软件学报 ISSN 1000-9825, CODEN RUXUEW
2024,35(12):5710−5724 [doi: 10.13328/j.cnki.jos.007084] [CSTR: 32375.14.jos.007084]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

SDAA: 面向申威智能加速卡的运行时系统*

赵玉龙[1], 张鲁飞[1], 许国春[2], 李宇轩[3], 孙茹君[1], 刘 鑫[4]

[1](数学工程与先进计算国家重点实验室, 江苏 无锡 214125)

[2](无锡先进技术研究院, 江苏 无锡 214125)

[3](清华大学 计算机科学与技术系, 北京 100084)

[4](国家并行计算机工程技术研究中心, 北京 100083)

通信作者: 刘鑫, E-mail: yyylx@263.net

摘 要: 自主研制的申威智能加速卡上搭载了脉动阵列增强的申威众核处理器, 其智能计算能力与主流 GPU 相当, 但仍缺少配套的基础软件. 为降低申威智能加速卡的使用门槛, 有效支撑人工智能应用开发, 设计面向申威智能加速卡的运行时系统 SDAA, 语义与主流的 CUDA 运行时保持一致. 针对内存管理、数据传输、核函数启动等关键路径, 采用软硬协同的设计方法实现卡上段页结合的多级内存分配算法、可分页内存多线程多通道的传输模型、多异构部件自适应的数据传输算法和基于片上阵列通信的快速核函数启动方法, 使得 SDAA 运行时性能优于主流 GPU. 实验结果表明, SDAA 运行时系统的内存分配速度是 NVIDIA V100 对应接口的 120 倍, 数据传输开销是对应接口的 1/2, 数据传输带宽达到对应接口的 1.7 倍, 核函数启动时间与对应接口相当. SDAA 运行时已支撑主流框架和实际模型训练在申威智能加速卡上的高效运行.

关键词: 运行时系统; 申威智能加速卡; 人工智能; 软件定义

中图法分类号: TP303

中文引用格式: 赵玉龙, 张鲁飞, 许国春, 李宇轩, 孙茹君, 刘鑫. SDAA: 面向申威智能加速卡的运行时系统. 软件学报, 2024, 35(12): 5710–5724. http://www.jos.org.cn/1000-9825/7084.htm

英文引用格式: Zhao YL, Zhang LF, Xu GC, Li YX, Sun RJ, Liu X. SDAA: Runtime System for Shenwei AI Acceleration Card. Ruan Jian Xue Bao/Journal of Software, 2024, 35(12): 5710–5724 (in Chinese). http://www.jos.org.cn/1000-9825/7084.htm

SDAA: Runtime System for Shenwei AI Acceleration Card

ZHAO Yu-Long[1], ZHANG Lu-Fei[1], XU Guo-Chun[2], LI Yu-Xuan[3], SUN Ru-Jun[1], LIU Xin[4]

[1](State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214125, China)

[2](Wuxi Institue of Advanced Technology, Wuxi 214125, China)

[3](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

[4](National Research Center of Parallel Computer Engineering and Technology, Beijing 100083, China)

Abstract: The homegrown Shenwei AI acceleration card is equipped with the Shenwei many-core processor based on systolic array enhancement, and although its intelligent computing power can be comparable to the mainstream GPU, there is still a lack of basic software support. To lower the utilization threshold of the Shenwei AI acceleration card and effectively support the development of AI applications, this study designs a runtime system SDAA for the Shenwei AI acceleration card, whose semantics is consistent with the mainstream CUDA. For key paths such as memory management, data transmission, and kernel function launch, the software and hardware co-design method is adopted to realize the multi-level memory allocation algorithm with segment and paged memory combined on the card, pageable memory transmission model of multiple threads and channels, adaptive data transmission algorithm with multi-heterogeneous components, and fast kernel function launch method based on on-chip array communication. As a result, the runtime performance of

* 基金项目: 国家重点研发计划 (2018ZX01028102)
收稿时间: 2023-03-15; 修改时间: 2023-08-18; 采用时间: 2023-11-08; jos 在线出版时间: 2024-03-27
CNKI 网络首发时间: 2024-03-29

© 中国科学院软件研究所 http://www.jos.org.cn

SDAA is better than that of the mainstream GPU. The experimental results indicate that the memory allocation speed of SDAA is 120 times the corresponding interface of NVIDIA V100, the memory transmission overhead is 1/2 of the corresponding interface, and the data transmission bandwidth is 1.7 times the corresponding interface. Additionally, the launch time of the kernel function is equivalent to the corresponding interface, and thus the SDAA runtime system can support the efficient operation of mainstream frameworks and actual model training on the Shenwei AI acceleration card.

Key words: runtime system; Shenwei AI acceleration card; artificial intelligence; software-defined

自主研制的申威众核处理器 (SW-AI) 采用双模态 PCIe 模块, 配置为端点 (endpoint, EP) 模式时, 可以作为加速卡使用. 申威智能加速卡与 GPU 使用模式相同, 能够充分利用主机侧丰富的人工智能软件生态, 在算子库层屏蔽底层异构硬件的差异, 方便用户编程使用, 拓展应用场景.

运行时系统作为加速卡软件栈的基础和核心, 是连接加速卡和主机的桥梁, 在保证性能的同时提供良好的软件生态. 申威智能加速卡运行时系统在设计上面临多方面的挑战, 一是申威智能加速卡上仅有一个通用主核, 与 GPU 由多个性能较低的专用微控制器 (microcontroller)[1]合作完成资源管理调度不同, 其硬件定制优化能力弱于 GPU, 但是软件定义能力强于 GPU, 对卡上主核系统的设计提出了新的要求; 二是申威智能加速卡支持页式和段式两种内存使用方式, 对内存分配管理方法提出了新的挑战; 三是数据传输带宽已成智能应用训练的瓶颈之一[2], 保证高效的数据传输性能是运行时软件设计的难点之一; 四是模型训练过程中算子调用频繁, 而 SW-AI 计算核心采用精简 RISC 架构, 启动和同步开销大, 无法满足模型需求.

为此, 本文设计实现了面向申威智能加速卡的运行时系统 SDAA (software defined accelerator architecture), 程序语义和 GPU 的运行时系统 CUDA (compute unified device architecture)[3]类似, 并针对申威智能处理器的体系结构特点, 充分利用加速卡内部通用处理器的管理控制能力, 通过软件定义的方法定制软件队列和控制通路, 实现多级的内存管理、高效的数据传输、快速的核函数启动等功能, 运行时性能优于主流 GPU. 本文的贡献如下.

1) 设计了面向申威智能加速卡的编程模型和运行时架构, 基于 SDAA 架构实现了卡上轻量级系统以及主机侧驱动、运行时库, 为申威智能处理器加速卡模式的推广应用提供了基础支撑.

2) 针对卡上段页式内存需求特点, 设计了主机侧锁页内存分配算法和设备侧段页结合的多级内存分配算法, 内存分配速度是 V100 对应接口的 120 倍.

3) 针对主机-加速卡、加速卡-加速卡高效数据传输的要求, 设计实现了多线程多通道的传输模型和多异构部件自适应的传输算法, 数据传输开销是对应接口的 1/2, 数据传输带宽达到对应接口的 1.7 倍.

4) 针对模型训练过程中算子调用频繁的情况, 采用软硬协同思想设计了基于片上阵列通信的快速核函数启动方法, 核函数启动时间与对应接口相当.

本文第 1 节介绍背景及相关工作. 第 2 节提出了 SDAA 编程模型和运行时系统架构. 第 3 节是 SDAA 软硬协同设计优化技术. 第 4 节通过实验评估 SDAA 运行时系统的关键接口性能. 最后给出总结和展望.

1 相关工作

常见的人工智能芯片有英伟达公司的 GPU、寒武纪公司的思元芯片[4]、百度公司的昆仑芯片[5]、阿里巴巴公司的含光芯片[6]、华为公司的昇腾 910 芯片 (Ascend 910)[7], 出于商业考虑, 各大公司对其加速卡内部管理相关软硬件实现均未开源.

英伟达 2006 年推出了灵活便捷的 CUDA 运行时及编程模型, 2014 年英伟达发布 cuDNN 深度神经网络加速库, 并集成到 AI 框架中, 进一步提升了 AI 计算提升性能和易用性等, 使得 AI 开发和研究人员可以更加专注于神经网络结构的设计. 经过不断发展完善, CUDA 生态已在人工智能领域内处于垄断地位. CUDA 生态由算子库、CUDA 运行时、编程模型、编译器、驱动和底层硬件架构构成.

