CS205 · C/C++ Programming

Project4 Report:　A 2D GPU Mat

赖海斌
12211612

[1]

摘要：本次项目的重点在于开发了一个功能强大的 GPU 矩阵类，该类实现了多数据输入、运算符重载、感兴趣区域（ROI）操作、内存管理以及跨 GPU 运算等关键功能。在此过程中，我们深入研究了 GPU 内存与通信设计概念，并获得了对 GPU CUDA 编程的实践经验。这一项目不仅使我们对 C/C++的特性有了更深入的了解，而且为我们进一步探索并发编程和 GPU 加速计算提供了坚实的基础。

关键词：cvMat；CUDA；System Design；GpuMat；memory-safe；

[1] 【NVIDIA 黄仁勋 跳科目三】
https://www.bilibili.com/video/BV1Ye411e7y1/?share_source=copy_web&vd_source=72eac555730ba7e7a64f9fa1d7f2b2d4

目录 Content

Part 0: 对前 3 次 Project 的总结与对后 2 次 Project 的展望

什么是于++？这个问题困扰了很多计系学生。一个简单的“Yu”字，意味着给分不确定，不理解，高难度。这门课令许多人谈虎色变，以至于我上学期末跟李卓钊老师谈起，这学期我想选 C/C++时，导师大惊：“万分小心！听说那门课工作量很多！”。然而，我是一个喜欢 Project 的人，我也不喜欢听别人说啥就是啥，5 个 Project 驱使着我加入到了这学期的 100 个倒霉蛋里。但是在三次 Project 结束后，突然有一天我在想，这 5 个 Project 到底意味着什么？我为什么要做这 5 个 Project？

Project1 是从计算器这类基础运算开始的。在以往的 Java,Python 里，我们大多时候不是很关注数据类型，比如 float 和 double，亦或者这些语言都有很好的数据类封装及 api。在实现 Project1 的时候，指针、栈、正则表达式开始引起我们注意。**Project1 是想告诉我们，C/C++关注于底层，它是比 Java 颗粒度更小、更精细化的语言。**在这门语言上设计一个小系统，才能体验到精细化的感觉。

Project2 是 Java 和 C 的比较。我们在过去的优化里大多只是关注算法层面的优化，比如剪枝、更好地局部搜索。但是，我们的代码是如何执行的，C 为什么快于 Java，似乎研究的同学更少。**Project2 是想告诉我们，C/C++的精细化与对底层的接近，使得它的程序性能更高，有更多的优化方向。**同时，计算机软硬件构成的复杂系统让 C/C++执行情况更复杂，执行时不能一概而论。

Project3 是优化浮点数矩阵乘法。在了解到 C/C++的高性能后，我们开始实践技术。我们复习并运用了 SIMD，OpenMP。但是我们惊讶的发现，OpenBLAS 是个强劲的对手。突然，我们的 toy 程序被一个精密优化的复杂系统所折服。**Project3 是想告诉我们，作为一门精细的语言，无数程序员用 C/C++对程序做了系统性的优化，我们在学习优化同时，也要明白这门语言所创造的系统工程。**

Project4 是矩阵类的设计。在这里，我们会参考 cv::Mat,学习构建一个大型工程内的一个类。跟之前的底层和优化相比，这回特性和系统占据了主导。封装管理、内存泄露、软 copy、运算符重载开始运用。**Project4 可能是想告诉我们，如何开发一个系统，如何用 C/C++搭建系统，如何管理好系统。**

Project5 往年 Project 是神经网络，这两年 CUDA 与 GPU 的活跃，让我们看到 C/C++在这方面的大放异彩，我也准备在这里看看 GPU 的应用。C/C++是一门古老、但至今仍活跃在种工业界的语言。

C/C++的 Project 是变化的，每年计算器、矩阵、神经网络都会轮着来。但是其核心，想必我们从上面的描述中已经发现了，**“优化——系统——应用”是我们学习这门底层语言的流程。**这是它的特性决定的。因此，单纯地掌握 C/C++语法，其实根本没有入门。这就像二战时期日本的“知美派”，知道美国有几艘航空母舰、多少飞机并不能打败美国；了解美国人的出击战术、航母部署方式、思考方式，才能真正击败对手（说的就是你，山本五十六）。

Project 不是卷起来的，而是学到了的。知道了它真正的意图，我们才能像头号玩家里的主角，破解“于”的奥秘，发现游戏里的彩蛋。本次 Project4 属于是优化到系统的承接段。我把主要的方向建立在“系统”上。我将展示我的搭建蓝图与工程过程。同时，另一个关键词是“CUDA”，李老师布置了这个任务，我准备学习离 cv::Mat 很近的 cv::cuda::GpuMat，我会在后续继续使用 CUDA。还有就是，我计划为“应用”做准备，把我的矩阵跟 image 结合起来。

Part 1: 需求分析

我们需要一个什么样的 Matrix 类？我的第一个想法是，复刻一个跟 OpenCV 一样的就好了。我甚至已经想到，一个 good example 里一定有个哥们实现了这个。但是 OpenCV 为什么要这么设计？更多的时候是为了满足图像处理的需要。但是我想做出一个更普通的 GpuMat，它可能没有 channel，但是可以当作一个还算不错的矩阵类。我因此便提出了我对这个类的期望：

1. 安全需求

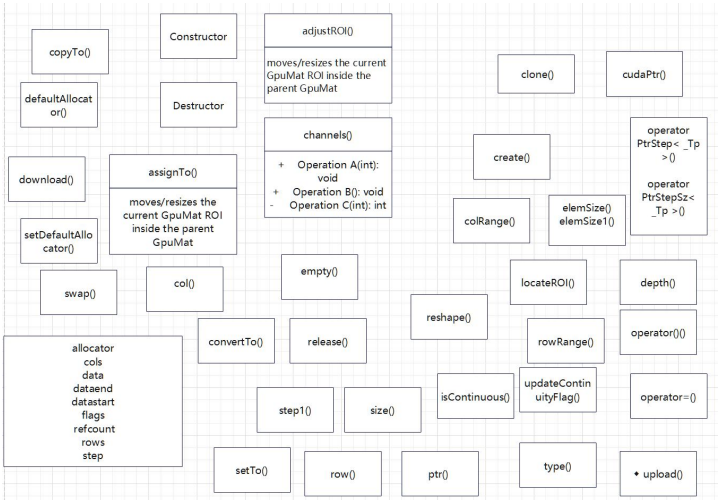
可以安全申请内存，释放内存
数据不会二次释放

2. 计算性能需求

数据复制到 GPU 的速度尽可能快
启动最佳的线程数进行计算
更多有趣的运算

3. 可用性需求

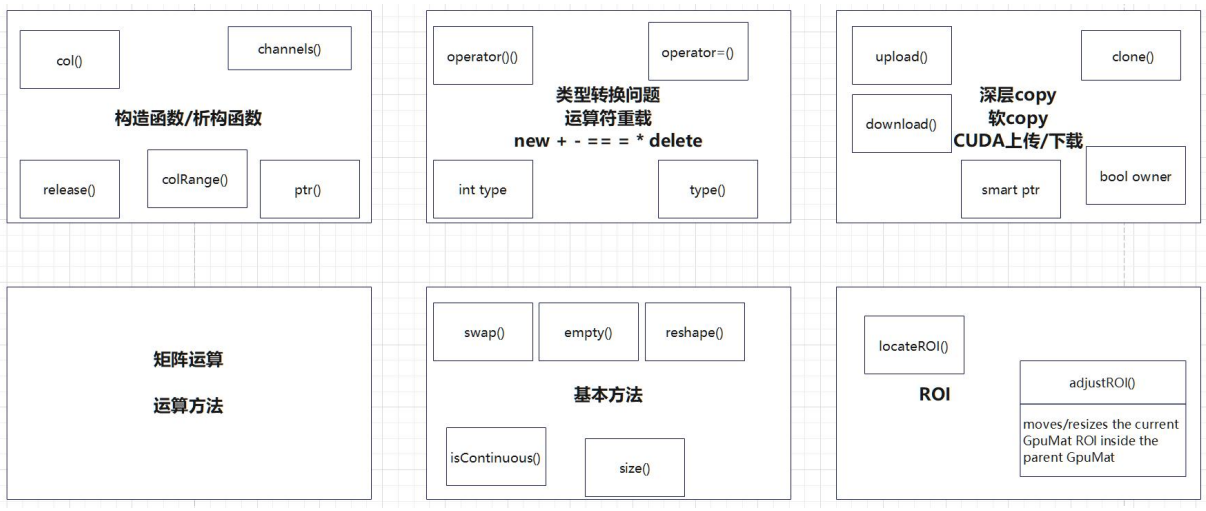
更多的适合的数据类型，如 FP16
可以使用多个 GPU 同时计算
Python 支持（有点大）
ROI 设计



Part 2: 我们怎么开始：参(fu)考(zhi)

虽然以前写过一些简单的 CUDA 样板函数，但是写一个跟 GPU 深度结合的类我还是第一次。我首先参考 cv::GpuMat。这个类没有 cv::Mat 那么庞大，拥有跟 Mat 类似的功能和 api，同时它也是个二维矩阵，非常适合学习及改进。

我将它的成员都丢了下来，随后开始分析，把它内部的设计总结为重要的 6 个部分：构造/析构函数，类型转换/运算符重载，copy 问题，运算 api，基本方法，ROI。在我的实现中，我也是按照这几个部分进行一一设计。



另外，从 Project3 对 GPU 硬件的了解中我们很容易明白，CUDA != C/C++，有这么几个性质我们必须注意：

1. 核函数谨慎使用模版！在 CUDA 分配内存时在计算上，CUDA 对于不同的数据类型调用的计算核是不一样的，面对不同数据类型矩阵的计算需要我们提前转换。

2. 谨慎处理__global__函数。__global__函数传参是通过常量内存传入 GPU 的，且规定参数大小不得大于 4KB。__global__函数不支持可变长参数。另外，开发者不能将一个操作符函数（如 operator+、operator-）声明为__global__，__global__函数不支持递归，不支持作为类的静态成员函数，支持类的友元声明但不支持在友元声明同时进行定义。

3. CUDA 尚不支持类的 static 静态数据成员。

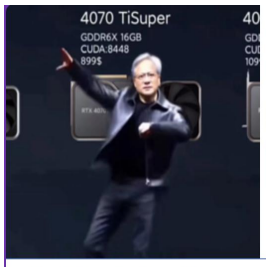
4. nvcc 版本 11.0 及更高版本支持所有 C++17 语言功能，但受此处[2]描述的限制的约束。

5. 没有 smart pointer 的支持[3]，需要自己包装实现。

Part 3: 实现过程 & 难点分析

我在 Github 上简单做了一个网页文档：https://laihb1106205841.github.io/GpuMat.github.io/annotated.html 在这里我也会上传源代码：Laihb1106205841/GpuMat.github.io: A matrix class for GPU on SUSTech 2024Spring C/C++ Project4

难绷↓



GpuMat 1.0.1

A GpuMat for CS205 C/C++ Project4

Main Page Topics Classes Search

Public Member Functions | List of all members

Matrix Class Reference

Base MAT class for GPU memory with reference counting. More...

