[(inner_k+1)&1], sa, Mt=4, Kt, inner_k+1,⋯);
7:     vload(Bv[(inner_k+1)&1], sb, Nt=8, Kt, inner_k+1,⋯);
8:     Cres += Av[inner_k] × Bv[inner_k];
9:   }
10:  item_ct1.barrier();
11:  sa = pref_Av;
12:  sb = pref_Bv;
13:  item_ct1.barrier();
14:  vload(Av[0], sa, Mt=4, Kt, 0,⋯);
15:  vload(Bv[0], sb, Nt=8, Kt, 0,⋯);
16: }
17: vload(Cv, C, Mt=4, Nt=8);
18: Cres = alpha × Cres + beta × Cv;
19: range<3> blockDim(1, 1, 512);
20: range<3> gridDim(1, CEIL_DIV(N, 128), CEIL_DIV(M, 128));
21: get_default_queue().submit([&](handler&cgh) {
22: cgh.parallel_for(nd_range<3>(gridDim × blockDim, blockDim),
23:                  (nd_item<3> item_ct1) {sgemm(⋯);});});
```

Fig. 13 SYCL GEMM code optimized by reducing thread tile size

图 13 减少线程分块大小优化 SYCL GEMM 代码

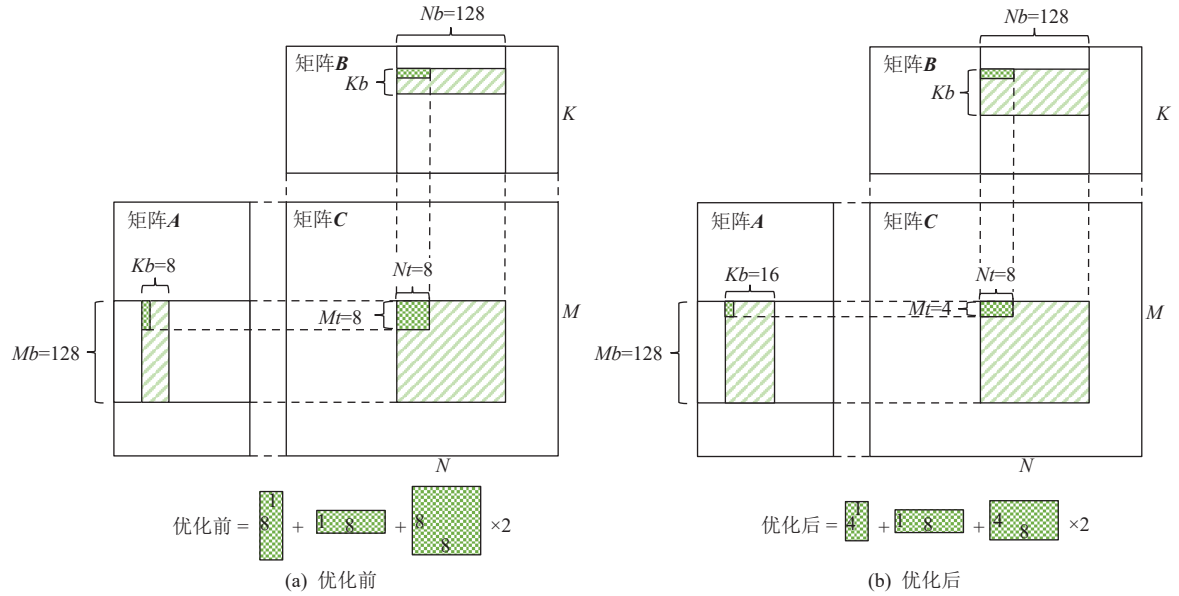


Fig. 14 Changes in tile size for *Mt* and *Nt* after optimization

图 14 经过优化后的 *Mt*，*Nt* 分块尺寸变化

5 实验结果与分析

本节评测了 paraTRANS 优化后的 SYCL GEMM 性能. 主要对 4 个 GEMM 对象进行评价，分别是：1)CUDA GEMM，即被移植的 CUDA GEMM；2) SYCL GEMM(DPCT)，即 DPCT 移植得到的 SYCL GEMM；3)paraTRANS(寄存器优化)，即 paraTRANS 在相同体系结构 GPU(NVIDIA RTX 3090)下对 SYCL GEMM 进一步优化得到的一种 SYCL GEMM 实现；4)paraTRANS(分块参数调优)，即 paraTRANS 在不同体系结构 GPU(AMD MI100)下对 SYCL GEMM 进一步优化得到的一种 SYCL GEMM 实现.

实验过程中的硬件软件环境如表 2 所示.