通过对开源驱动 Nouveau[8]的分析, 结合学术界对 CUDA 的功能分析[9–14]以及运行环境、资源管理、数据传输、核函数启动等功能的优化研究[15–23]. 可以得出 GPU 与主机交互流程、CUDA 内存分配原理、GPU 内存传输机制、GPU 核函数启动流程等.

© 中国科学院软件研究所 http://www.jos.org.cn

GPU 与主机交互的流程为: 主机上的驱动程序在主机侧内核空间分配两个缓冲区, 一个命令缓冲区 (command buffer) 和一个环形缓冲区 (ring buffer), 然后将命令缓冲区内存映射到用户空间. 运行时 (CUDA) 将一组命令推送到命令缓冲区, 然后再将这组命令的大小和偏移量更新到通道的环形缓冲区, 然后通过 MMIO 更新一个名为 PUT 的寄存器, 当 PUT 寄存器更新时, 设备上的微控制器从缓冲区中获取命令组, 并更新 GET 寄存器以通知运行时命令已被处理. 总的来说, GPU 设计了两级命令缓冲, 首先是环形缓冲区用于记录命令的大小和偏移, 另一个是真正存放命令的命令缓冲区.

CUDA 内存分配的原理为: GPU 中主机侧和设备侧内存分配 (cudaMallocHost 和 cudaMalloc 等) 采用相同或者类似的算法进行分配, 驱动程序分配内存后要更新更新页目录和页表, 支持的典型页大小为 4 KB 和 128 KB. GPU 的内存分配器使用的是隔离列表 (segregated free lists) 的方式进行内存管理, 单个内存池大小是 2 MB, 这种分配器在分配内存时用时不均衡, 首次内存分配的时间开销明显大于后续的内存分配开销, 差距在 10 倍左右.

GPU 数据传输分为两种形式: 一种是针对可分页 (pageable memory) 内存使用双缓冲方式进行内存传输, CUDA 中的双缓冲技术中分配了两个大小为 1 MB 锁页内存缓冲区, 以实现稳定的 DMA [19]数据传输; 另一种是不使用缓冲区, 用户通过 cudaMallocHost 直接分配锁页内存, 然后进行数据传输. GPU 中填入 DMA 描述符的是虚地址, 底层数据传输使用的是物理地址, 之所以能这么做是因为 GPU 上有一个既能够把设备上的虚地址转为物理地址也能把 CPU 侧的虚地址转为物理地址的页表 (page table) 部件.

GPU 核函数启动的流程为: 主机侧用户调用核函数启动接口, 运行时准备好核函数启动命令信息 (包括启动命令、代码段基地址、参数等), 驱动程序将命令信息放入命令缓冲队列并通过 PCIe 总线传到加速卡上, 卡上的命令处理部件 (command engine) 解析命令信息, 根据解析出来的信息参数来初始化计算引擎, 启动计算核心进行计算, 计算结束后计算核心同步完成信息, 返回给主机侧的运行时. 总的来说, GPU 通过卡上专用硬件处理单元完成核函数的加载启动.

申威众核处理器结构如图 1 所示, 为环网总线的设计模式, 环网总线将管理控制核心 (management processing element, MPE, 也称为主核)、4 个从核组 (core group, CG, 也称为从核阵列) 和双模态 PCIe 部件连接在一起, 每个从核组内包含一个高速存储控制器和 4 行 8 列共 32 个运算核心 (computing processing element, CPE, 也称从核), 每个运算核心由超标量处理核心、核内局存与通信引擎、智能加速核心组成, 智能加速核心采用脉动阵列结构; PCIe 部件上还有一个进行主机-加速卡数据传输的 DMA 部件.

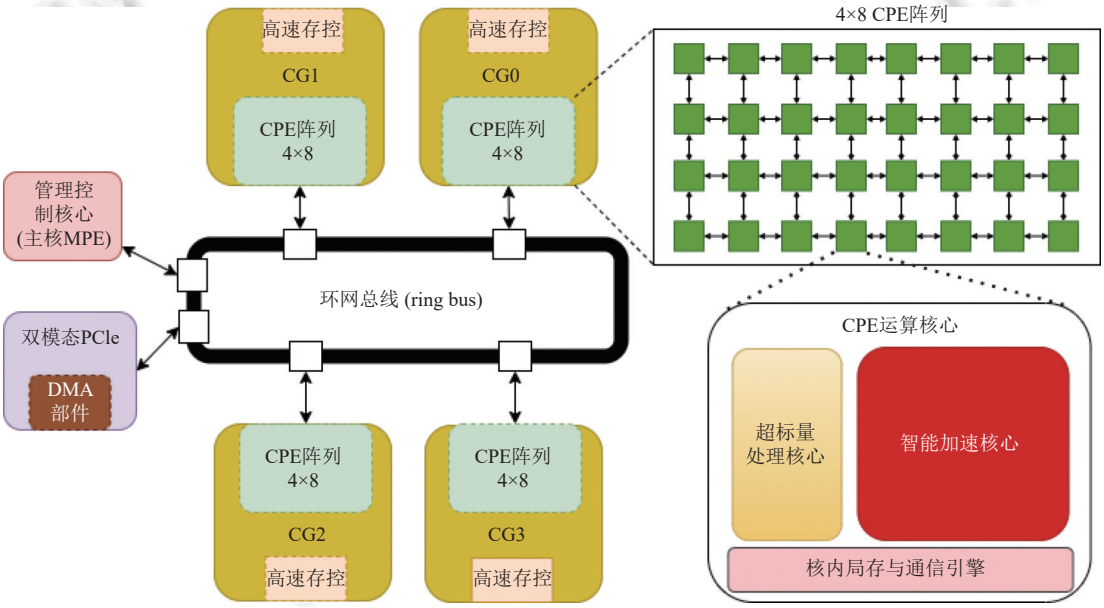


图 1 申威众核处理器基本结构图

© 中国科学院软件研究所 http://www.jos.org.cn

从图 1 可以看到, 与 GPU 卡上搭载多种类型处理能力较弱的微控制器不同, SW-AI 加速卡内部的主核是一个通用 64 位 RISC 处理器, 负责与主机侧通过 PCIe 总线进行通信及加速卡上计算、存储、传输资源的管理等, 需要用软件的方式把主核定义为不同种类的高效部件来完成不同的功能, 支撑上层 AI 应用开发.

2 系统总体架构

为适应主机和加速卡异构模式下新的编程需求, 本文设计了加速卡模式的 SDAA 编程模型及相应的运行时系统架构.

2.1 SDAA 编程模型

申威智能加速卡拥有与主机内存分离的设备内存空间, 本文针对申威智能加速卡的结构特点设计了 SDAA 编程模型, 对加速卡硬件资源进行抽象表示, 为用户提供统一编程接口, 解决跨 PCIe 总线计算、存储、通信等一系列问题. SDAA 编程模型属于显式异构编程模型[24], 采用主机和设备 (host/device) 分工的形式, 主机侧运行 CPU 控制代码, 设备侧运行加速部分代码, 通过__global__和__kernel__等关键字, 来标识设备端代码. 用户通过 SDAA 运行时库接口来使用 SDAA 编程模型, SDAA 运行时库的主要接口有设备管理、内存管理、数据传输、运行控制、流管理、事件管理等类型. 设备管理接口用于获取当前可用的加速卡设备, 并配置当前使用的设备; 内存管理接口用于在主机或设备内存上为用户程序申请和释放所需的内存空间; 设备传输接口实现主机-加速卡内存间以及加速卡-加速卡内存间的数据传输; 运行控制接口用于注册核函数、配置核函数参数、将包含核函数的代码段加载到加速卡设备内存并启动执行; 流管理用于创建或销毁流, 同一个流内的操作是严格串行的, 流之间是可以并发执行的, 用来提高设备利用率, 提升应用性能; 事件管理主要是在流中插入事件, 用来在不同的流之间进行同步等操作.

图 2 展示了 SDAA 编程模型的示例, 中间部分是手写数字识别应用的推理流程, 左侧是 CUDA 版本卷积算子示例, 右侧是 SDAA 版本卷积算子示例. 加速卡模式下 SDAA 程序运行过程可以概括为 4 个主要步骤: ① 使用内存管理接口分配设备内存; ② 数据传输接口将源数据从主机内存复制到设备内存; ③ 运行控制接口配置参数并在设备侧启动运行核函数; ④ 核函数执行完毕后将输出结果从设备复制回主机内存.

