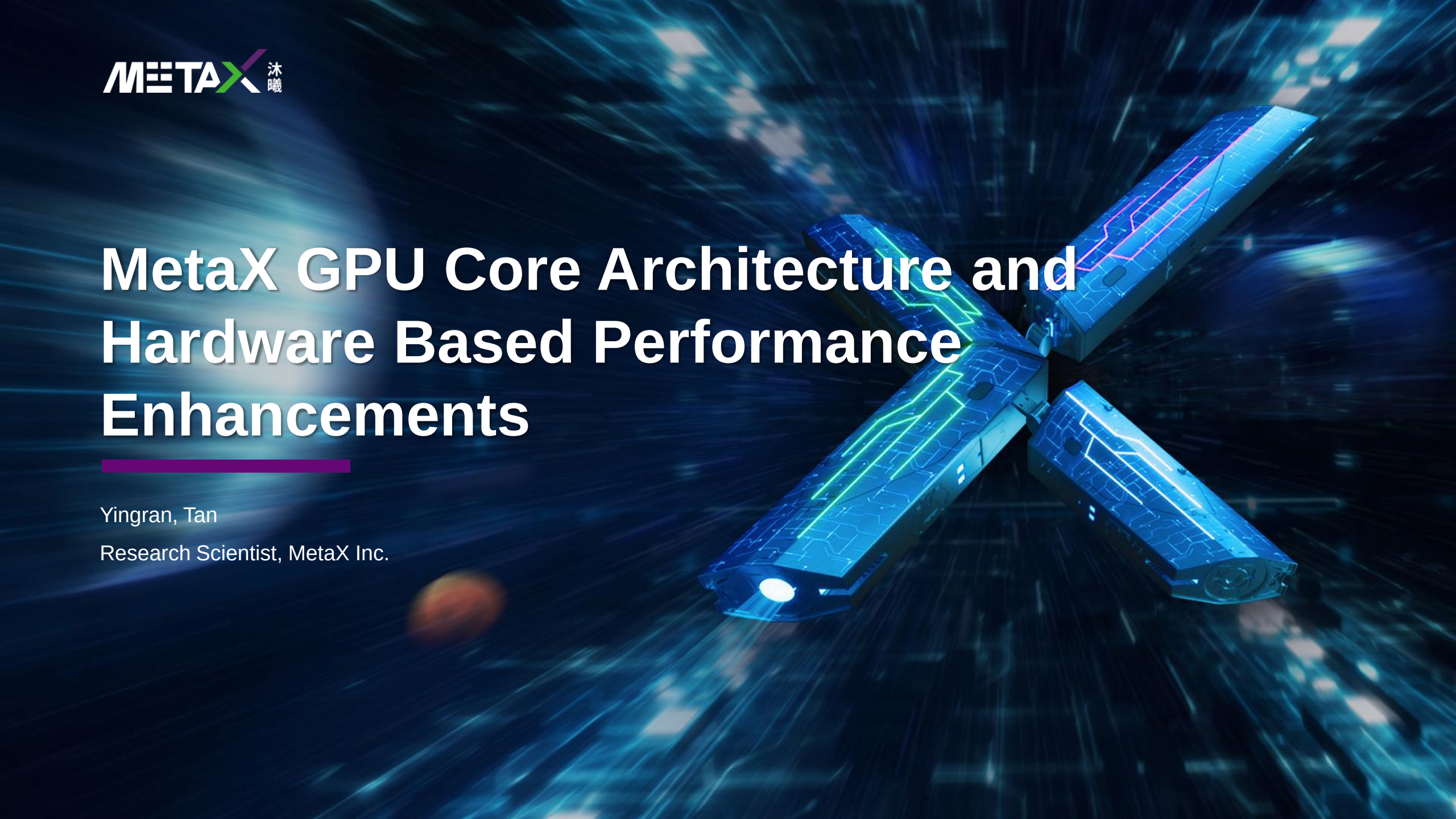


MetaX GPU Core Architecture and Hardware Based Performance Enhancements

Yingran, Tan

Research Scientist, MetaX Inc.

MetaX C系列概览

❯沐曦MACA软件栈国际主流生态兼容

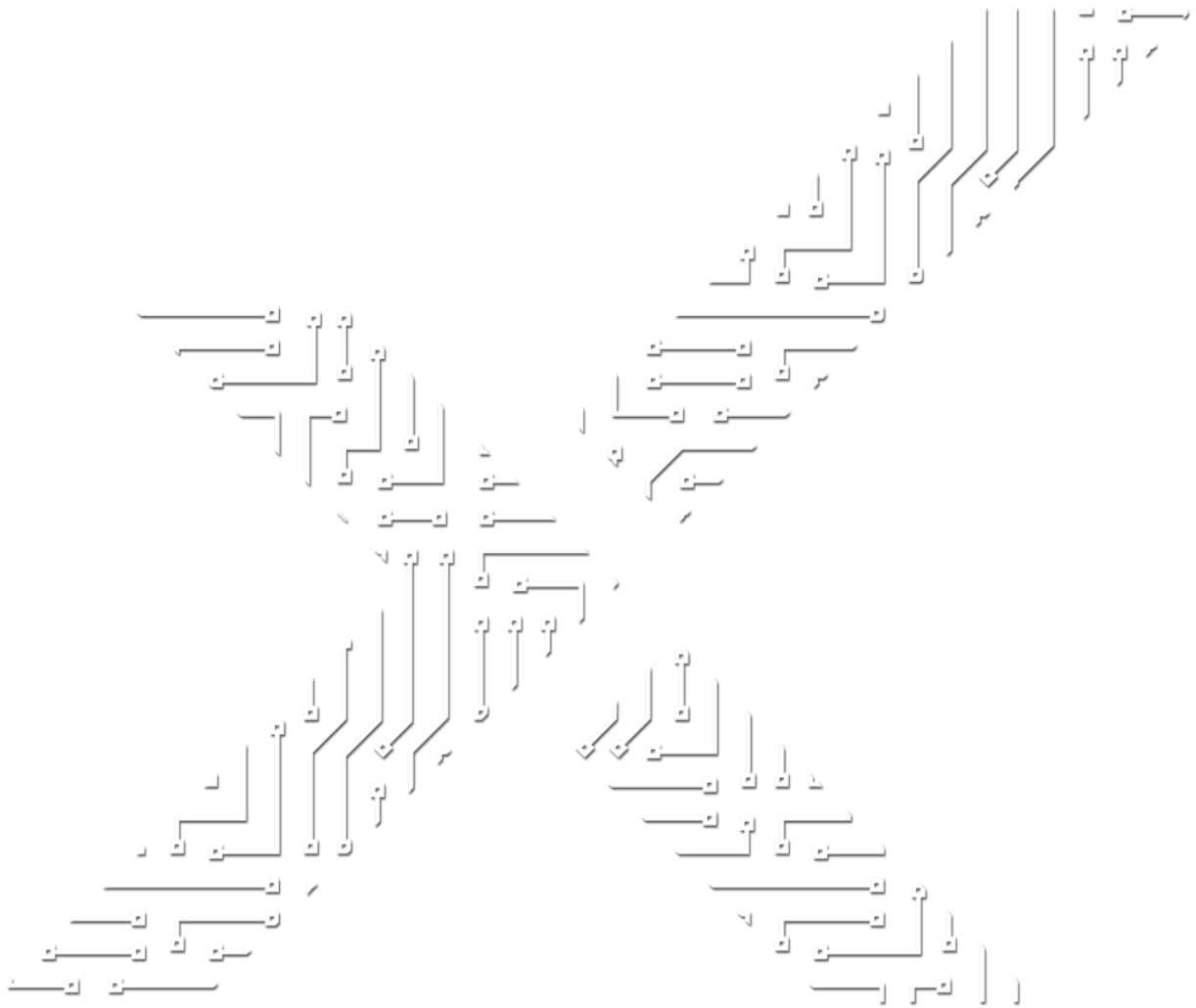
❯沐曦C系列性能达到国际主流水准

- ➤ 不同GPU产品性能相差极大
- ➤ 即便同一厂家，架构演进过程中，参数和指令集改动都可能很大
- ➤上一代的代码在下一代通常只能做到功能兼容，性能部分兼容。
  - ▻ 某GPU厂家基于A架构开发的半精度矩阵乘法kernel在B架构产品上核心ALU单元利用率不超过40%，基于B架构开发的半精度矩阵乘法kernel在C架构产品上利用率不超过50%。