Public Member Functions

	Matrix (size_t rows, size_t cols, int device=0)
	Matrix (const **Matrix** &other)
size_t	**getRows** () const noexcept
size_t	**getCols** () const noexcept
size_t	**getSize** () const noexcept
float *	**getData** () const noexcept
float	**getRoiColData** (size_t row, size_t col) const
__device__ __host__ float &	**operator()** (size_t row, size_t col)

template<typename T >

[2] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cpp17

[3] https://forums.developer.nvidia.com/t/smart-pointers-in-device-and-global-functions/159561 不过，比较好的是有人做了一个 cuda 的 share pointer。但它的大小看着很吓人 https://github.com/roostaiyan/CudaSharedPtr/blob/main/cudasharedptr.h

1. 构造器 & 析构函数

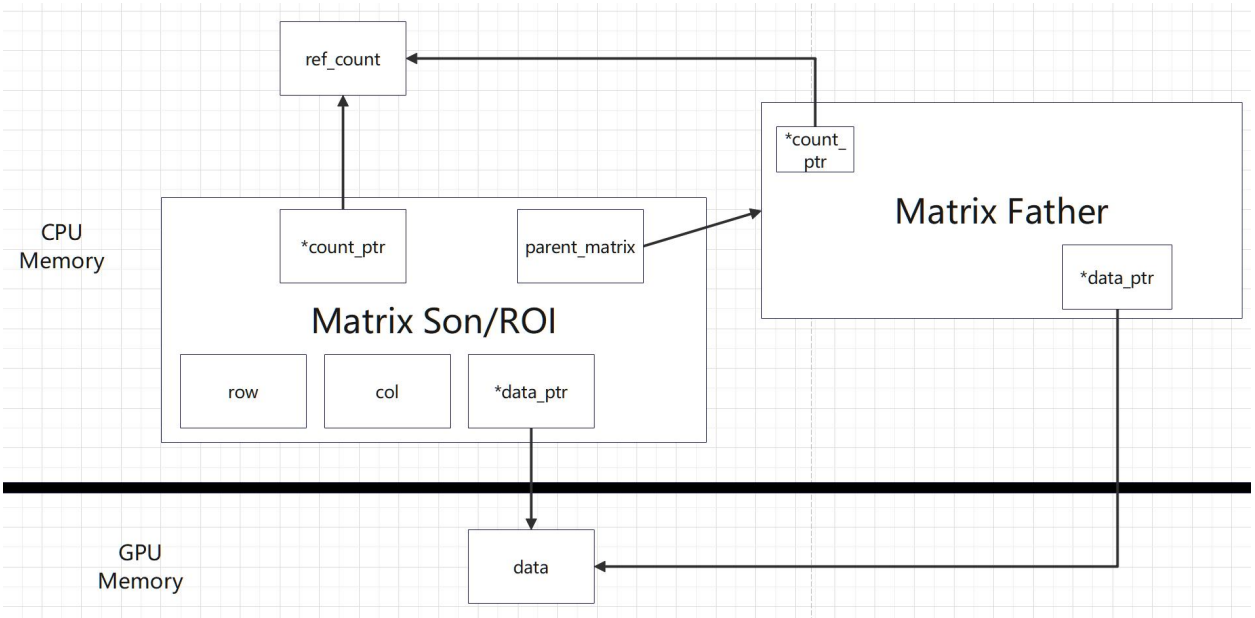
使用了 CUDA 的统一内存（Unified Memory）来分配内存，这样可以使内存在 CPU 和 GPU 之间自动进行数据迁移，从而简化了内存管理。

构造函数不会初始化矩阵的数据值，而是在需要时由用户自行设置。

支持多数据类型。Parent_matrix 为 ROI 的设计。

```cpp
Matrix(size_t rows, size_t cols, int device = 0)
    : rows(rows), cols(cols),  device(device) {

    std::cout << "hi from Matrix constructor";

    checkRowsCols(rows,cols);

    int deviceCount;
    cudaGetDeviceCount(&deviceCount);
    if(device >= deviceCount){
        std::cerr << "Invalid device num";
    }
    cudaSetDevice(device); // 选择 GPU

    ref_count = new int(1);
    // data_type = MAT_32;
    if (*ref_count==1) { // Allocate memory for the matrix
        std::cout << "\n Manage Memory for Matrix! \n";
        cudaMallocManaged(&data, rows * cols * sizeof(T));
    }


    parent_matrix = nullptr;
}
```

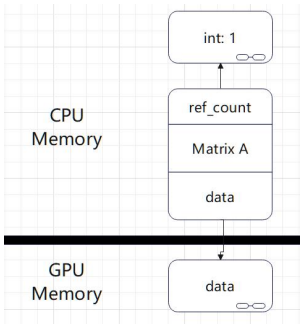
大致设计见下图：



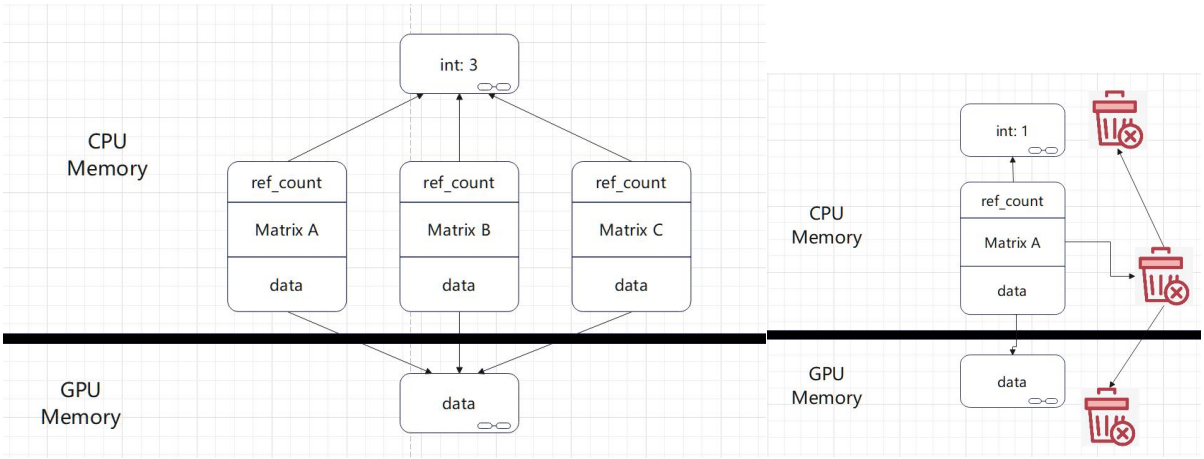
2. 如何正确地拷贝矩阵

2.1 浅拷贝

矩阵在拷贝的时候，我们设想的浅拷贝应该是只拿取原矩阵的大小、数值指针，对数据层面不会进行复制或重写等操作。为解决这一问题，我们引入了 ref_count 计数器，针对同一块矩阵对其使用次数进行计数。在初始化阶段，Matrix 在 GPU 上分配一段内存给矩阵使用，CPU 上记录当前矩阵的信息及引用次数。



随着新矩阵在“=”运算符后赋值、浅拷贝方法中创建，ref_count 会自增加 1。而每次析构矩阵时，ref_count 会减 1。当 ref_count 在最后一个引用的 Matrix 也将被释放时，我们再释放 GPU 上的 data。这里 ref_count 使用 int 是防止我们在设计时不小心多减成了负数，这样会非常难处理。



不过我的最初版本跟这个 count 的版本不同。我最初是想使用一个 bool owner 来管理 data。每当我创建一个新的浅拷贝时，新的矩阵 owner = true，旧的为 false。也就是把 data 交给最后一位拷贝的矩阵处理。但是这里面存在的问题是，假如我们中途不小心删除了最后一位，那么我们将永远无法释放 GPU 上的数据了。因此，我后面改用了 count 这一更安全的设计。这也是 share_pointer[4]与 opencv 里的想法。

要注意的是，GPU 上的数据须使用 cudaFree()函数将数据释放而不是 delete[]。面对空指针，cuda 文档里提到，“cudaFree If devPtr is 0, no operation is performed”[5]，这和 free 函数的效果是一样的。但是神奇的是，我们可以多次声明 cudaFree()函数而不引起程序崩溃，这可能是因为虽然在 GPU 上我们确实是二次释放了内存，但是我们在 CPU 上对错误异常没有进行处理。

