2023-2024学年春季学期

计算机体系结构安全

Computer Architecture Security

授课团队：史岗，陈李维

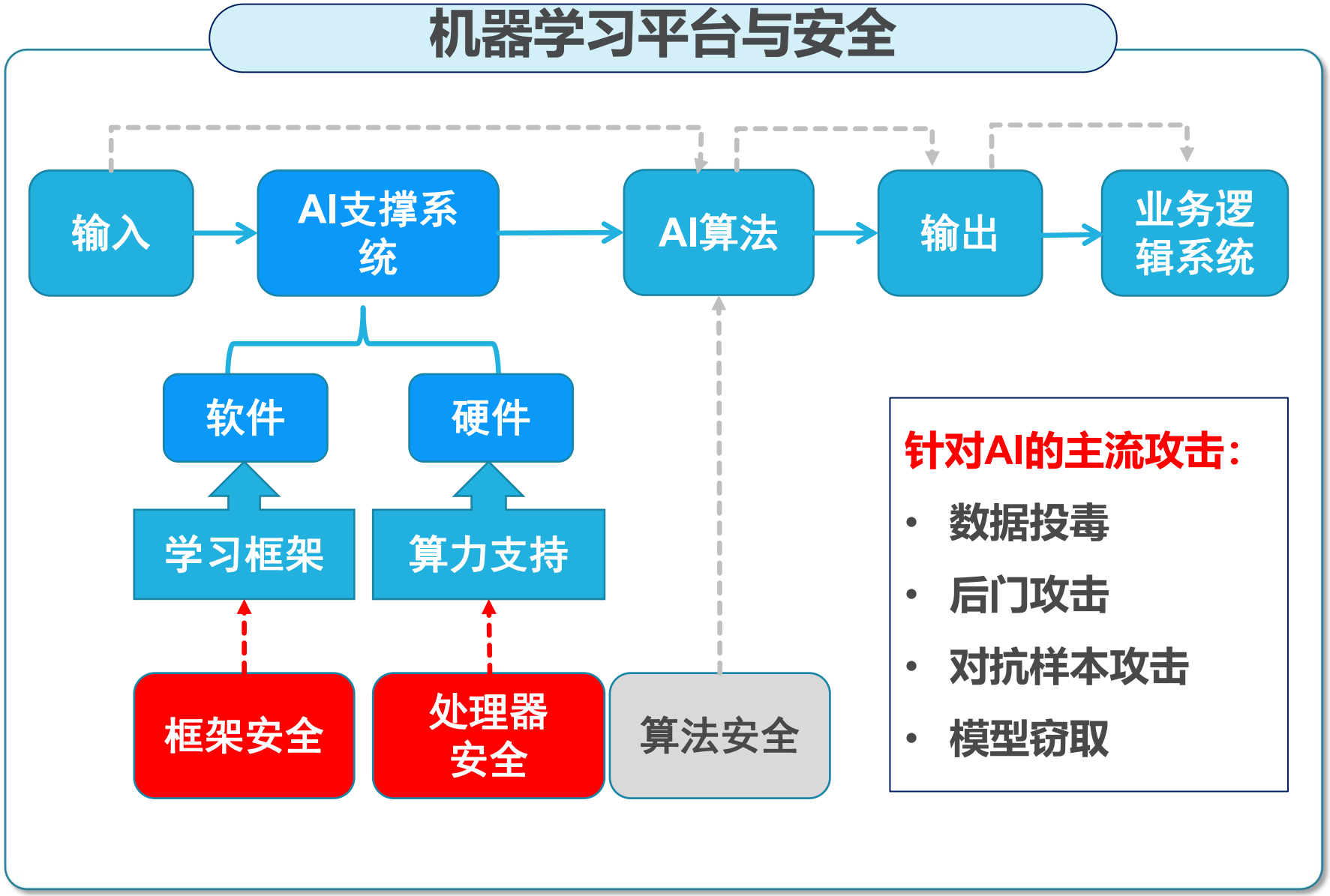
计算机体系结构安全

Computer Architecture Security

[第14次课] 机器学习平台安全架构

授课教师：史岗

授课时间：2024.5.27


机器学习平台与安全
输入
AI支撑系统
AI算法
输出
业务逻辑系统
软件
硬件
学习框架
算力支持
框架安全
处理器安全
算法安全
针对AI的主流攻击：
• 数据投毒
• 后门攻击
• 对抗样本攻击
• 模型窃取