❯对性能有比较大影响的差异在本PPT中标题加 ※。相关代码可能需要手动调优

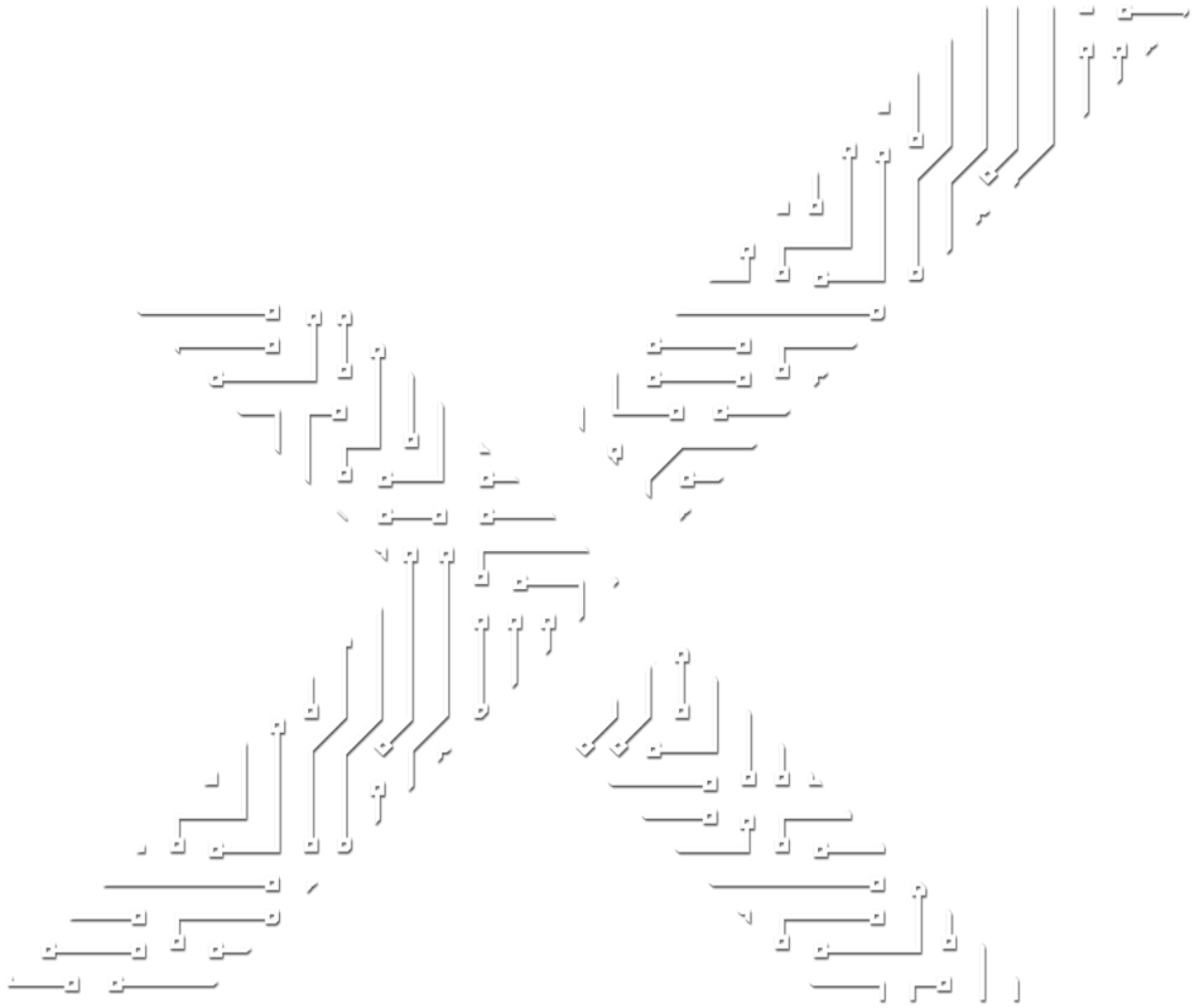
大纲

- 编程模型
- 架构对比
- 性能优化

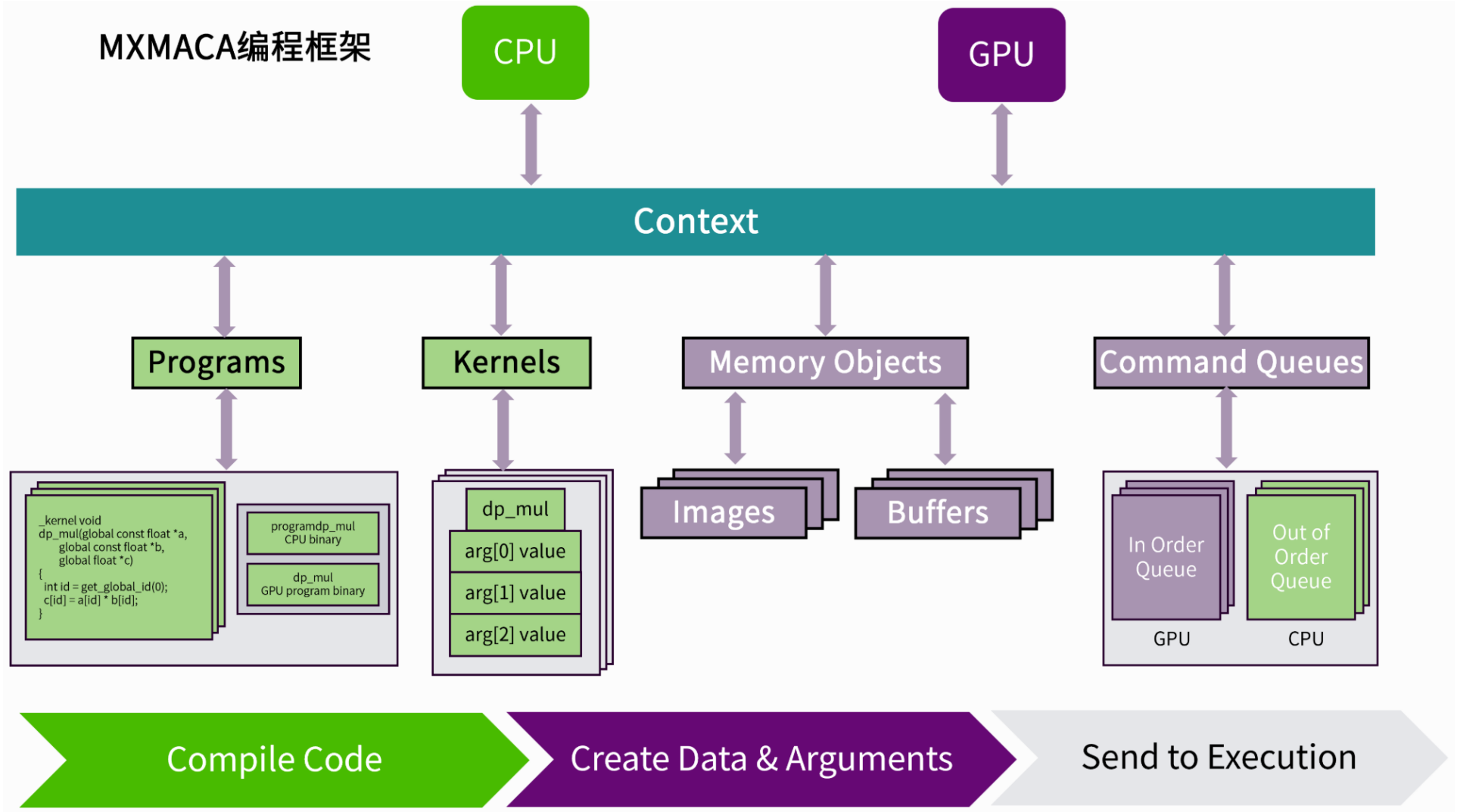


大纲

- 编程模型
  - Grid-Block-Thread
  - Memory Hierarchy
- 架构对比
- 性能优化

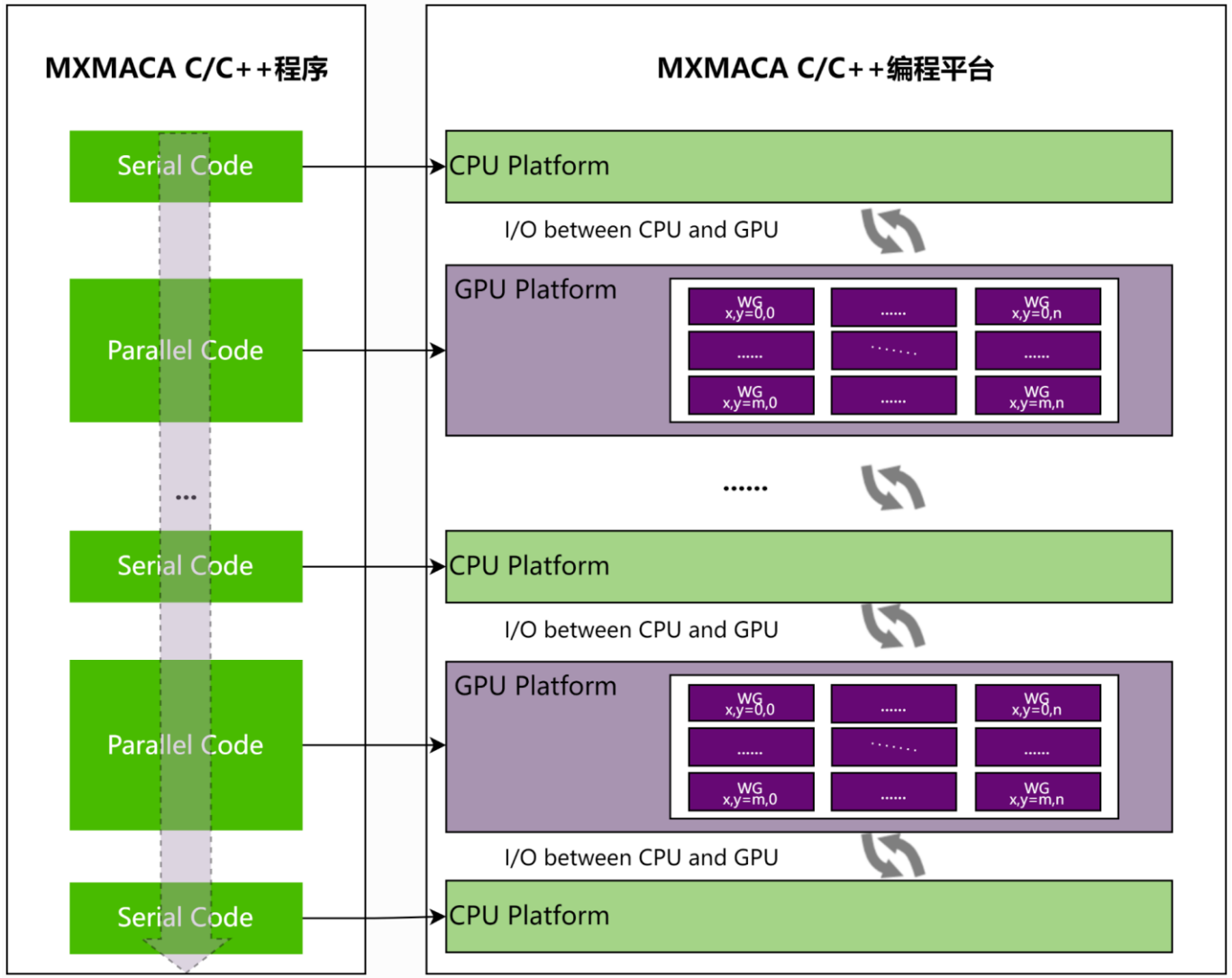


MXMACA程序结构



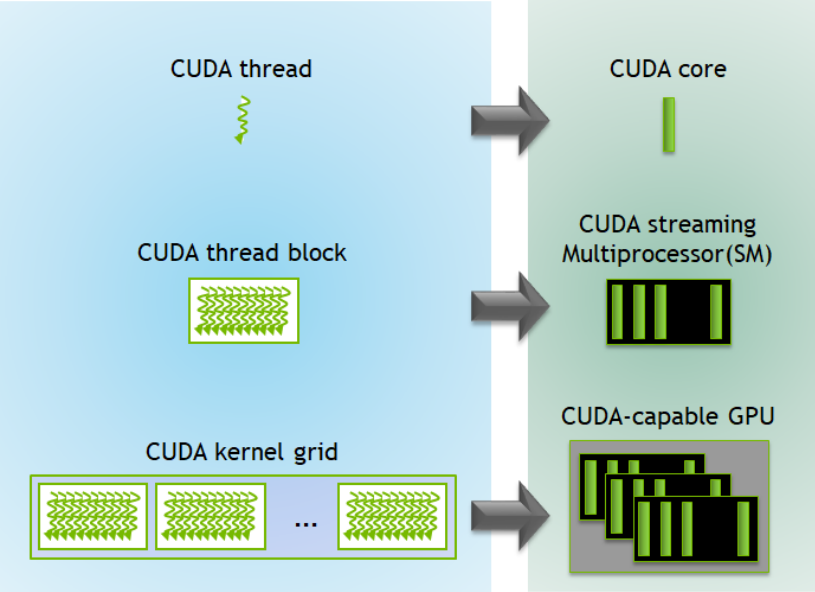
MXMACA程序结构

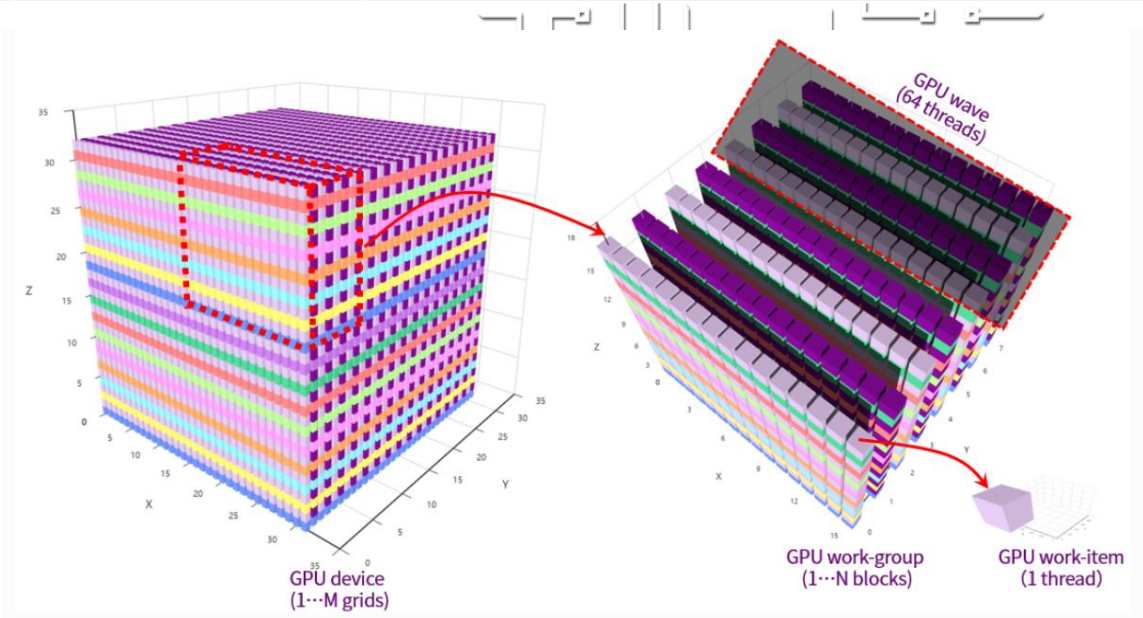
- 一个典型的MXMACA程序实现流程应遵循下面的模式
  - 把数据从CPU内存拷贝到GPU内存
  - 调用核函数对GPU内存的数据进行处理
  - 将数据从GPU内存传送回CPU内存