如果想处理这一异常，我们需要检查它返回的 cudaSuccess。

另外查看文档发现，cudaMalloc 使用的是指向指针的指针，CUDA 设计者选择使用返回值来携带错误状态[6]。这和我们之前用的 malloc 函数返回一个新的指针不同。我们也会在接下来的多 GPU 环节结合文章i 对 malloc 的介绍以及 Project3 学习的知识比较这两个函数。

2.2 深拷贝

我们要在 CUDA 上重新分配一次内存，并把数据重新拷贝过去。直接的设计非常简单，直接调用 cudaMalloc 和 cudaMemcpy 就可以了。

但是 GPU 跟 CPU 很不一样的是，CPU 中我们的地址都已经是逻辑地址，无论我们的机器有多少内存条等物理状况，在 BIOS 识别到内存后，OS 会用 MMU 给定的虚拟内存代替物理地址。但是 CUDA 的分配不同，cudaMalloc 不允许一个整块的矩阵一半分配在 GPU0，一半在 GPU1。

---

[4] https://learn.microsoft.com/zh-cn/cpp/standard-library/shared-ptr-class?view=msvc-170

[5] https://developer.download.nvidia.cn/compute/DevZone/docs/html/C/doc/html/group__CUDART__MEMORY_gb17fef862d4d1fefb9dba35bd62a187e.html

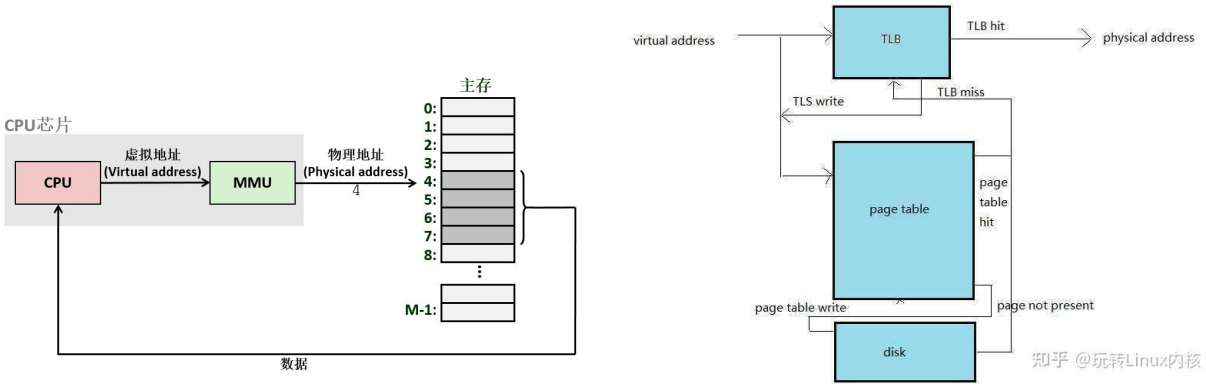
[6] 更多的讨论：https://stackoverflow.org.cn/questions/12936986

为了搞清楚情况，OS，启动！

```cpp
/**
@brief 深拷贝函数，用于在CUDA设备之间复制数据
*/
void deepCopy(const I *src_data, size_t size) const {
    cudaMalloc((T**)&src_data, size * sizeof(T)); // 在设备上分配新的内存
    cudaMemcpy(data, src_data, size * sizeof(T), cudaMemcpyDeviceToDevice); // 将数据从源地址复制到新分配的设备内存上
}

/**
@brief 深拷贝函数，用于在CUDA设备之间复制数据
*/
void deepCopy(const I *src_data, size_t size, int device) {
    cudaSetDevice(device);
    cudaMalloc((T**)&src_data, size * sizeof(T)); // 在设备上分配新的内存
    cudaMemcpy(data, src_data, size * sizeof(T), cudaMemcpyDeviceToDevice); // 将数据从源地址复制到新分配的设备内存上
}
```

在 MMU 中，它会完成地址转换的操作。当程序访问内存时，它使用的是逻辑地址，并将逻辑地址转换为物理地址，以便正确地访问内存中的数据和指令。这种地址转换是通过页表或段表等数据结构来实现的。

在虚拟内存管理中，页表（Page Table）是一种数据结构，用于将逻辑地址映射到物理地址。它将系统中的每一页（通常是固定大小的内存块，比如 4KB）映射到物理内存中的对应页框（或称为物理页）。每个进程都有自己的页表，用于管理其虚拟地址空间和物理内存的映射关系。



当我们的 CPU 程序访问内存时，程序提供的是逻辑地址。MMU 根据页表将逻辑地址生成 PTE 地址[7]，然后再进行实际的内存访问。这里边会存在两个状态：命中，缺失。

页命中（Page Hit）： 当程序访问的内存页面已经在物理内存中时，称为页命中。这意味着所需的数据或指令已经在物理内存中，程序可以直接访问它，而不需要从磁盘或交换空间中加载。

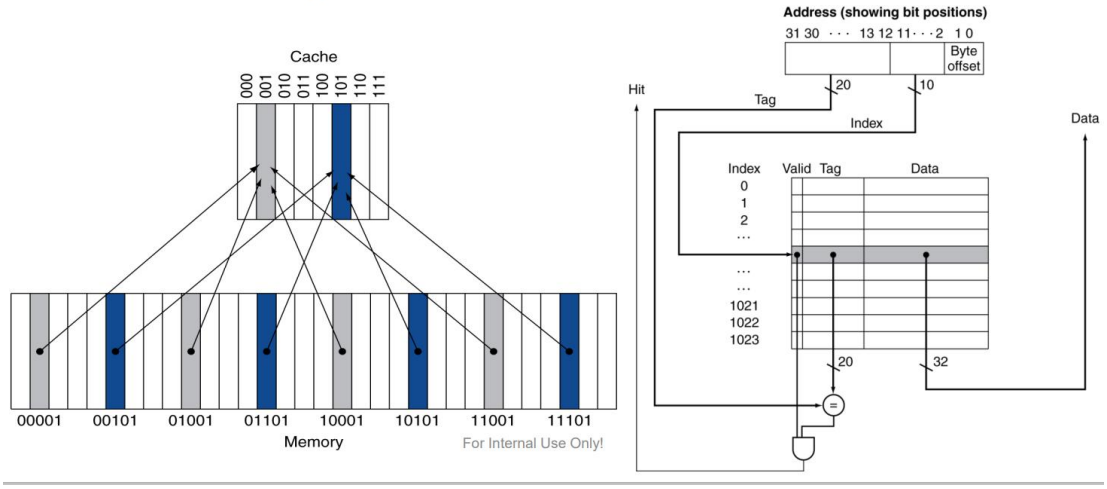
页缺失（Page Fault）： 当程序访问的内存页面不在物理内存中时，称为页缺失。这意味着所需的数据或指令尚未加载到物理内存中，可能因为被换出到磁盘或是第一次访问。当发生页缺失时，OS 会将相应的页面从磁盘或交换空间加载到物理内存中，更新页表。然后，程序被重新启动以继续执行。

这和我们的计组课上学习的 Cache 工作原理基本差不多。或者换句话说，其实内存中的思想基本上是一致的。[8]所以我们在这里看到一个很关键的因素，我们之所以可以在 CPU 上愉快地用逻辑地址，MMU

[7] PTE（Page Table Entry）地址是页表中的条目所对应的物理内存地址。

[8] 《计算机体系结构：量化研究方法 第 6 版》内存章节 P92 页对这部分进行了详细的说明，P105 页对 Intel Core i7 6700 的实现做了介绍

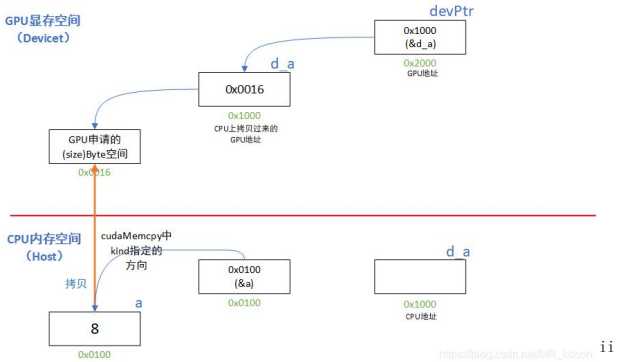
功不可没。



那么，GPU 没有连着 MMU 啊，那可咋整？

一开始我看到确实不行。最早期的 cudaMalloc 以及 cudaHostMalloc 分别只分配 device（GPU）和 Host（CPU）的内存，而想要通信，需要调用 cudaMemcpy 进行拷贝，并且拷贝时，CPU 的指针和 GPU 的指针是无法共通的，或者说，你的 CPU 指针不能指向 GPU 的内存。

此时的 cudaMemcpy 通过一个指向指向 GPU 的指针的指针来运作。我们以 cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice)为例子。a 是一个 GPU 指针，它指向我们的 GPU 内存。d_a 是一个 CPU 指针，它指向的是 a 这个指针。但是，我们不能用 d_a 直接操控 GPU，而是要 a 这个中间商来操控。



