精简CUDA教程-RuntimeAPI

Hello

1. 根据driver的错误代码处理，同样提出runtime的错误处理

```
#define checkRuntime(op)  __check_cuda_runtime((op), #op, __FILE__, __LINE__)

bool __check_cuda_runtime(cudaError_t code, const char* op, const char* file, int line){
    if(code != cudaSuccess){
        const char* err_name = cudaGetErrorName(code);
        const char* err_message = cudaGetErrorString(code);
        printf("runtime error %s:%d  %s failed. \n  code = %s, message = %s\n", file, line, op, er
        return false;
    }
    return true;
}
```

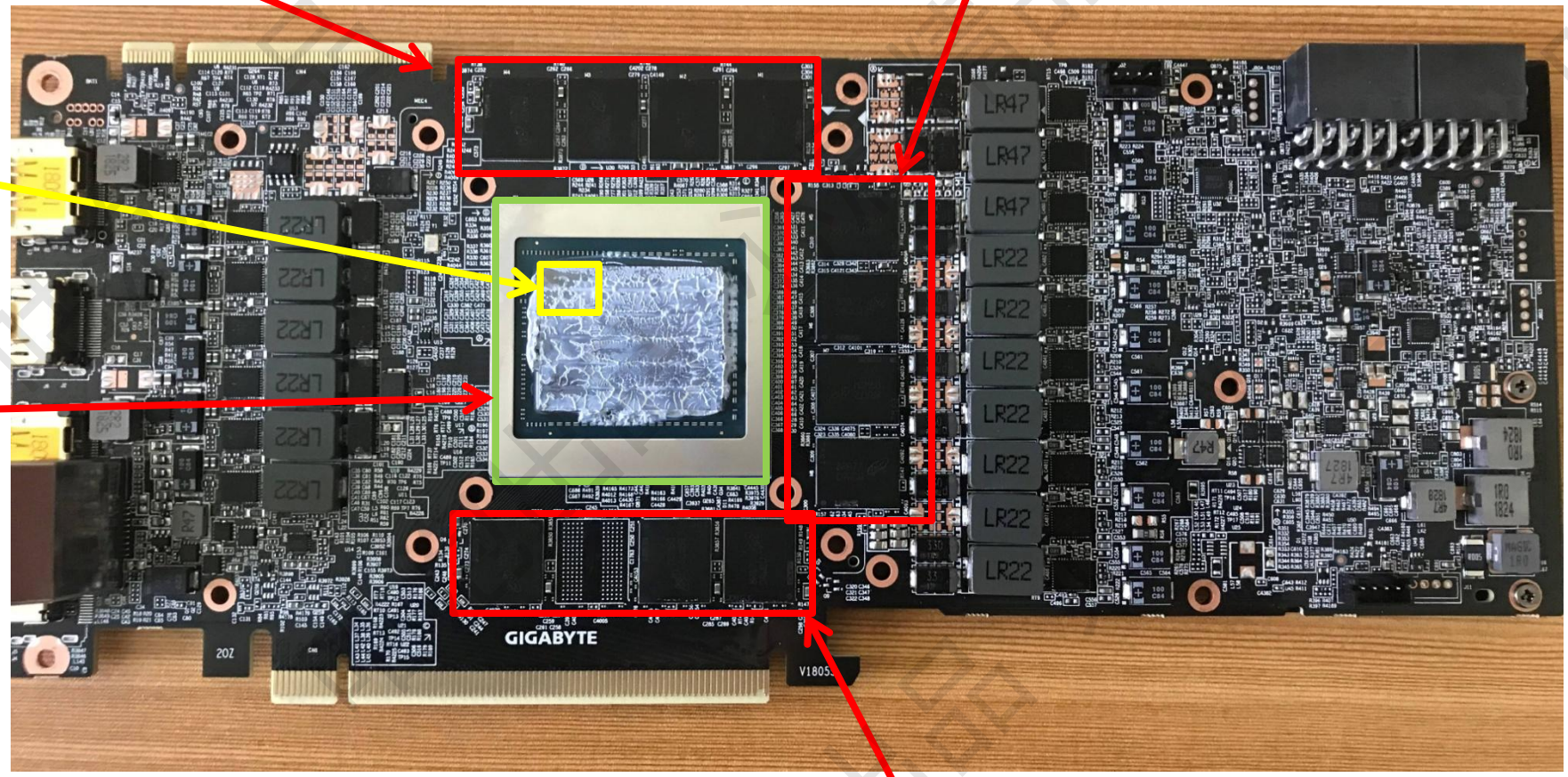
Memory

1. 内存模型是CUDA中很重要的知识点
2. 主要理解pinned memory、global memory、shared memory即可，其他不常用

CUDA内存模型

内存类型	是否在芯片上	缓存	访问方式	作用范围	生命周期	大小	速度	备注
Register	在	无	读/写	单个线程内	线程	每个线程63个 etc.	最快	由编译器决定如何使用寄存器，当寄存器不够用会使用local memory
Local	不在	无	读/写	单个线程内	线程	可用内存/SM数量/SM最大常驻线程数，81KB etc.	普通	本质是与global memory在一起。局部定义的变量，或者寄存器不够用时会使用到
Shared	在	无	读/写	block内的所有线程	block	48KB etc.	快	block内的所有线程共享访问，由__shared__修饰的变量
Global	不在	无	读/写	所有线程 + 主机	主机分配	11GB etc.	普通	全局所有线程任意时候都可以访问，由cudaMalloc、cudaMallocHost、cudaHostAlloc、cudaMallocManaged所涉及分配，具体还有Pinned Memory、Zero Copy Memory、Unified Memory
Constant	不在	有	读	所有线程 + 主机	主机分配	64KB etc.	快	只读内存，全局所有线程都可以访问，由cudaMemcpyToSymbol进行初始化
Texture	不在	有	读	所有线程 + 主机	主机分配	n/o	快	只读内存，全局所有线程都可以访问。针对3d编程使用