Grid-Block-Thread

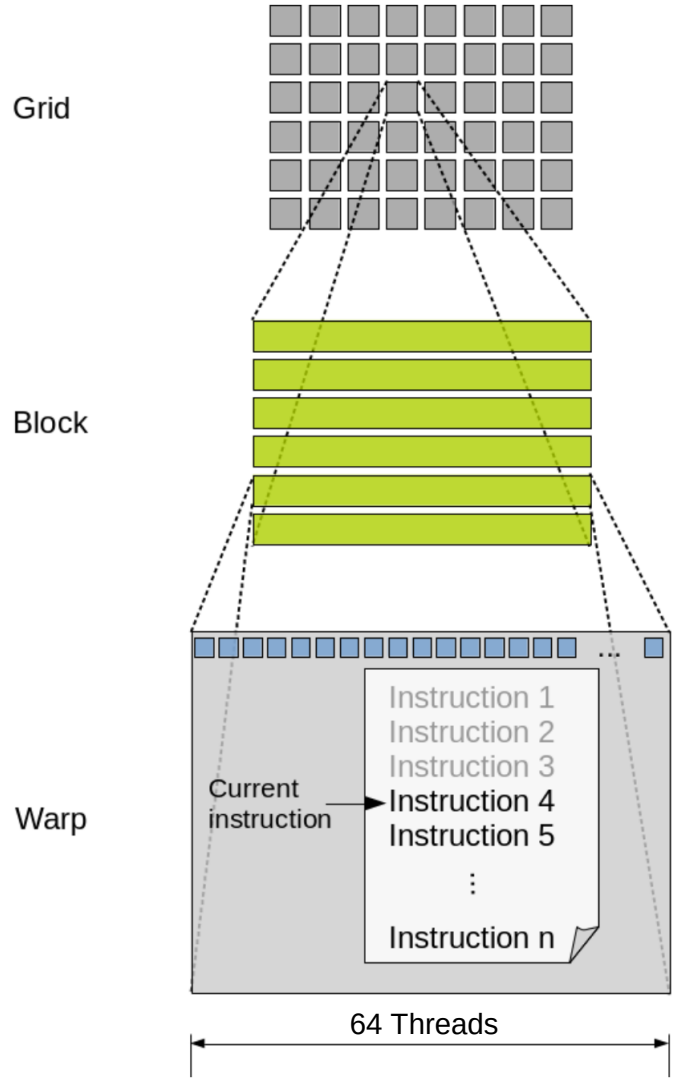
CUDA/MXMACA名称			曦云硬件资源		
网格	Grid				Block组成的三维结构,对应 kernel
线程块	Block		工作组	Workgroup	由相同内核执行的工作项集合
线程	Thread		工作项	Work-Item	任务的基本单元
线程束	Warp/Wave		线程束	Wave	指令同步执行的最小单元





硬件调度执行单位：Warp/Wave

- **Warp represent the smallest unit of work that can be scheduled**
  - All threads in a warp execute in lockstep
- **SPMD (Single Program Multiple Data) :**
  - All warps within a Grid execute the same instruction stream.
- **SIMT (Single Instruction Multiple Thread):**
  - Instructions are issued by the hardware at the warp level.
  - Multiple warps can occupy the same hardware for better utilization



※ Impact of Warp/Wave Size

- **Warp/Wave Shuffle需要考虑32和64的区别**
  - MACA: 64 threads
  - CUDA: 32 threads
- **float __shfl_down_sync(int mask, float sum, int width=warpSize)**
- **建议直接调用__reduce_add_sync(mask, sum) 等maca builtin function**

```cuda
template <unsigned int blockSize>
__device__ __forceinline__ float warpReduceSum(float sum){
    if(blockSize >= 32)sum += __shfl_down_sync(0xffffffff,sum,16);
    if(blockSize >= 16)sum += __shfl_down_sync(0xffffffff,sum,8);
    if(blockSize >= 8)sum += __shfl_down_sync(0xffffffff,sum,4);
    if(blockSize >= 4)sum += __shfl_down_sync(0xffffffff,sum,2);
    if(blockSize >= 2)sum += __shfl_down_sync(0xffffffff,sum,1);
    return sum;
}
```

CUDA

```maca
template <unsigned int blockSize>
__device__ __forceinline__ float warpReduceSum(float sum){
    if(blockSize >= 64) sum +=
__shfl_down_sync(0xffffffffffffffff,sum,32);
    if(blockSize >= 32) sum +=
__shfl_down_sync(0xffffffffffffffff,sum,16);
    if(blockSize >= 16) sum +=
__shfl_down_sync(0xffffffffffffffff,sum,8);
    if(blockSize >=  8) sum +=
__shfl_down_sync(0xffffffffffffffff,sum,4);
    if(blockSize >=  4) sum +=
__shfl_down_sync(0xffffffffffffffff,sum,2);
    if(blockSize >=  2) sum +=
__shfl_down_sync(0xffffffffffffffff,sum,1);
    return sum;
}
```

MACA

数值精度支持

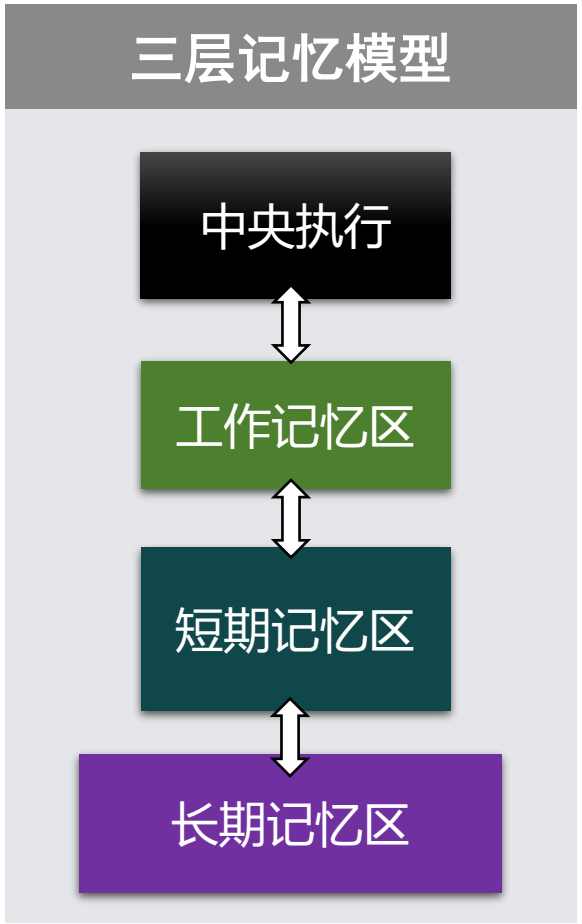
MetaX C系列某款GPU产品数据类型支持情况	
有符号定点	.s8, .s16, .s32, .s64
有符号定点	.u8, .u16, .u32, .u64
浮点	.f16, .f16x2, .bf16, .bf16x2 .f32, .f64
比特（无类型）	.b8, .b16, .b32, .b64, .b128, .b256

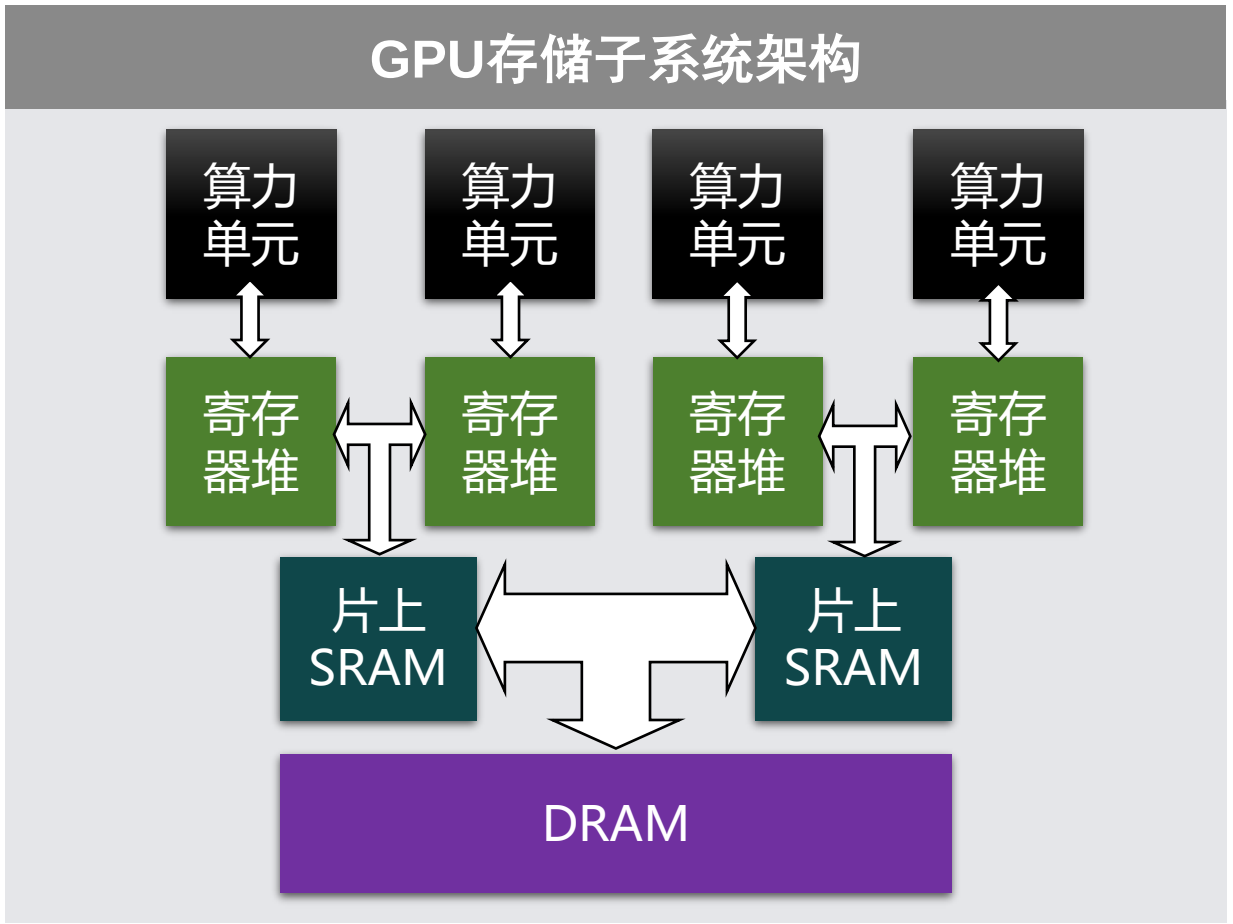
MXMACA C++ extension数值精度支持

- 内置类型：https://developer.metax-tech.com/api/client/document/preview/1175/split_files/c_%E8%AF%AD%E8%A8%80%E6%89%A9%E5%B1%95.html#n6vrjba27bh81
- 向量类型：https://developer.metax-tech.com/api/client/document/preview/1175/split_files/c_%E8%AF%AD%E8%A8%80%E6%89%A9%E5%B1%95.html#n8bnvtwg7pwy1

GPU存储子系统架构

认知科学三层记忆 (Memory)， 类比 GPU存储子系统架构(Memory Subsystem)