Table 2 Experimental Environment

表 2 实验环境

软硬件	版本/型号
Ubuntu	20.04.5 LTS
Python	3.8
CUDA	11.7
ROCm	5.4.3
Intel/LLVM	sycl 分支（commit id：ec064e）
DPCT	2023.0.0
TVM	sycl 分支（commit id：6842ca）
CPU	Intel® Xeon® Silver 4110 CPU @ 2.10 GHz
NV GPU	NVIDIA Geforce RTX 3090
AMD GPU	AMD Instinct MI100

5.1 针对基于 SYCL 的 GEMM 算子性能测试

本节分别从不同大小的 GEMM 移植和不同实现的 GEMM 移植 2 个角度来说明 GEMM 移植前后的性能问题，并且给出了 paraTRANS 的性能提升情况.

5.1.1 CUDA 到 SYCL 移植后对不同矩阵大小的测试

本实验测试了移植前、移植后以及针对移植得到的程序进一步优化后的 GEMM 在不同矩阵输入大小下的性能情况. 实验选择的 GEMM 大小参考了百度的 DeepBench[9] 基准程序，然后为矩阵分块进行了针对输入大小的手动补齐.

本实验旨在直观给出不同输入下，移植后存在的性能下降情况以及优化后的性能提升情况. 移植后的 GEMM 分别在 NVIDIA 与 AMD 的显卡上进行了测试.

如图 15 所示，在 NVIDIA 的 GPU 测试有 2 个现象：1)移植后的 GEMM 普遍出现性能下降；2)通过放弃预取的方式减少寄存器开销和换取 warp 占用率的方式在 54.67% 的输入样例中能够发挥作用，而在剩余的输入大小时不能发挥作用. 图 16 给出了 AMD 的 GPU 测试情况，表明优化后的程序相比于移植得到的程序的性能提升. 在 AMD Instinct MI100 上的加速效果平均能够达到 4.49 倍.

当移植后的程序运行在 NVIDIA RTX 3090 上时，因为移植后的程序寄存器的占用增加，导致 warp 占用率下降，程序性能下降. 而通过简化算法放弃预取，减少寄存器的数量的方法来提高程序 warp 占用率的方法能够在半数以上的矩阵大小下发挥作用. 当移植的程序运行在 AMD MI100 上时，移植后的程序在

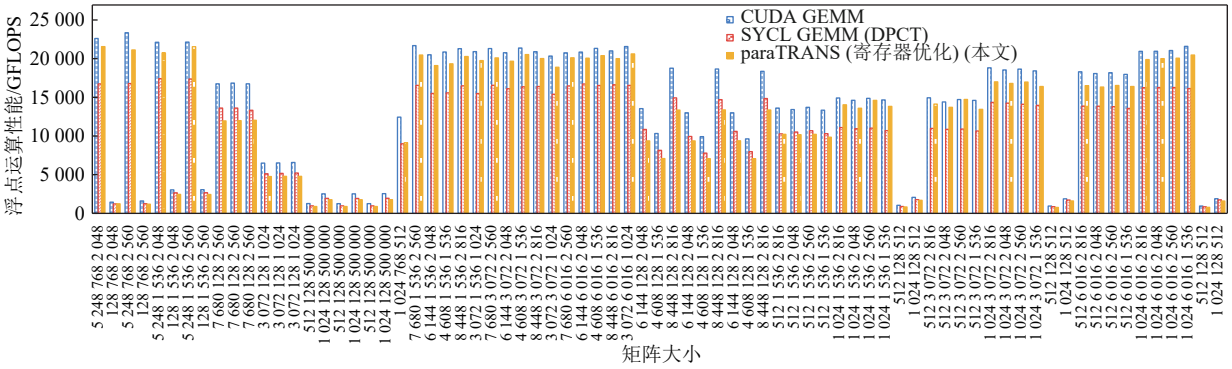


Fig. 15 Performance of NVIDIA RTX 3090-transplanted and paraTRANS-optimized GEMM with different input sizes

图 15 在 NVIDIA RTX 3090 上移植的与 paraTRANS 优化的 GEMM 在不同输入大小下的性能

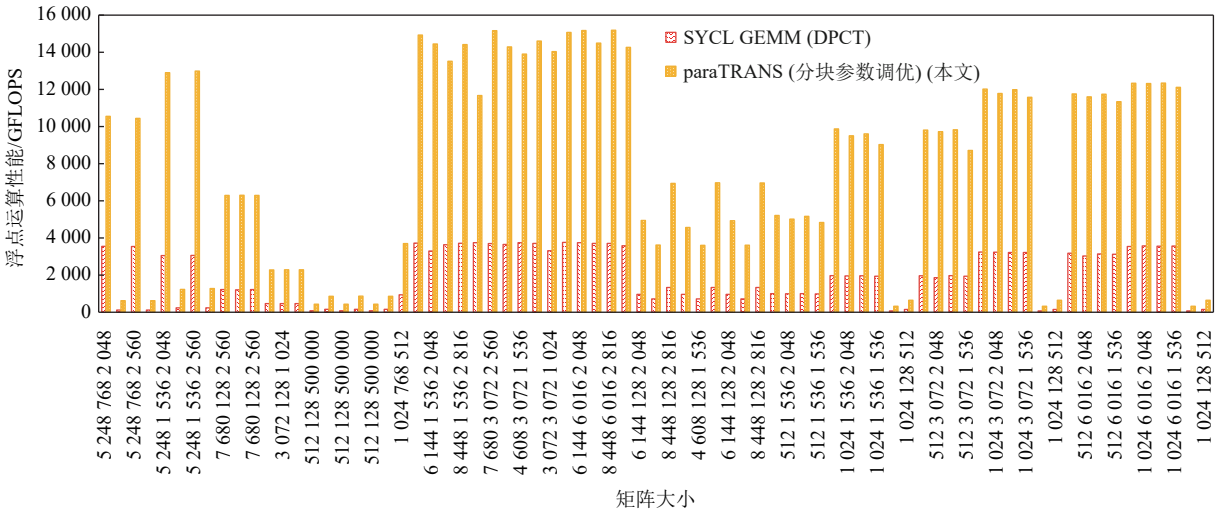


Fig. 16 Performance of AMD Instinct MI100-transplanted and paraTRANS-optimized SYCL GEMM with different input sizes

图 16 在 AMD Instinct MI100 上移植的与 paraTRANS 优化的 SYCL GEMM 在不同输入大小下的性能

AMD MI100 上运行时会带来大量的 scratch memory 的访问，导致访存延迟时间占比高，程序性能下降. 而通过减小矩阵分块大小，单个线程的计算量减少，能够减少 scratch memory 的访问，所以优化后的 SYCL 程序性能得到进一步提升.