Memory



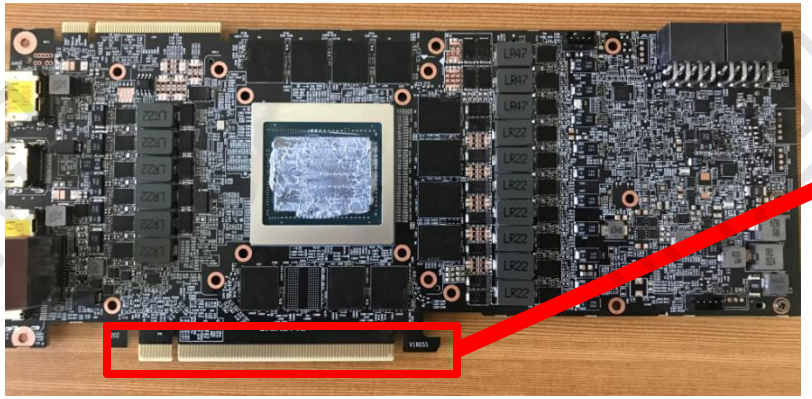
这是一个不太严谨但是辅助理解的图

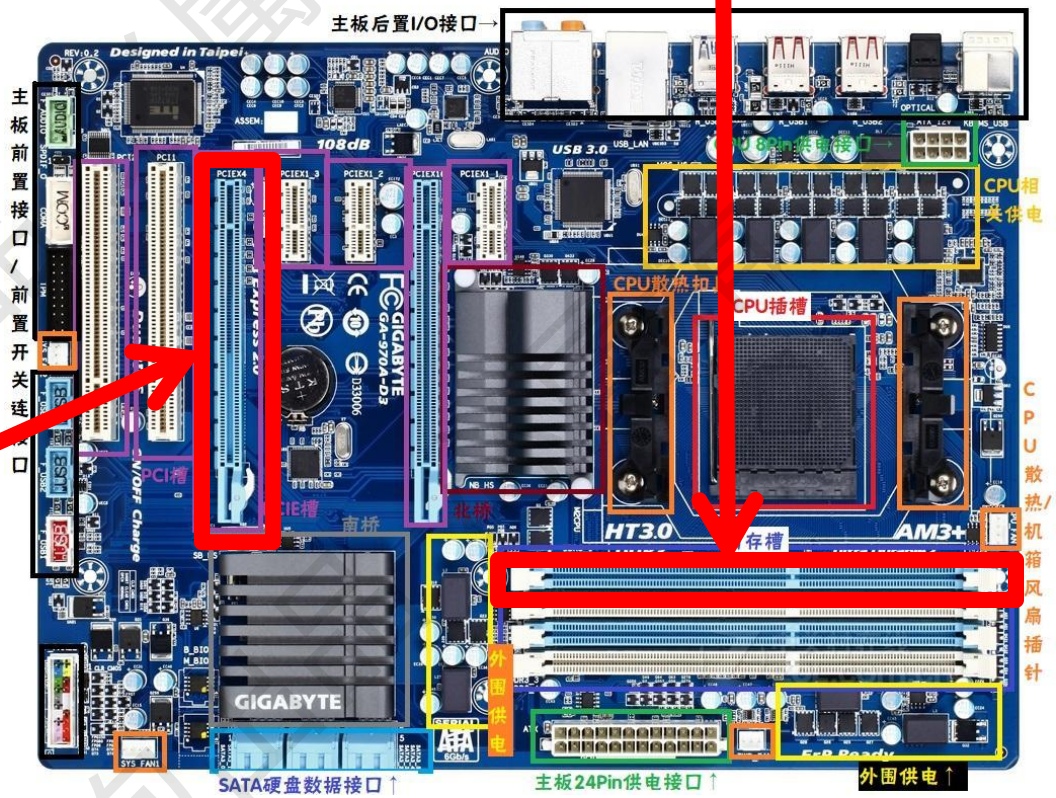
Memory

主机的内存
host memory
cpu内存
pinned memory

内存条

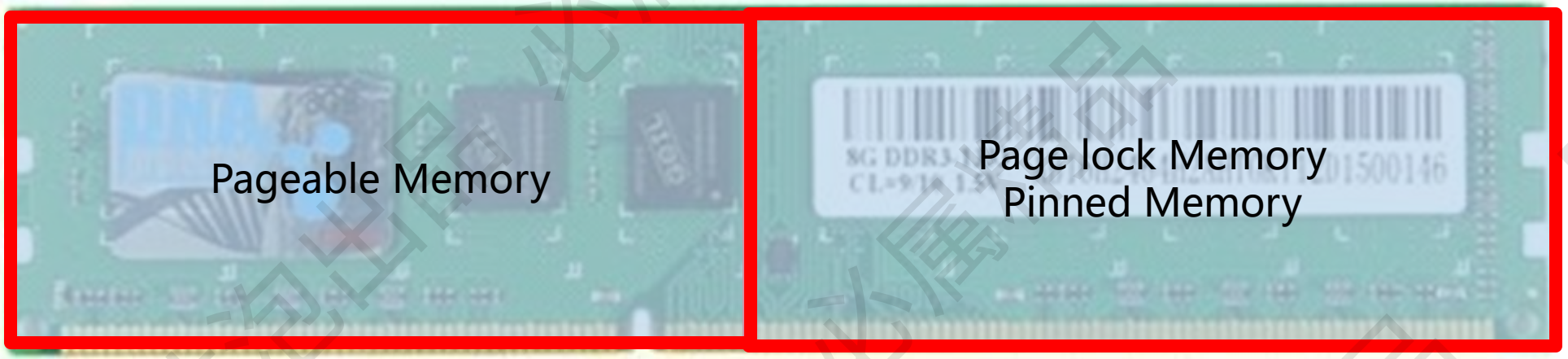
显卡





电脑主板

Pinned Memory