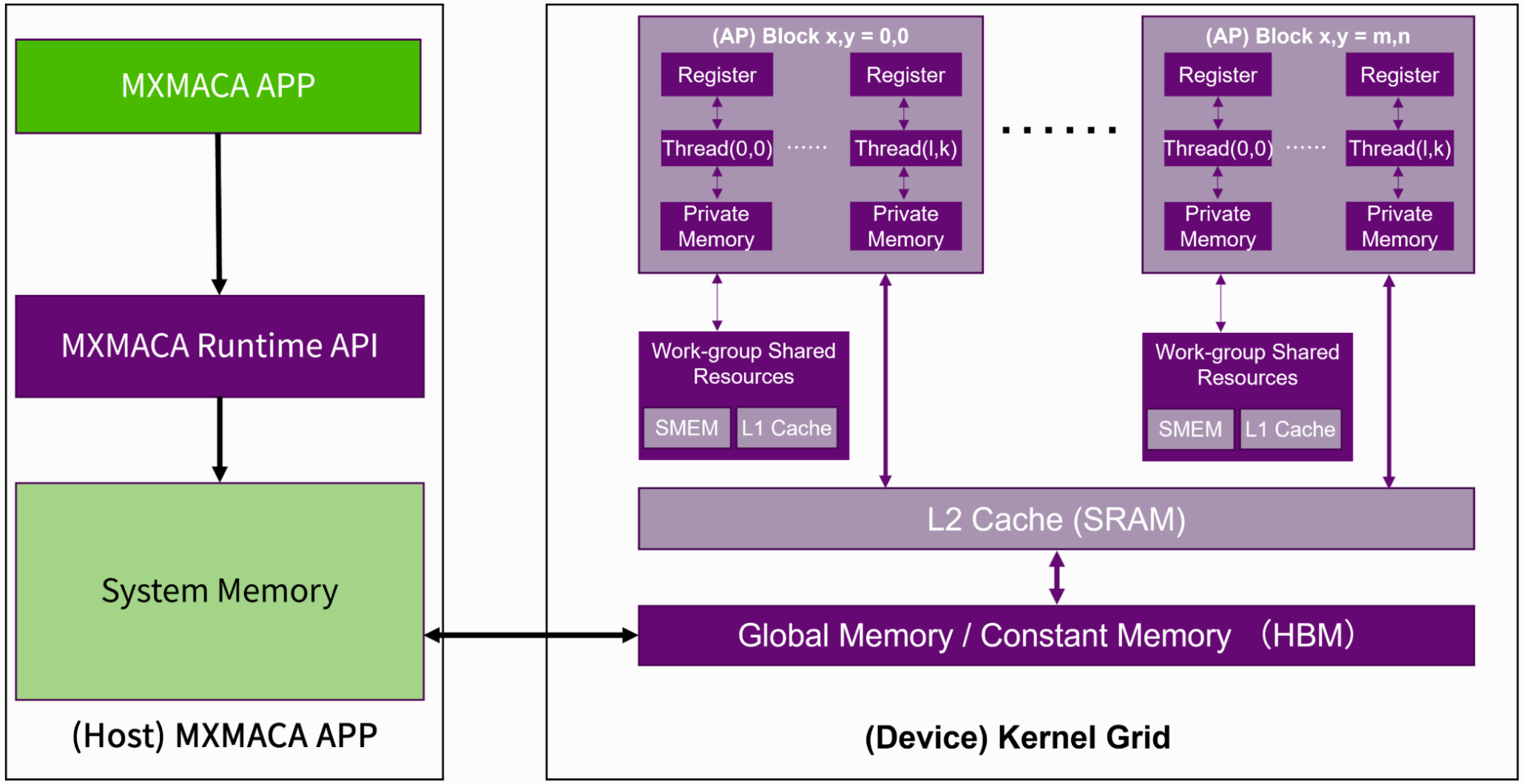




SRAM
- 容量小
- 速度快
- 使用数字逻辑工艺
- 片上SRAM

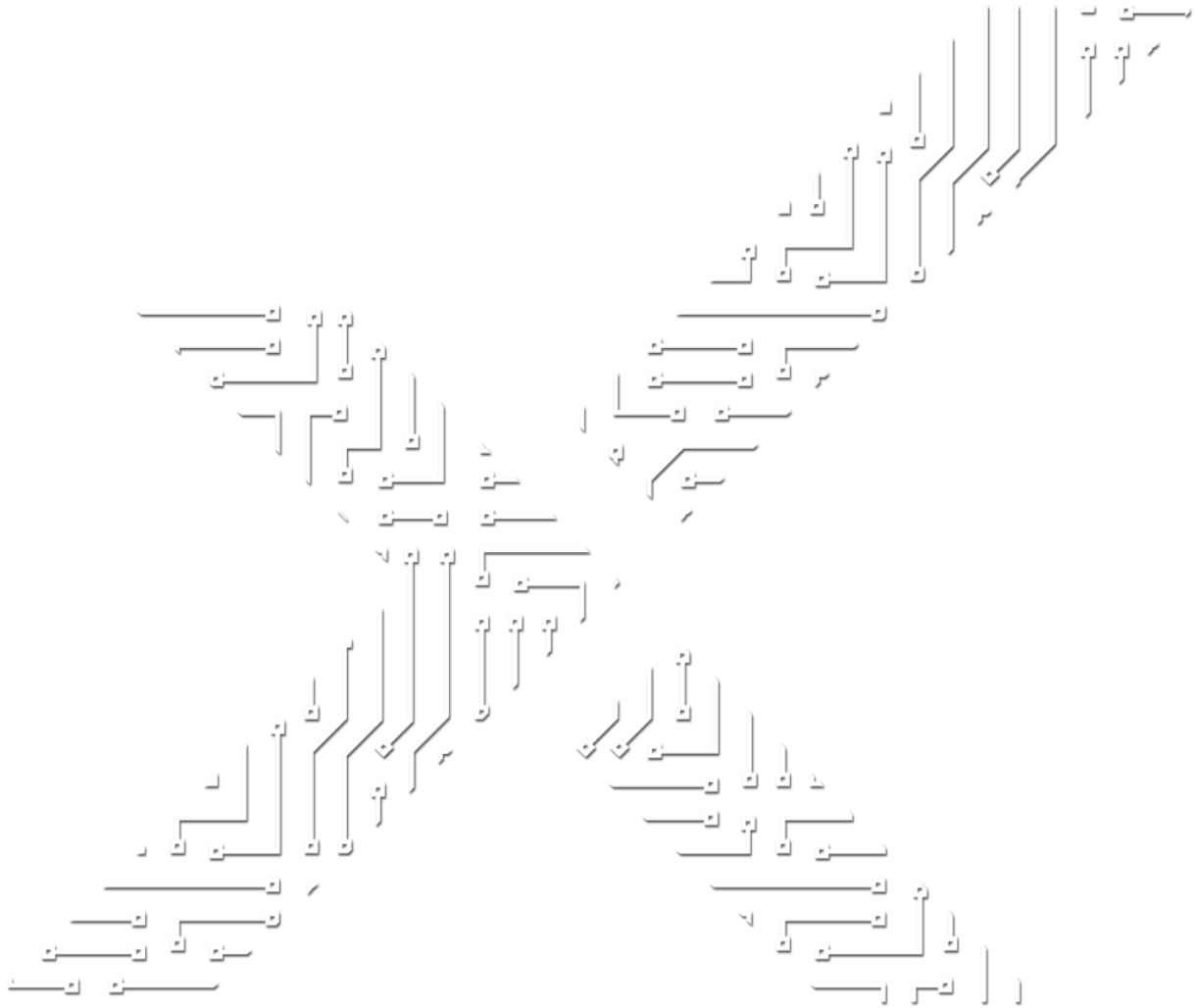
DRAM
- 容量大
- 速度慢
- 专用DRAM工艺
- 片外DRAM

MXMACA编程的内存层次模型

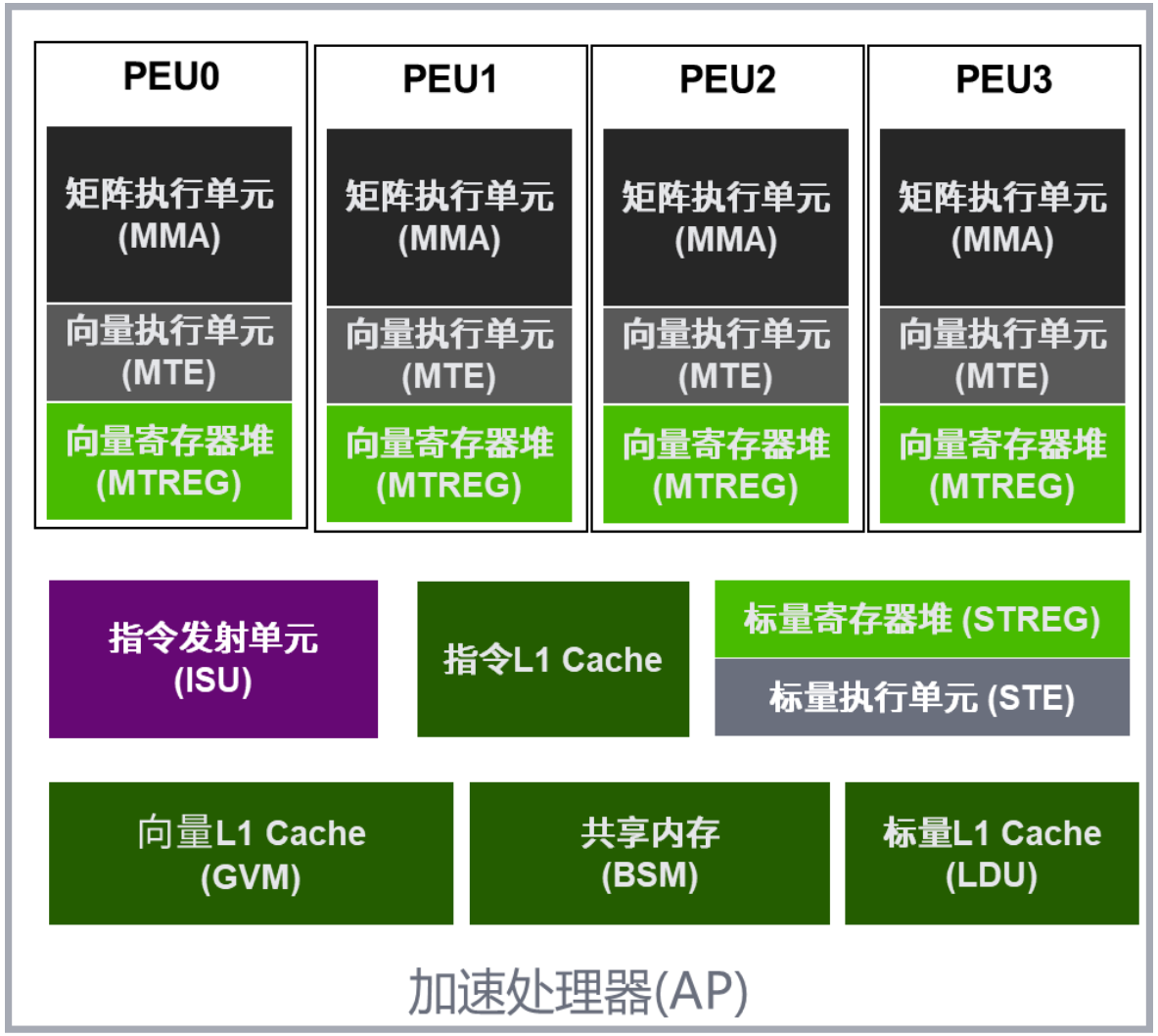


大纲

- 编程模型
- 架构对比
  - 体系结构对比
  - 算力参数对比
  - 存储容量对比
  - 读写带宽对比
- 性能优化



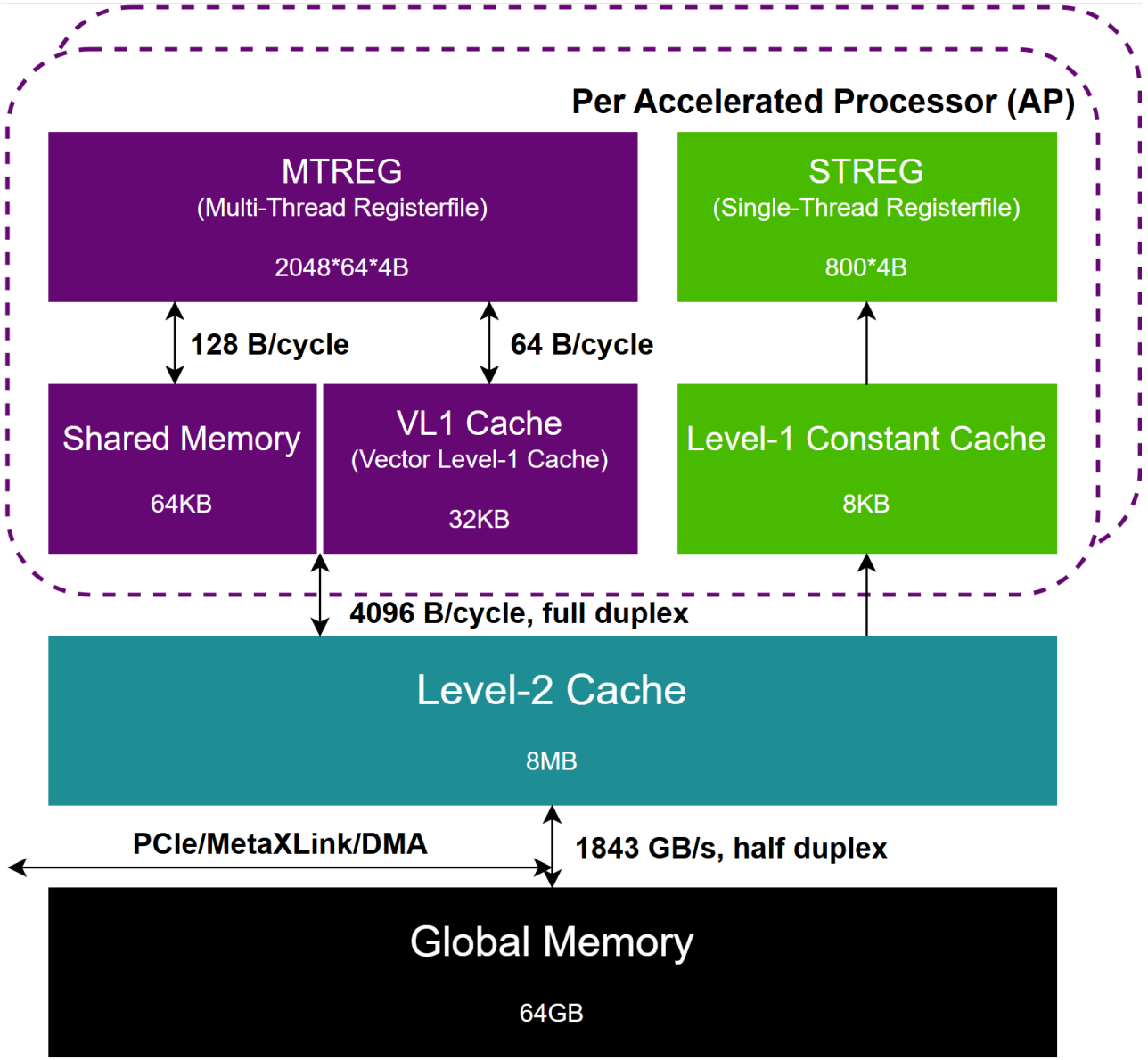
AP架构参数



FLOP/Cycle	某款主流GPU	曦云某款产品
#SM/AP (个)	108	104
标量执行单元/AP	N/A	1
向量执行单元/AP	64	64
矩阵执行单元/AP	4	4
FP32向量/AP	128	可通过测试集获得指令吞吐
FP32矩阵/AP	N/A	
FP16(BF16)矩阵/AP	2048	
INT8矩阵/AP	4096	
INT4矩阵/AP	8192	N/A
共享内存/AP	0~164KB	64KB
向量寄存器堆/AP	4 SMSP * 512 reg/t * 32 t * 4B = 256KB	4 PEU * 512 reg/t * 64 t * 4B = 512KB