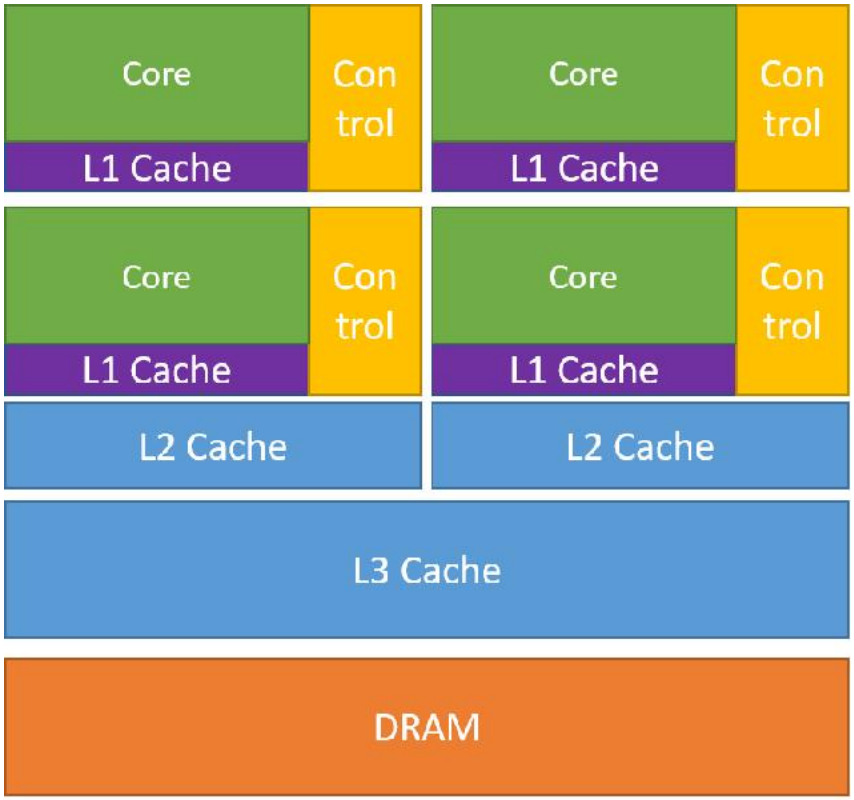
内容概要

- GPU微处理器架构
- GPU微结构安全风险
- AI学习框架安全风险
- 总结

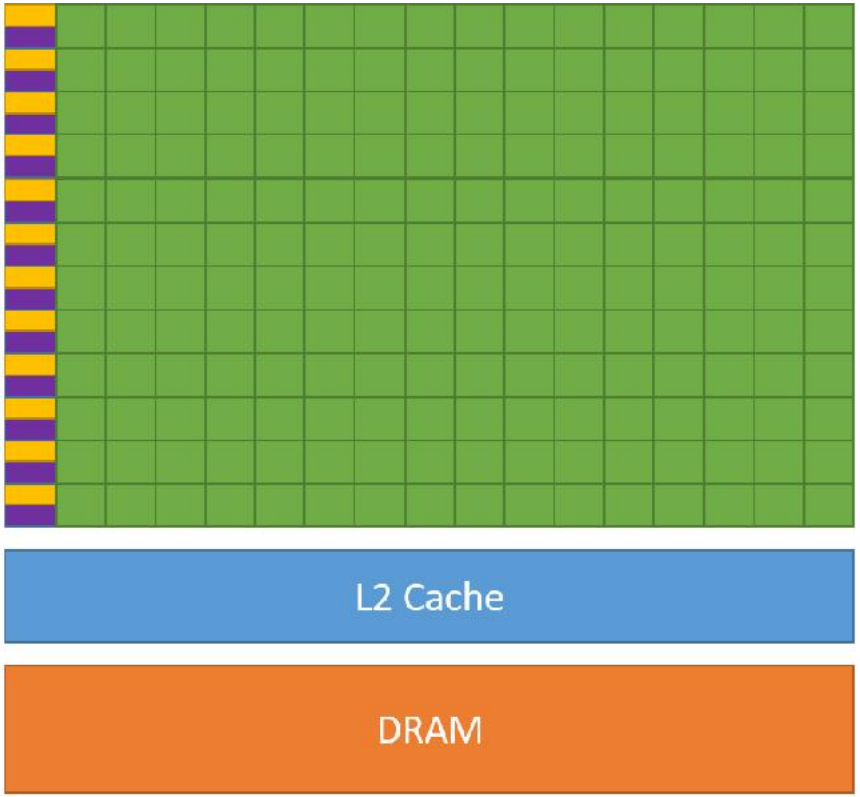
内容概要

- GPU微处理器架构
- GPU微结构安全风险
- AI学习框架安全风险
- 总结

CPU和GPU的不同



CPU



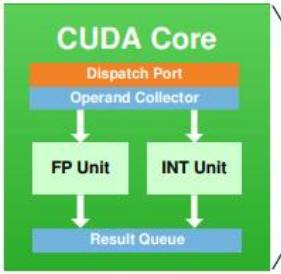
GPU

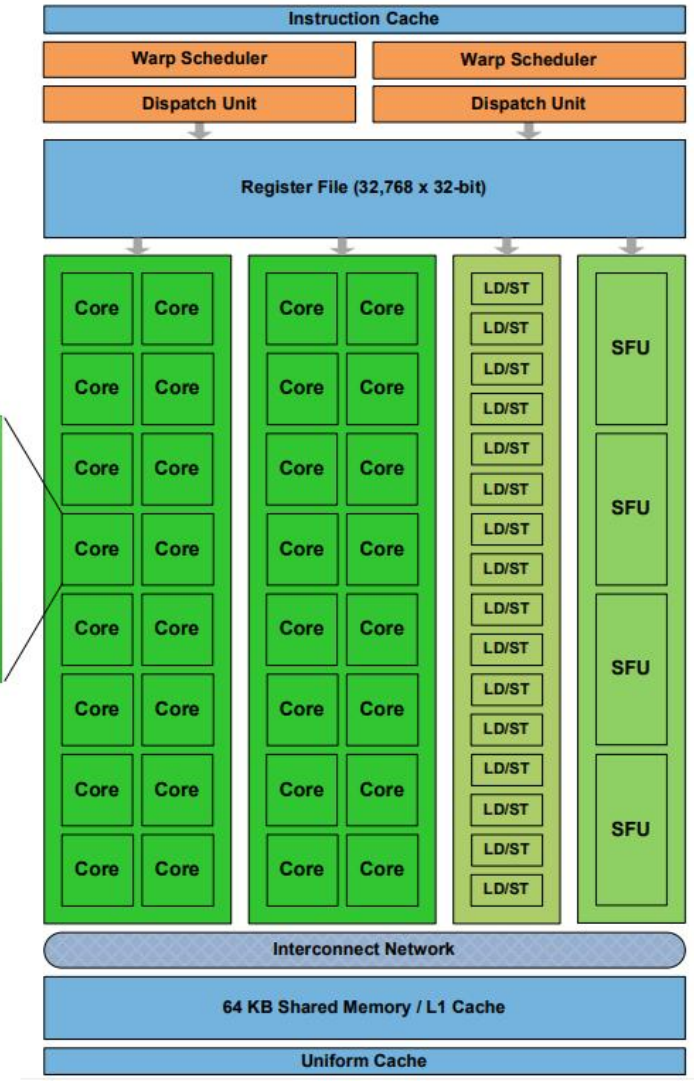
- GPU将更多的晶体管用来数据处理（更多的Core），而不是数据缓存与控制
- GPU相比与CPU有更高的指令吞吐量和内存带宽
- GPU的Core相对CPU的Core更简单，主要进行乘加操作，控制逻辑也更简单

GPU处理器微架构

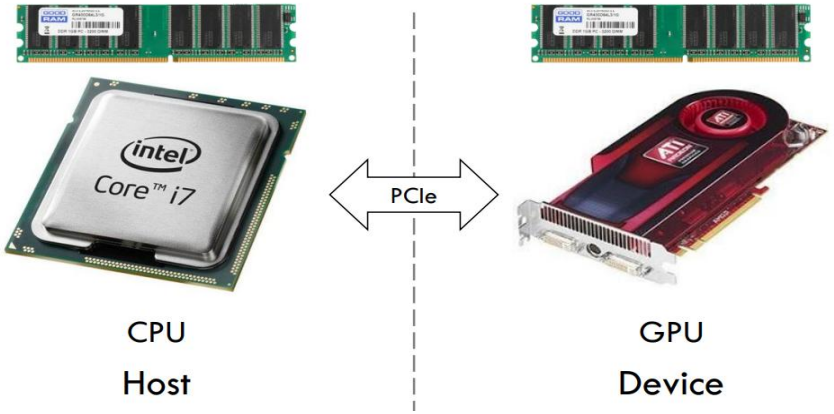
- Nvidia 流式多处理器（Streaming Multiporcessor，SM）
  - 计算部件：CUDA Core 完成整型和浮点计算

    SFU 完成三角函数、开根等特殊计算
  - LD/ST单元：数据读取到缓存或DRAM中
  - L1 Cache：减少访存延迟
  - Shared Memory：用于线程间共享
  - Uniform Cache：常数变量的缓存
  - Warp Scheduler：线程束调度器
  - Dispatch Unit：指令分配到计算和存储单元

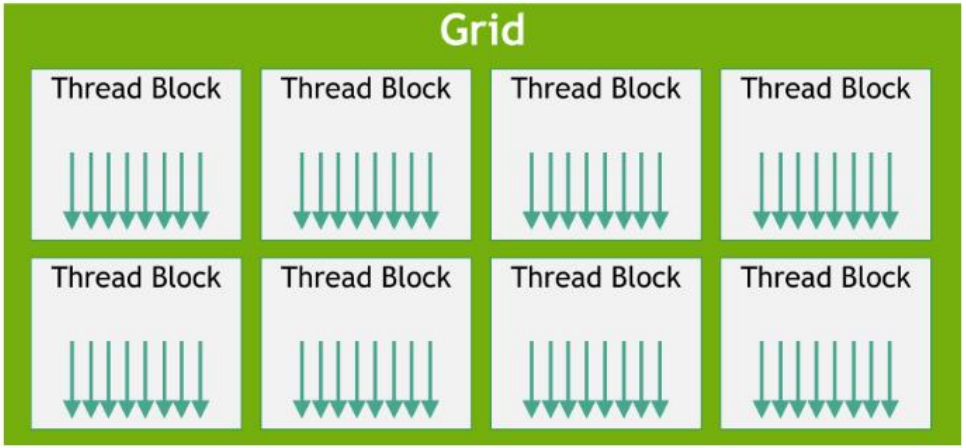




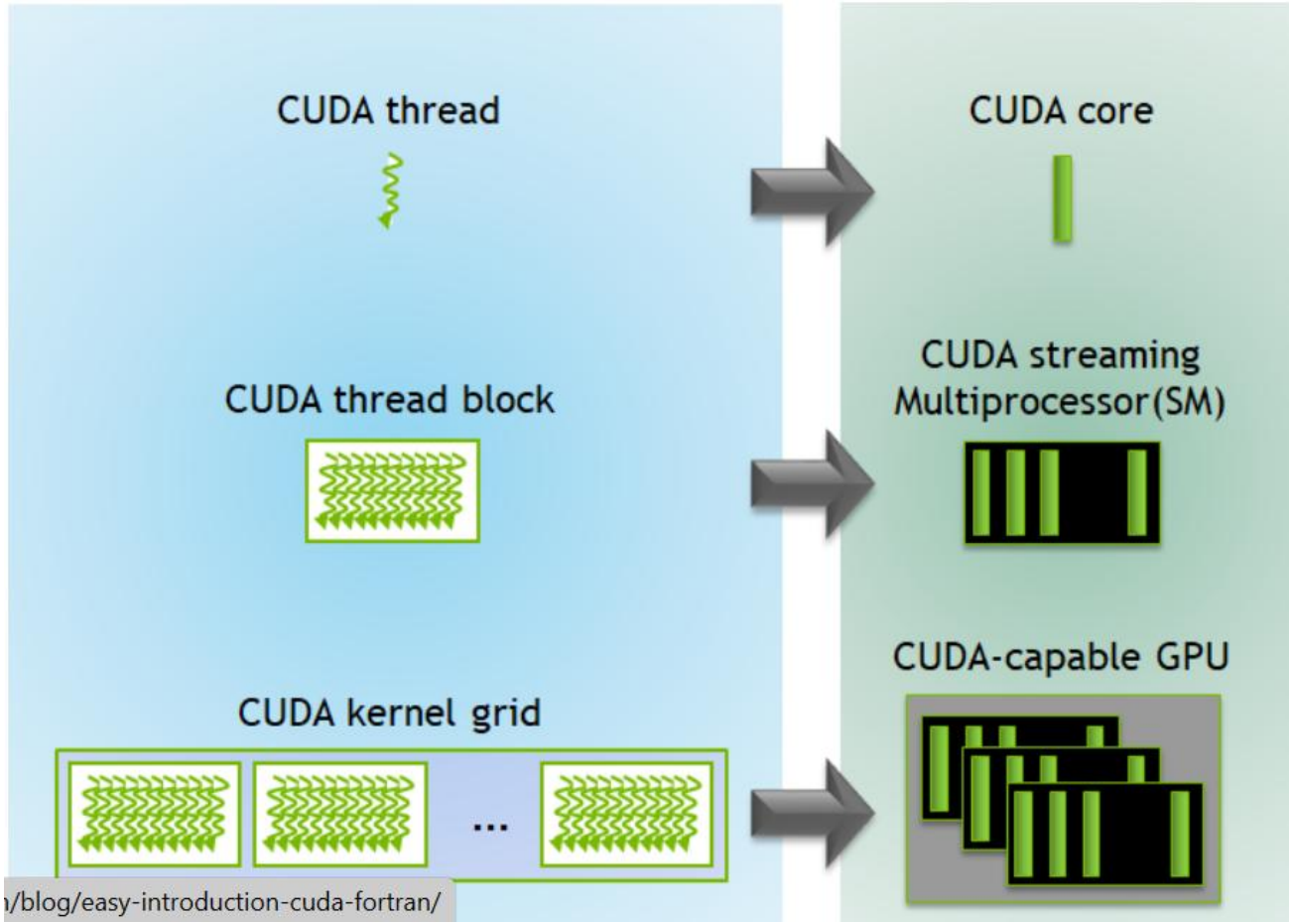
- CUDA/GPU的编程模型
  - 术语
    - Host Vs. Device



    - Grid -> Block -> Thread
      - Thread：最小的执行单元
      - Block：一组可以共享和同步的线程组成的单元。块之间则没有同步关系。
      - Grid：一组Block组成的单元，完成一个更大规模计算任务。