在 CUDA 4 后，NVIDIA 引入了 UVA：**同一虚拟寻址**（不是统一），它把 GPU 们和 CPU 连成一块。

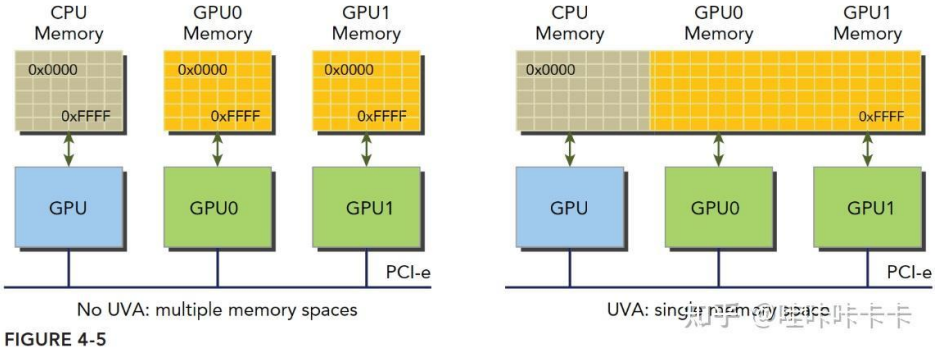
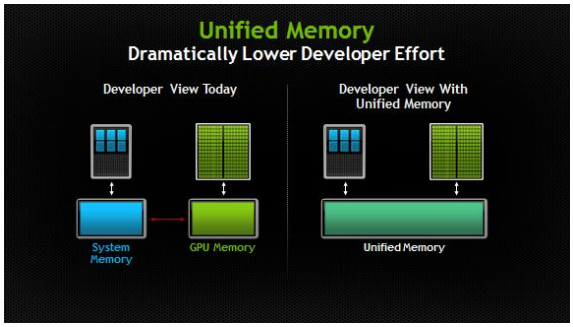


FIGURE 4-5

通过 UVA，cudaHostAlloc 函数分配的固定主机内存具有相同的主机和设备地址，可以直接将返回的地址传递给核函数。

UVA 为系统中的所有内存提供了一个单一的虚拟内存地址空间，无论指针位于系统中的哪个位置，

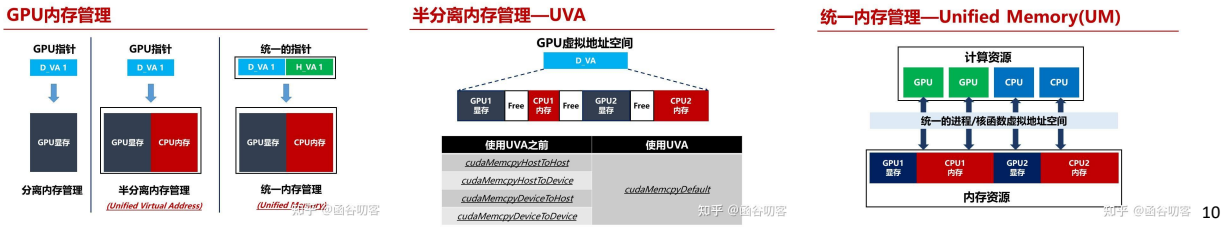
无论是设备内存（在相同或不同的 GPU 上）、主机内存还是片上共享内存，都能从 GPU 代码中访问这些指针。它还允许 cudaMemcpy 在各个设备上的使用，而无需指定输入和输出参数的确切位置。UVA 支持 "零拷贝"（Zero-Copy）内存，即设备代码可通过 **PCIe** 直接访问针脚主机内存，而无需使用 memcpy。零拷贝 "提供了统一内存的一些便利，但没有性能，因为它总是通过 PCI-Express 的低带宽和高延迟进行访问。



到了 CUDA6.0 时代，NVIDIA 做出了**统一内存寻址**（UM）。在 UM 实现中，他们在 CPU 和 GPU 各创建了一个托管内存池，内存池中已分配的空间可以通过相同的指针直接被 CPU 和 GPU 访问，底层系统在统一的内存空间中自动的进行设备和主机间的传输。[9]

UVA 不会像 UM 那样自动将数据从一个物理位置迁移到另一个物理位置。由于 Unified Memory 能够在主机和设备内存之间的单个页面级别自动迁移数据，数据传输对应用是透明的，大大简化了代码。

所以，虽然我们的深拷贝函数只有短短几行，但背后是 ~~NVIDIA 嗯造的屎山~~ NVIDIA 的多代技术累积。我们可以轻松地将自己矩阵声明的指针直接指向 GPU 内，进而才有我们下面的 upload 与 download 函数。



3 数据 upload & download

CUDA 内的矩阵可以被 CPU 内的分配，也可以下载到 CPU。

应用程序可以通过将适当的参数传递给 cuMemAddressReserve 来保留虚拟地址范围。获得的地址范围不会有任何与之关联的设备或主机物理内存。保留的虚拟地址范围可以映射到属于系统中任何设备的内存块，从而为应用程序提供由属于不同设备的内存支持和映射的连续 VA 范围。应用程序应使用 cuMemAddressFree 将虚拟地址范围返回给 CUDA。用户必须确保在调用 cuMemAddressFree 之前未映射整个 VA 范围。这些函数在概念上类似于 mmap/munmap（在 Linux 上）[11]

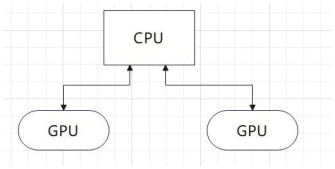
[9] https://developer.nvidia.com/blog/unified-memory-in-cuda-6/
[10] https://zhuanlan.zhihu.com/p/430101220
[11] https://developer.nvidia.com/zh-cn/blog/cuda-virtual-address-management-cn/

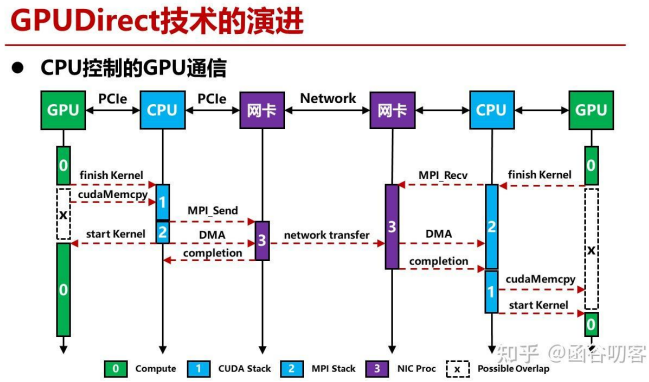
```cpp
/**
@brief 上传数据到 GPU
@param 上传到GPU上的数据的指针
*/
void upload(T* device_data) {
    cudaSetDevice(device);
    cudaMalloc((void**)&device_data, rows * cols * sizeof(T)); // 在 GPU 上分配内存
    cudaMemcpy(device_data, data, rows * cols * sizeof(T), cudaMemcpyHostToDevice); // 复制到设备
    if (data != nullptr && ref_count == 0) { // 把这个矩阵重新写了
        cudaFree(data);
    }
    data = device_data; // 更新数据指针
}

/**
@brief 从 GPU 下载数据到 CPU
@param 返回主机上的数据
@public
*/
T* download() {
    T* host_data = new T[rows * cols]; // 在主机上分配内存
    cudaMemcpy(host_data, data, rows * cols * sizeof(T), cudaMemcpyDeviceToHost); // 复制到主机
    return host_data;
}
```

4 在多张 GPU 上分配矩阵

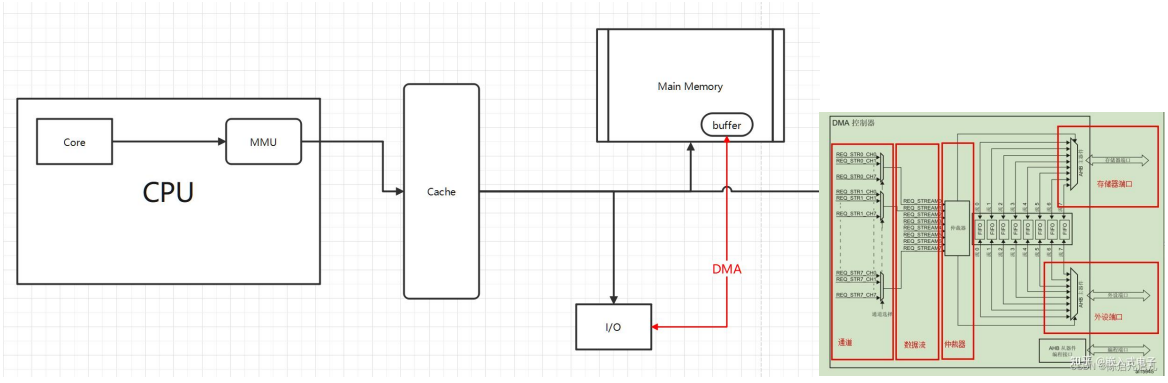
好了，刚刚我们了解了 CPU 与 GPU 间的内存交互故事。那么，GPU 跟 GPU 间可以相互交互内存吗？

最早期的 GPU 间通信还是依靠 CPU，GPU 把数据上传到 CPU 后，CPU 再把数据发送给另一个 GPU。这种方法的效率非常低下。所以很快就被替代掉了。

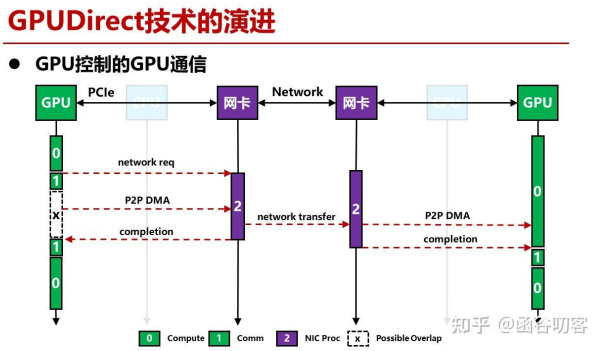


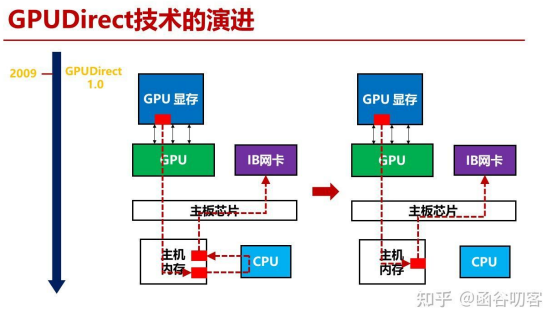


最早绕过 CPU 执行存储访问的，是一个叫 DMA 的东西。DMA 用于在外设与存储器之间以及存储器与存储器之间提供高速数据传输。可以在无需任何 CPU 操作的情况下通过 DMA 快速移动数据。这样节省的 CPU 资源可供其它操作使用。这个东西被广泛用在 CPU 能力较弱的单片机上，比如 stm32。