对于整个Host Memory内存条而言，操作系统区分为两个大类（逻辑区分，物理上是同一个东西）：

1. Pageable memory，可分页内存
2. Page lock memory，页锁定内存

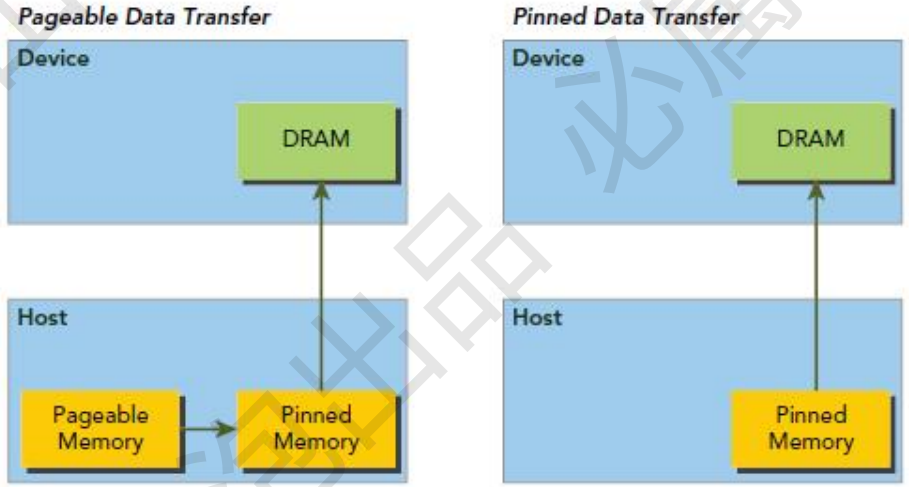
你可以理解为Page lock memory是vip房间，锁定给你一个人用。而Pageable memory是普通房间，在酒店房间不够的时候，选择性的把你的房间腾出来给其他人交换用，这就可以容纳更多人了。**造成房间很多的假象，代价是性能降低**