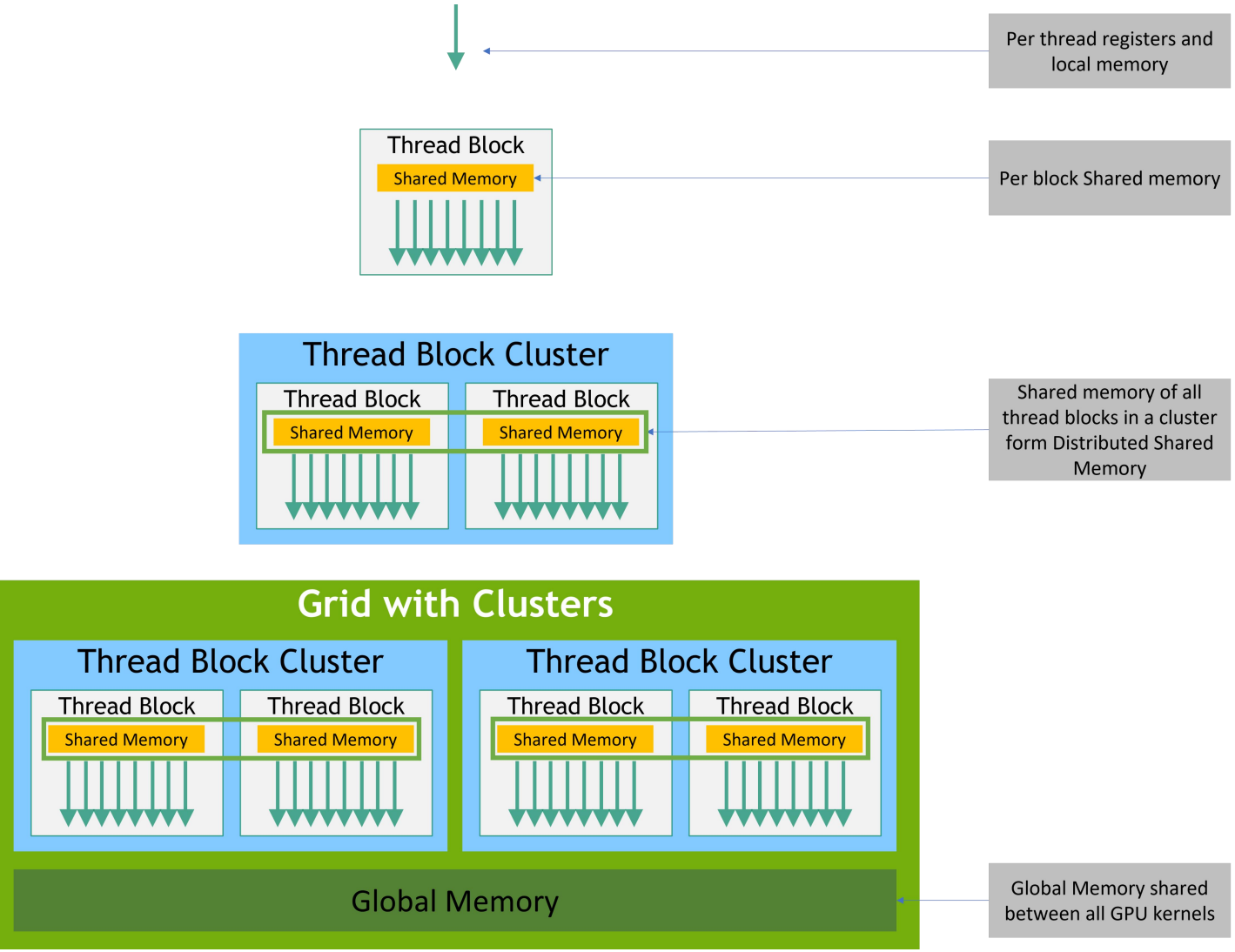


- CUDA/GPU的编程模型
  - 编程抽象与物理计算资源的对应关系

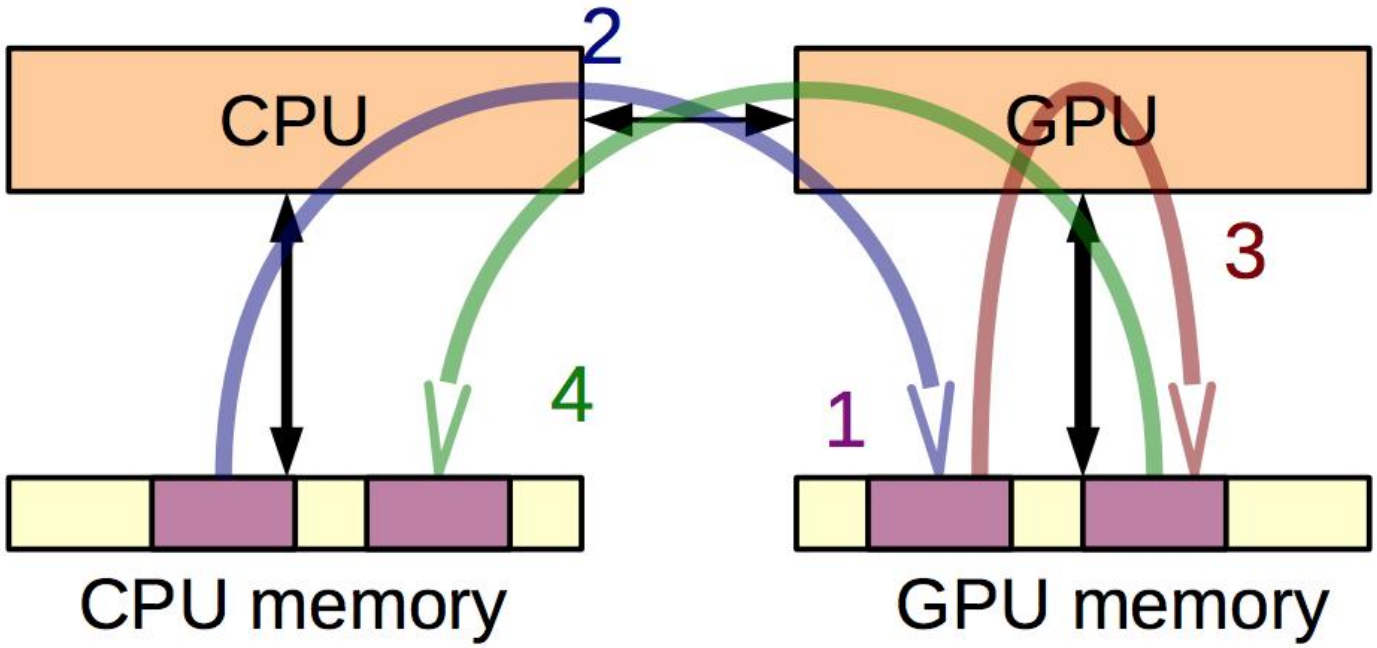


CUDA/GPU的编程模型

存储模型



- CUDA/GPU的编程模型
  - 存储模型



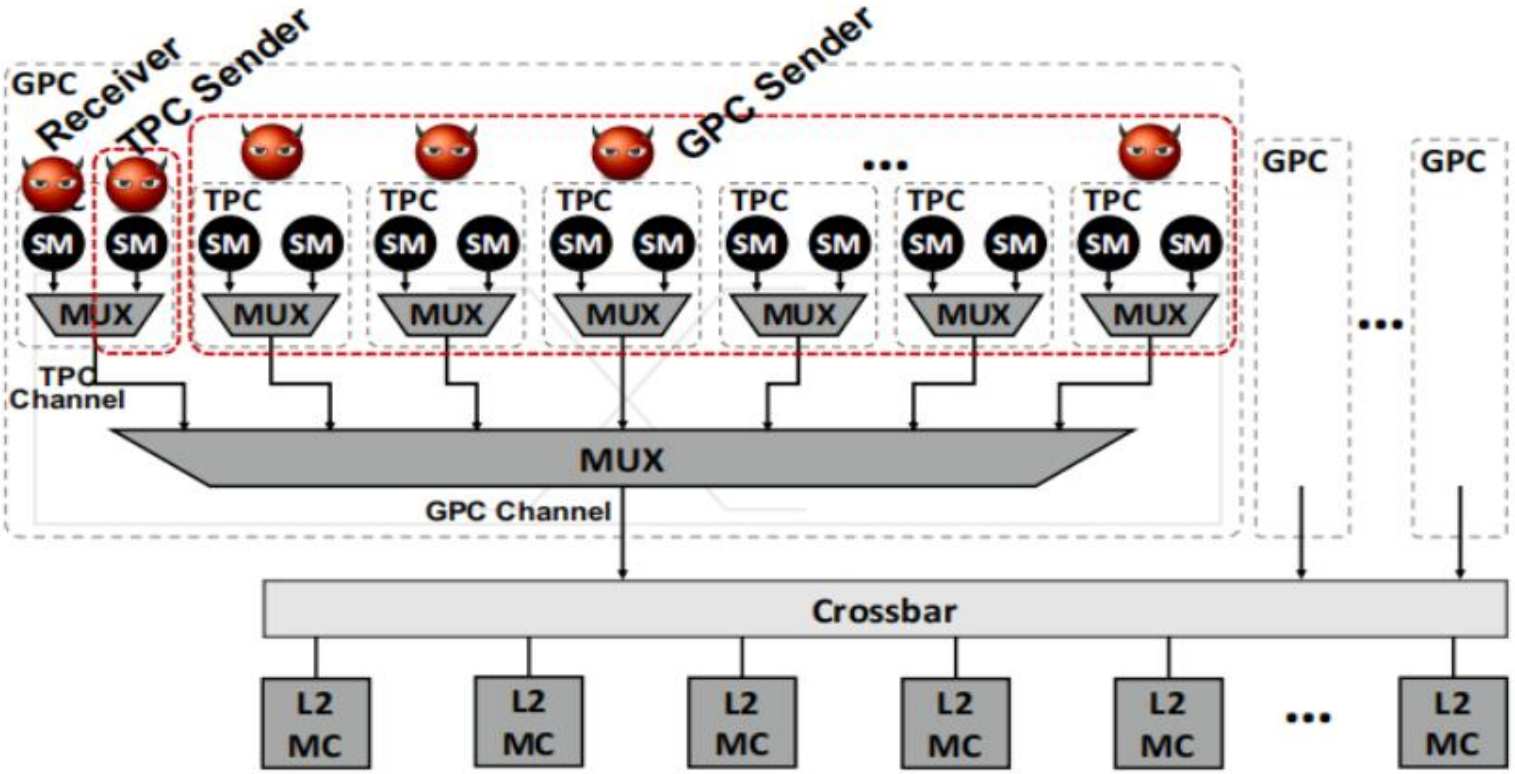
1. 分配GPU的内存
2. 输入数据从Host CPU拷贝到GPU Device
3. 在GPU内存里进行计算
4. 计算结果从GPU Device传会给Host CPU

内容概要

- GPU微处理器架构
- GPU微结构安全风险
  - 隐蔽信道风险
  - 存储泄漏风险
- AI学习框架安全风险
- 总结

GPU侧信道攻击

- “Network-on-Chip Microarchitecture-based Covert Channel in GPUs” （Micro’ 2021)
  - **主要方法：利用GPU层次结构和互连中的多路复用器建立一个隐蔽通道。**
  - **预备知识+逆向工程获得：**
    - TPC：2个SM构成，形成一个纹理处理簇（Texture Processing Cluster ）
    - GPC：多个TPC构成，形成一个图形处理族（Graphic Processing Cluster ）
    - GPU：一个Nvidia V100 由6个GPC组成，每个GPC由7个TPC组成