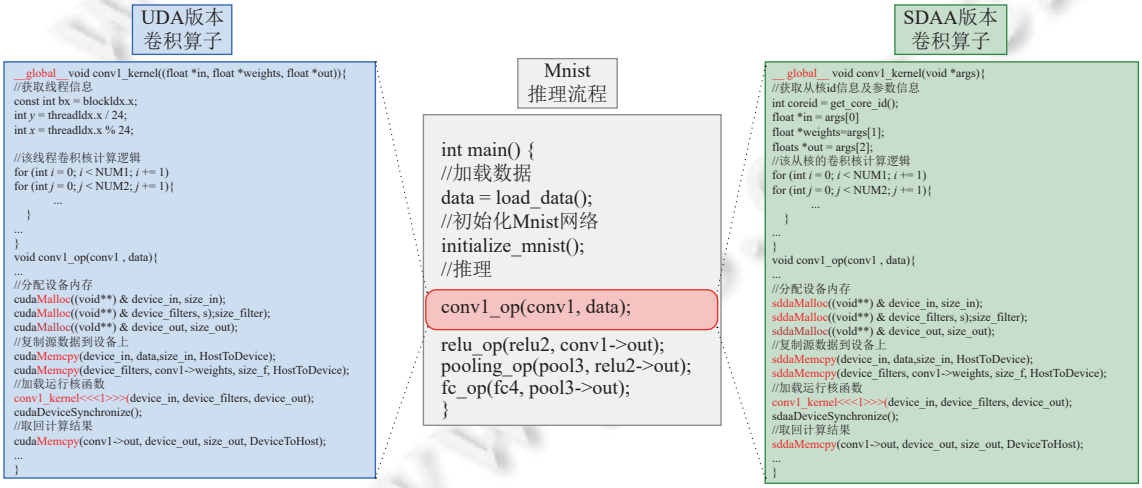


图 2 SDAA 编程模型示例

SDAA 运行时接口名称和语义与对应的 CUDA 运行时类似, 这种方式能屏蔽底层硬件差异, 降低用户学习成本、使用门槛, 便于上层算子库、框架进行自动化迁移, 大大降低迁移成本, 对申威智能加速卡生态完善发展有积极促进作用.

© 中国科学院软件研究所 http://www.jos.org.cn

2.2 运行时系统架构

SDAA 运行时系统架构是为支撑 SDAA 编程模型而设计, 如图 3 所示, 整个系统分为主机侧和加速卡侧: 主机侧有 SDAA 运行时库和设备驱动 AI-Driver, 加速卡侧的主核上运行轻量级系统 AI-Kernel, 从核上有守护程序 Slave_Daemon. 为发挥主核优势, SDAA 系统首先使用软件定义的方式把加速卡上的主核灵活的定义成内存资源管理器、计算资源管理器、PCIe 总线上的 DMA 控制器、主机任务接收分发器 (dispatcher) 等一系列角色, 然后将卡上系统和主机侧驱动运行时紧密耦合, 最后使用定制的软件队列和控制通路实现多级的内存管理、高效的数据传输、快速的核函数启动等功能.

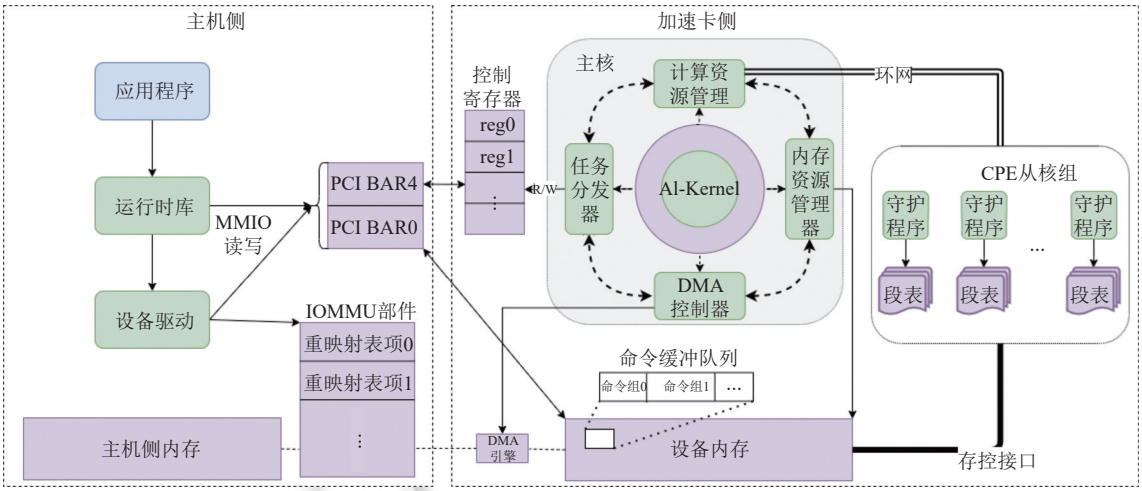


图 3 SDAA 运行时系统架构图

为尽可能在系统层面减少开销, 不同于 GPU 的两级命令缓冲模式, SDAA 运行时系统基于主核较强的逻辑处理能力设计了单层的命令传输模型, 命令发送的流程为: 主机上的驱动程序在内核空间分配一块加速卡侧的内存作为命令缓冲区 (command buffer), 然后将命令缓冲区内存映射到用户空间.

主机侧的 SDAA 库提供用户 API 接口, 完成对加速卡模式下计算任务通用的、低延迟的、无特殊硬件支持的调度和管理. AI-Driver 通过 BAR4 (base address register) 空间来访问设备上的控制寄存器进行控制信息交互, 使用 BAR0 空间访问设备内存中的命令缓冲队列来传递命令, 通过主机侧 IOMMU 部件的重映射表来支持主机侧和设备侧内存的高效 DMA 传输.

在加速卡内部主核上设计实现了一个基于 Unikernel 架构[25]的轻量级系统 AI-Kernel, 把通用主核灵活的定义成不同的角色, 来管理卡上的内存资源、计算资源、DMA 部件等, 并通过控制寄存器与主机侧的设备驱动运行时进行交互, 以响应主机侧的各种指令. AI-Kernel 是一个单用户、单进程、单地址空间的系统, 减少了用户身份验证、地址切换、系统调用等各种开销, 能够极其快速的响应主机侧发送的命令; AI-Kernel 的启动时间在 20 ms 左右, 在同样的芯片上, 传统的 Linux 系统启动时间在 10 s 左右; AI-Kernel 编译后的镜像文件在 100 KB 左右, 而传统的 Linux 镜像一般在 10 MB 以上.

3 软硬协同的系统优化

加速卡运行时系统是用户申请调度加速卡资源的总入口, 提高 SDAA 系统接口性能是系统设计实现的重要目标之一. 为应对运行时系统面临的挑战, 降低系统层面开销, 提升上层算子应用的性能, 本文采用软硬协同的方式设计了加速卡段页结合的多级内存分配算法、可分页内存多线程多通道的传输模型、多异构部件自适应的数据传输算法和基于片上阵列通信的快速核函数启动方法, 对 SDAA 运行时中内存管理、数据传输、核函数启动等接口进行优化.

© 中国科学院软件研究所 http://www.jos.org.cn

3.1 段页结合的多级内存分配算法

SW-AI 缺少类似 GPU 的页表部件[21], 锁页内存需要主机侧分配物理上连续的内存来满足 DMA 部件传输要求, 但 Linux 系统中缺少分配物理上连续大块内存的手段, 因此 SDAA 驱动中利用硬件 IOMMU 部件设计了锁页内存分配算法. 针对卡上主核使用页式、从核使用段式内存的特色, 结合智能应用内存需求, 设计了段页结合的多级内存分配算法.

● 锁页内存分配算法. 如算法 1 所示, 内核分配的锁页内存虚空间连续物理上不连续, 使用 IOMMU 重映射表把物理上不连续内存定向为 DMA 可用的连续空间. 该算法在分配锁页内存时仅在主机侧的 IOMMU 中填入重映射项, 无需把信息发送到卡上, 速度要快于 GPU 对应的锁页内存分配算法.

算法 1. 锁页内存分配算法.