※二级缓存系统 (容量)

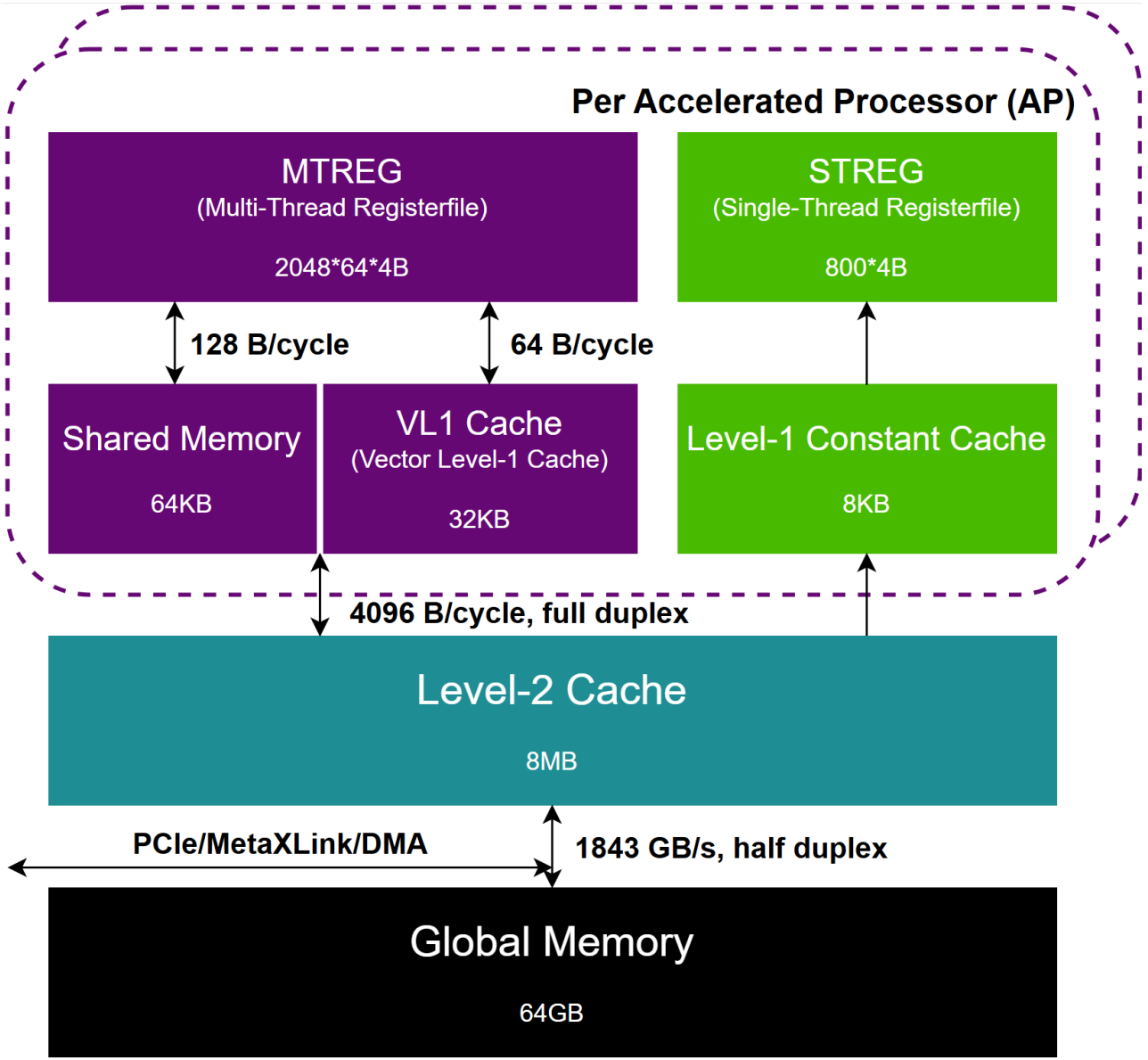
	某款主流GPU	曦云某款产品
显存	80GB	64GB
L2	40MB	8MB
向量寄存器堆/AP	4 SMSP * 512 reg/t * 32 t * 4B = **256KB**	4 PEU * 512 reg/t * 64 t * 4B = **512KB**
共享内存	**0~164KB**	**64KB**
L1D	16~192KB	32KB*
L1C	8KB (vector)	8KB (Scalar)
标量寄存器堆/AP	Unknow (uniform)	800 * 4B



- 此处提到的曦云某款产品中 L1D为streaming cache

二级缓存系统 (理论带宽)

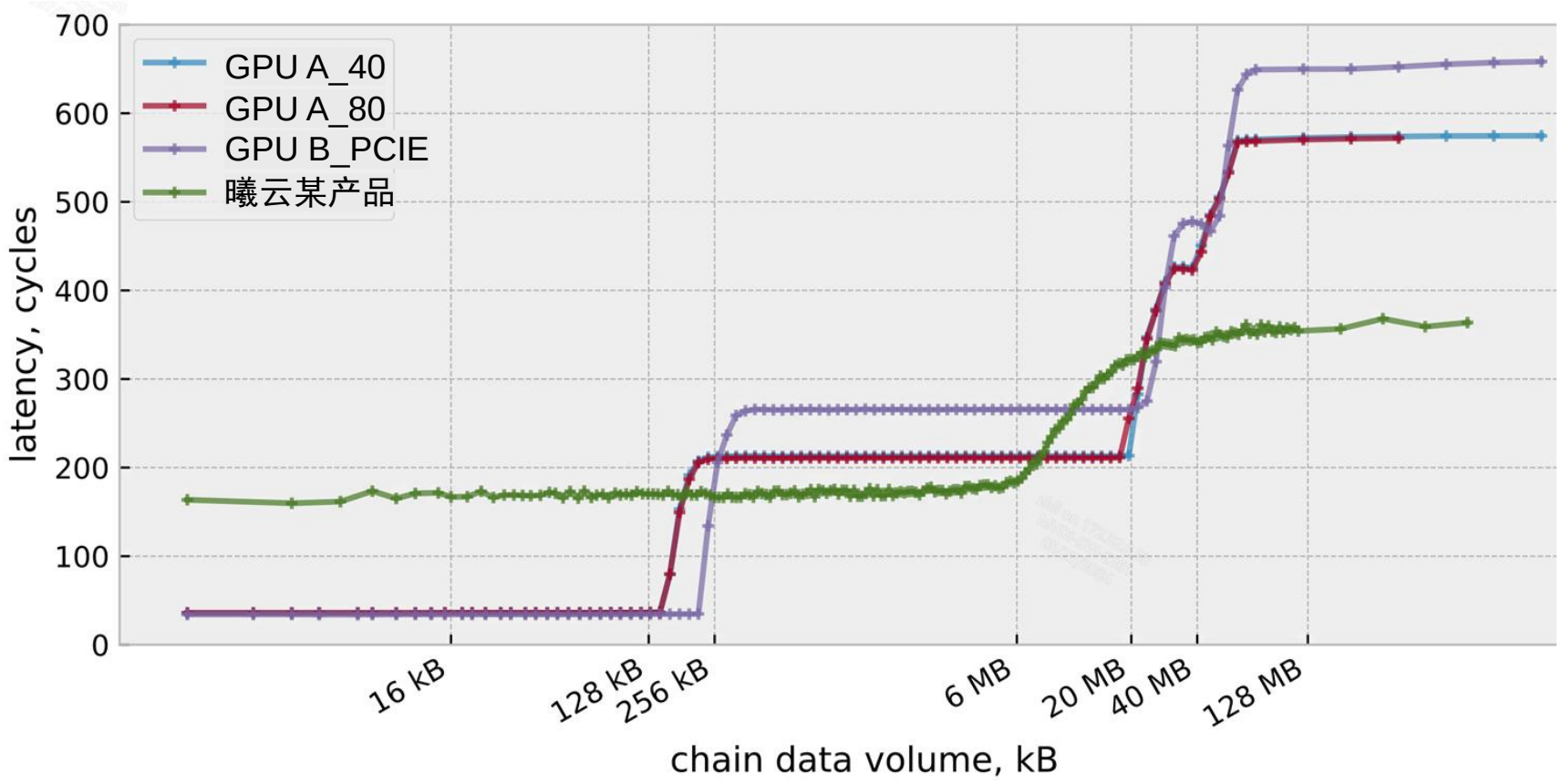
	某款主流GPU	曦云某款产品
理论DRAM-L2 (GB/s)	1950	1843
L2-L1 (Byte/cycle)	5120	4096
L1-MTREG (Byte/cycle)	128	64*
Shared-MTREG (Byte/cycle)	128	128
Host-HBM (单向 GB/s)	32	64