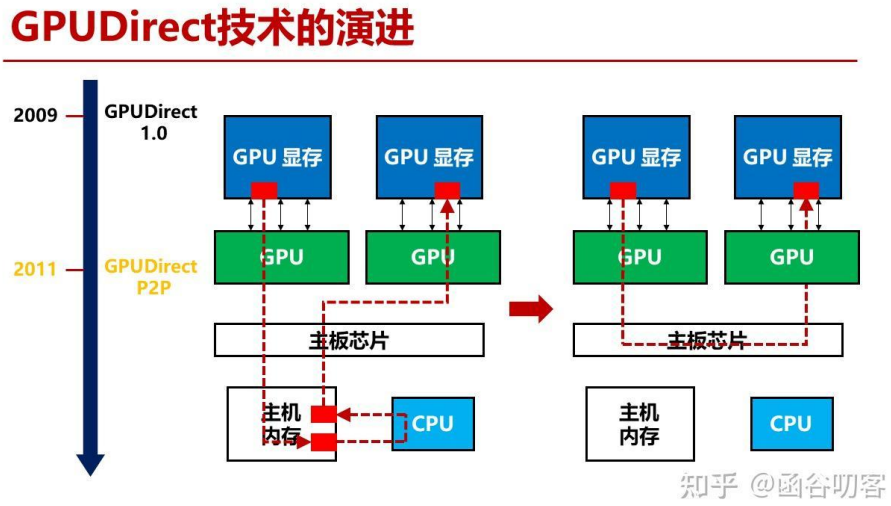


由此，NVIDIA 工程师想到了用 DMA，将 GPU 与网卡连接起来。实现 GPU-网卡-网卡-GPU 通信。就这样，2009 年，GPUDirect1.0 诞生了。

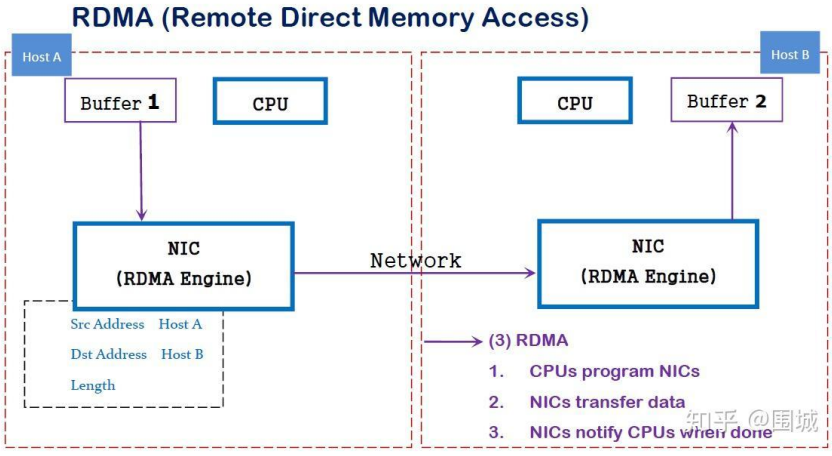




在这一思想驱动下，二代技术很快出现。第二代 GPUDirect 技术被称作 GPUDirect P2P，重点解决的是节点内 GPU 通信问题。两个 GPU 可以通过 PCIe P2P 直接进行数据搬移，避免了主机内存和 CPU 的参与。



一台机子上不同设备可以访问内存，那，多台机可不可以？这就是 RDMA。



在网络传输中，传统的 TCP/IP 技术在数据包处理过程中，要经过操作系统及其他软件层，需要占用大量的服务器资源和内存总线带宽，数据在系统内存、处理器缓存和网络控制器缓存之间来回进行复制移动，给服务器的 CPU 和内存造成了沉重负担。

RDMA 是一种新的直接内存访问技术，让计算机可以直接存取其他计算机的内存，而不需要经过处理器的处理，不对操作系统造成任何影响。

在实现上，RDMA 实际上是一种网卡与软件架构充分优化的远端内存直接高速访问技术，通过将 RDMA 协议固化于硬件(即网卡)上，以及支持 Zero-copy 和 Kernel bypass 这两种途径来达到其高性能的远程直接数据存取的目标。使用 RDMA 的优势如下：

零拷贝(Zero-copy) - 应用程序能够直接执行数据传输，在不涉及到网络软件栈的情况下。数据能够被直接发送到缓冲区或者能够直接从缓冲区里接收，而不需要被复制到网络层。

内核旁路(Kernel bypass) - 应用程序可以直接在用户态执行数据传输，不需要在内核态与用户态之间**做上下文切换**。

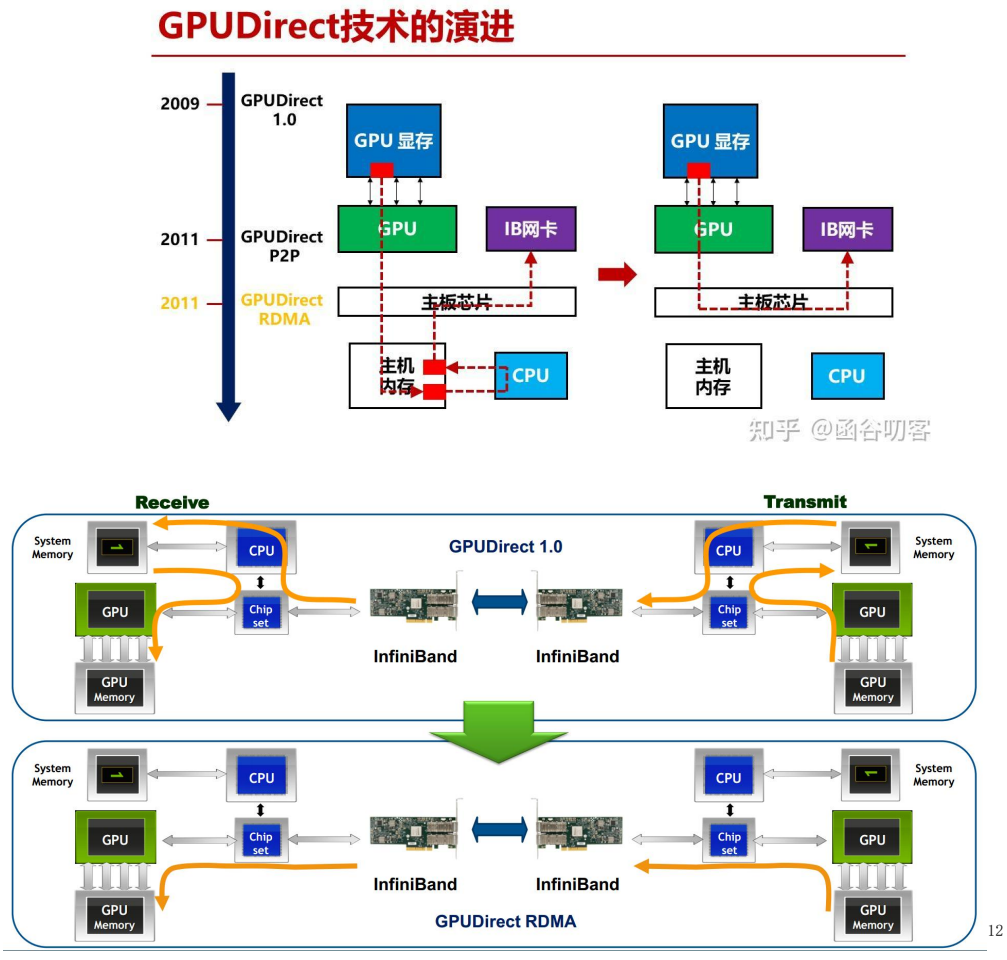
不需要 CPU 干预(No CPU involvement) - 应用程序可以访问远程主机内存而不消耗远程主机中的任何 CPU。远程主机内存能够被读取而不需要远程主机上的进程（或 CPU)参与。远程主机的 CPU 的缓存(cache)不会被访问的内存内容所填充。

消息基于事务(Message based transactions) - 数据被处理为离散消息而不是流，消除了应用程序将流切割为不同消息/事务的需求。

支持分散/聚合条目(Scatter/gather entries support) - RDMA 原生态支持分散/聚合。也就是说，读取多个内存缓冲区然后作为一个流发出去或者接收一个流然后写入到多个内存缓冲区里去。

在主流的 RDMA 技术中，可以划分为两大阵营。一个是 IB 技术，另一个是支持 RDMA 的以太网技术(RoCE 和 iWARP)。

NVIDIA 一看：这东西好啊！于是做出了 GPUDirect RDMA。将 IB 互相连接起来，就实现了 GPU 间的通信。

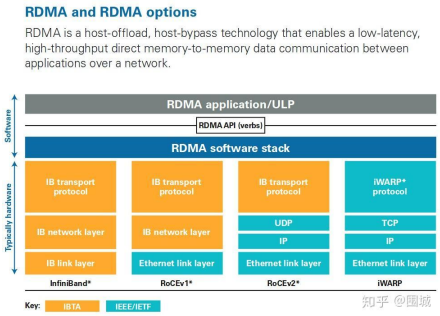


[12] https://zhuanlan.zhihu.com/p/55142557

在两个对等体之间设置 GPUDirect RDMA 通信时，从 PCI Express 设备的角度来看，所有物理地址都是相同的。传统上，BAR 窗口等资源使用 CPU 的 MMU 作为内存映射 I/O （MMIO） 地址映射到用户或内核地址空间。但是，由于当前操作系统没有足够的机制在驱动程序之间交换 MMIO 区域，因此 NVIDIA 内核驱动程序会导出函数以执行必要的地址转换和映射[13]。