Pinned Memory

基于前面的理解，我们总结如下：

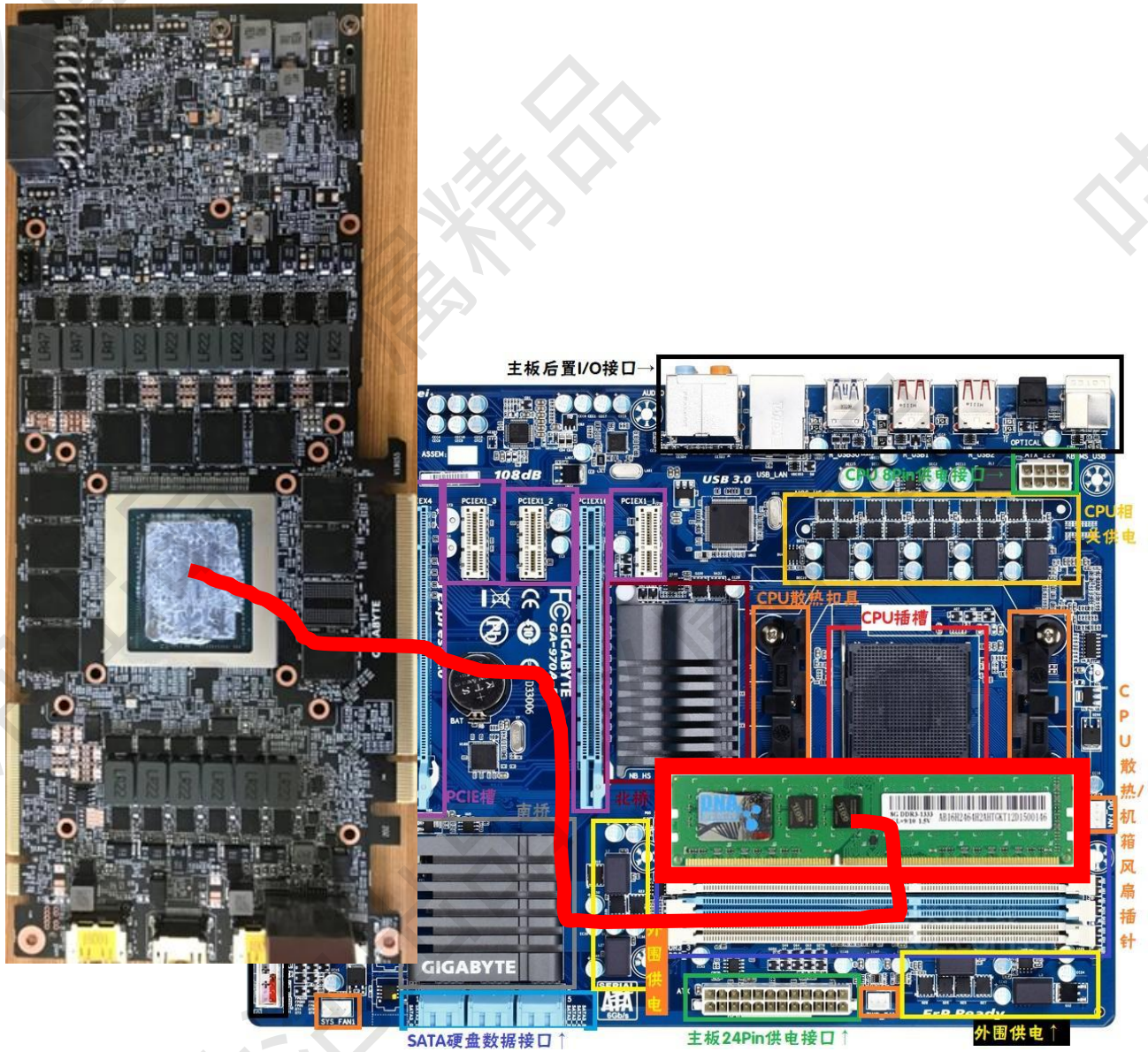
1. pinned memory具有锁定特性，是稳定不会被交换的（这很重要，相当于每次去这个房间都一定能找到你）
2. pageable memory没有锁定特性，对于第三方设备（比如GPU），去访问时，因为无法感知内存是否被交换，可能得不到正确的数据（每次去房间找，说不准你的房间被人交换了）
3. pageable memory的性能比pinned memory差，很可能降低你程序的优先级然后把内存交换给别人用
4. pageable memory策略能使用内存假象，实际8GB但是可以使用15GB，提高程序运行数量（不是速度）
5. pinned memory太多，会导致操作系统整体性能降低（程序运行数量减少），8GB就只能用8GB。注意不是你的应用程序性能降低，这一点一般都是废话，不用当回事
6. **GPU可以直接访问pinned memory而不能访问pageable memory（因为第二条）**