输入: 待分配内存大小 size;
输出: 分配内存的地址.

```
1. vaddr ← vmalloc(size) //分配锁页内存
2. area ← get_vm_area(size) //获取重映射地址空间
3. iommu_addr ← area.addr
4. num_pages ← (size >> PAGE_SHIFT)
5. for i 0 to num_pages–1 do
6.     remap_vmalloc_to_user(i, vma, vaddr) //把地址映射到用户空间
7.     set_iommu_entry(i, domain, iommu_addr, vaddr) //配置 IOMMU 重映射表
8. end
9. return vma → vaddr
```

● 段页结合的多级内存分配算法. 算法 2 展示了主要流程, 首先会根据传入的用途标志位 flag 获得要分配内存的类型是主核内存 (页式内存) 还是从核内存 (段式内存), 然后多级内存分配器会通过 first fit 和 next fit 混合搜索算法查找出大于待分配内存的最小内存块, 并修改块内存的元数据, 将其设置为已分配状态, 最后如果是段式内存则需要更新段表. 原理如图 4 所示, 第 1 级使用 bitmap 进行粗粒度分配, 结合 SW-AI 硬件特点, 将位图存储区按照 cache 行进行对齐, 使用向量优化方法, 提高了位图检索以及处理的速度. 第 2 级是使用 free_list 链表串起来的细粒度空间管理算法, 内存块的头部存放该块内存的元数据 (meta data); 分配时, 从链表上分配一个对象出去; 释放时, 把对象插入到链表. 链表指针直接分配在待分配内存中, 不需要额外的内存开销. 该算法能够支持段页统一管理, 具有分配回收速度快、数据块合并效率高、空间利用率高等优势.

算法 2. 段页结合的多级分配算法.

输入: 待分配内存大小 size, 用途标志位 flag;
输出: 分配内存的地址.

```
1. if flag == MPE then //分配主核内存
2.     max_size ← MPE_PAGE_SIZE/2
3. else
4.     max_size ← CPE_SEG_SIZE/2
5. end
6. if size >= max_size then
7.     addr ← malloc_from_bitmap(flag, size) //需要的内存块大, 直接从第 1 级 bitmap 中分配内存
```

© 中国科学院软件研究所 http://www.jos.org.cn

```
8. else
9.      addr ← malloc_from_free_list(flag, size) //需要的内存块小, 从第 2 级 free_list 中分配内存
10. end
11. if flag == CPE then //分配从核内存
12.      update_seg_table(addr, size) //更新段表
13. end
14. return addr
```

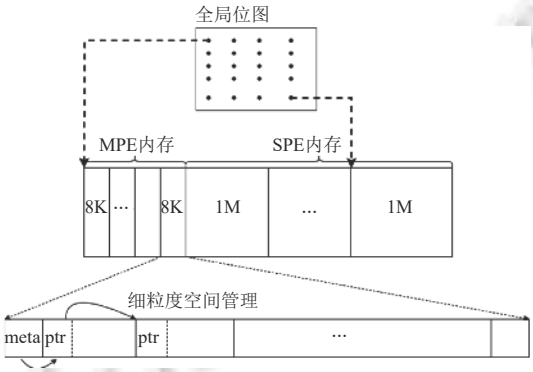


图 4 段页结合的多级内存分配算法示意图

● 加速卡内存分配时间开销分析. 申威加速卡和 GPU 内存分配的时间开销如公式 (1) 所示. 其中 T 为内存分配所需要的总时间; t_{host} 指的是主机 CPU 侧命令发起以及结果返回的时间开销, 与链路延时密切相关; t_{malloc} 是设备侧从接收到命令到内存分配完成的时间; t_{update} 是更新段 (页) 表的时间, GPU 中要通过 PCIe 通路把多条页表信息填入页表中, 更新页表的时间如公式 (2) 所示, t_0 是一个 μs 级[18]的常量, 对于申威加速卡, 只需主核通过 I/O 填入一个段表项, 时间如公式 (3) 所示, t_1 为 ns 级的常量.

$$T = t_{\text{host}} + t_{\text{malloc}} + t_{\text{update}} \tag{1}$$

$$t_{\text{gpu-update}} = t_0 \times size/PAGE\_SIZE \tag{2}$$

$$t_{\text{sw-update}} = t_1 \tag{3}$$

针对公式 (1) 中的 t_{host} 时间开销, sdaaMalloc 通过使用 PCIe 共享寄存器来进行主机-加速卡间的信息传递, 大幅降低了传输延时. 通过段页结合的多级内存分配算法, 公式 (1) 中的 t_{malloc} 时间开销大大降低. 得益于软硬协同的架构设计以及算法优化, SDAA 的加速卡内存分配效率预期比 CUDA 要高.

3.2 多线程多通道的传输模型及多异构部件自适应的数据传输算法

为充分利用多通道 DMA 的硬件能力, 提高数据传输带宽, 设计实现了可分页内存多线程多通道 (multi-threads multi-channels, MTMC) 的传输模型, 提高主机-加速卡 (包括 host to device, H2D, device to host, D2H) 数据传输性能; 针对申威智能加速卡 PCIe 上的 DMA 部件仅支持在主机-加速卡间数据传输, 无法直接支持上层应用加速卡-加速卡传输的问题, 使用软硬协同的方式设计了多异构部件自适应的传输算法, 实现高效的 D2D (device to device) 数据传输功能.

● 可分页内存多线程多通道的传输模型 (multi-threads multi-channels, MTMC). 模型结合加速卡上仅有一个主核且 DMA 部件有 8 通道的特点, 参考 CUDA 中的双缓冲机制以及一些改进的多线程双队列模型[19,21], 在主机侧根据需求分配多个锁页缓冲区, 把用户可分页内存中的数据使用多线程的方式拷贝到不同的缓冲区中, 然后再把命令信息放入命令缓冲队列中, 这种命令中有可以并发执行的标识位, 加速卡侧的主核接收到命令信息后把所有

© 中国科学院软件研究所 http://www.jos.org.cn

带有并发标识位的命令分配给不同的 DMA 通道进行数据传输, 通过这种方式可以有效提高 PCIe 带宽利用率, 提高数据传输效率.

$$8\times S_c = S_{\text{B2D}} \leqslant N_t \times S_{\text{H2B}} \tag{4}$$

$$S_{\text{B2D}} + N_t \times S_{\text{H2B}} \leqslant B_{\text{MEM}} \tag{5}$$

MTMC 模型需同时满足公式 (4) 和公式 (5), 其数据传输的时间线如图 5 所示. 公式 (4) 中 S_c 对应图中单通道的 BufferToDevice 平均传输速率, S_{B2D} 是使用 Channel 0 至 Channel 7 共计 8 通道 DMA 并发 BufferToDevice 的总速率, S_{H2B} 是主机上单线程把数据从可分页内存拷贝到锁页缓存的带宽, N_t 是主机侧线程数量, B_{MEM} 是主机侧访存带宽. 其中, 公式 (4) 和公式 (5) 均取“=”的线程数量可以达到 PCIe 带宽的最大利用率.

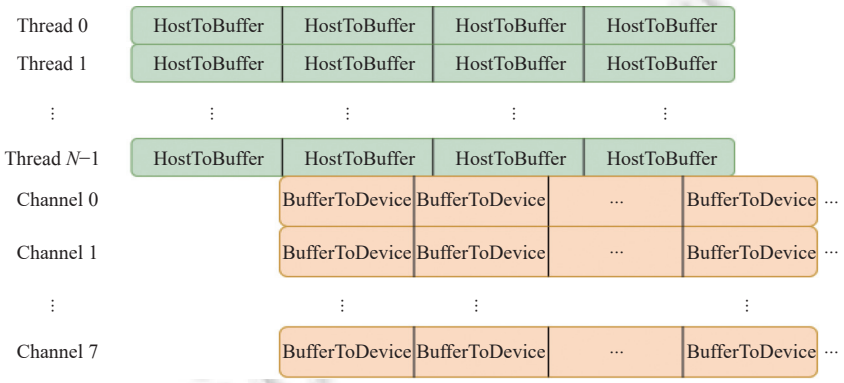


图 5 多线程多通道数据传输时间线

● 多异构部件自适应的传输算法. 在申威智能加速卡上, 支持数据搬运功能的部件有主核、从核、PCIe 上的 DMA 部件、RC 上的 IOMMU 部件, 各数据传输部件的情况汇总如表 1 所示.