在很多时候，我们的计算需要 GPU-GPU 的 P2P 通信，NVIDIA 就做出了 GPUDirect P2P。GPUDirect P2P 通信技术将数据从源 GPU 复制到同一节点中的另一个 GPU，不再需要将数据临时暂存到主机内存中。如果两个 GPU 连接到同一 PCIe 总线，GPUDirect P2P 允许访问其相应的内存，而无需 CPU 参与。

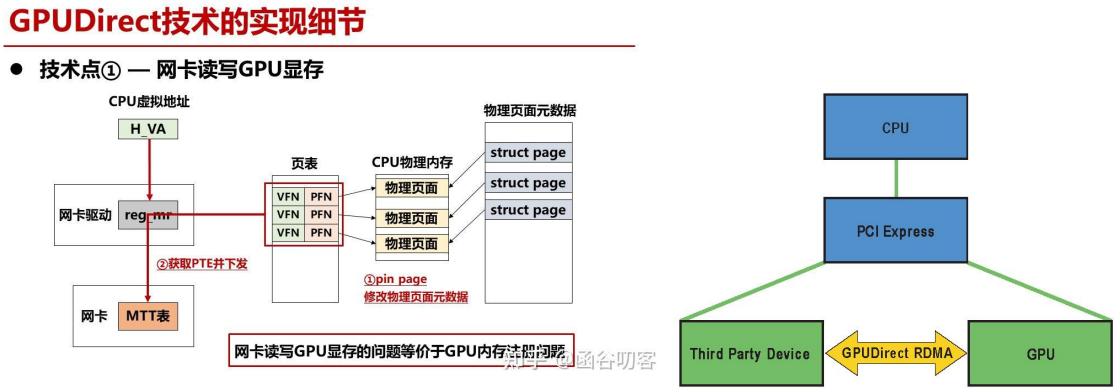
我们实现了基于 P2P 通信的一个 changeGPU 函数，它可以将原本放在 GPU0 的矩阵转移到 GPU1 去。




```cpp
void changeGPU(int device_num) {
    // 在GPU num上分配内存用于存储目标矩阵
    T* deviceDestMatrix;
    cudaSetDevice(device_num); // 选择GPU
    cudaMalloc(&deviceDestMatrix, rows * cols * sizeof(T));

    // 将源矩阵数据从GPU0复制到GPU1
    cudaSetDevice(device); // 选择原来的GPU

    // from device to device_num
    cudaMemcpyPeer(deviceDestMatrix, device_num, data, device, rows * cols * sizeof(T));


    // 释放device上的目标矩阵内存
    cudaSetDevice(device_num); // 选择GPU1
    cudaFree(deviceDestMatrix);
}
```

这里边的核心函数就是 CudaMemcpyPeer，将 GPU device 的数据拷贝到 GPU device_num 上。

由于篇幅限制，对 Direct 技术的介绍就暂时到这里。但是 https://zhuanlan.zhihu.com/p/430101220 的介绍真的非常精彩。



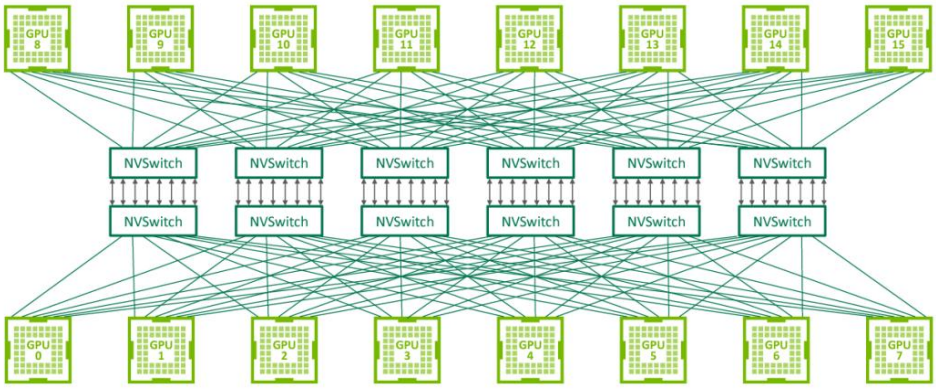
在刚刚的通信中，PCIe3.0*16 的双向带宽不足 32GB/s，当训练数据不断增长时，PCIe 的带宽满足不了需求，会逐渐成为系统瓶颈。为提升多 GPU 之间的通信性能，充分发挥 GPU 的计算性能，NVIDIA 于 2016 年发布了全新架构的 NVLink。NVLink 是一种高速、高带宽的互连技术，用于连接多个 GPU 之间或连接 GPU 与其他设备（如 CPU、内存等）之间的通信。NVLink 提供了直接的点对点连接，具有

[13] https://docs.nvidia.com/cuda/gpudirect-rdma/index.html#how-gpudirect-rdma-works

比传统的 PCIe 总线更高的传输速度和更低的延迟。

NVLink 支持多种连接配置，包括 2、4、6 或 8 个通道，可以根据需要进行灵活的配置和扩展。这使得 NVLink 适用于不同规模和需求的系统配置。

但是，只有 8 个通道还是太少了。NVIDIA 在 2018 年又发布了 NVSwitch，实现了 NVLink 的全连接。NVIDIA NVSwitch 是首款节点交换架构，可支持单个服务器节点中 16 个全互联的 GPU，并可使全部 8 个 GPU 对分别达到 300GB/s 的速度同时进行通信。



当然，我们这里就用不上了，但是我们可以看到，通信技术的不断增长，背后影射的是计算速度的提升与需求的提高，他们推动着通信的研究。

5. 类型转换 & 计算

我们在设计类时为了方便，在 CPU 上的 C++使用了泛型。因此，它只能支持同类型（int + int, float+float）等的转换。不过这问题不大，反正到计算时 GPU 也只能进行这种操作。

我们这里没有对核函数进行优化，他们将在 Project5 中使用。这样，Project4 的矩阵类也可以作为 Project5 的模板，非常的方便。

```cpp
template <typename T>
__global__
void matrixAddKernel(T *a, T *b, T *result, size_t rows, size_t cols) {
    size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    size_t stride = blockDim.x * gridDim.x;

    for (size_t i = idx; i < rows * cols; i += stride) {
        result[i] = a[i] + b[i];
    }
}
```

```cpp
template <typename T>
__global__
void matrixSubtractKernel(T *a, T *b, T *result, size_t rows, size_t cols) {
    size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    size_t stride = blockDim.x * gridDim.x;

    for (size_t i = idx; i < rows * cols; i += stride) {
        result[i] = a[i] - b[i];
    }
}

```

```cpp
template <typename T>
__global__
void matrixMulKernel(T *a, T *b, T *result, size_t aRows, size_t aCols, size_t bCols) {
    size_t row = blockIdx.y * blockDim.y + threadIdx.y;
    size_t col = blockIdx.x * blockDim.x + threadIdx.x;// 尝试了改变顺序，但是都很大

    if (row < aRows && col < bCols) {
        T sum = 0;
        for (size_t k = 0; k < aCols; ++k) {
            sum += a[row * aCols + k] * b[k * bCols + col];
        }
        result[row * bCols + col] = sum;
    }
}
```

另外提供了一些小函数，比如 LU 分解。

```cpp
template <typename T>
__global__ void LUDecomposition(T *A, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    // Perform LU decomposition
    for (int k = 0; k < n - 1; k++) {
        if (tid >= k + 1 && tid < n) {
            A[tid * n + k] /= A[k * n + k];
            for (int i = k + 1; i < n; i++) {
                if (tid >= i && tid < n) {
                    A[tid * n + i] -= A[tid * n + k] * A[k * n + i];
                }
            }
        }
        __syncthreads();
    }
}
```

6. 重载运算符

重载了+，*，-，=，()等运算符。注意到在加减乘方法中，我们声明了一个新矩阵。

```cpp
// Matrix multiplication
    Matrix operator*(const Matrix& other) {
        if (cols != other.rows) {
            std::cerr << "Matrix dimensions do not match for multiplication!\n";
            return Matrix(0, 0, false);// Return an empty matrix
        }

        Matrix result(rows, other.cols);

        // Launch CUDA kernel for matrix multiplication
        dim3 blockDim(32, 32);
        dim3 gridDim((result.getCols() + blockDim.x - 1) / blockDim.x, (result.getRows() + blockDim.y - 1) / blockDim.y);
        matrixMulKernel<<<gridDim, blockDim>>>
                (data, other.data, result.data, rows, cols, other.cols);
        cudaDeviceSynchronize();

        // *result.ref_count += 1;

        return result;
    }
```

是的没错，如果我们没有之前的软 copy 的话，我们的矩阵将会把数值重新复制一遍。但是在重写了拷贝构造函数后，我们的矩阵只会复制指针。在 result 矩阵生命周期停止时，它只会将矩阵的指针传递给我们的结果 Matrix，不会对大数据造成影响。

```
Matrix operator*(const Matrix& other) {
    if (cols != other.rows) {
        std::cerr << "Matrix dimensions do not match
        return Matrix(0, 0, false);// Return an empty
    }

    Matrix result(rows, other.cols);

    // Launch CUDA kernel for matrix multiplication
    dim3 blockDim(32, 32);
    dim3 gridDim((result.getCols() + blockDim.x - 1)
    matrixMulKernel<<<gridDim, blockDim>>>
            (data, other.data, result.data, rows, col
```

```
Matrix<float> E = A;    // copy constructor
Matrix<float> D = A * B;    // 重写*
```