数据传输到GPU



显卡访问Pinned Memory轨迹

通过PICE接口，到主板，再到内存条



内存方面总结

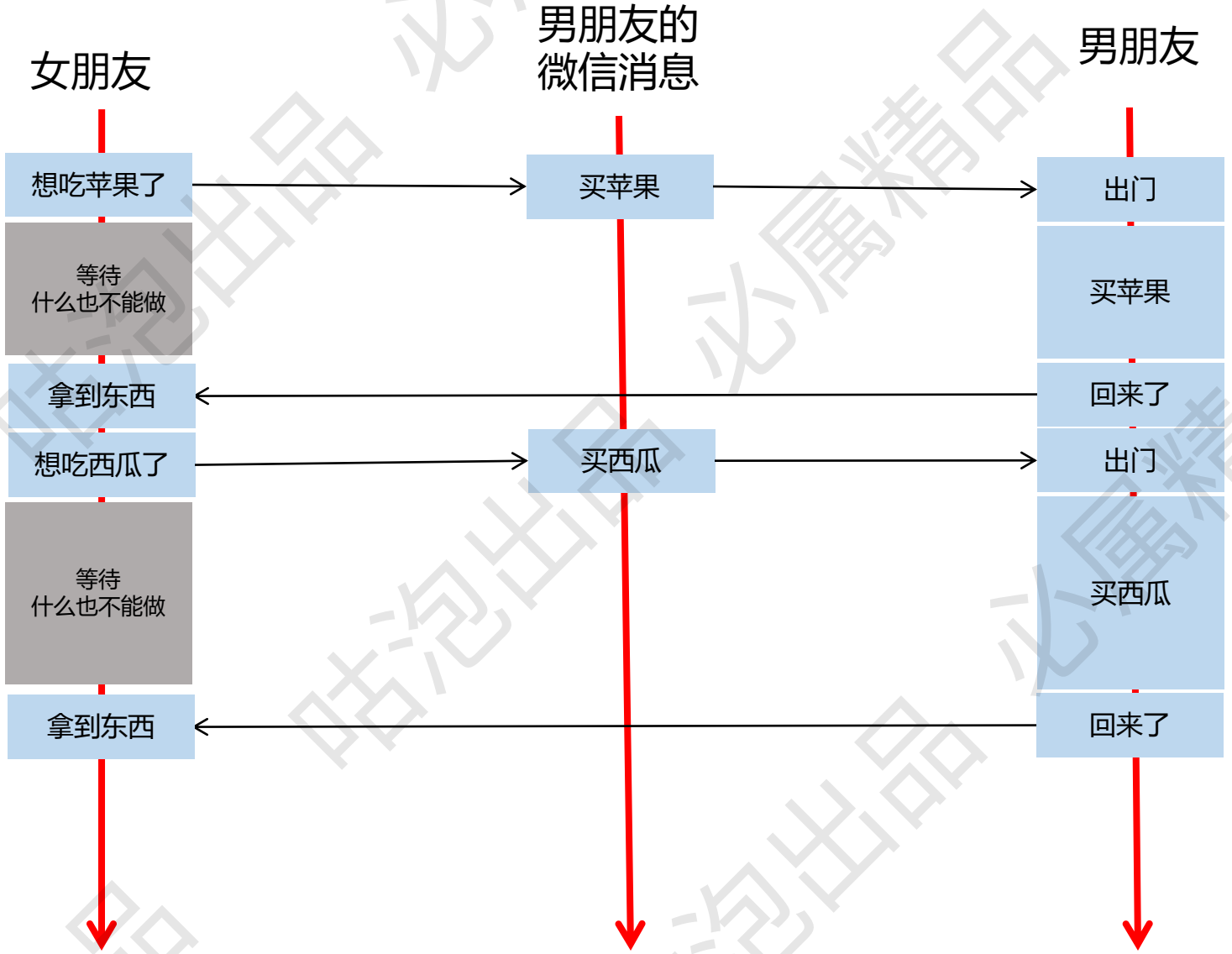
原则：

1. GPU可以直接访问pinned memory，称之为（DMA Direct Memory Access）
2. 对于GPU访问而言，距离计算单元越近，效率越高，所以 PinnedMemory<GlobalMemory<SharedMemory
3. 代码中，由new、malloc分配的，是pageable memory，由cudaMallocHost分配的是PinnedMemory，由cudaMalloc分配的是GlobalMemory
4. 尽量多用PinnedMemory储存host数据，或者显式处理Host到Device时，用PinnedMemory做缓存，都是提高性能的关键