表 1 数据传输部件汇总表

D2D数据传输部件	启动开销	最大带宽	能否和从核计算互相隐藏 (overlap)
主核	小	低	一定程度上可以
从核	大	高	否
DMA部件	中	中等, 带宽上限为PCIe带宽一半	是
IOMMU部件	中	中高, 带宽上限为PCIe带宽	是

该算法综合利用了申威智能加速卡上的主核、从核、PCIe 上的 DMA 部件、RC 上的 IOMMU 部件等硬件资源, 结合不同部件启动开销、最大带宽、能否和从核计算互相隐藏 (overlap) 等特点, 根据传输数据量的大小计算不同部件的时间开销 T, 选择最优的部件进行 D2D 数据传输, T 组成如下:

$$T = T_h + T_d \tag{6}$$

其中, T_h 是主机侧的开销, T_d 是加速卡侧的开销, 在不同机制下 T_d 各不相同, 不同机制下分别为 $T_{d\text{-}mpe}$、$T_{d\text{-}cpe}$、$T_{d\text{-}dma}$、$T_{d\text{-}iommu}$, mpe_cpy_bw 是系统在启动过程中会自动测试获取主核数据复制的性能参数, T_{cpe_oh} 是系统发起一次从核 D2D 的开销时间, cpe_cpy_bw 是使用从核组进行数据复制的性能参数, $dmar\_bw$, $dmaw\_bw$ 是 PCIe 的 DMA 部件发起读写的实际带宽性能, T_{dma_oh} 是发起一次 DMA 操作的启动开销, T_0 是数据从设备传到主机上的缓冲区再从缓冲传回设备中间的时间间隔.

$$T_{d\text{-}mpe} = S/mpe\_cpy\_bw \tag{7}$$

$$T_{d\text{-}cpe} = T_{cpe\_oh} + S/cpe\_cpy\_bw \tag{8}$$

$$T_{d\text{-}dma} = (T_{dma\_oh} + S/dmar\_bw) + (T_{dma\_oh} + S/dmaw\_bw) + T_0 \tag{9}$$

$$T_{d\text{-}iommu} = T_{dma\_oh} + S/dma\_cpy\_bw \tag{10}$$

© 中国科学院软件研究所 http://www.jos.org.cn

$$dma\_bw = (dmar\_bw + dmaw\_bw)/2 \tag{11}$$

该算法在系统启动时能够预先根据不同传输数据量 S 搜集各部件传输性能, 自动选择最优的实现方式进行 D2D 数据传输, 其原理如图 6 所示, 在支持 IOMMU 部件的情况下, 数据量小于 S_0 时, 使用的是主核进行数据传输, 数据量在 S_0 和 S_1 之间时, 使用的是 IOMMU 部件进行传输, 数据量大于 S_1 后, 使用的是从核进行传输; 从图中也可以看到, 在不支持 IOMMU 的情况下, 数据量在 A 和 B 两点之间时, 会使用 DMA 部件进行传输.

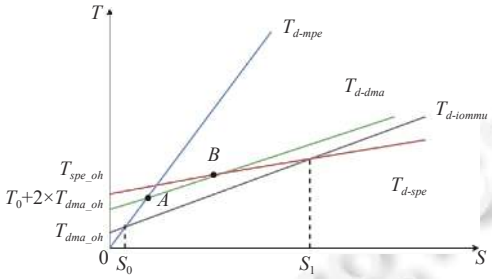


图 6 多异构部件的自适应数据传输原理

3.3 基于片上阵列通信的快速核函数启动方法

模型训练推理需要调用大量的算子, 每个算子都对应一次核函数启动, 降低核函数启动开销对上层应用意义重大[22]. 申威智能加速卡中从核是精简的 RISC 架构, 传统从核启动方法开销在 ms 级, 任务完成同步需要查看每个从核的状态, 效率较低, 无法满足应用需求, 为此, 本文首先分析了加速卡核函数启动延时 (kernel launch latency, *KLL*), 然后结合硬件特点设计实现了基于片上阵列通信的快速核函数启动方法.

● 加速卡核函数启动延时分析. *KLL* 是指从用户在主机 CPU 上发起核函数启动命令, 到核函数执行结束, 在主机 CPU 侧同步完成的总时间. 不论是 GPU 还是申威智能加速卡, 均有两部分组成, 如下列公式所示.

$$KLL = T_{\text{exec}} + KLO \tag{12}$$

$$KLO = O_{\text{cpu}} + O_{\text{cmd-tran}} + O_{\text{device}} \tag{13}$$

$$O_{\text{sw-device}} = O_{\text{command}} + O_{\text{init}} + Num \times (O_{\text{start}} + O_{\text{query}}) \tag{14}$$

其中, *KLL* 是核函数启动总的时间延时, T_{exec} 是核函数在卡内计算单元运行的时间, 这部分时间是真正执行计算的有效时间, *KLO* (kernel launch overhead) 是核函数启动执行总的开销, *KLO* 由 3 部分组成, CPU 侧的开销 O_{cpu}, 命令传输开销 $O_{\text{cmd-tran}}$, 加速卡侧的开销 O_{device}. O_{cpu} 是主机侧的时间开销, 不同设备运行时接口准备命令及参数信息的过程差别不大. 对于 $O_{\text{cmd-tran}}$, GPU 使用的是 ring buffer 和 command buffer 两级命令缓冲模式, 申威智能加速卡使用的是一级命令缓冲模式, 速度更快. 加速卡侧的开销 O_{device} 两者差别较大, GPU 是使用命令引擎 (command engine) 来读取以及解析命令, 然后配置启动计算引擎, 计算完成后同步结束信息, 在 GPU 上 O_{device} 的操作由硬件执行, 耗时在 μs 级; 申威智能加速卡上的 O_{device} 操作由主核上轻量级系统执行, 包括读取并解析命令 (O_{command}), 配置并初始化从核组信息 (O_{init}), 启动从核组 (O_{start}), 从核计算完成后同步结束信息 (O_{query}) 等. 传统启动方法耗时大主要是因为每次运行都要进行配置初始化, O_{init} 开销在数百 μs. 传统启动任务和查询任务结束等操作需依次查询 32 个从核状态, Num 为 32, 单次开销 ($O_{\text{start}} + O_{\text{query}}$) 为 0.5 μs 左右.

算法 3. 函数启动算法.

```
1. void (*kernelFunc)(void*);
2. void *kernel_args = NULL;
3. while TRUE do
4.     asm("halt") //从核进入浅睡眠状态, 等待主核唤醒
5.     kernelFunc ← (void*)_PC //获取 PC
```

© 中国科学院软件研究所 http://www.jos.org.cn

```
6.      kernel_args ← (void*)_ARG //获取参数
7.      *done ← 0
8.      kernelFunc(kernel_args) //执行核函数
9.      sync() //从核列通信同步
10.     synr() //从核行通信同步
11.     *done ← 3
12. end
```

● 基于片上阵列通信的快速核函数启动方法. 该方法流程如图 7 所示, 从核使用浅睡眠和行列同步、主核使用寄存器广播等硬件特性来降低加速卡侧的开销 O_{device}. 从核利用浅睡眠的功能来隐藏从核组初始化的时间开销, 在加速卡启动时对从核资源进行初始化, 从核开始执行守护程序 Slave_Daemon, 不再需要对从核资源进行初始化, O_{init} 降为 0. 函数启动如算法 3 所示, 从核启动后进入浅睡眠模式, 可以通过“按门铃”的方式来启动从核函数, 函数执行完成后加入了行列同步的代码, 查询任务状态时只需要查询 0 号从核的即可, Num 由 32 变为 1, 所需时间是传统方式 (需要依次配置 32 个从核并逐个轮询任务的状态) 的 1/32. 综上, 基于片上阵列通信的快速核函数启动方法中, 使用从核守护程序 Slave_Daemon 把开销 O_{device} 由 ms 级降为 μs 级.

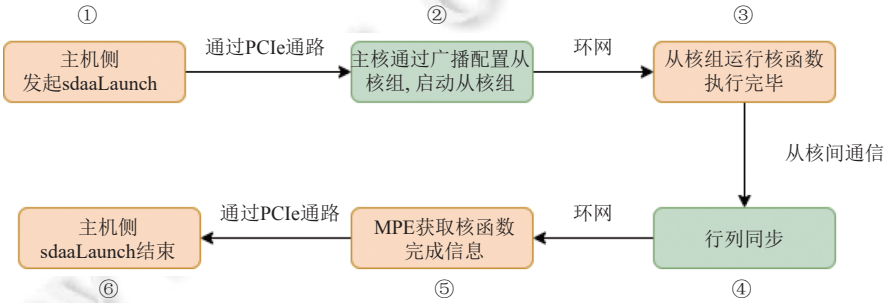


图 7 申威智能加速卡核函数启动流程

4 实验分析

实验分析主要针对关键技术和关键路径接口性能, 关键路径接口包括内存管理接口 sdaaMalloc 和 sdaaMallocHost、数据传输接口 sdaaMemcpy (包含 D2H、H2D、D2D 等)、核函数启动接口 sdaaLaunch 等.