- 此处提到的曦云某款产品中 L1D为streaming cache

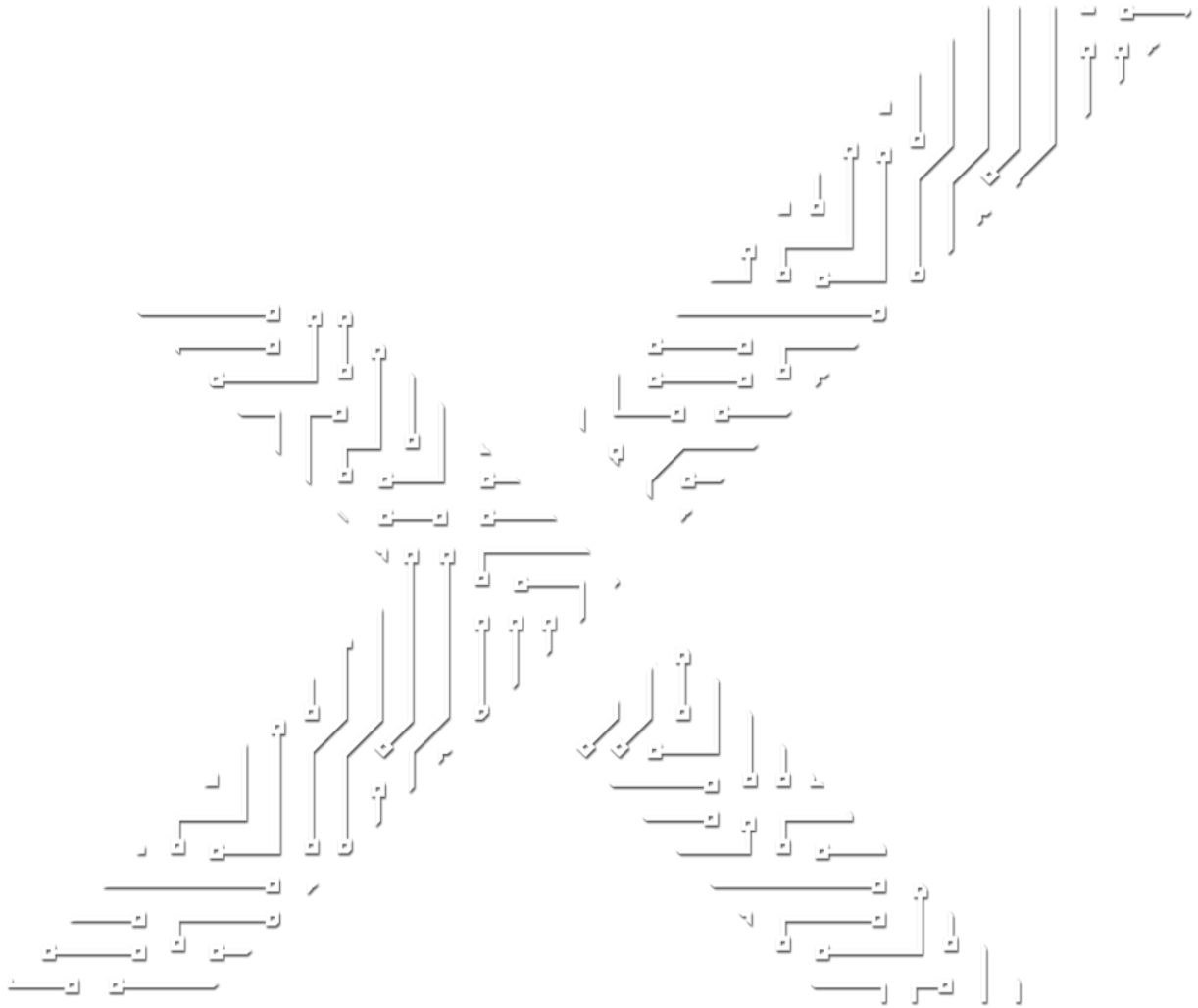
二级缓存系统 (时延)

- 曦云某款产品 L2 hit/miss时延小于某款主流GPU A_40/A_80/B_PCIE
- 曦云某款产品 L1 is streaming cache. No L1 Cache hit. Memory coalescing @ L1.



大纲

- 编程模型
- 架构对比
- 性能优化
  - 基本方法
  - 优化案例



性能调优基本方法

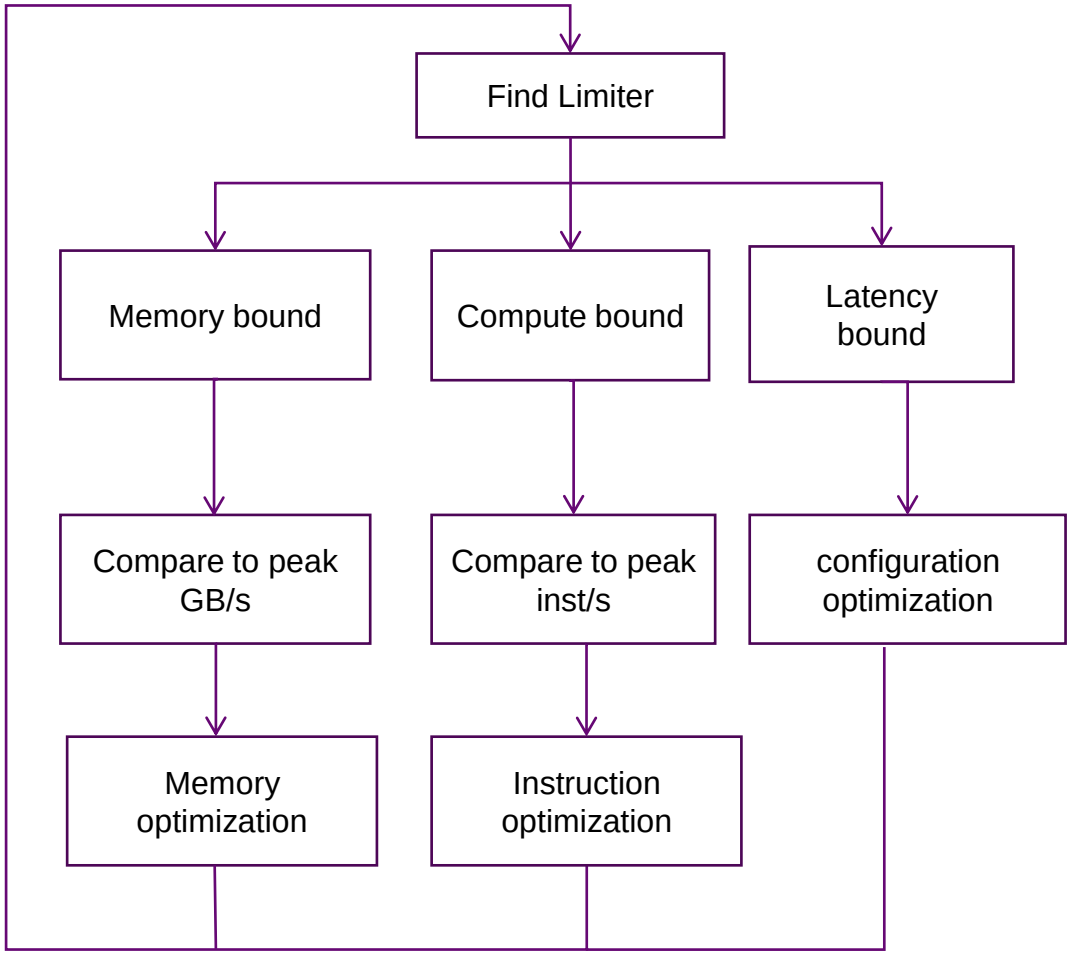
- **对于真卡数据进行profile，先测量，后优化**
- **性能评价方式，找到耗时最高的10 GPU kernels, 遵循Amdahl定律**
  - 找到薄弱点和优化方向
  - 找到关键性能瓶颈
  - 验证最差耗时硬件性能单元
  - 对比分析硬件性能
  - 增加性能测试用例
  - 优化并行算法

性能调优基本方法

硬件性能是找到花费时间最多的模块

Max

- Kernel Launch, Complete Signal, Cache Flush
- PCIe, read + write, latency
- RM(wave/thread launching speed)
- HBM read + write
- Vector/Scaler ALU throughput
- GlobalMem Load throughput Latency
- SharedMem throughput and latency
- 基于理论峰值（算力/带宽）和算法特性，设定一个合理的性能目标



※体系结构相关的优化案例

Occupancy

- Case 1: Active Block
- Case 2: Active Warp
- Case 3: Private Memory Spill

Coalescing

- Case 4: Global Memory Access Coalescing
- Case 5: Partial Write

Bank Conflict

- Case 6: Shared Memory Bank Conflict

Latency Hiding

- Case 7: Latency Hiding

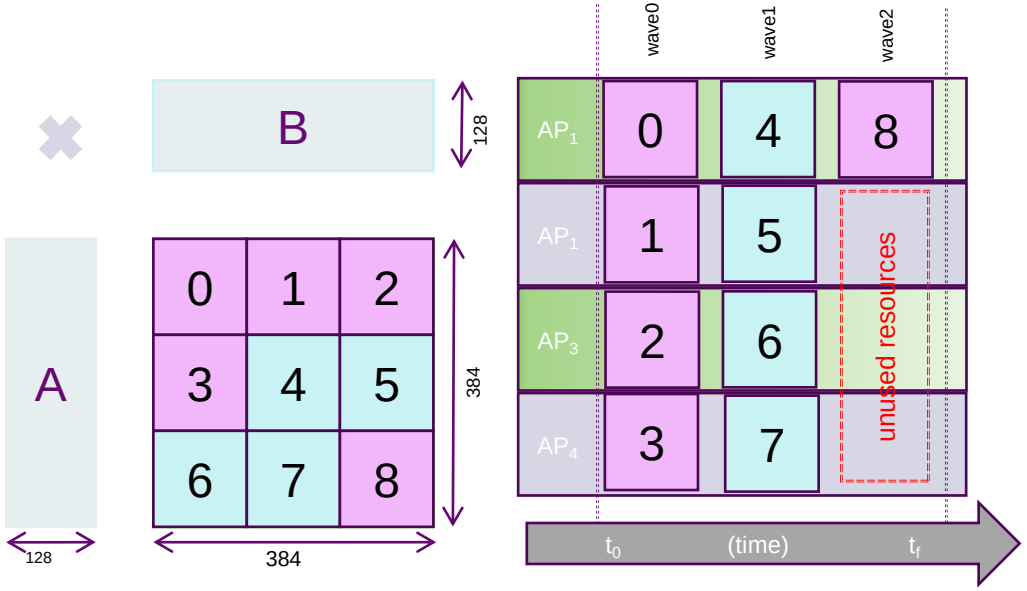
※Occupancy

资源类型	MetaX C系列资源限制	对占用率的影响
每个warp的线程数	64	
每个AP的warp数	最多32个warp（2048个线程）	限制最大的并行线程数
每个AP的block数	最多32个block	限制并行block的数量
每个AP内共享内存容量	64KB	直接影响活动Block的数量
每个block的线程数	每个block最多1024个线程	影响线程槽的分配方式
每个AP的向量寄存器容量	2048个64 × 32-bit向量寄存器	直接影响Active Warp的数量
每个线程的向量寄存器数	最多256个32-bit寄存器	直接影响Active Warp的数量
每个AP内标量寄存器容量	最多800个32-bit寄存器	直接影响Active Warp的数量