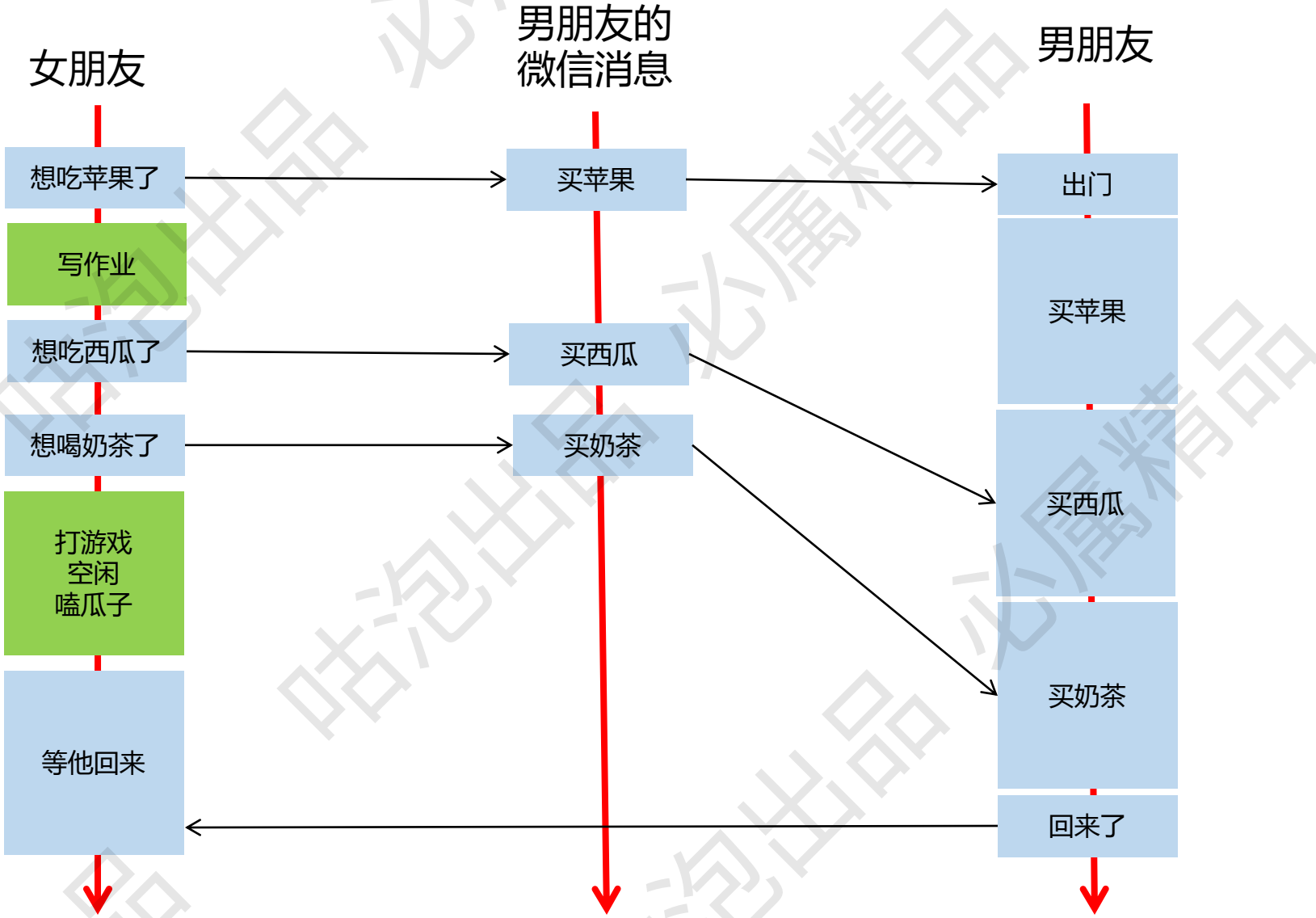
stream - 流

1. 流是一种基于context之上的任务管道抽象，一个context可以创建n个流
2. 流是异步控制的主要方式
3. nullptr表示默认流，每个线程都有自己的默认流

流 - 女朋友让男朋友买东西-时序图



流 - 女朋友让男朋友买东西-时序图



stream - 流

1. 上面的例子中，**男朋友的微信消息**，就是任务队列，流的一种抽象
2. 女朋友发出指令后，他可以做任何事情，无需等待指令执行完毕，（指令发出的耗时也是极短的）
3. 即，异步操作，执行的代码，加入流的队列后，立即返回，不耽误时间
4. 女朋友发的指令被送到流中排队，男朋友根据流的队列，**顺序执行**
5. 女朋友选择性，**在需要的时候**等待所有的执行结果
6. 新建一个流，就是新建一个男朋友，给他发指令就是给他发微信，**你可以新建很多个男朋友**
7. 通过cudaEvent可以选择性等待任务队列中的部分任务是否就绪