4.1 测试环境

测试硬件环境为一台型号为 NF5468M5 的浪潮英信服务器, 在其 PCIe 插槽上分别安装一块 GPU 加速卡和一块申威智能加速卡, 以确保主机侧环境一致性. 测试环境如表 2 所示.

表 2 SDAA 测试软件环境

系统	服务器配置	操作系统	加速卡配置	驱动	运行时
SDAA CUDA	CPU至强® Silver4214 CPU @ 2.20 GHz DDR4内存 128 GB	CentOS7.8	申威智能加速卡 NVIDIA V100	AI-driver1.0 drm-535.86.10	SDAA 1.0 CUDA 12.2

4.2 内存分配算法及内存管理接口性能

设备上段页结合的多级内存分配算法是 sdaaMalloc 设备内存分配接口的核心, 为验证该算法的有效性, 本文使用申威系统中 C 标准库 (libc) 中的内存分配器 Malloc 作为参照对象, 分别分配 1 KB、16 KB、256 KB、4 MB、8 MB、64 MB、1 GB 等不同大小的内存块, 反复运行 100 次后取平均, 测试结果如图 8 所示, 从结果中可以看出, 针对不同大小的内存分配请求, 段页结合的多级内存分配算法性能均优于标准库中的 Malloc 算法.

© 中国科学院软件研究所 http://www.jos.org.cn

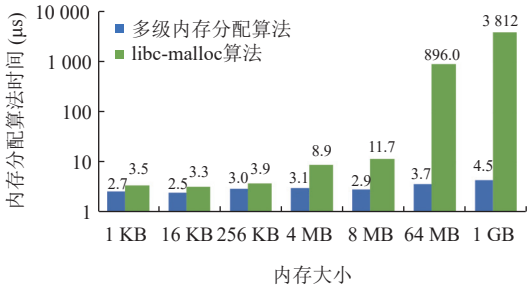


图 8　内存分配算法性能对比

接着对比设备内存分配管理接口 sdaaMalloc 和 cudaMalloc, 使用主机侧系统接口 gettimeofday 记录设备内存分配时间, 然后释放内存, 反复测试 100 次取平均结果. 然后对比主机锁页内存分配接口 sdaaMallocHost 和 cudaHostAlloc, 使用主机侧系统接口 gettimeofday 记录主机锁页内存分配时间, 然后释放内存, 反复测试 100 次取平均结果. 使用 SDAA 和 CUDA 接口分别分配大小为 1 KB、16 KB、256 KB、4 MB、8 MB、64 MB、1 GB 的内存块, 测试结果如图 9 所示.

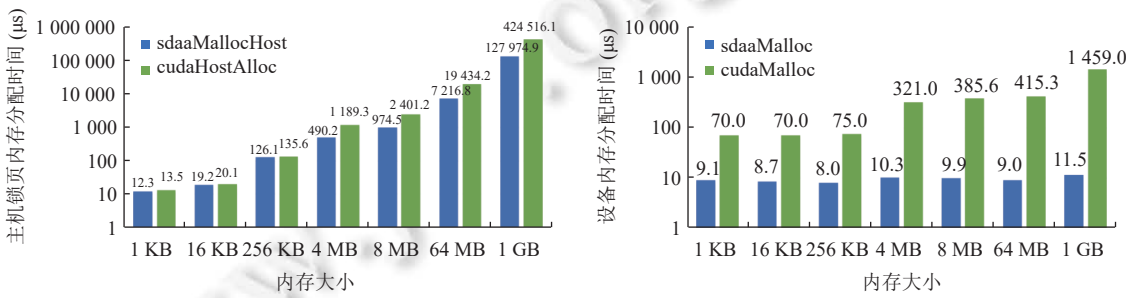


图 9　内存管理接口测试结果

图 9 中横轴是待分配的内存块大小, 纵轴是分配接口耗时, 使用的是对数坐标, 从测试结果可以看出: 锁页内存分配中, 内存大小不大于 256 KB 时 SDAA 和 CUDA 的分配开销相当, 差别不大, 大块内存分配时, sdaaMalloc-Host 的速度是 cudaHostAlloc 的 3 倍左右; 加速卡内存分配中, sdaaMalloc 开销比较稳定, 在 10 μs 左右, SDAA 的设备内存分配性能明显优于 CUDA, 大块内存分配速度是 CUDA 的 120 倍左右.

4.3　MTMC 传输模型、自适应数据传输算法及数据传输接口性能

针对可分页内存传输设计的 MTMC 传输模型, 本文通过与单线程单通道的传输方式进行对比, 来测试 MTMC 模型的有效性, 传输分主机到设备 (H2D) 和设备到主机 (D2H), 传输的数据量大小分别为 1 KB、16 KB、256 KB、4 MB、8 MB、64 MB、1 GB, 两者的测试结果如图 10 所示, 从结果中可以看到多线程多通道的传输模型相比于单线程单通道传输模型, 在性能上有极大提升, 小块数据传输延时相当, 大块数据传输带宽最大提升 2.6 倍.

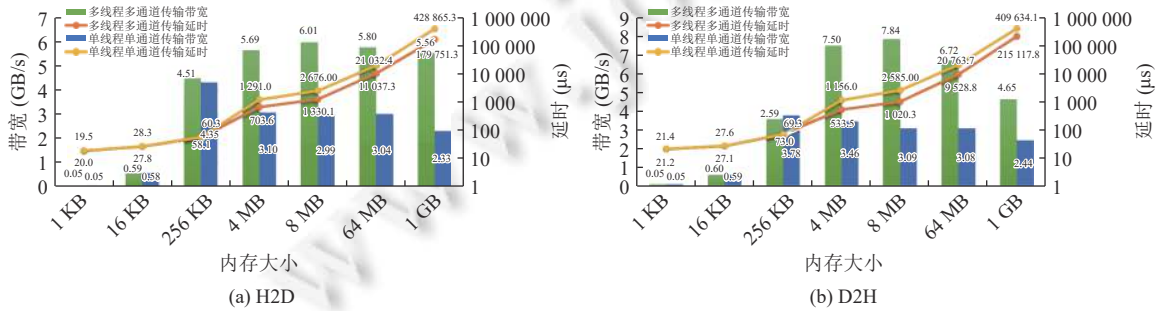


图 10　不同传输模型传输性能对比

© 中国科学院软件研究所 http://www.jos.org.cn

多异构部件自适应的数据传输算法是 D2D 传输的关键部分, 为验证算法有效性, 本文分别使用主核、从核、DMA 部件、IOMMU 部件、综合使用各种部件的自适应算法进行了数据传输性能测试, 结果如图 11 所示, 从测试结果可以看出, 自适应的数据传输算法优于使用各单独部件进行 D2D 数据传输.