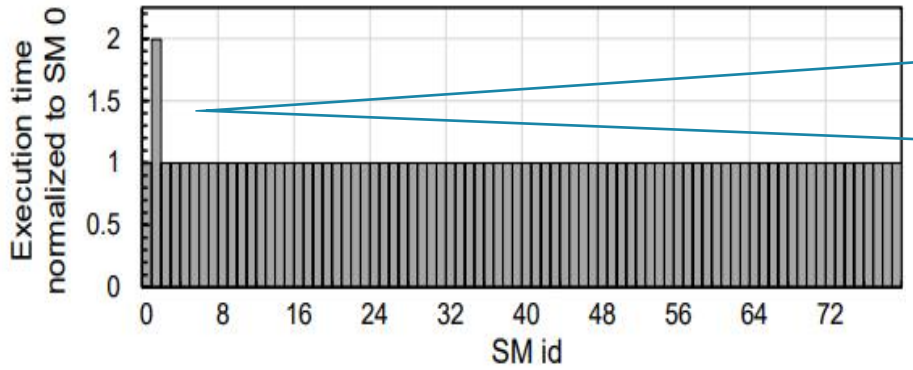


TPC通道的拥塞发现

Algorithm 1: Reverse Engineering TPC Organization

```
input: config_smid   // SM id to be activated
Procedure Memory Write Test(config_smid)
    sm_id = get_smid()
    Amount = array_size / thread_block_size
    base = Amount · thread_id
    if sm_id == 0 then
        for i ← 0 to Amount do
            arr_A[base + i] = thread_id
        end
    else if sm_id == config_smid then
        for i ← 0 to Amount do
            arr_B[base + i] = thread_id
        end
    return
```

通过激活固定的sm_id为0的SM以及其他给定sm_id的SM来进行内存写操作，测试过程中的耗时长短来判断SM之间的层次关系。



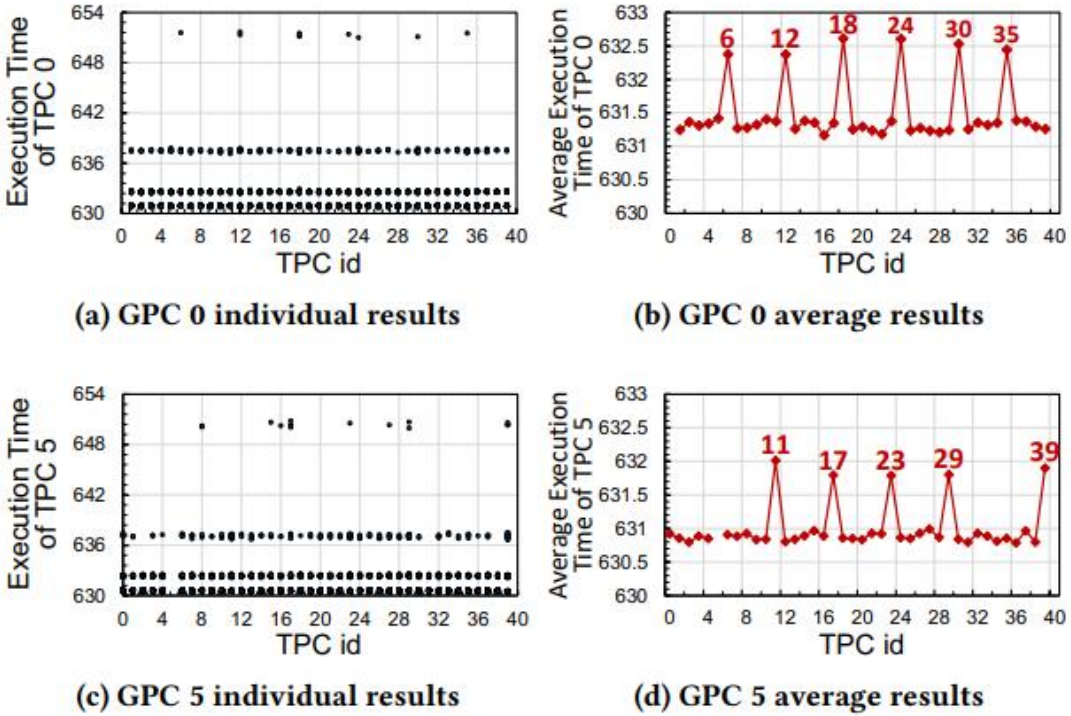
发现：

最长耗时恰好为其他耗时的两倍，

推断：

SMi and SMi + 1两者存在带宽争用的情况，构成隐蔽时间信道的条件。

GPC通道的拥塞发现



(a) GPC 0 individual results

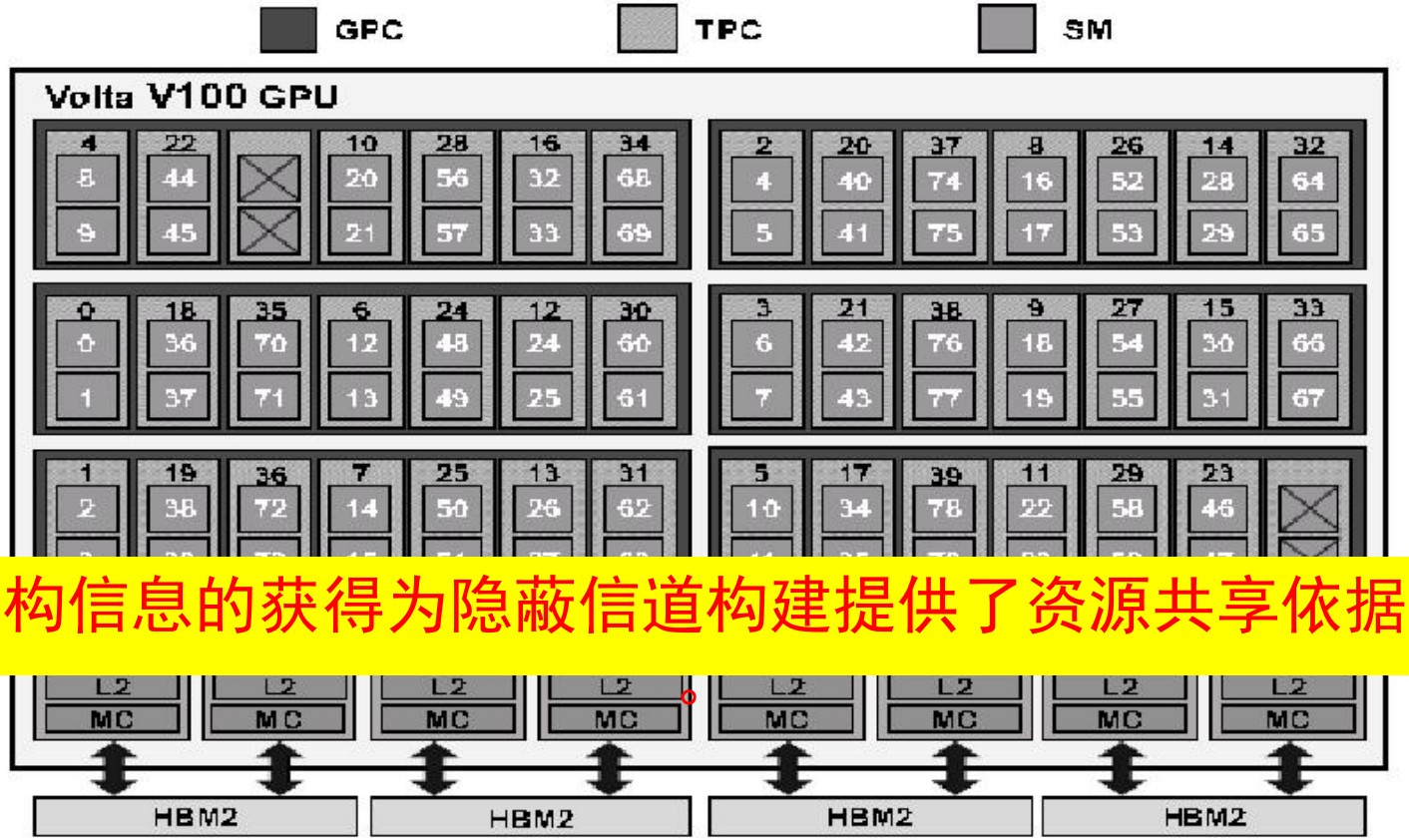
(b) GPC 0 average results

(c) GPC 5 individual results

(d) GPC 5 average results

1. 固定一个每次都运行的TPC，再依次遍历其它TPC，1个TPC每次运行1个SM
2. 随机选择5个TPC，每个TPC运行1个SM，作为背景流量，重复200次求平均
3. 统计平均时间长的与固定的那个TPC位于同一个GPC
4. 重复上述测试找出每个GPC包含的TPC

逆向还原出来的Nvidia V100架构



结构信息的获得为隐蔽信道构建提供了资源共享依据。

Figure 4: NVIDIA Volta V100 GPU's Logical to Physical Core Mapping.

构成隐蔽信道的时间差来源分析

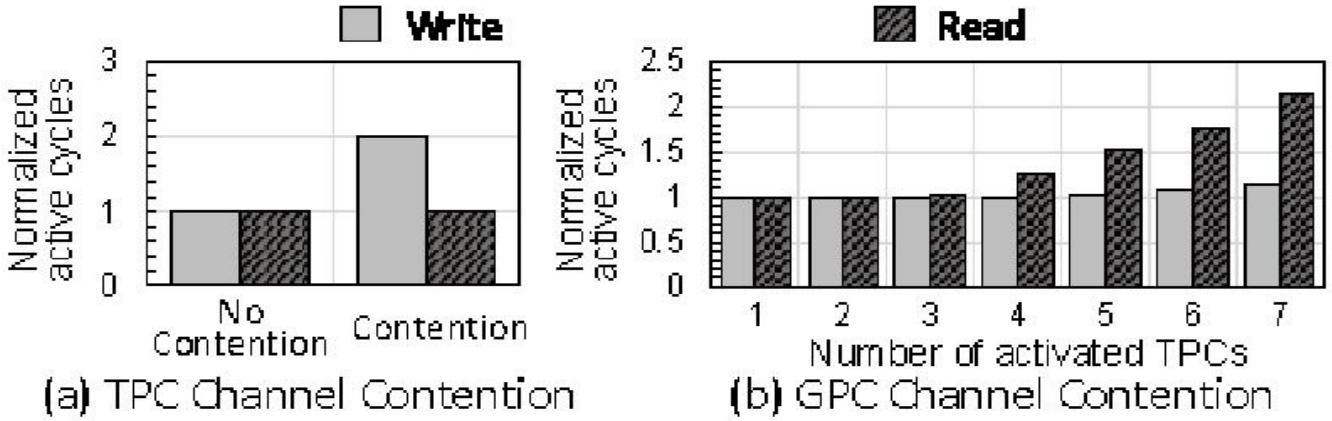


Figure 5: Performance impact of read and write memory accesses on (a) TPC channel and (b) GPC channel.