stream - 流

```cpp
int main(){

    int device_id = 0;
    checkRuntime(cudaSetDevice(device_id));

    cudaStream_t stream = nullptr;
    checkRuntime(cudaStreamCreate(&stream));

    // 在GPU上开辟空间
    float* memory_device = nullptr;
    checkRuntime(cudaMalloc(&memory_device, 100 * sizeof(float)));

    // 在CPU上开辟空间并且放数据进去，将数据复制到GPU
    float* memory_host = new float[100];
    memory_host[2] = 520.25;
    checkRuntime(cudaMemcpyAsync(memory_device, memory_host, sizeof(float) * 100, cudaMemcpyHostToDevice, stream)); // 异步复

    // 在CPU上开辟pin memory,并将GPU上的数据复制回来
    float* memory_page_locked = nullptr;
    checkRuntime(cudaMallocHost(&memory_page_locked, 100 * sizeof(float)));
    checkRuntime(cudaMemcpyAsync(memory_page_locked, memory_device, sizeof(float) * 100, cudaMemcpyDeviceToHost, stream));
    checkRuntime(cudaStreamSynchronize(stream));

    printf("%f\n", memory_page_locked[2]);

    // 释放内存
    checkRuntime(cudaFreeHost(memory_page_locked));
    checkRuntime(cudaFree(memory_device));
    checkRuntime(cudaStreamDestroy(stream));
    delete [] memory_host;
    return 0;
}
```

异步复制时，发出指令立即返回，并不等待复制完成

同样不等待复制完成，但是在流中排队

统一等待流队列中所有操作结束

stream - 流

1. 要十分注意，指令发出后，流队列中储存的是指令参数，**不能加入队列后立即释放参数指针**，这会导致流队列执行该指令时指针失效而出错
2. 应当在十分肯定流已经不需要这个指针后，才进行修改或者释放，否则会有非预期结果出现
3. 举个粒子：你给钱让男朋友买西瓜，他刚到店拿好西瓜，你把转的钱撤回去了。此时你无法预知他是否会跟店家闹起来矛盾，还是屁颠的回去。如果想得到预期结果，必须得让卖西瓜结束再处理钱的事情

核函数

1. 核函数是cuda编程的关键
2. 通过xxx.cu创建一个cudac程序文件，并把cu交给nvcc编译，才能识别cuda语法
3. __global__表示为核函数，由host调用。__device__表示为设备函数，由device调用
4. __host__表示为主机函数，由host调用。__shared__表示变量为共享变量
5. host调用核函数：function<<<**gridDim**, **blockDim**, sharedMemorySize, stream>>>(args...);
6. 只有__global__修饰的函数才可以用<<<>>>的方式调用
7. 调用核函数是传值的，不能传引用，可以传递类、结构体等，核函数可以是模板
8. 核函数的执行，是异步的，也就是立即返回的
9. 线程layout主要用到**blockDim**、**gridDim**
10. 核函数内访问线程索引主要用到**threadIdx**、**blockIdx**、**blockDim**、**gridDim**这些内置变量

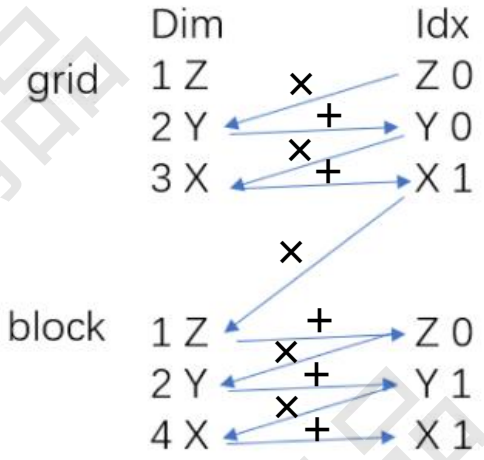
核函数

1. 核函数里面，把blockDim、gridDim看作shape，把threadIdx、blockIdx看做index
2. 则可以按照维度高低排序看待这个信息：

dims	indexs
gridDim.z	blockIdx.z
gridDim.y	blockIdx.y
gridDim.x	blockIdx.x
blockDim.z	threadIdx.z
blockDim.y	threadIdx.y
blockDim.x	threadIdx.x

```
Pseudo code:
position = 0
for i in 6:
    position *= dims[i]
    position += indexs[i]
```