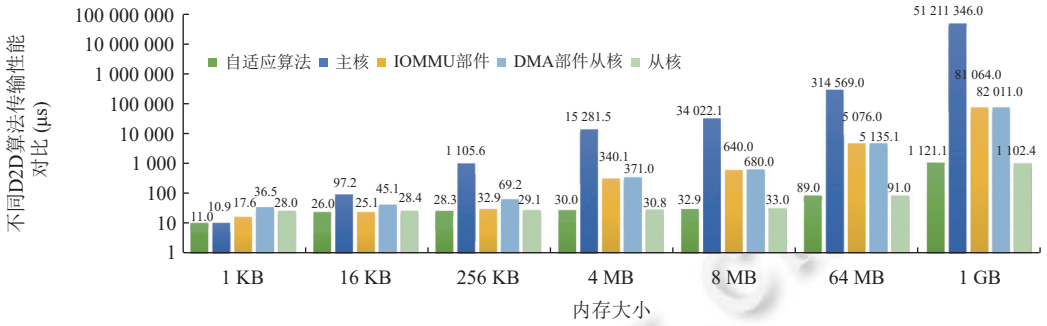
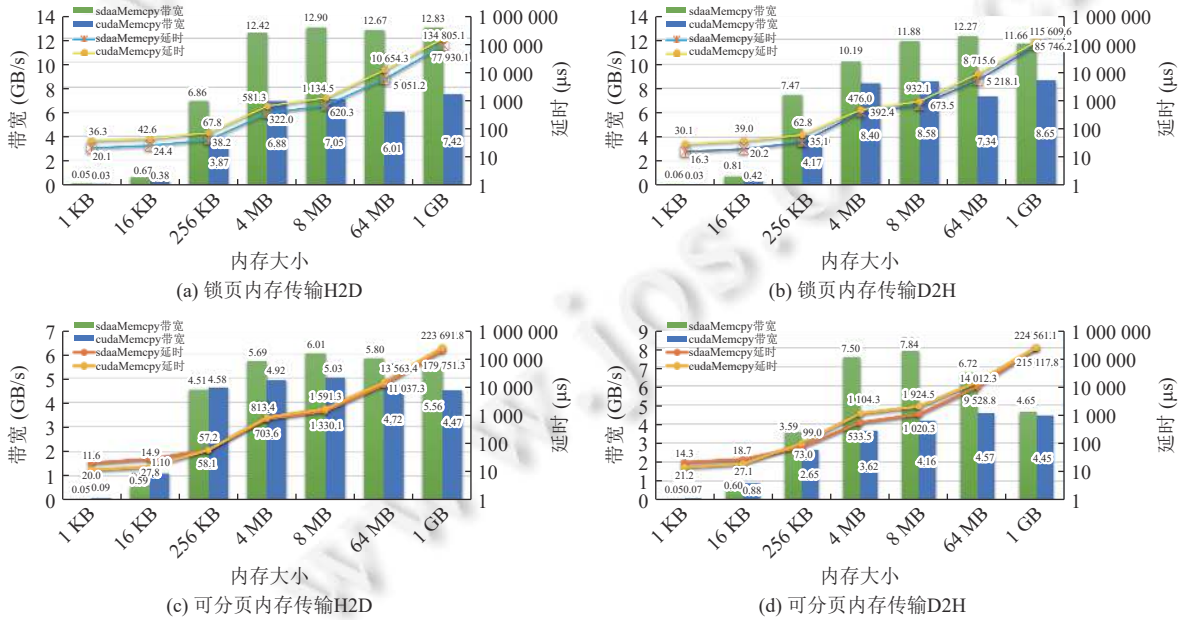


图 11 不同 D2D 算法传输性能对比

SDAA 中内存传输函数 sdaaMemcpy 和 CUDA 的 cudaMemcpy 使用方法基本一致, 用户传入主机侧内存地址, 还需指定数据传输的方向 (包括 H2D, D2H, D2D), 系统会自动根据传入的主机侧内存地址的类型 (锁页内存和可分页内存) 选择对应的方法进行传输, 对于可分页内存要先采用缓存算法把数据拷贝到锁页缓存中, 然后再进行传输. 测试方法是首先在主机侧分别使用内存分配接口 (锁页内存分配接口、可分页内存分配接口、设备内存分配接口) 分配对应的内存, 并初始化该内存数据, 然后在主机侧调用数据传输接口进行测试, 并在主机 CPU 侧记录数据传输用时, 最后进行数据校验, 测试 10 次取平均值. 在 SDAA 和 CUDA 中设定同样大小的传输数据量, 分别为 1 KB、16 KB、256 KB、4 MB、8 MB、64 MB、1 GB. 测试结果如图 12 所示.

图 12 中横轴是待传输的内存块大小, 左侧主纵轴是传输带宽, 单位是 GB/s, 右侧次纵轴表示数据传输接口延时, 单位是 μs, 次纵轴使用的是对数坐标. 从图 12 中测试结果可以看出: SDAA 的锁页内存传输启动开销和最大带宽均优于 CUDA, 启动开销是 CUDA 对应接口的 1/2, 最大带宽是 CUDA 的 1.4–1.7 倍; 可分页内存传输中, 启动开销方面 CUDA 优于 SDAA, 但是最大带宽方面 SDAA 明显优于 CUDA, 为其 1.6 倍左右; D2D 传输中, SDAA 启动开销低于 CUDA, CUDA 卡内 D2D 的带宽低于 SDAA.



(a) 锁页内存传输H2D

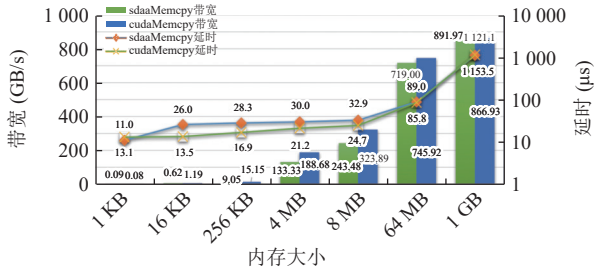
(b) 锁页内存传输D2H

(c) 可分页内存传输H2D

(d) 可分页内存传输D2H

图 12 数据传输接口测试结果

© 中国科学院软件研究所 http://www.jos.org.cn



(e) 设备内存传输D2D

图 12 数据传输接口测试结果 (续)

4.4 快速核函数启动方法及核函数启动接口性能

传统的核函数启动方法中, 在从核运行之前需要对从核进行一系列操作包括从核资源初始化、创建线程组、启动从核运行, 最后再通过轮询各从核状态等待线程结束. 基于片上阵列通信的快速核函数启动方法正是为改进传统核函数启动方法而设计, 测试中使用两种方法启动内容为空的核函数并等待任务结束, 各运行 100 次, 快速启动方法平均耗时 2.3 μs, 传统启动方式平均耗时 883.6 μs.

SDAA 中核函数启动函数为 sdaaLaunch, CUDA 中为 cudaLaunchKernel, 两者均支持同步模式和异步模式. 测试中启动的核函数均为空函数, 核函数参数为 3 个. 同步模式测试时, 在 CUDA 中配置 CUDA_LAUNCH_BLOCKING=1 参数, 在主机侧运行 100 次启动接口, 计算单次平均时间; 异步模式测试时, 先运行 100 次启动接口, 然后再进行设备同步, 计算单次平均时间. 测试结果如图 13 所示.

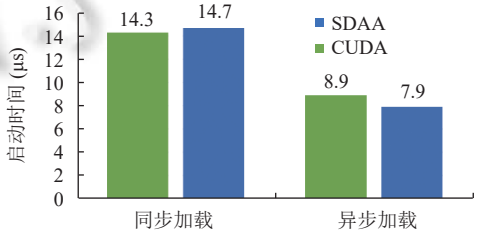


图 13 核函数启动测试结果

从图 13 中可以看出: 同步模式下 SDAA 的核函数启动性能优于 CUDA, 异步模式下 CUDA 的核函数启动性能优于 SDAA. 总的来说, SDAA 核函数性能与 CUDA 对应接口性能相当.

5 结束语

为了充分发挥国产申威众核处理器的体系结构优势, 有效支撑人工智能应用的开发和优化, 本文设计了 SDAA 运行时系统, 包含主机侧运行时库、设备驱动和加速卡侧轻量级系统、守护程序. 通过软件定义的方法设计了单层的命令传输模型, 针对人工智能应用的关键路径进行性能优化, 包括多级内存分配机制, 高效数据传输机制和快速核函数启动方法. 实验结果表明, SDAA 运行时接口性能优于 GPU 加速卡使用的 CUDA 运行时, 同时保持和 CUDA 类似的程序语义.

目前 SDAA 已经能够支撑上层算子库、框架、多类型模型应用运行, 支持的框架包括 PyTorch、TensorFlow、PaddlePaddle 等, 支持的模型包括 Mnist、BERT、YOLOv5m、Wav2Vec2.0、ResNet50、VGG 等. 后续工作有两个方向, 一个是根据应用需求进一步优化运行时接口性能, 并提高加速卡的资源利用效率; 第二加强对大模型应用的基础支撑, 充分利用加速卡软件定义主核的特点, 引入 Capuchin[26], ZeRO[27]等应用层软件的优化机制, 让 SDAA 运行时系统能够直接高效支撑上层大模型的训练推理应用.