移植后的程序在不同硬件下、不同的输入矩阵大小下普遍会存在移植后性能下降的问题. 性能下降的问题能够通过一定的算法、代码的改变进行缓解，达到接近移植前的水平.

5.1.2 对基于不同编译 schedule 调优策略的算子测试

本实验旨在验证针对相同矩阵大小的 GEMM 算子，由 TVM 编译器生成的 SYCL 代码通过不同编译 schedule 调优策略后，是否也会出现 4.1 节和 5.1 节中所出现的不良寄存器分配和访存拥塞问题. 同时验证 paraTRANS 对这些问题的优化效果.

本实验基于 NVIDIA RTX 3090 平台，分别利用 TVM 的 CUDA 和 SYCL 两种编译后端，来生成具有相同 GEMM 尺寸（M=N=K=4 096），但具有不同调度实现的核函数代码. 同时通过与 paraTRANS 的寄存器和分块参数调优优化后的 SYCL 代码性能进行比较. 实验对大小为 M=N=K=4 096 的矩阵乘法利用 TVM auto-scheduler 分别对 SYCL 和 CUDA 进行代码调优，且经过 2000 轮调优后达到性能收敛为止. 实验选取了调优过程中的 50 个最好的调度配置，记为图 17~19 的 50 组 GEMM 核函数.

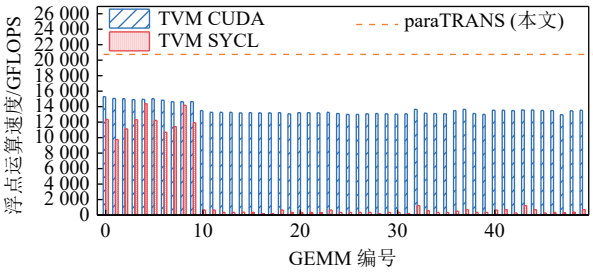


Fig. 17 Overall performance disparities of CUDA and SYCL operators generated by TVM

图 17 TVM 生成的 CUDA 和 SYCL 算子的整体性能差异

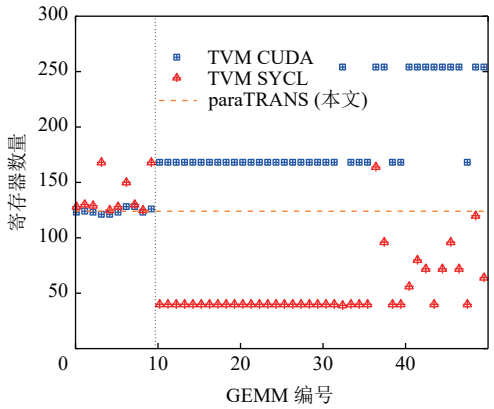


Fig. 18 Utilization disparities in a single thread register between CUDA and SYCL operators generated by TVM

图 18 TVM 生成的 CUDA 和 SYCL 算子的单个线程寄存器占用差异

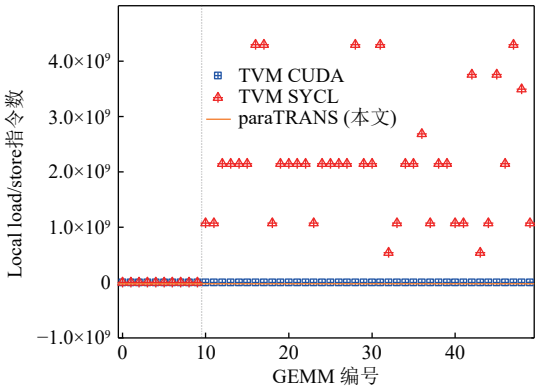


Fig. 19 Disparities in the count of local memory access instructions for CUDA and SYCL operators generated by TVM

图 19 TVM 生成的 CUDA 和 SYCL 算子对 local memory 访问指令数的差异

为进一步追踪基于 TVM 所生成的 SYCL 算子的性能下降以及 paraTRANS 性能提升的原因，本节实验将通过图 18 和图 19 分别对各核函数占用的寄存器数量和 local load/store 指令数进行了进一步测试和分析.