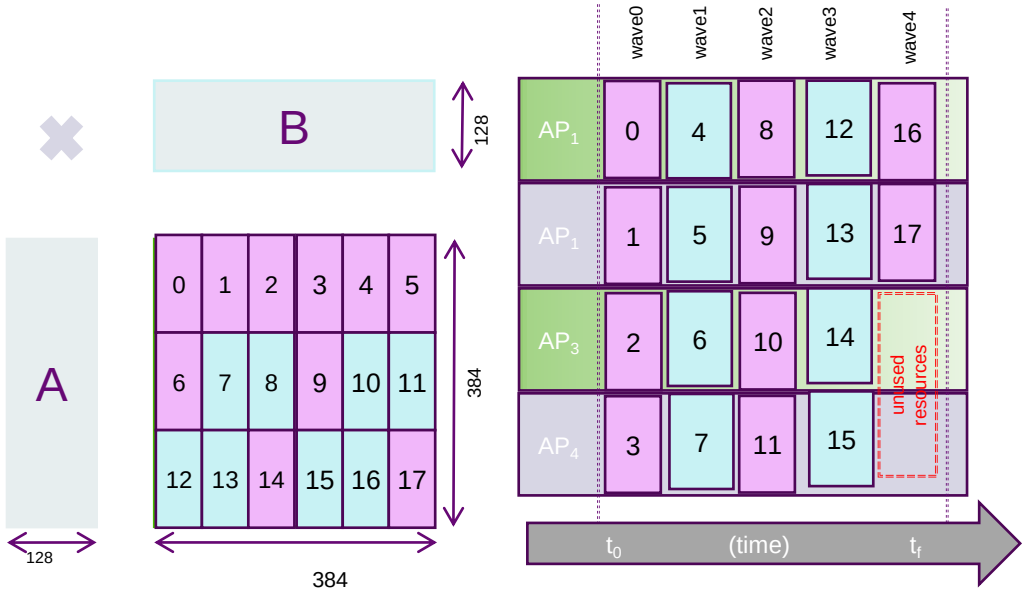
※ Case 1: Active Block

➢ 假设一个有四个AP型GPU环境下的384x384x128的 GEMM 操作所采取的数据并行执行计划：

- 采用grid size = 9的数据并行分解方式，单个计算单元处理数据为128*128*128
- AP利用率上限为75%



- 采用grid size = 18的数据并行分解方式，单个计算单元处理数据为128*64*128，
- AP利用率上限为90%



※Occupancy

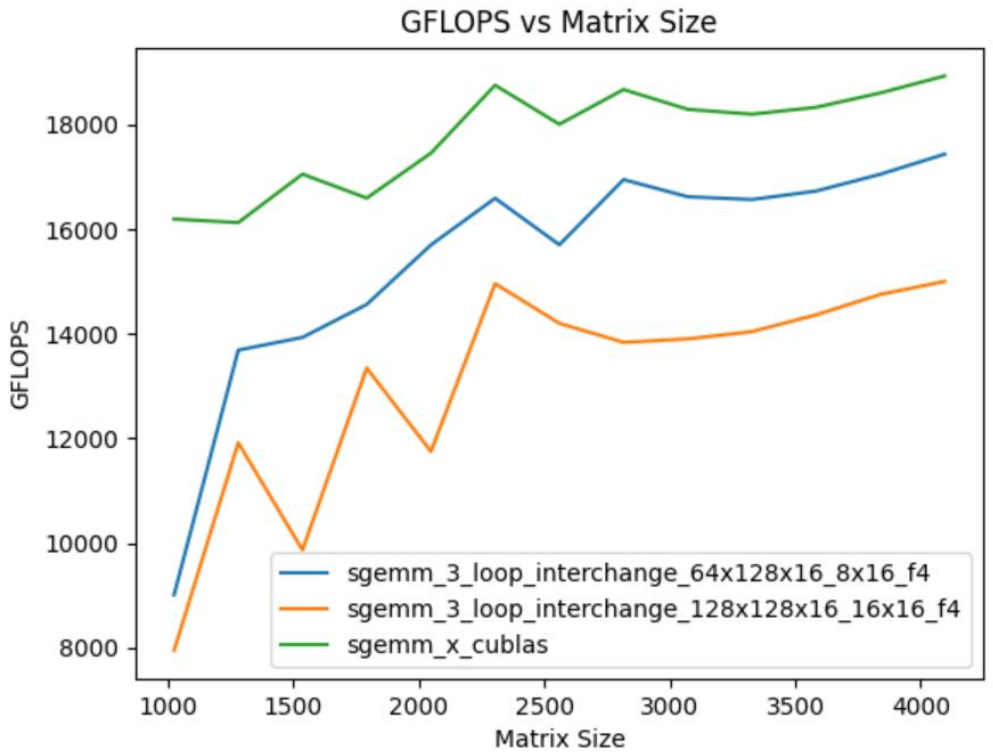
- GPU的Occupancy指的是一个AP的Static Active Warp数量与该AP最大支持的warp数量之比。即
- $Occupancy$ = 每个AP上$Static\ Active\ Warp$数量 / AP最大支持的$Warp$数量
- Active Warp指的是AP上同时被调度执行的Warp。Active Warp数量真正决定了延迟隐藏的效果
- 编译时使用 --resource-usage，查看一个线程的寄存器使用量，从而计算最大的Static active warp数量

※Case 2：Active Warp

- 为何128x128x16性能比64x128x16更差？
- Active Warp per SM的影响

```
nvcc  -std=c++17 -O2 -g --resource-usage -lineinfo -gencode arch=compute_80,co
 maca info   : Function properties for  _Z25sgemm_128x128x16_16x16_f4iiiPfS_S_
  0 bytes stack frame
 maca info   : Used  152 MTregisters, 20 STregisters, 8192 bytes shared mem
 maca info   : staticMaxWarps/PEU : 3
```

```
nvcc  -std=c++17 -O2 -g --resource-usage -lineinfo -gencode arch=compute_80,code=sm_80  -c sge
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z25sgemm_128x128x16_16x16_f4iiiPfS_S_' for 'sm_80'
ptxas info    : Function properties for _Z25sgemm_128x128x16_16x16_f4iiiPfS_S_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 128 registers, 8192 bytes smem, 392 bytes cmem[0]
```



Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	12.50	Block Limit Registers [block]	1
Theoretical Active Warps per SM [warp]	8	Block Limit Shared Mem [block]	1
Achieved Occupancy [%]	12.49	Block Limit Warps [block]	8
Achieved Active Warps Per SM [warp]	7.99	Block Limit SM [block]	32

※Case 3：Private Memory

查看private memory使用量的方法

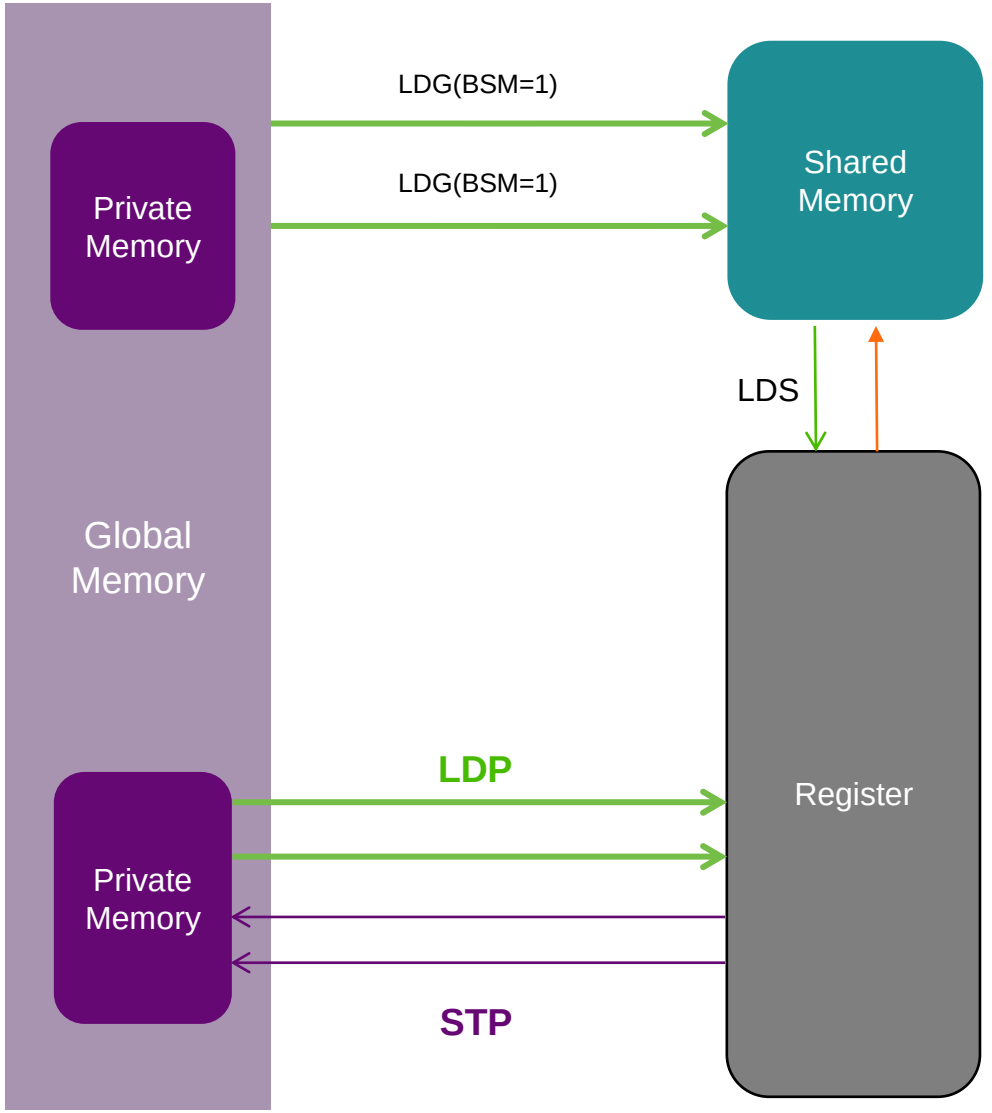
- 添加编译选项 “--resource-usage”
- 查看“bytes stack frame size”

常见产生private memory的语法

- Function Call： Caller & Callee 保存调用现场，通过栈传递调用参数
- 静态申请空间 & 动态申请空间
- 寄存器溢出
- 变量使用Volatile
- 类型长度不一的类型强转

Private memory性能损失

- 1 private memory：20% perf Loss



※Case 3：Private Memory

Function Call

- 函数调用使用inline/__forceinline__
- 间接调用换为 直接调用
- 使用inline需要综合评估，可能：编译耗时增加，寄存器溢出

静态申请/动态申请栈空间

- 数组，结构体等默认放在private memory， 编译器会进行优化 ，优化成功后才会放置在寄存器，影响优化结果的因素
  - 变量作为参数传递给子函数(对变量的操作行为未知)
  - 其他编译器判断优化代价大于收益的情形
- 可采取的优化策略
  - 改变变量/mem的使用方式
  - 减少变量的使用周期
  - 减少数组的长度/结构体规模

reg spill

- 程序使用变量过多
- launch bound设置不合理 **(512blocksize可使用156 mtreg， 1024blocksize可使用128 mtreg)**
- 循环展开次数过多
- inline范围过大

※Case 3：Private Memory -- 高级语言层级消除建议示例

runtime address

```
__global__ void test(int* res, int n, int t){

int a[100];
int id;
if(n>t){
   id = 1;
}else {
   id = 2;
}
a[id] = 20;

res[n]= a[1];

}
```

```
__global__ void test(int* res, int n, int t){

   int2 a[100];
   for(int i = 0; i<100; i++){
       a[i].x = i;
       a[i].y = 0;
   }

   int id;
   if(n>t){
       id = 1;
   }else {
       id = 2;
   }
    res[n] = a[id].x;

}
```

```
__global__ void test(int* res, int n, int t){

int a[100];
//int id;
if(n>t){
   //id = 1;
   a[1] = 20;
}else {
   //id = 2;
   a[2] = 20;
}
//a[id] =20;

res[n]= a[1];

}
```

改成这种
写法不会产生private memory

```
__global__ void test(int* res, int n, int t){

   int2 a[100];
   for(int i = 0; i<100; i++){
       a[i].x = i;
       a[i].y = 0;
   }

   //int id;
   int temp1, temp2;
   if(n>t){
       //id = 1;
       temp1 = a[1].x;
   }else {
       //id = 2;
       temp2 = a[2].x;
   }
    //res[n] = a[id].x;
    res[n] = n>t? temp1 : temp2;

}
```

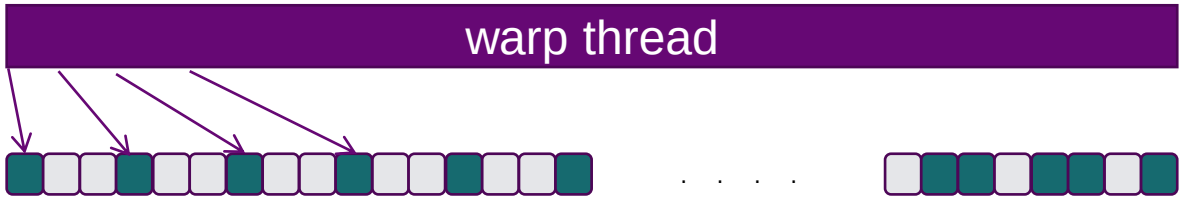
改成这样写法不会产生private memory

※ Case 4: Global Memory Access Coalescing

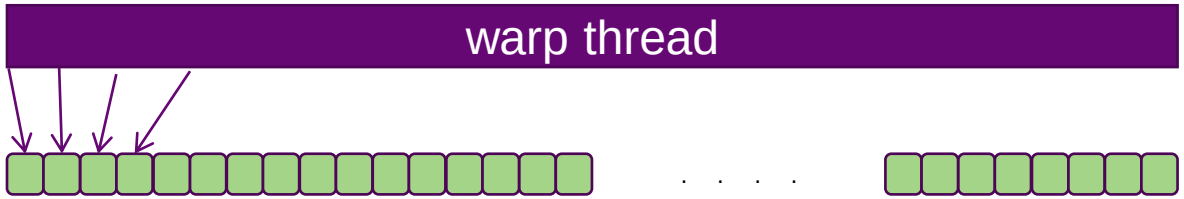
- Stride Accesses to Coalesced Access
- 访存是按照cache-line的长度进行访存，当每次访存时实际的有效数据较小时会造成带宽浪费
- AoS（Array of Structures）与SoA（Structure of Arrays）

```
typedef struct {
    float x, y, z;
} Point3D;
__global__ void example(Point3D*idata, float* odata)
{
   int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
   odata[xid] = idata[xid].x;
}
```

```
typedef struct {
    float* x, *y, *z;
} Point3D;
__global__ void example(Point3D idata, float* odata)
{
   int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
   odata[xid] = idata.x[xid];
}
```



在该例子种实际访问的有效数据仅为1/3



Coalesced Access

※Case 5: Partial Write降低HBM写带宽

背景

- MetaX C系列某产品开启HBM ECC后，kernel写global或者private中的数据时，如果写地址没有对齐到64B，或者没有将64B写满，会出现写指令完成时间特别长的问题。