© 中国科学院软件研究所 http://www.jos.org.cn

References:

[1] Fujii Y, Azumi T, Nishio N, Kato S. Exploring microcontrollers in GPUs. In: Proc. of the 4th Asia-Pacific Workshop on Systems. Singapore: ACM, 2013. 2. [doi: 10.1145/2500727.2500742]

[2] Gholami A, Yao ZW, Kim S, Mahoney MW, Keutzer K. AI and memory wall. 2021. https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8

[3] Garland M, Le Grand S, Nickolls J, Anderson J, Hardwick J, Morton S, Phillips E, Zhang Y, Volkov V. Parallel computing experiences with CUDA. IEEE Micro, 2008, 28(4): 13–27. [doi: 10.1109/MM.2008.57]

[4] Liu SL, Du ZD, Tao JH, Han D, Luo T, Xie Y, Chen YJ, Chen TS. Cambricon: An instruction set architecture for neural networks. In: Proc. of the 43rd ACM/IEEE Annual Int'l Symp. on Computer Architecture (ISCA). Seoul: IEEE, 2016. 393–405. [doi: 10.1109/ISCA.2016.42]

[5] Ouyang J, Noh M, Wang Y, Qi W, Ma Y, Gu CH, Kim S, Hong KI, Bae WK, Zhao ZB, Wang J, Wu P, Gong XZ, Shi JX, Zhu HF, Du XL. Baidu Kunlun an AI processor for diversified workloads. In: Proc. of the 2020 IEEE Hot Chips 32 Symp. (HCS). Palo Alto: IEEE, 2020. 1–18. [doi: 10.1109/HCS49909.2020.9220641]

[6] Jiao Y, Han L, Long X. Hanguang 800 NPU—The ultimate AI inference solution for data centers. In: Proc. of the 2020 IEEE Hot Chips 32 Symp. (HCS). Palo Alto: IEEE, 2020. 1–29. [doi: 10.1109/HCS49909.2020.9220619]

[7] Liao H, Tu JJ, Xia J, Zhou XP. DaVinci: A scalable architecture for neural network computing. In: Proc. of the 2019 IEEE Hot Chips 31 Symp. (HCS). Cupertino: IEEE, 2019. 1–44. [doi: 10.1109/HOTCHIPS.2019.8875654]

[8] freedesktop.org. Nouveau: Accelerated open source driver for NVIDIA cards. 2023. https://nouveau.freedesktop.org/

[9] Kato S, McThrow M, Maltzahn C, Brandt S. Gdev: First-class GPU resource management in the operating system. In: Proc. of the 2012 USENIX Annual Technical Conf. Boston: USENIX Association, 2012. 37.

[10] Kato S, Lakshmanan K, Rajkumar R, Ishikawa Y. TimeGraph: GPU scheduling for real-time multi-tasking environments. In: Proc. of the 2011 USENIX Annual Technical Conf. Portland: USENIX Association, 2011. 2.

[11] Kato S, Aumiller J, Brandt S. Zero-copy I/O processing for low-latency GPU computing. In: Proc. of the 2013 ACM/IEEE Int'l Conf. on Cyber-physical Systems. Philadelphia: IEEE, 2013. 170–178. [doi: 10.1145/2502524.2502548]

[12] Kato S. Implementing open-source CUDA runtime. In: Proc. of the 54th Programming Symp. 2013.

[13] Kato S, Lakshmanan K, Ishikawa Y, Rajkumar R. Resource sharing in GPU-accelerated windowing systems. In: Proc. of the 17th IEEE Real-time and Embedded Technology and Applications Symp. Chicago: IEEE, 2011. 191–200. [doi: 10.1109/RTAS.2011.26]

[14] Kato S, Lakshmanan K, Kumar A, Kelkar M, Ishikawa Y, Rajkumar R. RGEM: A responsive GPGPU execution model for runtime engines. In: Proc. of the 32nd Real-time Systems Symp. Vienna: IEEE, 2011. 57–66. [doi: 10.1109/RTSS.2011.13]

[15] Volos S, Vaswani K, Bruno R. Graviton: Trusted execution environments on GPUs. In: Proc. of the 13th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 681–696.

[16] Calderón AJ, Kosmidis L, Nicolás CF, Cazorla FJ, Onaindia P. Understanding and exploiting the internals of GPU resource allocation for critical systems. In: Proc. of the 2019 IEEE/ACM Int'l Conf. on Computer-aided Design (ICCAD). Westminster: IEEE, 2019. 1–8. [doi: 10.1109/ICCAD45719.2019.8942170]

[17] Neugebauer R, Antichi G, Zazo JF, Audzevich Y, López-Buedo S, Moore AW. Understanding PCIe performance for end host networking. In: Proc. of the 2018 Conf. of the ACM Special Interest Group on Data Communication. Budapest: ACM, 2018. 327–341. [doi: 10.1145/3230543.3230560]

[18] Fujii Y, Azumi T, Nishio N, Kato S, Edahiro M. Data transfer matters for GPU computing. In: Proc. of the 2013 Int'l Conf. on Parallel and Distributed Systems. Seoul: IEEE, 2013. 275–282. [doi: 10.1109/ICPADS.2013.47]

[19] Wilt N. The CUDA Handbook: A Comprehensive Guide to GPU Programming. Upper Saddle River: Addison-Wesley, 2013.

[20] Suzuki Y, Kato S, Yamada H, Kono K. GPUvm: GPU virtualization at the hypervisor. IEEE Trans. on Computers, 2016, 65(9): 2752–2766. [doi: 10.1109/TC.2015.2506582]

[21] Cho S, Hong J, Choi J, Han H. Multithreaded double queuing for balanced CPU-GPU memory copying. In: Proc. of the 34th ACM/SIGAPP Symp. on Applied Computing. Limassol: ACM, 2019. 1444–1450. [doi: 10.1145/3297280.3297426]

[22] Kim S, Oh S, Yi Y. Minimizing GPU kernel launch overhead in deep learning inference on mobile GPUs. In: Proc. of the 22nd Int'l Workshop on Mobile Computing Systems and Applications. ACM, 2021. 57–63. [doi: 10.1145/3446382.3448606]

[23] Zhang LQ, Wahib M, Matsuoka S. Understanding the overheads of launching CUDA kernels. In: Proc. of the 48th Int'l Conf. on Parallel Processing. Kyoto: ACM, 2019.

[24] Wu JX, Qi XF, Gao YZ. Review of programming models for heterogeneous parallel computing. Aerospace Shanghai (Chinese & English), 2021, 38(4): 1–11 (in Chinese with English abstract). [doi: 10.19328/j.cnki.2096-8655.2021.04.001]

© 中国科学院软件研究所 http://www.jos.org.cn

[25] Madhavapeddy A, Mortier R, Rotsos C, Scott D, Singh B, Gazagnaire T, Smith S, Hand S, Crowcroft J. Unikernels: Library operating systems for the cloud. ACM SIGARCH Computer Architecture News, 2013, 41(1): 461–472. [doi: 10.1145/2490301.2451167]

[26] Peng X, Shi XH, Dai HL, Jin H, Ma WL, Xiong Q, Yang F, Qian XH. Capuchin: Tensor-based GPU memory management for deep learning. In: Proc. of the 25th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2020. 891–905. [doi: 10.1145/3373376.3378505]

[27] Rajbhandari S, Rasley J, Ruwase O, He YX. ZeRO: Memory optimizations toward training trillion parameter models. In: Proc. of the 2020 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Atlanta: IEEE, 2020. 1–16. [doi: 10.1109/SC41405.2020.00024]

附中文参考文献:

[24] 邬江兴, 祁晓峰, 高彦钊. 异构计算并行编程模型综述. 上海航天 (中英文), 2021, 38(4): 1–11. [doi: 10.19328/j.cnki.2096-8655.2021.04.001]

赵玉龙(1984－), 男, 博士生, 助理研究员, 主要研究领域为人工智能基础软件.

李宇轩(1995－), 男, 博士生, CCF 专业会员, 主要研究领域为异构平台的科学计算应用.

张鲁飞(1986－), 男, 博士, 工程师, CCF 学生会员, 主要研究领域为高性能计算, 操作系统, 机器学习.

孙茹君(1990－), 女, 博士, 助理研究员, 主要研究领域为高性能计算机体系结构, 并行计算模型.

许国春(1979－), 男, 助理研究员, 主要研究领域为操作系统, 计算机体系结构.

刘鑫(1979－), 女, 博士, 研究员, 博士生导师, CCF 杰出会员, 主要研究领域为并行算法和应用.

© 中国科学院软件研究所 http://www.jos.org.cn