分析结论：

TPC通道在两个SM同时写操作时发生拥塞
GPC通道在4个及以上TPC同时用读操作时发生拥塞

以上特点为隐蔽信道构建提供状态变化的依据。

构成隐蔽信道的同步机制

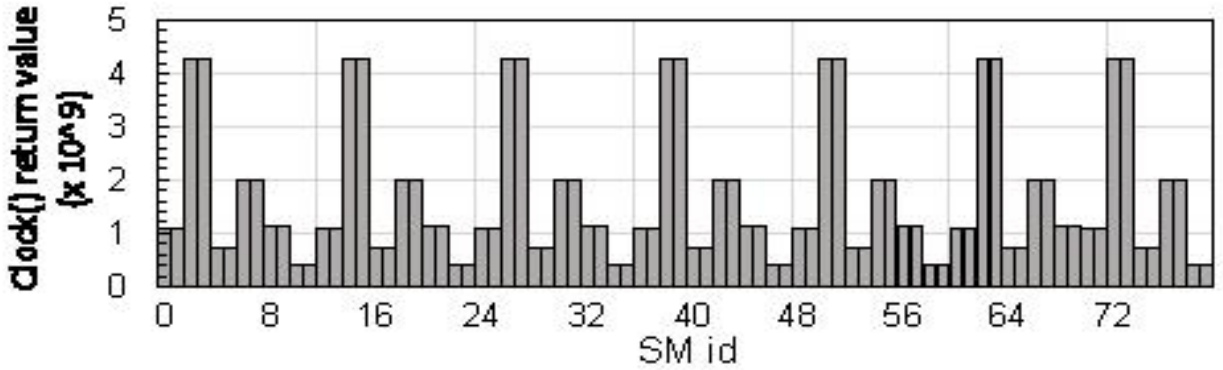


Figure 6: Distribution of clock() return values across the different SMs in the Volta GPU.

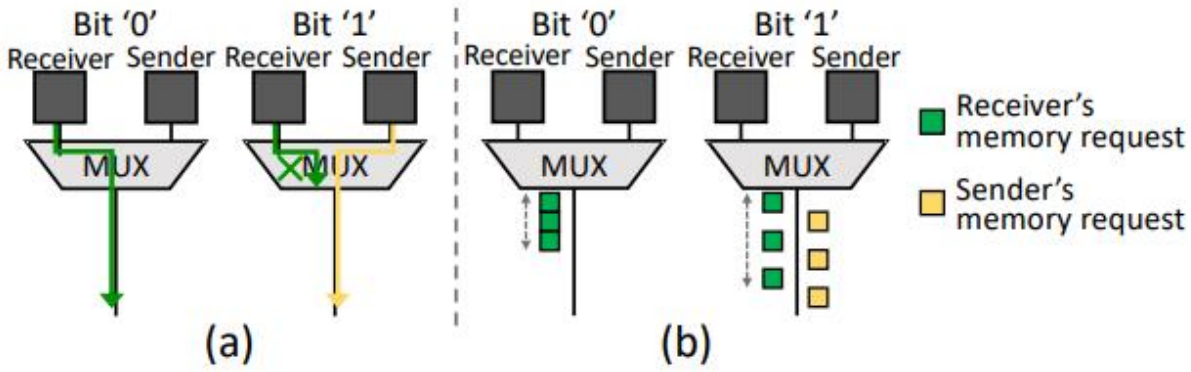
规律分析：

同一个TPC的SM时钟寄存器值的偏差（个位数）远小于读取L2 cache的时间（200~250个时钟周期）

同样，同一个GPC的SM时钟寄存器值的偏差远小于读取L2 cache的时间

使用SM内时钟寄存器为侧信道构建参考时钟。

隐蔽信道的构成与信息泄漏



"0"：发送SM不向L2 Cache提交请求，接收SM向L2 Cache提交请求，NOC上不发生拥塞

"1"：发送SM向L2 Cache提交请求，接收SM向L2 Cache提交请求，NOC上发生拥塞

注意：为了实现隐蔽信道，还需要知道GPU的调度机制，将Send/Receive线程加载到合适的SM上

隐蔽信道的算法示意

Algorithm 2: Interconnect-based Covert Channel

$D_{send}[N]$, $D_{receive}[N]$: N bit data to transmit and receive, respectively.
T_{slot}: Size of timing slot shared between sender and receiver.

```
Procedure Sender operations()
    Synchronization()
    for i ← 0 to N do
        if D_send[i] == 1 then
            Access L2 cache    // Invoke channel-contention
        else
            Do nothing         // No channel-contention
        end

        busy waiting for remaining T_slot
        if i % Sync_period == 0 then
            Synchronization()
        end
    end
    return
```

```
Procedure Receiver operations()
    Synchronization()
    for i ← 0 to N do
        AccessTime[i] = Measured L2 access latency
        if AccessTime[i] > Threshold then
            D_receive[i] = 1   // Channel is under contention
        else
            D_receive[i] = 0   // Channel is contention free
        end
        busy waiting for remaining T_slot
        if i % Sync_period == 0 then
            Synchronization()
        end
    end
    return
```

发送方：根据Dsend中的数据（假设为泄漏的数据）发起L2 Cache访问
接收方：根据访问L2的延迟，计算并估计泄漏的值
过一段时间进行一次同步，以消除时间误差的积累

攻击效果：同一个TPC的隐蔽信道信息泄漏带宽约为24Mb/s，错误率接近0

此种隐蔽信道的扩展讨论

1. 实验验证，在NVIDIA kepler、Pascal、Truing等架构下同样存在

2. NVIDIA Ampere架构GPU采用MIG（multi-instance GPU)为不同的用户或实例提供独立的GPC和存储分区，导致该方案部分失效。也就是说木马程序和间谍程序在同一个实例中才有效。

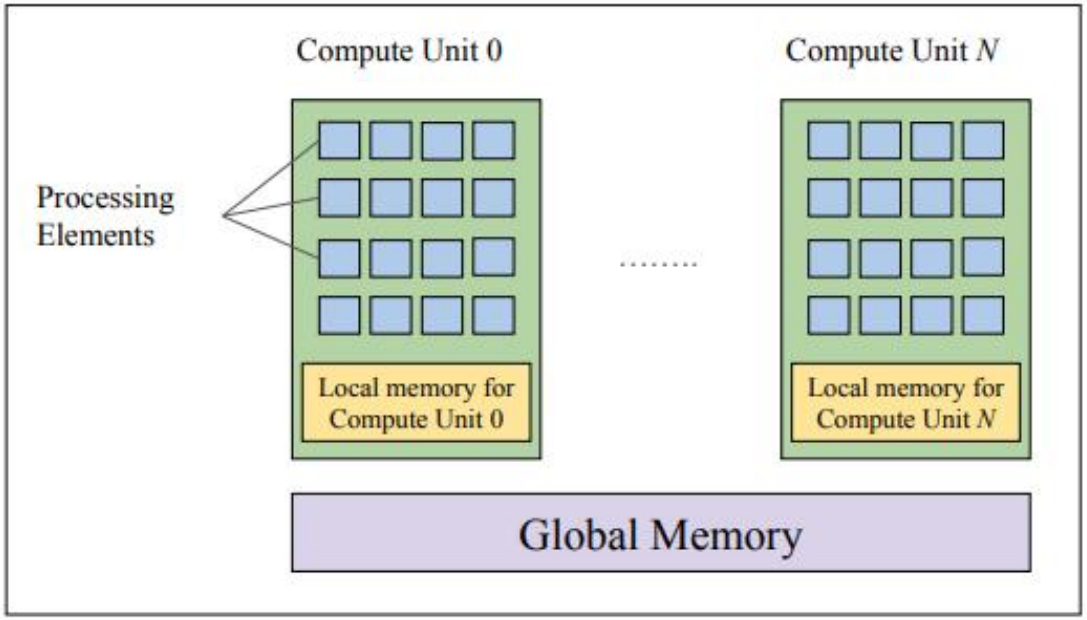
3. AMD架构的GPU采用OpenCL编程模型，没有提供和CUDA一样的可以探测GPU片上网络架构的接口，也没用时钟获取接口，因此该方案在AMD架构下也无法生效。

内容概要

- GPU微处理器架构
- **GPU微结构安全风险**
  - 隐蔽信道风险
  - **存储泄漏风险**
- AI学习框架安全风险
- 总结

GPU Local Memory泄露利用

- “LeftoverLocals: Listening to LLM Responses Through Leaked GPU Local Memory”
  - **主要方法：利用GPU存储隔离缺陷，实现Apple、高通、AMD的GPU敏感信息泄露**
  - **预备知识：**
    - 不同的计算计算单元（CU）拥有不同大小的local memory,一般在16KB~64KB之间，并被CU内部的处理元件（PE）共享
    - local memory一般存放深度学习模型的输入/输出和权重



GPU Local Memory泄露利用

Listener Kernel

```
1  __kernel void listener(__global volatile
2                         int *dump)
3  {
4
5    local volatile int lm[L_SIZE];
6
7    for (int i =  get_local_id(0);
8             i <  L_SIZE;
9             i += get_local_size(0))
10   {
11
12     dump[L_SIZE*get_group_id(0)+i]=lm[i];
13   }
14 }
```