把内存地址看作长条数组
则通用的内存定位计算如上



方便的记忆办法，是左乘右加

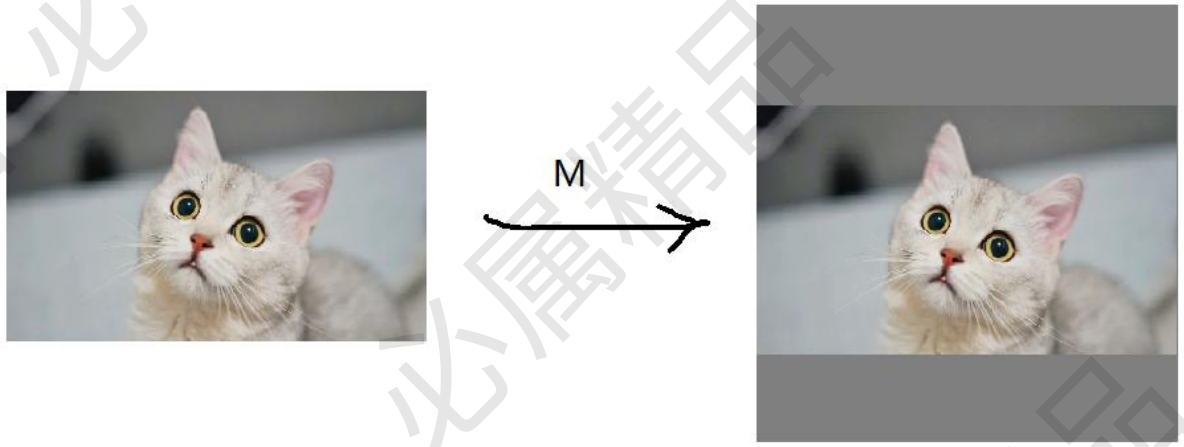
无论tensor维度多复杂，这个方法都适用

共享内存

1. 共享内存因为更靠近计算单元，所以访问速度更快
2. 共享内存通常可以作为访问全局内存的缓存使用
3. 可以利用共享内存实现线程间的通信
4. 通常与__syncthreads同时出现，这个函数是同步block内的所有线程，全部执行到这一行才往下走
5. 使用方式，通常是在线程id为0的时候从global memory取值，然后syncthreads，然后再使用

Warpaffine

主要解决图像的缩放和平移
来处理目标检测中常见的
预处理行为



Warpaffine

1. warpaffine是对图像做**平移缩放旋转**变换进行**综合统一**描述的方法
2. 同时也是一个很容易实现cuda并行加速的算法
3. 在深度学习领域通常需要做预处理，比如CopyMakeBorder，RGB->BGR，减去均值除以标准差，BGRBGRBGR -> BBBGGGRRR
4. 如果使用cuda进行并行加速实现，那么可以对整个预处理都进行统一，并且性能贼好
5. 由于warpaffine是标准的矩阵映射坐标，并且可逆，所以逆变换就是其变换矩阵的逆矩阵
6. **对于缩放和平移的变换矩阵，其有效自由度为3**

scale	0	**ox**
0	**scale**	**oy**
0	0	1

只有缩放和平移时的矩阵

```
img = cv2.resize(img)
img = cv2.copyMakeBorder(img)
img = cv2.cvtColor(img, BGR2RGB)
img = (img - mean)/std
img = img.transpose(2, 0, 1)[None]
```

```
img = warpAffine(img)
```

YoloV5后处理

Yolov5是目标检测中比较经典的模型，学习对其后处理进行解码是非常有必要的。在这里我们仅使用核函数对Yolov5推理的结果进行解码并恢复成框，掌握后处理所解决的问题，以及对于性能的考虑

经验之谈：

1. 对于后处理的代码研究，可以把PyTorch的数据通过转换成numpy后，tobytes再写到文件，然后再到c++中读取的方式，能够快速进行问题研究和排查，此时不需要tensorRT推理也可以做后处理研究。这也叫变量控制法
2. fast_nms_kernel会在极端情况少框，但是这个极端情况一般不会出现，实测几乎没有影响
3. fast nms在cuda实现上比较简单，高效，不用排序

yolov5的输出tensor(n x 85)
其中85是cx, cy, width, height, objness, classification * 80

YoloV5后处理

CPU解码重点：

1. 避免多余的计算，需要知道有些数学运算需要的时间远超过很多if，减少他们的次数就是性能的关键
2. nms的实现是可以优化的，例如remove flag并且预先分配内存，reserve对输出分配内存

GPU解码重点：

1. 表示输出数量不确定的数组，用[count, box1, box2, box3]的方式，此时需要有最大数量限制
2. 通过atomicAdd实现数组元素的加入，并返回索引
3. 一样的，不必要的计算，尽量省掉

thrust

是的，只是为了告诉你它的存在，这里蜻蜓点水

error

1. 错误处理是理解如何控制cuda发生的错误，和捕获错误的技术
2. 在写cuda相关代码时，错误检查是错误处理的一种手段
3. 在这里着重拿出来讲可恢复与不可恢复的错误，以及其传播的特性

谢谢!