整体实验结果如图 17 所示，TVM 生成的代码中，基于 SYCL 的 GEMM 算子性能要低于基于 CUDA 的 GEMM 算子. 其中，核函数 ID 大于 10 之后的算子实现性能下降尤其明显. 此外，本实验也验证了本文所提出 paraTRANS 系统，能够有效提高 GEMM 算子的计算峰值性能，超过 TVM 通过自动调优所得到的 SYCL 算子性能.

虽然 TVM 所生成的 CUDA 与 SYCL 核函数代码两者在算法、共享内存使用、循环结构与计算元素的顺序等多个方面都是相同的. 但是，由图 17 可知，基于 SYCL 生成的 GEMM 核函数性能要低于基于 CUDA 的算子实现. 在前 10 组中，SYCL 的性能平均是 CUDA 的 80.74%. 在剩余 40 组中，SYCL 的性能平均只有 CUDA 的 3.41%. 总体性能方面，50 组核函数性能均无法超过本文所提出的 paraTRANS 系统对寄存器和分块参数优化后的 GEMM 算子性能.

图 18 和图 19 给出了不同 GEMM 核函数在寄存器和 NVIDIA local memory 访问上的情况. 其中，在访存层次结构上，NVIDIA local memory 与 AMD 的 scratch memory 是同等级别的访存资源. 首先，由图 19 可知，前 10 组 TVM 所生成的 CUDA 代码与 SYCL 代码均不存在 local memory 访问. 但是，根据图 18 可知，前 10 组中 2 种编程模型所对应的算子对寄存器的占用量明显上升. 在后 40 组中，图 18 显示 TVM 基于 SYCL 所产生的核函数代码的寄存器占用量出现了显著的减少. 图 19 显示 SYCL 核函数对 local memory 有了大量的访问操作.

通过对 50 个 TVM 生成的核函数实现的分析可知，在不同的核函数实现中，SYCL 性能均低于 CUDA 性能. 而性能下降的原因总结如下：

1)在前 10 组核函数实验中，SYCL 性能小幅度下降是由寄存器占用量增加引起的，导致 warp 占用率下降和性能下降原理如 3.1 节所述.

2)在后 40 组实验中，SYCL 性能大幅度下降是因为寄存器占用量降低，导致出现更多次对 local memory 访存操后出现访存阻塞情况，性能下降原理如 4.1 节所述.

综上，基于 3.2 节所提出的基于寄存器分配优化和 4.2 节所提出的分块参数调优优化，paraTRANS 系统能够有效解决以上 2 点问题，能够有效控制寄存器数量进而提高 warp 占用率，同时降低 local memory 访存操作次数，缓解访存阻塞问题.

5.2 对不同大小 GEMM 的 roofline 性能测试

本实验直观地给出 DPCT 移植后 SYCL GEMM 的优化潜力以及优化后的 SYCL GEMM 与理论性能上限的距离.

由图 4 可知，GEMM 的理论最大性能受到硬件的计算能力限制. 在本实验中，为了清楚对比图 20 中计算强度的变化，缩放了坐标轴，图 20 中未画出硬件的访存瓶颈线.

在图 20(a)中，DPCT 移植后的 SYCL GEMM 总是在 CUDA GEMM 的下方，距离硬件的计算能力上限更远. 去掉预取的 SYCL GEMM 则在低计算强度

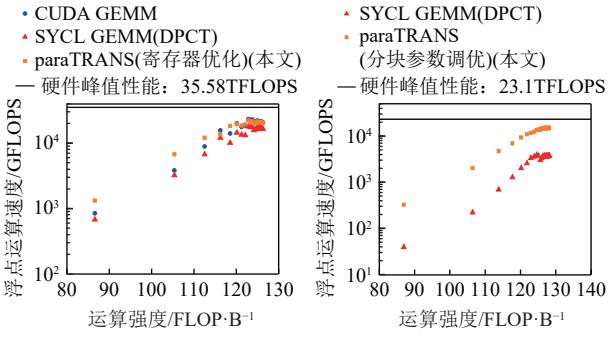


Fig. 20 Optimization effect of Roofline model for different sizes of GEMM

图 20 基于 Roofline 模型对不同大小的 GEMM 的优化效果

下处于 CUDA GEMM 上方，随计算强度不断上升，其浮点运算速度基本与 CUDA GEMM 重合或略微低于 CUDA GEMM.