L_SIZE：local memory大小
lm：local memory中的空间
dump：全局内存中的空间

Listener中多个线程并行读取local memory中的内容。

Writer Kernel

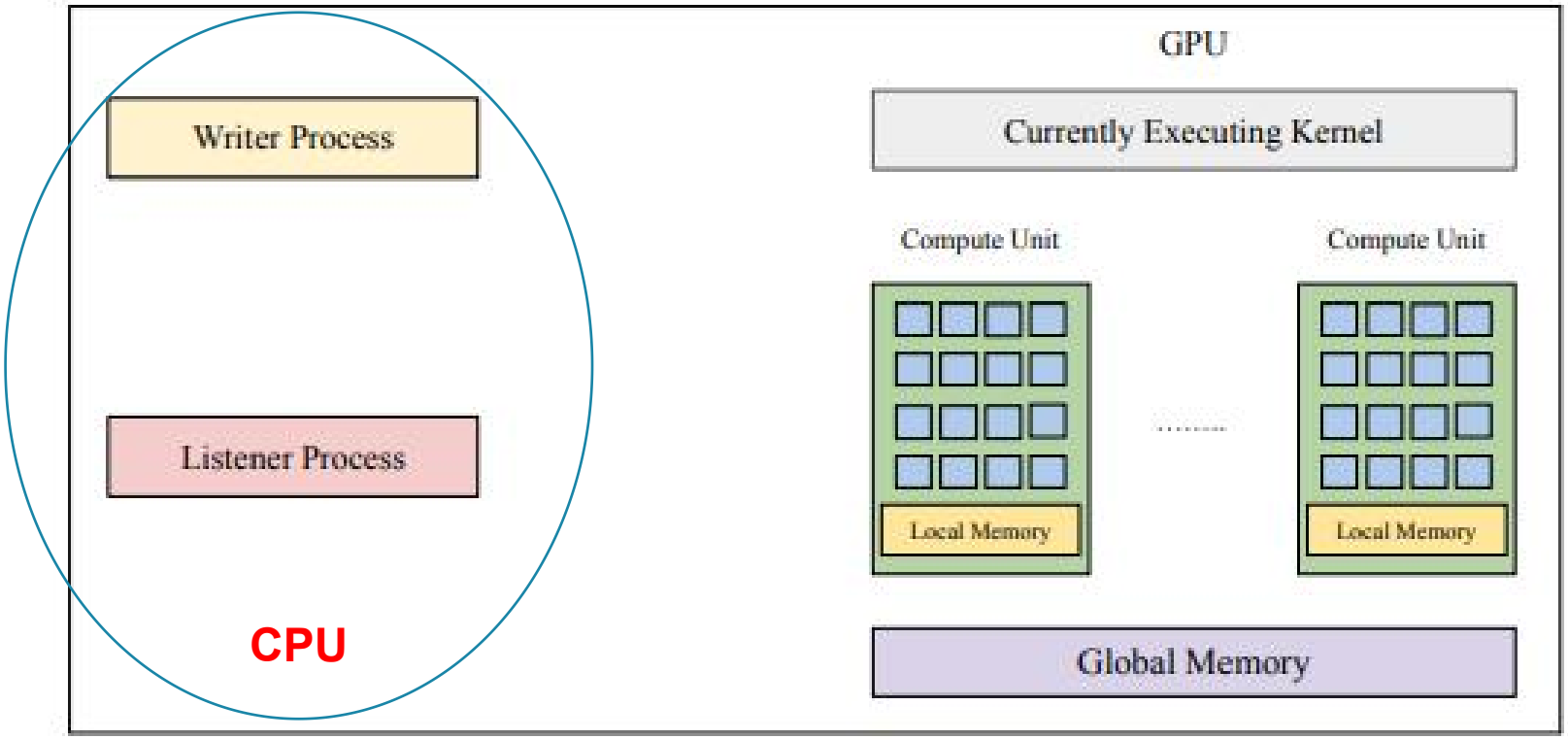
```
1  __kernel void writer(__global volatile
2                       int *canary)
3  {
4    local volatile int lm[L_SIZE];
5    for (uint i = get_local_id(0);
6              i < L_SIZE;
7              i += get_local_size(0))
8    {
9      lm[i] = canary[i];
10   }
11 }
```