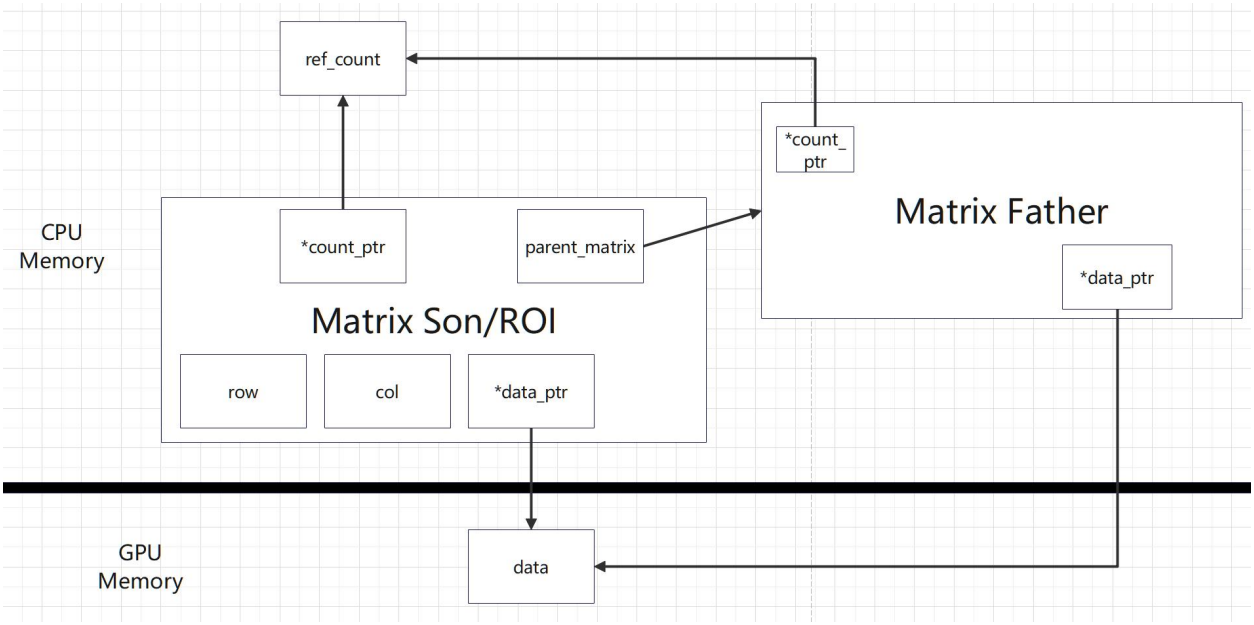
```
/**
 * @brief
 @param 拷贝构造函数，增加引用计数
 */
Matrix(const Matrix& other) : rows(other.rows), cols(other.cols), data(other.data) {
    std::cout<< "Hi from copy constructor\n";
    device = other.device;
    ref_count = (other.ref_count);
    (*ref_count) += 1;
}
```

我们的 Matrix 类实现了拷贝构造函数和赋值运算符重载，返回的 result 对象会被正确地拷贝到 C 中，而非直接赋值。

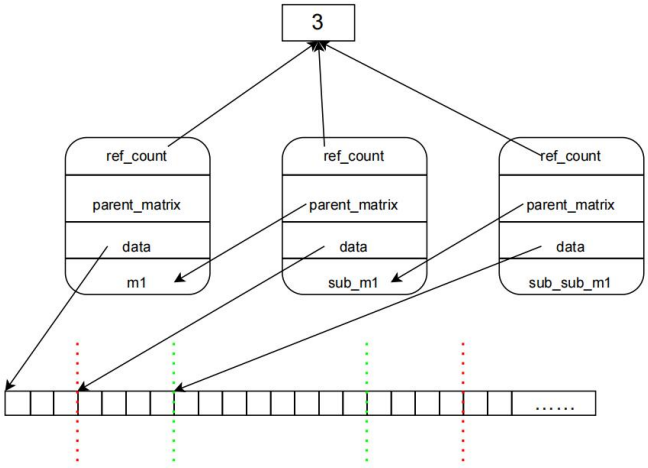
```
Matrix& operator=(const Matrix& other) {
    if (this != &other) { // 检查自我赋值
        (*ref_count)--;
        if (data != nullptr && *ref_count==0) {
            cudaFree(data);
            free(ref_count);
        }
        std::cout << "Hi from =\n";
        rows = other.rows;
        cols = other.cols;
        data = other.data;
        ref_count = other.ref_count;
        (*ref_count)++;
    }
    return *this;
}
```

6. ROI

回到我们的设计图中，没错，我们将 ROI 看做是另一个矩阵的子矩阵，或者说我们是对同样一份数据但是指针位置不同的矩阵。他们共享一块 ref_count 和 data，他们的指针指向同一个方向的数据。但是他们的



ROI 会遇到一个问题，就是当我们有多个 ROI 时，我们应该以根矩阵为“参考系”，来计算我们感兴趣的地区。这是董学长在他的报告中指出的：



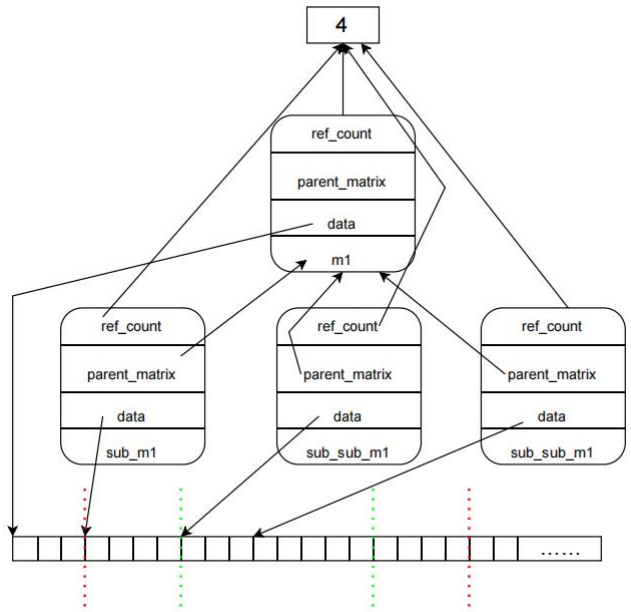
It is unacceptable because `sub_sub_m1[i][j]==sub_sub_m1.data[i * sub_sub_m1->parent_matrix->ncols + j]==sub_sub_m1.data[i * sub_m1.ncols + j]` and it is wrong.

为此我们设计了一个寻找根矩阵的流程。

```
const Matrix* par = &other;  // 获取给定矩阵的父矩阵指针

// 找到给定矩阵的根矩阵（没有父矩阵的矩阵）
while (par->parent_matrix != nullptr) {
    par = par->parent_matrix;
}

// 将当前矩阵的父矩阵指针设置为根矩阵
parent_matrix = par;
```

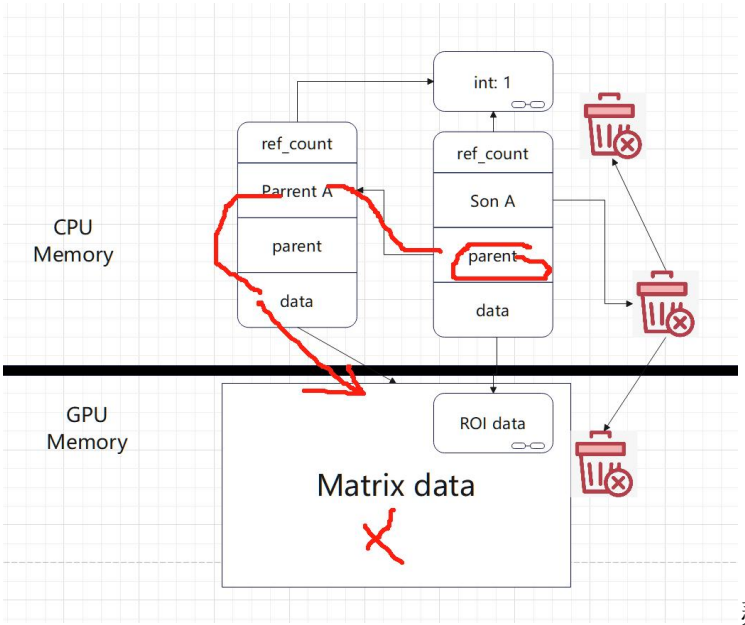
```cpp
// 设置ROI（Region of Interest）
    void setROIas(size_t startRow, size_t startCol,size_t numRows,size_t numCols, const Matrix& other) {

        if (data != nullptr && *ref_count==0) {
            cudaFree(data);
            free(ref_count);
        }

        const Matrix* par = &other;  // 获取给定矩阵的父矩阵指针

        // 找到给定矩阵的根矩阵（没有父矩阵的矩阵）
        while (par->parent_matrix != nullptr) {
            par = par->parent_matrix;
        }

        // 将当前矩阵的父矩阵指针设置为根矩阵
        parent_matrix = par;

        if (parent_matrix == nullptr) {
            throw std::logic_error("Cannot set ROI on a non-parent matrix");
        }

        // 合法性检查
        if (startRow + numRows > parent_matrix->getRows() || startCol + numCols > parent_matrix->getCols()) {
            throw std::out_of_range("ROI out of bounds of parent matrix");
        }
        *(parent_matrix->ref_count) +=1;
        ref_count = parent_matrix->ref_count;

        // 计算数据偏移量
        size_t offset = startRow * parent_matrix->getCols() + startCol;
        // 设置新的数据指针
        data = parent_matrix->getData() + offset;
        // 更新行数和列数
        rows = numRows;
        cols = numCols;
        // 父矩阵指针保持不变
    }
```