在图 20(b)中，DPCT 移植得到的 SYCL GEMM 距离硬件计算能力上限很远. 而缩小分块后的 SYCL GEMM 在相同计算强度下取得更高的浮点运算速度，更接近硬件的计算能力上限.

在 NVIDIA RTX 3090 上时，DPCT 移植得到的 SYCL GEMM 性能下降，远离硬件计算能力上限，优化空间更大. 而去掉预取后达到了与移植前相同的优化水平. 在 AMD MI100 上时，DPCT 移植得到的 SYCL GEMM 远离硬件计算能力上限，优化空间大. 缩小分块后，SYCL GEMM 更加逼近硬件的峰值计算能力.

DPCT 移植得到的 SYCL GEMM 在原 NVIDIA RTX 3090 上和其他 AMD MI100 上性能不佳，优化潜力大. 而在各自平台下进行优化后都更加逼近硬件峰值计算能力，达到移植前水平.

5.3 对 SM 利用率与 warp 占用率的测试

本实验以输入矩阵大小满足 $M=N=K$ 时为例，测试 GEMM 移植后运行在 NVIDIA RTX 3090 上因为寄存器占用增加而造成 warp 占用率下降的情况. 具体测试了修改移植后 GEMM 算法后，程序的 warp 占用率的提升以及 SM 利用率的提升. 旨在说明移植后，进一步优化后的 GEMM 的寄存器、warp 占用率、SM 利用率的改变.

图 21 中的 3 条曲线给出了移植前、移植后、对移植的 SYCL GEMM 进一步优化后的情况. 移植后的 SYCL GEMM 性能出现了明显下降，对移植后的 SYCL GEMM 简化算法之后，SYCL GEMM 能够达到移植之前的 96.95%. 表 3 给出了相应的寄存器、warp

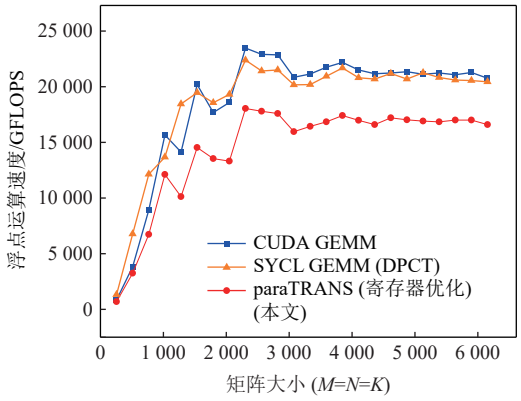


Fig. 21 Performance of transplanted and paraTRANS-optimized GEMM on NVIDIA RTX 3090 for a set of inputs

图 21 移植后与 paraTRANS 优化后的 GEMM 在 NVIDIA RTX 3090 上一组输入下的性能

Table 3 Register Usage and warp Occupancy

表 3 寄存器占用和 warp 占用率

GEMM 核函数类别	寄存器数/线程	warp 占用率 /%
CUDA GEMM	128	33.3
SYCL GEMM（DPCT）	129	16.7
SYCL GEMM（寄存器优化）	124	33.3

占用率的变化情况. 可见移植后的 SYCL GEMM 寄存器数目从 128 提升到了 129，warp 占用率相应从 33.3% 下降到 16.7%. 而简化算法后的 SYCL GEMM 的寄存器从 129 回落到 124，warp 占用率相应从 16.7% 回升到 33.3%. 图 22 则给出了去掉预取后 SM busy 指标从移植后平均 56.58% 提升到 65.68%.

移植后，寄存器数目的增加是 DPC++与 NVCC 的不同实现导致的. 而寄存器影响程序 warp 占用率，进而影响程序的并行执行效率. 所以能够观察到在完成移植之后寄存器数目上升、warp 占用率下降、整体的浮点运算速度下降. 使用更少变量和更加简

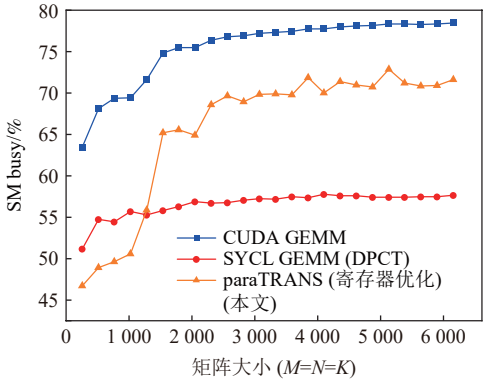


Fig. 22 SM busy test of GEMM on NVIDIA RTX 3090

图 22 GEMM 在 NVIDIA RTX 3090 上 SM busy 测试

单的算法, 会减少寄存器的占用情况. 相应的在对移植后的 SYCL GEMM 去掉预取之后, 寄存器占用下降, warp 占用率回到移植前水平, SM busy 提升, 整体的浮点运算速度也回到接近移植前的水平.