L_SIZE：local memory大小
lm：local memory中的空间
canary：全局内存中的初始值

Writer中多个线程并行地将全局内存中数据读入local memory。

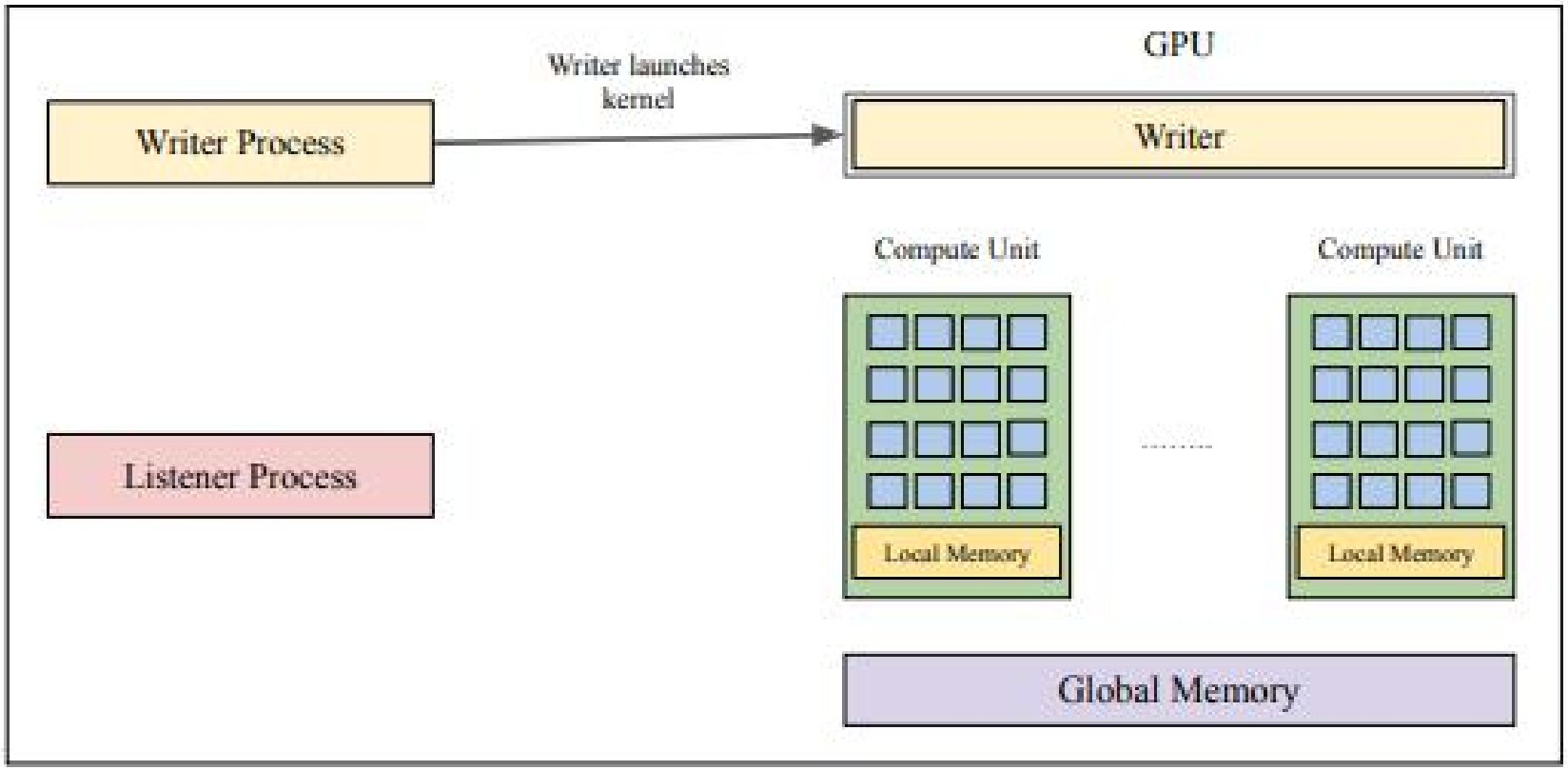
GPU Local Memory泄露利用

攻击步骤1： GPU初始化，攻击者(监听进程）和受害者（写进程）在同一个GPU设备上运行



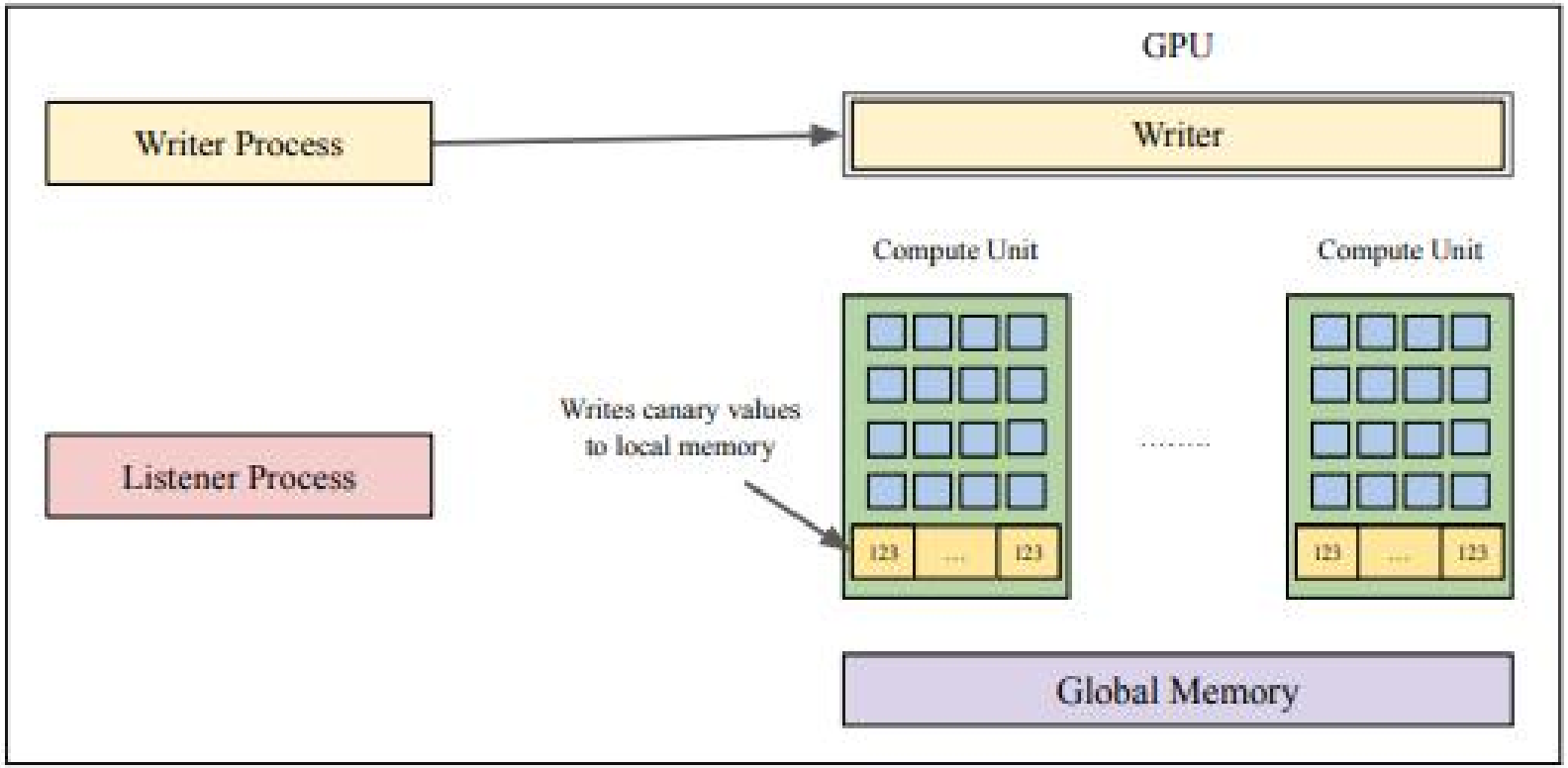
GPU Local Memory泄露利用

攻击步骤2：受害者（写进程）启动核函数进行local memory写操作



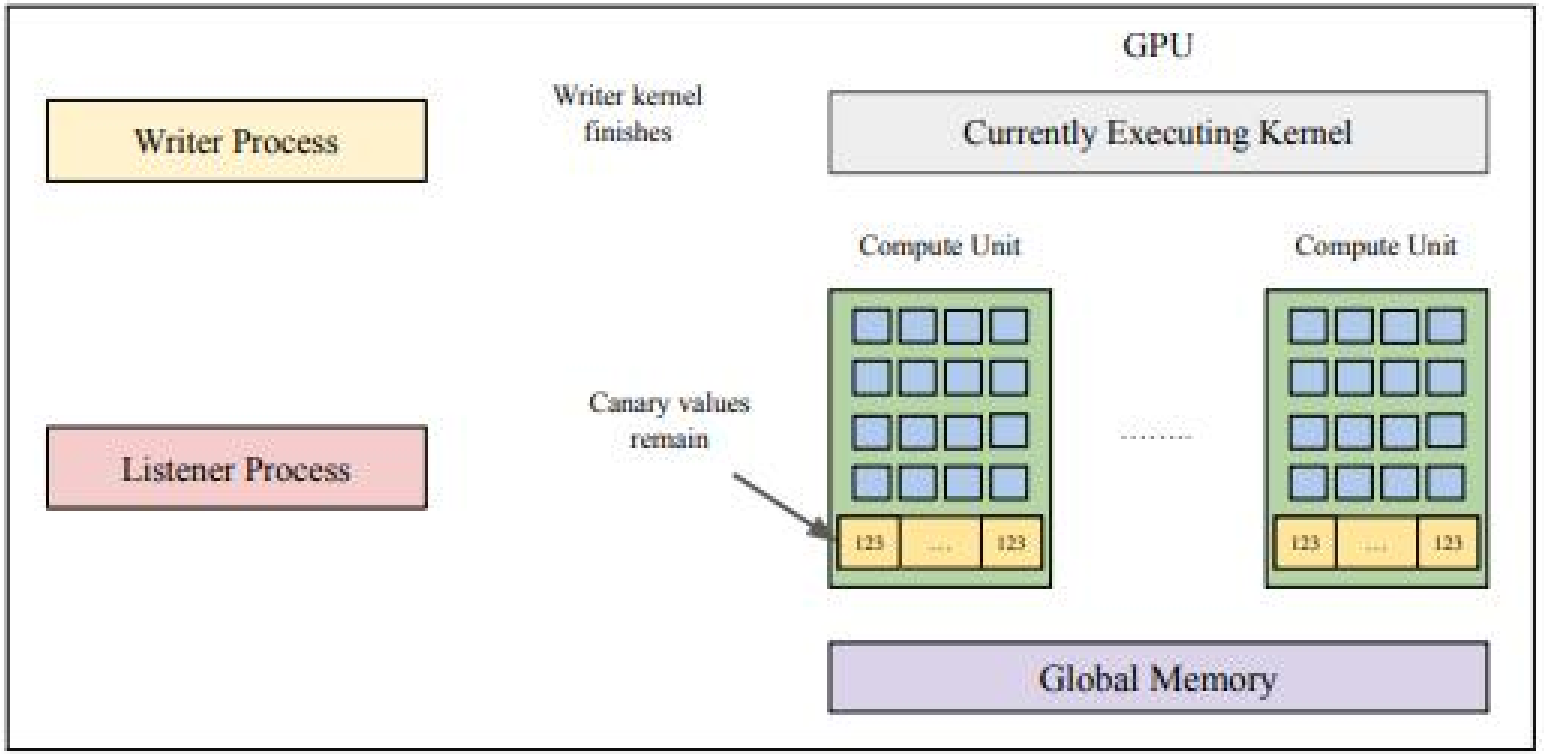
GPU Local Memory泄露利用

攻击步骤3：受害者（写进程）的核函数在所有计算单元的local meomory写入数值“123”