现象

- 使用smi关闭ECC后，性能有了显著提升

解决方法

- 内存访问合并，内存访问重排，预取到Shared Memory再随机写

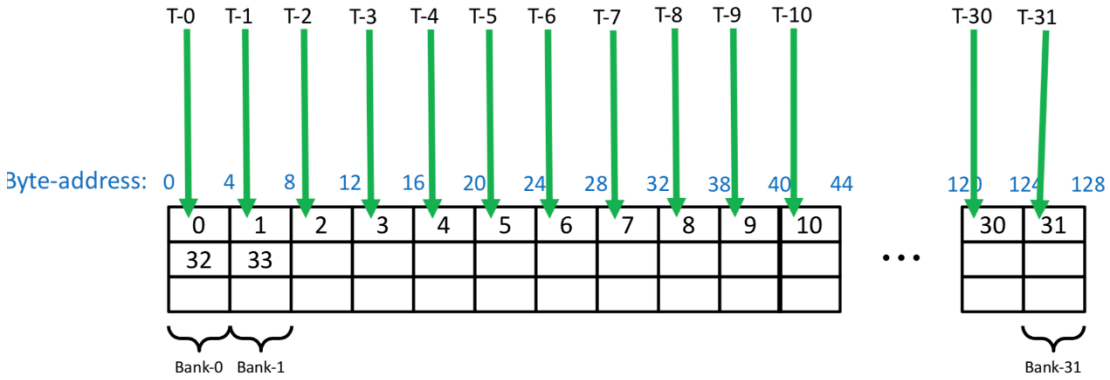
直观解释

- HBM Channel = 候车区 + 上车区。 假设某产品有32 Channel，等效于32个等待上车的出口
- Partial Write: 即将上车的乘客行李太多堵住了出口，造成一段时间内只有一个人可以上车
- ECC需要HBM controller做Read-Modify-Write，此时为了避免打断原子操作，阻塞后续访问请求

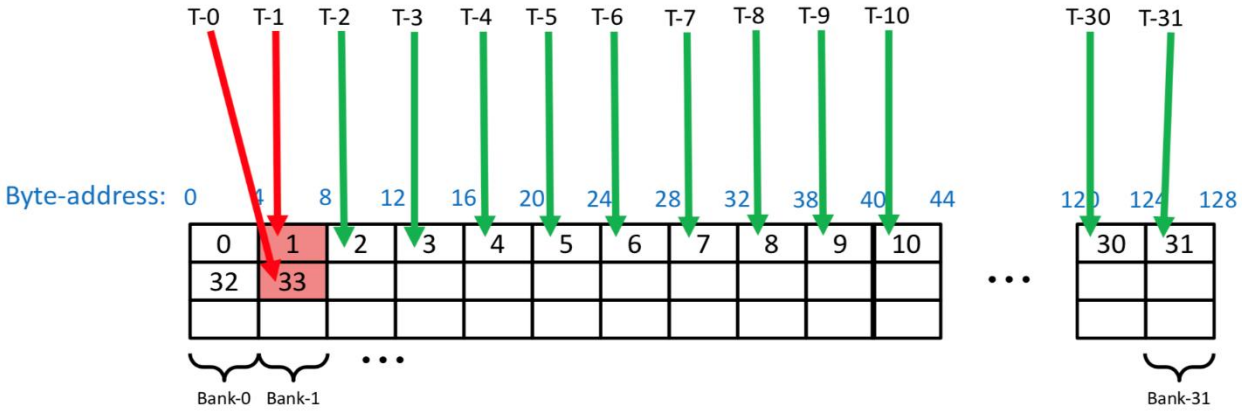
※Case 6: Shared Memory Bank Conflict

- 假定某款GPU产品的Shared Memory有32个Bank，每个bank 4Byte宽。每时钟周期每个bank可以读+写一次。
- Shared Memory每个时钟周期只能服务一个Warp。
- 如果如果一个内每个thread加载4 Byte数据，且所有请求没有bank conflict，那么只需要1 (2) 个时钟周期就可以完成请求。
- 如果2 (4) 个thread同时访问一个bank的bank conflict，则需要 2 (4) 个时钟周期，即shared带宽峰值减半

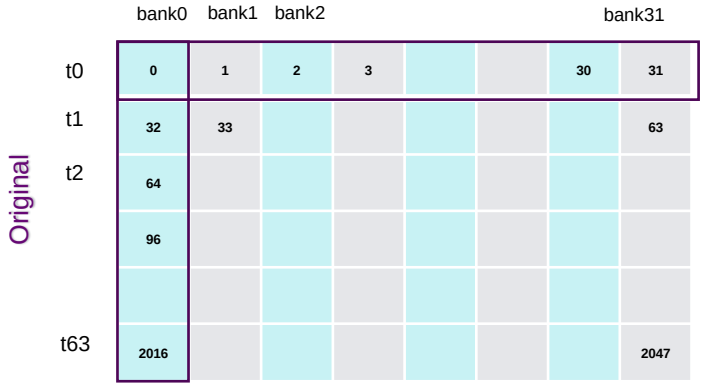
No Bank Conflicts



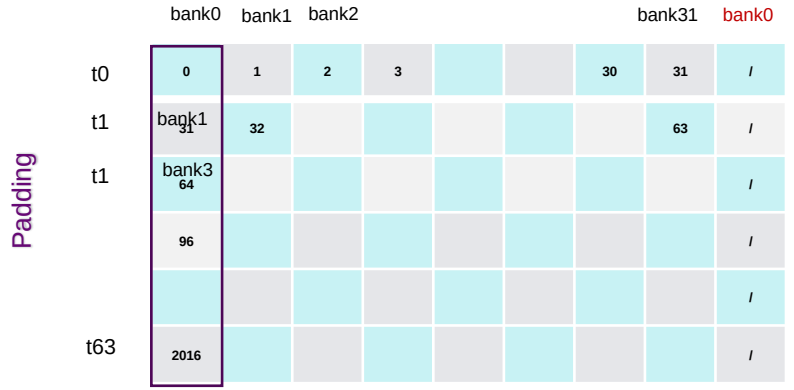
2-way Bank Conflict



※Case 6: Shared Memory Bank Conflict



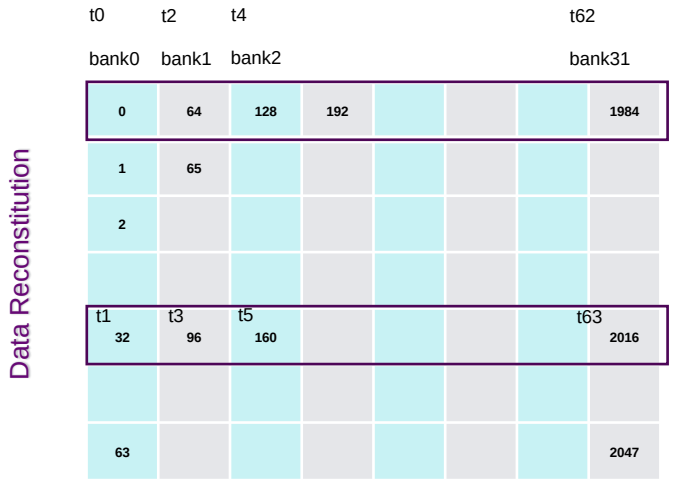
两次事务进行访问，每次事务中32路bank冲突



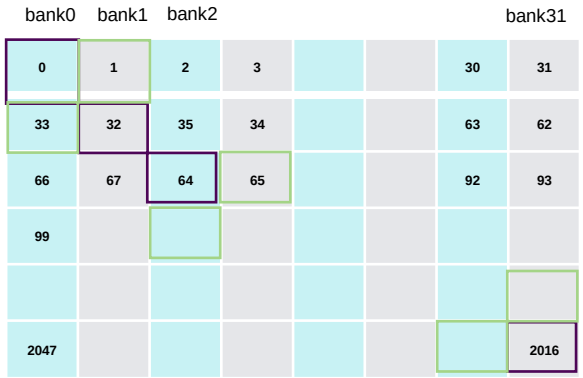
需使用额外shared Memory

两次事务进行访问，每次事务中无bank冲突

Swizzle：通过位运算重排索引，让同一 wave 内不同线程访问的 shared memory 地址分布在不同的 bank 上



两次事务进行访问，每次事务中无bank冲突
写数据是如何保证无bank冲突

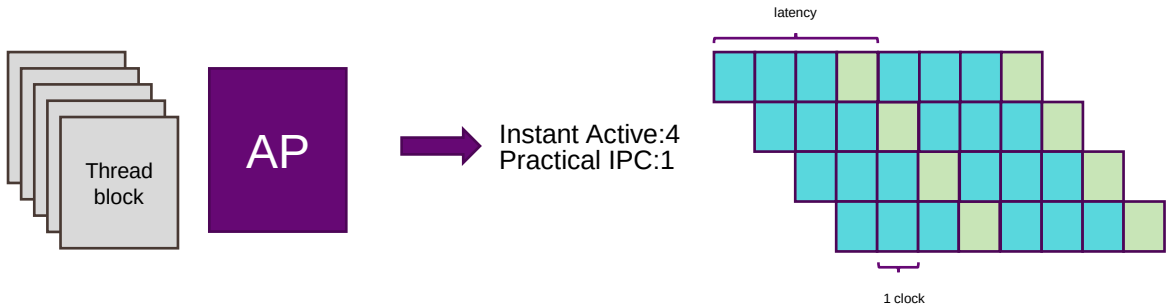


index=row×n+(col⊕(row mod n))

※Case 7: Latency Hiding

举例说明

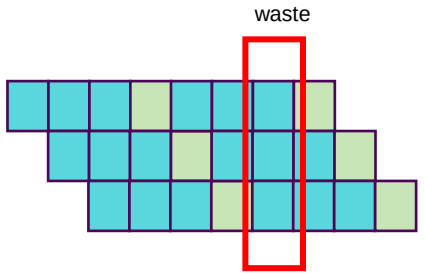
- 每个thread block中，有1条访存指令，latency为3cycle; 紧接着有1条计算指令，latency为1 cycle;
- AP中的计算单元同时可以处理1条计算指令
- 为了达到latency hiding的目的，需要同时在SM上驻留4 x1=4个thread block。



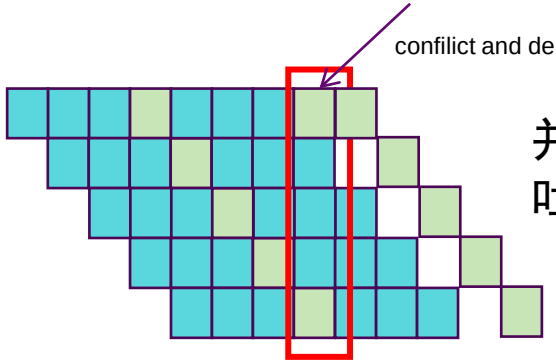
※Case 7: Latency Hiding

- 若Arrival Rate(或Departure Rate)有上确界R_max,

- 给定Latency,则Rx = $\begin{cases} R\_max \\ \frac{1}{L} \times Te \end{cases}$



并发数不足，导致hide latency的能力不够，AP上的计算单元出现空余周期



并发数过高，SM的计算单元不足以吞吐足够的指令，需要进行stall

Thanks!

DISCLAIMER & ATTRIBUTION

This presentation may contain privileged and confidential information and is intended only for MetaX Integrated Circuits (Shanghai) Co., Ltd. and/or its affiliates (hereinafter collectively referred to as “MetaX”). The information presented in this presentation should be kept strictly confidential and should not be allowed to make available to others without written consent from MetaX.

The information presented in this presentation is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors. MetaX makes no representation or warranty of any kind, express or implied, regarding the reliability, availability or completeness of any information on the Slides.

MetaX is the owner of the presentation and all portions thereof. Except as expressly provided, you are prohibited from copying, modifying, distributing, displaying or transmitting any of the contents of the Slides for any purposes.

MetaX, 沐曦, the MetaX logo and combinations thereof are trademarks of MetaX. Other product and company names used in this presentation are for identification purposes only and may be trademarks or trade names of their respective owners.

http://www.metax-tech.com/

TileLang开源训练营

第二讲：国产GPU核心架构与硬件加速指令深度解析

共建AI编译开源生态

DataWhale社区 | GTOC社区 | CNB社区 | AI Infra开源社区 | 沐曦股份 | TileLang项目

GitLink 确实·开源

TileLang开源训练营 | 2026年5月

扫码报名TileLang开源训练营