这样，我们的 ROI 就可以准确定位到我们最想要的矩阵上了。



第二个问题产生在释放的时候。如果我们的矩阵是 ROI，那么我们如果在释放的时候还是释放的 data*，那我们只是释放了 ROI 的数据，对整个矩阵的数据没有进行释放。

对此，我们可以判断，如果我们的是一个 ROI，我们就销毁父矩阵所指的数据区域，也就是全部的数据区域。



灵魂画手

```cpp
// Destructor
~Matrix() {
    *(ref_count) -= 1;
    if (*(ref_count) == 0 && data != nullptr) {

        if(parent_matrix == nullptr){
            cudaError_t cudaStatus = cudaFree(data);
            if (cudaStatus != cudaSuccess) {
                std::cerr << "CUDA Free Error: " << cudaGetErrorString(cudaStatus) << std::endl;
            }
            free(ref_count);
            ref_count = nullptr;
        }else{
            cudaError_t cudaStatus = cudaFree(parent_matrix->data);
            if (cudaStatus != cudaSuccess) {
                std::cerr << "CUDA Free Error: " << cudaGetErrorString(cudaStatus) << std::endl;
            }
            free(ref_count);
            ref_count = nullptr;
        }
        // std::cout << "hi from destructor\n";
    }
}
```

Part 4: 简单测试

我们在 Main.cu 文件中对矩阵类进行了测试。测试分为可用性测试及速度测试。

1.可用性测试

多类型

测试了 unsigned char, short, int, float, double, 均可使用。同时，没有出现内存泄露及二次释放。

```
end of mainCUDA Free Error: invalid argument
S12211612@lab01:~/Project/Project4/final$ nvcc main.cu -o Main
S12211612@lab01:~/Project/Project4/final$ ./Main
hi from Matrix constructor
 Manage Memory for Matrix!
hi from Matrix constructor
 Manage Memory for Matrix!
Hi from copy constructor
hi from Matrix constructor
 Manage Memory for Matrix!
Hi from copy constructor
END OF CREATING TEST
hi from Matrix constructor
 Manage Memory for Matrix!
hi from Matrix constructor
 Manage Memory for Matrix!
Print TEST
 Matrix A:
Matrix A END
hi from Matrix constructor
 Manage Memory for Matrix!
Hi from copy constructor
Hi from =

Matrix A * B: finish
time=0.000229
end of mainS12211612@lab01:~/Project/Project4/final$
```

多计算

Float，double，int，unsigned char, short。均可计算。不过不同数据类型的无法进行计算。

```cpp
std::cout << "Print TEST \n Matrix A:\n";
A+=B;
// std::cout<< A;
std::cout << "Matrix A END\n";

Matrix<uchar> N(3,3,3);
Matrix<short> L(3,3,3);
Matrix<uchar> P(3,3,3);

Matrix<uchar> N2(3,3,3);

Matrix<uchar> X = N + N2;
```

多卡

测试成功，矩阵 B 移植到了 GPU1 上，与在 GPU0 的矩阵 A 进行运算。有趣的是，A*B 的计算是在

A 上完成的。主要原因是*作为 A 的运算符重载。

```
+-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 2080 Ti     Off |   00000000:04:00.0 Off |                  N/A |
| 30%   48C    P2           193W /  250W |     542MiB /  11264MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA GeForce RTX 2080 Ti     Off |   00000000:0B:00.0 Off |                  N/A |
| 27%   34C    P2            62W /  250W |     158MiB /  11264MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   2  NVIDIA GeForce RTX 2080 Ti     Off |   00000000:13:00.0 Off |                  N/A |
| 27%   31C    P8            13W /  250W |       4MiB /  11264MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   3  NVIDIA GeForce RTX 2080 Ti     Off |   00000000:1B:00.0 Off |                  N/A |
| 28%   34C    P8            19W /  250W |       4MiB /  11264MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A    697888      C   ./Main                                     154MiB |
|    1   N/A  N/A    697888      C   ./Main                                     154MiB |
+---------------------------------------------------------------------------------------+
s12211612@lab01:~/Project/Project4$
```

2.速度测试

在速度测试中，我尝试复现 Project3 中周益贤同学所提到的反写线程降速问题。然而，我发现我并没有遇到这个问题，甚至在调换之后，矩阵乘法所用的时间反而增加了。我认为可能的原因是线程数和块的数量分配问题。

2. 记得我们在 2.3.3.1 CUDA 试水的示例代码中，故意反着写了 `threadIdx.x` 和 `threadIdx.y`，但是当时并未解释。

   这是因为一个非常抽象的现象：如果正常使用它们，效率会变得无比低下。

```
//origin(slow)
size_t x = threadIdx.x + blockDim.x * blockIdx.x;
size_t y = threadIdx.y + blockDim.y * blockIdx.y;
//swap(fast)
size_t x = threadIdx.y + blockDim.x * blockIdx.x;
size_t y = threadIdx.x + blockDim.y * blockIdx.y;
```

GPU 矩阵乘法计算在 1000 前后都是 0.05 秒，在升高到 8000 后来到了 0.8s。注意，这里是矩阵 B 和矩阵 A 在不同的 GPU 存储与运算的效果。基本和周同学的测试结果中的 plain 一致。

Part 5: 总结

在本次 Project 中我们制作了一个 GPU 矩阵类，实现了多数据输入，运算符重载，ROI，内存管理，跨 GPU 运算等操作，有了基本的系统设计概念与 GPU CUDA 编程的了解，对 C/C++的特性了解地更深刻了。

Reference

[1] United States Department of Transportation. Core System Requirements. Available: https://www.its.dot.gov/index.htm

[2] Specification (SyRS)"IEEE Guide for Information Technology - System Definition - Concept of Operations (ConOps) Document," in IEEE Std 1362-1998 , vol., no., pp.1-24, 22 Dec. 1998, doi: 10.1109/IEEESTD.1998.89424. Available: https://ieeexplore.ieee.org/document/761853

[3] ISO/IEC/ IEEE 42010, Systems and software engineering Architecture description. International Standard

[4] 一文彻底理解 DMA - 知乎 (zhihu.com)

[5] 一文搞懂 MMU 工作原理 - 知乎 (zhihu.com)

时间比较赶，所以我都在一文搞懂（乐）。

附录：代码

为了防止我在 Project 上传时 cu 文件无法上传，我先将代码丢在了这里。当然，也可以去 Laihb1106205841/GpuMat.github.io: A matrix class for GPU on SUSTech 2024Spring C/C++ Project4 我会上传对应的代码。

```h
#ifndef MATRIX_H
#define MATRIX_H

#include "h.cu"

#endif  // MATRIX_H
```

```cpp
#include <iostream>

#include <vector>



class Matrix {

private:



    size_t rows;

    size_t cols;

    size_t* ref_count; // 引用计数器

public:

std::vector<std::vector<float>> *data; // 指向矩阵数据的指针

    // 构造函数

    Matrix(size_t rows, size_t cols) : rows(rows), cols(cols) {

        data = new std::vector<std::vector<float>>(rows, std::vector<float>(cols, 0.0f));

        ref_count = new size_t(1); // 初始引用计数为 1

        std::cout << "HI from cons\n";

    }

    // 拷贝构造函数

    Matrix(const Matrix& other) : rows(other.rows), cols(other.cols), data(other.data), ref_count(other.ref_count) {

        (*ref_count)++;

        std::cout << "HI from copy\n";
```

```
}

// 析构函数

~Matrix() {

    (*ref_count)--;

    if (*ref_count == 0) {

        delete data; // 释放数据

        delete ref_count; // 释放引用计数器

        std::cout << "Delete!\n";

    }

    std::cout << "De\n";

}

// 赋值运算符重载

Matrix& operator=(const Matrix& other) {

    if (this != &other) {

        (*ref_count)--;

        if (*ref_count == 0) {

            delete data;

            delete ref_count;

        }

        rows = other.rows;

        cols = other.cols;

        data = other.data;

        ref_count = other.ref_count;

        (*ref_count)++;

    }

    return *this;

}

// 重载 * 运算符以实现矩阵乘法

Matrix operator*(const Matrix& other) {

    if (cols != other.rows) {

        std::cerr << "Error: Matrix dimensions are incompatible for multiplication." << std::endl;

        return Matrix(0, 0); // 返回一个空矩阵表示错误

    }

    Matrix result(rows, other.cols);

    for (size_t i = 0; i < rows; ++i) {

        for (size_t j = 0; j < other.cols; ++j) {

            for (size_t k = 0; k < cols; ++k) {

                result.data->at(i).at(j) += data->at(i).at(k) * other.data->at(k).at(j);

            }

        }

    }

    return result;

}

// 打印矩阵

void print() {

    for (size_t i = 0; i < rows; ++i) {
```

```
            for (size_t j = 0; j < cols; ++j) {
                std::cout << data->at(i).at(j) << " ";
            }
            std::cout << std::endl;
        }
    }
};
int main() {
    Matrix A(2, 3);
    (*A.data) = {{1, 2, 3},
                 {4, 5, 6}};
    Matrix B(3, 2);
    (*B.data) = {{7, 8},
                 {9, 10},
                 {11, 12}};
    std::cout << "Matrix A:" << std::endl;
    A.print();
    std::cout << "\nMatrix B:" << std::endl;
    B.print();
    Matrix C = A * B; // 执行矩阵乘法
    Matrix D = A;
```

```
    std::cout << "\nMatrix C (Result of A * B):" << std::endl;
    C.print();
    return 0;
}
```

还有两篇文章：

---

[i] 内存分配不再神秘：深入剖析 malloc 函数实现原理与机制 - 知乎 (zhihu.com)

[ii] CUDA 编程入门之处理 Device 显存的三个 CUDA API cuda 有哪些 api-CSDN 博客