移植后的 SYCL 程序运行在 NVIDIA RTX 3090 上时, 会因为编译器不同导致移植后程序寄存器占用数目不同, 进而影响程序的 warp 占用率与程序的并行度, 最终影响 SM 利用率. 如果此问题是性能瓶颈, 可以考虑简化算法, 减少寄存器占用以换取更高的 warp 占用率与程序的并行度, 提高 SM 利用率.

5.4 对访存、计算时间占比的测试

本实验以输入矩阵大小满足 M=N=K 时为例, 测试 GEMM 移植后运行在 AMD MI100 上存在的显著的访存问题, 以及通过减小分块的方式来提升移植后 SYCL GEMM 的性能.

图 23 比较了移植前、移植后、对移植后 GEMM 更改分块后这 3 组 GEMM 在不同输入大小下能够达到的硬件峰值性能情况. 可以观察到, 移植后的 SYCL GEMM 能够达到的峰值性能下降到 16.76%, 调整分块后的 SYCL GEMM 峰值性能达到 65.11%, 与移植前同一水平. 图 24 给出了优化前后的计算、访存的时间占比情况, 在减小分块之后, 访存阻塞的时间占比下降, 且访存总时间占比下降, 计算时间成为主要部分. 进一步分析程序的 scratch memory 访问量, 在减小分块后访问量从每线程 180 B 下降到每线程 16 B.

移植后运行在 AMD MI100 上的 SYCL GEMM 访问 scratch memory 的开销大, 而移植前的 GEMM 则不

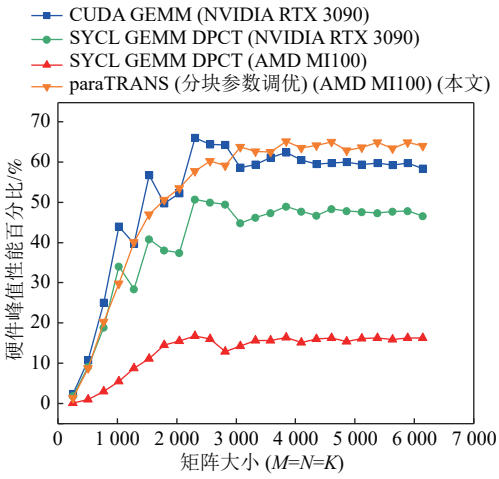


Fig. 23 Percentage of peak performance achieved by GEMM on NVIDIA RTX 3090 and AMD MI100

图 23 GEMM 在 NVIDIA RTX 3090 和 AMD MI100 上达到的峰值性能百分比

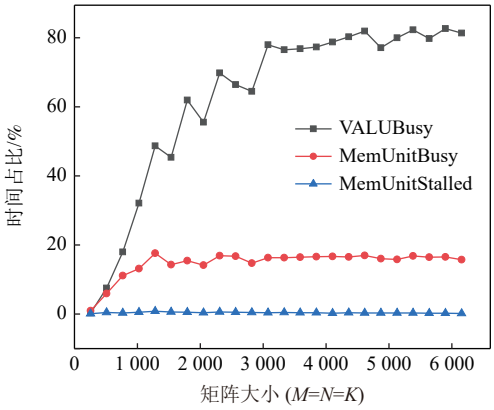


Fig. 24 Time proportion of SYCL GEMM on AMD MI100 in each processing flow after reducing the tiles

图 24 减少分块后 SYCL GEMM 在 AMD MI100 的各处理流程的时间占比

存在此开销. 原因是 DPC++在 AMD MI100 上编译的程序 warp 占用率更高, 但因为 AMD Instinct MI100 的寄存器资源有限, 所以导致额外的 scratch memory 开销. 而 scratch memory 位于 off-chip 的 device memory (HBM2)上, 访问速度慢, 从而造成了严重的访存等待, 移植后的 SYCL GEMM 性能不高. 在减小分块之后, 单个线程量任务减少, 寄存器占用减少, scratch memory 访问大幅降低, 程序性能得到显著提升.

移植后的 SYCL 程序运行在 AMD MI100 上时, 因为编译器的优化策略不同, 引入大量的 scratch memory 访问, 从而影响程序的性能. 若此问题是移植后程序的性能瓶颈, 可以考虑减少单个线程的计算量以及减少寄存器占用, 从而减少 scratch memory 的访问.

6 相关工作

SYCL 相比于 CUDA 来说, 是更新的编程模型, 许多特性依然值得探讨, 性能表现依然需要验证. 最近几年, 伴随着 Intel DPC++ 版本的发布更新, 许多文章分析了 SYCL 的性能表现. 在文献 [10] 中, 研究了 Rodinia benchmark 从 CUDA 移植到 SYCL 的过程. 该文发现 DPCT 能够将 87% 的 CUDA 代码自动翻译. 然后研究了 NVIDIA GPU 和 Intel CPU 的性能情况, 发现访存操作与原生代码相近, 但是存在一些算子出现了指令、访存、硬件利用率等方面的性能问题. 文献 [11] 研究了将 epistasis detection 算法从 CUDA 移植到 SYCL 的过程, 通过移植后的手动调优, 实现了在 NVIDIA GPU 上能够取得与 CUDA 性能相近的表现. 文献 [12] 将一份针对 NVIDIA GPU 编写的

stencil 代码，从 CUDA 移植到了 SYCL，达到了 75% 的性能移植表现. 文献 [13] 研究了分子动力学模拟软件 LAMMPS 在不同的 GPU 平台上的性能移植情况，提供了通过 roofline 模型来研究软件在不同架构下的 GPU 平台下的加速情况. 文献 [14] 将 Bioinformatics 应用从 CUDA 移植到 SYCL 上，从软件的调用接口和 PTX 代码生成的角度分析了移植后的 SYCL 代码，解释了在 NVIDIA GPU 上存在性能问题的原因. 文献 [15–18] 也介绍了 SYCL 应用在不同的 GPU 上的性能情况. 文献 [19–20] 也介绍了 SYCL 代码的可移植性问题，并且在调研过程中发现，编译器的优化欠佳导致移植的程序性能不佳不是孤例. 文献 [21] 发现 GTC-P 程序在移植到 OpenACC 时，会因为编译器带来性能瓶颈，需要额外的优化. 总结来说，伴随 SYCL 的逐渐发展，评测分析 SYCL 的性能情况以及研究 SYCL 的编译器实现是 SYCL 被更多人接受的关键.

以上工作研究了不同的应用从 CUDA 到 SYCL 的移植，分析其在 NVIDIA GPU 上的运行效果. 而本文讨论了 CUDA GEMM 的移植，并且从移植后的程序的寄存器、访存的角度分析了在 NVIDIA GPU 和 AMD GPU 上不同的性能表现. 研究了典型应用在跨编程模型和跨硬件平台上的性能变化情况.

7 总　　结

本文介绍了一个面向跨异构编程模型 SYCL 代码移植过程的共性算子优化工具 paraTRANS 以及将一个高性能的 CUDA GEMM 移植到 SYCL 的过程，并评价了移植后的性能表现. 而 paraTRANS 定位了体系结构在相同与不同的 GPU 上性能下降的原因，进行了针对不同平台的优化.

移植得到的 SYCL GEMM 运行在体系结构相同的 NVIDIA RTX 3090 上时，paraTRANS 分析了性能瓶颈，解决了 SYCL GEMM 各优化方法对寄存器资源的占用与 SM 占用率之间的策略平衡，提出了基于变量占用分析的代码简化算法，选择通过预取换取 warp 占用率的方式，最终在 NVIDIA RTX 3090 上得到接近移植前性能水平的 SYCL GEMM.

移植得到的 SYCL GEMM 运行在体系结构不同的 AMD MI100 上时，paraTRANS 定位了性能瓶颈，解决了 SYCL GEMM 在体系结构改变时（NVIDIA 到 AMD）存在的显著访存阻塞的挑战，并通过面向参数调优的搜索算法，选择通过缩小线程分块的方式，最终在 AMD MI100 上得到达到移植前性能水平的 SYCL GEMM.

而本文的研究对象相对单一，paraTRANS 当前也局限在 GEMM 移植优化的问题上. 在未来的工作中，我们将会尝试对更多的对象做移植，检查它们移植前后的性能情况，提供更加普适的问题解决算法. 并且本文涉及的硬件也只覆盖了 NVIDIA 与 AMD 这 2 款 GPU，而在 Intel 的 GPU、海光 DCU、FPGA 等硬件上的性能如何依然未知. 在未来的工作中会在更多的实验平台上讨论其性能可移植的问题.

作者贡献声明：马兆佳与邵恩共同提出了算法思路实验方案并撰写论文；马兆佳完成实验；狄战元、马立贤提出建议与技术指导并修改论文.

参　考　文　献

[1] Schneider D. The exascale era is upon us: The frontier supercomputer may be the first to reach 1, 000, 000, 000, 000, 000, 000 operations per second[J]. IEEE Spectrum, 2022, 59(1): 34−35

[2] Brodtkorb A R, Dyken C, Hagen T R, et al. State-of-the-art in heterogeneous computing[J]. Scientific Programming, 2010, 18(1): 1−33

[3] Munshi A. The openCL specification[C/OL]//Proc of the 21st IEEE Hot Chips Symp (HCS). Piscataway, NJ: IEEE, 2009[2023-04-19]. http://ieeexplore.ieee.org/document/7478342

[4] Heroux M A, McInnes L C, Thakur R, et al. ECP software technology capability assessment report[R/OL]. 2020[2023-04-23]. https://doi.org/10.2172/1760096

[5] Intel. Intel/LLVM: Intel staging area for llvm.org contribution[CP/OL]. (2023-04-19)[2023-04-19]. https://github.com/intel/llvm

[6] AMD ROCm Software. HIPIFY[CP/OL]. (2023-06-26)[2023-06-27]. https://github. com/ROCm-Developer-Tools/HIPIFY

[7] Intel. Migrate CUDA* to DPC++ code: Intel DPC++ compatibility tool[EB/OL]. [2023-07-07]. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html

[8] Zhai Yujia. How to optimize SGEMM on NVIDIA GPUs[CP/OL]. (2023-04-19)[2023-04-21]. https://github.com/yzhaiustc/Optimizing-SGEMM-on-NVIDIA-Turing-GPUs

[9] Baidu Research. DeepBench[CP/OL]. (2023-06-28)[2023-07-04]. https: //github. com/baidu-research/DeepBench

[10] Castaño G, Faqir-Rhazoui Y, García C, et al. Evaluation of Intel's DPC++ compatibility tool in heterogeneous computing[J]. Journal of Parallel and Distributed Computing, 2022, 165: 120−129

[11] Jin Zheming, Vetter J S. Performance portability study of epistasis detection using SYCL on NVIDIA GPU[C/OL]//Proc of the 13th ACM Int Conf on Bioinformatics, Computational Biology and Health Informatics. New York: ACM, 2022[2023-04-23]. https://dl.acm.org/doi/10.1145/3535508.3545591

[12] Christgau S, Steinke T. Porting a legacy cuda stencil code to oneapi[C]//Proc of the 34th IEEE Int Parallel and Distributed Processing Symp Workshops (IPDPSW). Piscataway, NJ: IEEE, 2020: 359−367

[13] Hagerty N, Vergara V G M, Tharrington A. Studying performance portability of lammps across diverse gpu-based platforms[R/OL]. 2022[2023-04-23]. https://doi.org/10.1002/cpe.7895

[14] Jin Zheming, Vetter J S. Understanding performance portability of Bioinformatics applications in SYCL on an NVIDIA GPU[C]//Proc of the 16th IEEE Int Conf on Bioinformatics and Biomedicine (BIBM). Piscataway, NJ: IEEE, 2022: 2190−2195

[15] Deakin T, McIntosh-Smith S. Evaluating the performance of HPC-style SYCL applications[C/OL]//Proc of the 8th Int Workshop on OpenCL. New York: ACM, 2020[2023-04-19]. https://dl.acm.org/doi/10.1145/3388333.3388643

[16] Jin Zheming, Vetter J S. Evaluating nonuniform reduction in HIP and SYCL on GPUs[C]//Proc of the 8th Int Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD). Piscataway, NJ: IEEE, 2022: 37−43

[17] Volokitin V, Bashinov A, Efimenko E, et al. High performance implementation of Boris particle pusher on DPC++. A first look at oneAPI[C]//Proc of the 14th Int Conf on Parallel Computing Technologies. Berlin: Springer, 2021: 288−300

[18] Costanzo M, Rucci E, Sánchez C G, et al. Assessing opportunities of SYCL and Intel oneAPI for biological sequence alignment[J]. arXiv preprint, arXiv: 2211.10769, 2022

[19] Da Silva H C, Pisani F, Borin E. A comparative study of SYCL, OpenCL, and OpenMP[C]//Proc of the 2016 Int Symp on Computer Architecture and High Performance Computing Workshops (SBAC-PADW). Piscataway, NJ: IEEE, 2016: 61−66

[20] Tsai Y M, Cojean T, Anzt H. Porting sparse linear algebra to Intel GPUs[C]//Proc of the 27th European Conf on Parallel Processing. Berlin: Springer, 2021: 57−68

[21] Wang Yichao, Lin Xinhua, Cai Linjin, et al. Porting and optimizing GTC-P on TaihuLight supercomputer with OpenACC[J]. Journal of Computer Research and Development, 2018, 55(4): 875−884 (in Chinese)
(王一超, 林新华, 蔡林金, 等. 太湖之光上利用 OpenACC 移植和优化 GTC-P[J]. 计算机研究与发展, 2018, 55(4): 875−884)



Ma Zhaojia, born in 2000. Master candidate. His main research interest includes high-performance computing.
马兆佳, 2000 年生. 硕士研究生. 主要研究方向为高性能计算.



Shao En, born in 1988. PhD, senior engineer, master supervisor. Senior member of CCF. His main research interests include computer interconnect network, SYCL, and high-performance computing and systems software.
邵 恩, 1988 年生. 博士, 高级工程师, 硕士生导师. CCF 高级会员. 主要研究方向为计算机互连网络、SYCL、高性能计算和系统软件.



Di Zhanyuan, born in 1999. PhD candidate. His main research interests include parallel computing, compilation optimization, and code generation.
狄战元, 1999 年生. 博士研究生. 主要研究方向为并行计算、编译优化、代码生成.



Ma Lixian, born in 1993. PhD candidate, engineer. His main research interests include high performance computing and machine learning.
马立贤, 1993 年生. 博士研究生, 工程师. 主要研究方向为高性能计算、机器学习.