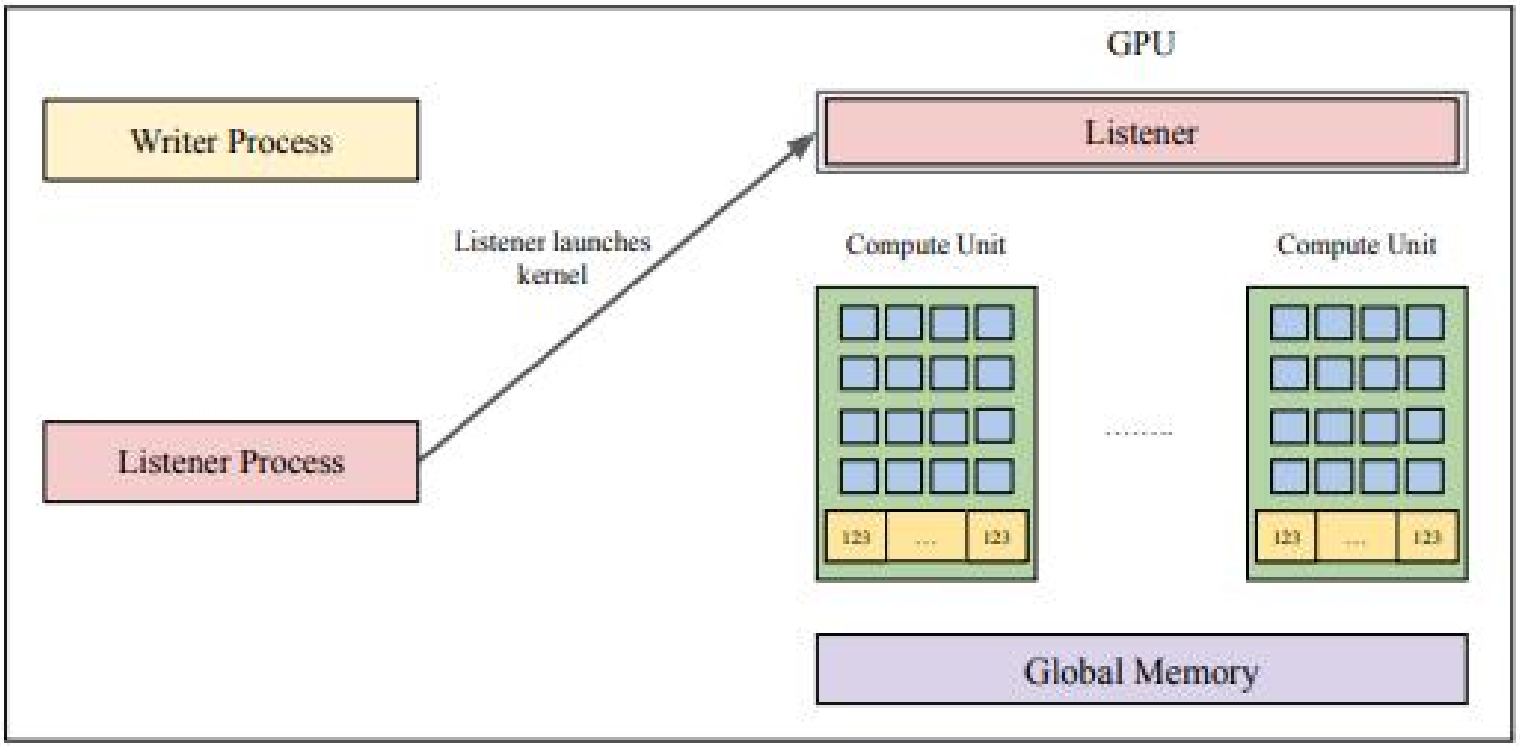
GPU Local Memory泄露利用

攻击步骤4：受害者（写进程）核函数写操作完成。在存在缺陷的系统环境中，Local memory中留存了写入的内容。



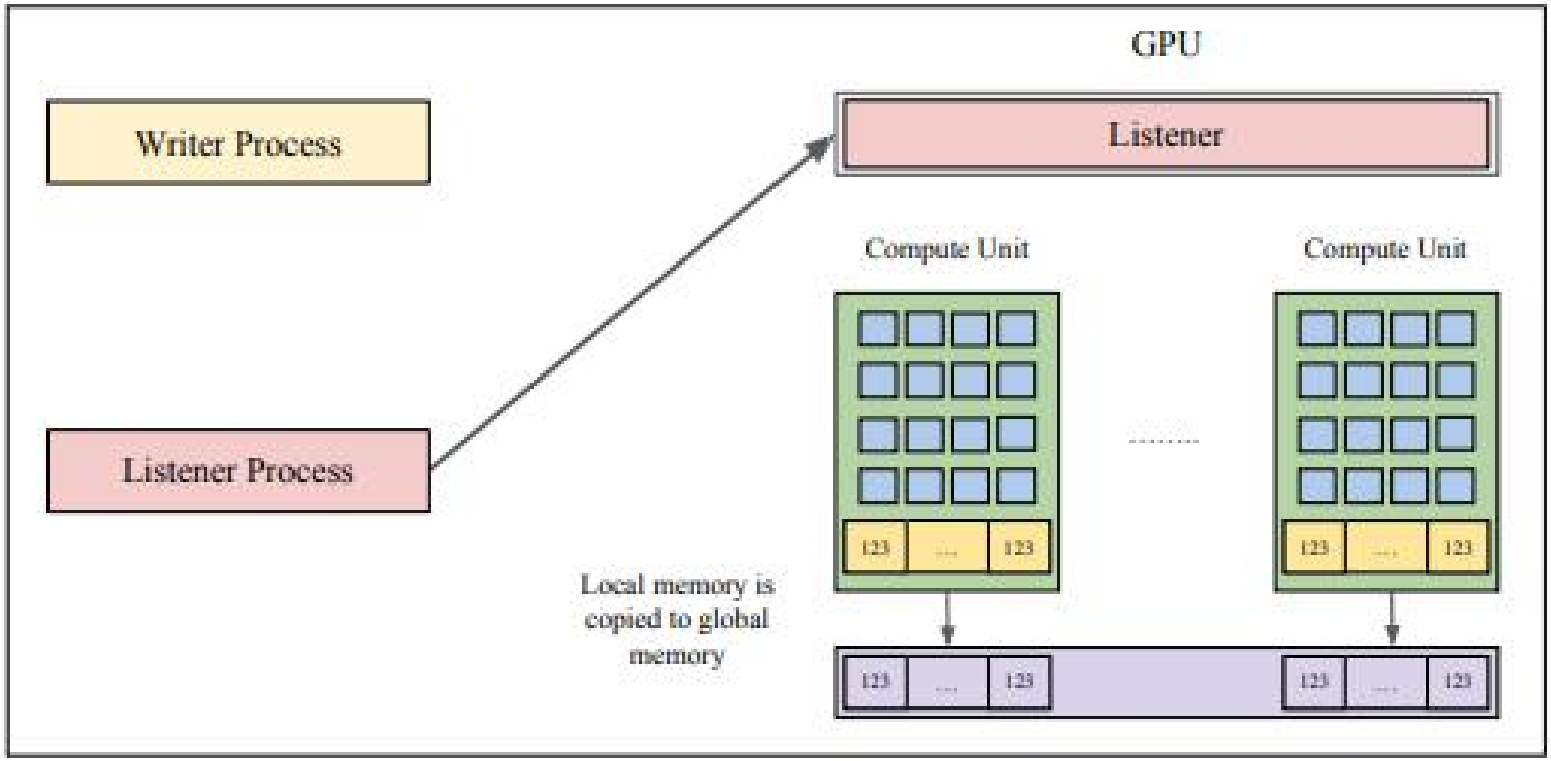
GPU Local Memory泄露利用

攻击步骤5：攻击者（监听进程)启动核函数独占GPU。



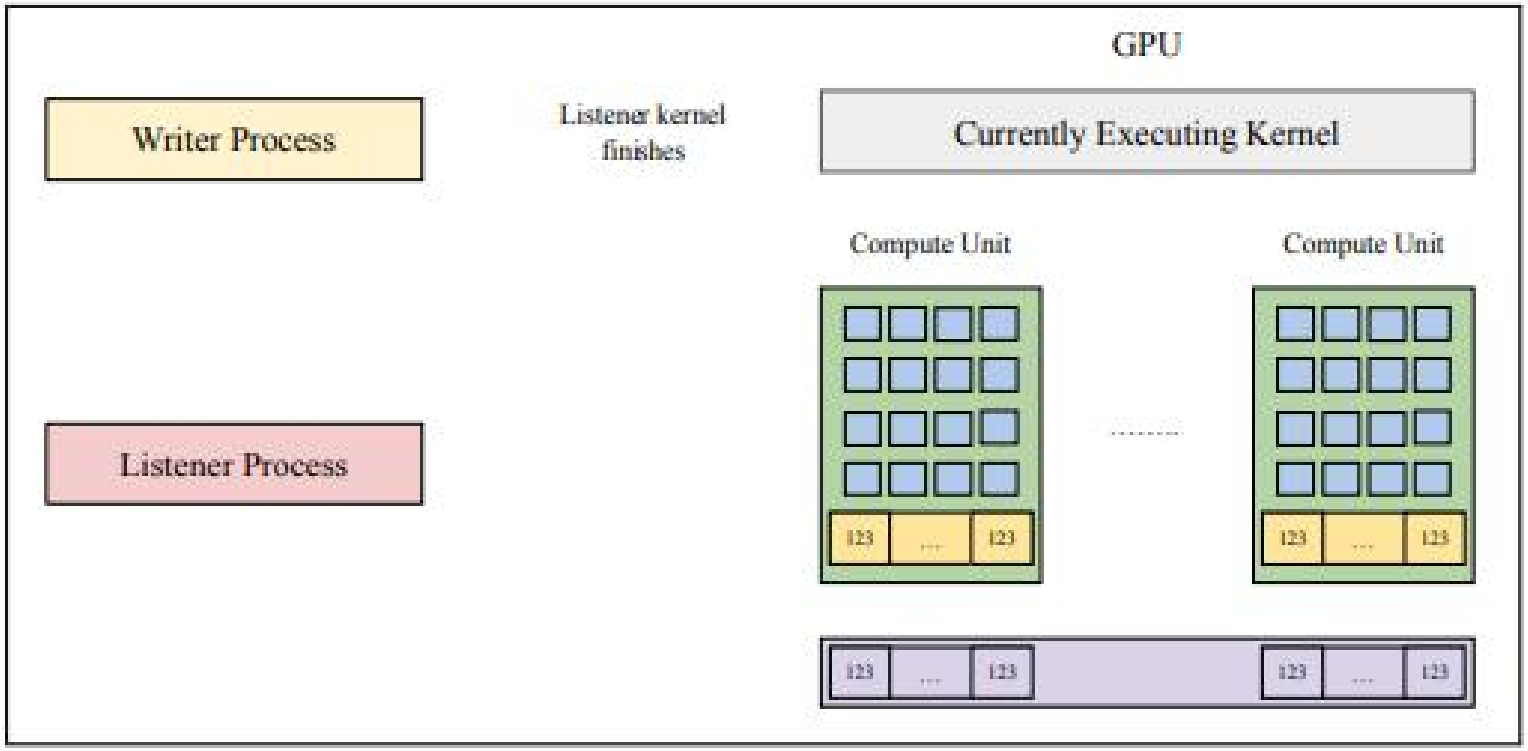
GPU Local Memory泄露利用

攻击步骤6：攻击者（监听进程）将所有local memory的值加载到global memory中。



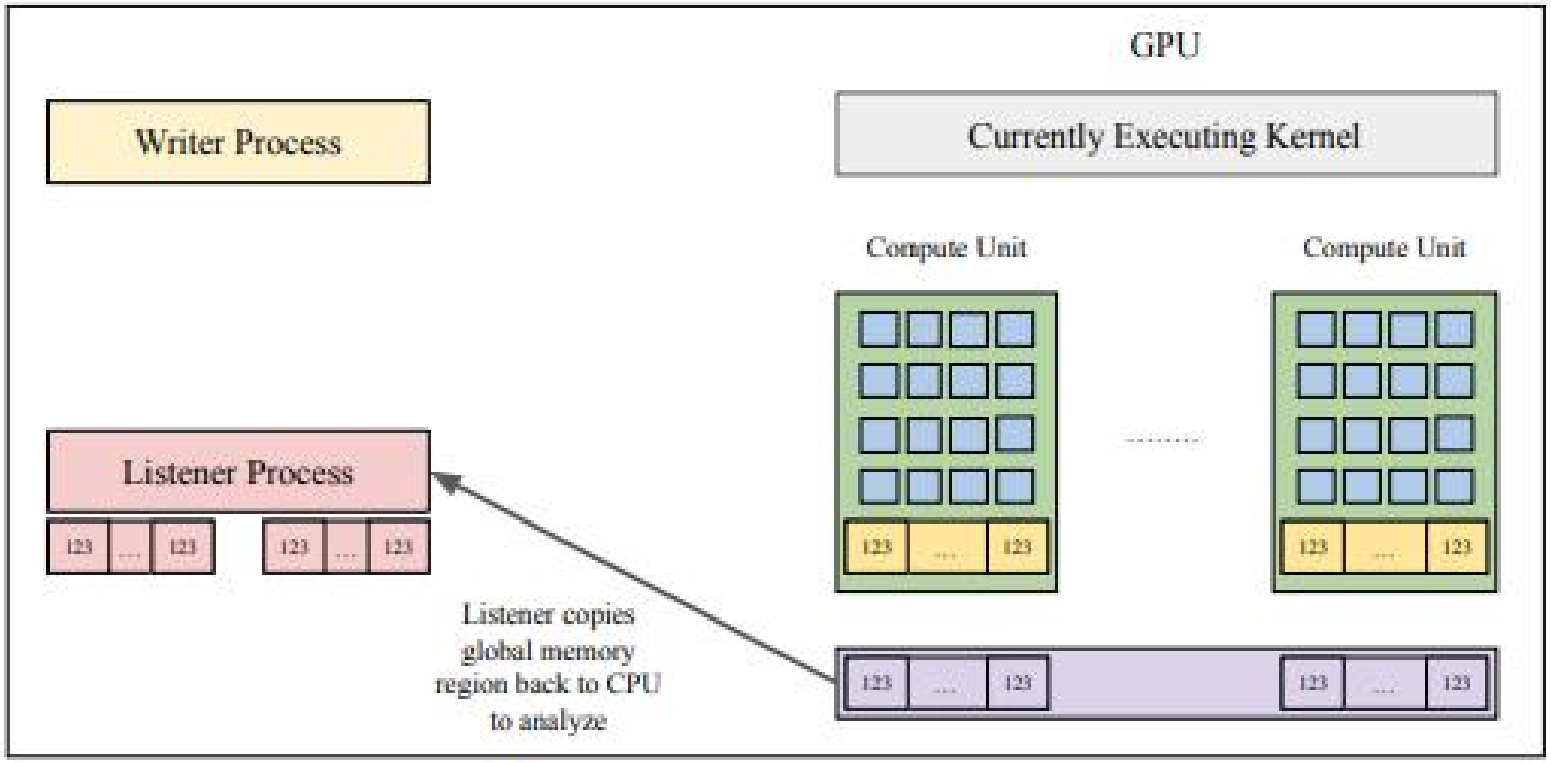
GPU Local Memory泄露利用

攻击步骤7：攻击者（监听进程）核函数完成。 此时，全局内存中包含本地内存的转储



GPU Local Memory泄露利用

攻击步骤8：攻击者（监听进程）可以将global memory内容复制回 CPU 并检查泄漏的数据。



Local Memory泄露的扩展讨论

1. 该漏洞影响面覆盖Apple, Qualcomm和 AMD 等厂商，且利用难度低，在移动设备、云平台等环境下都可被利用。

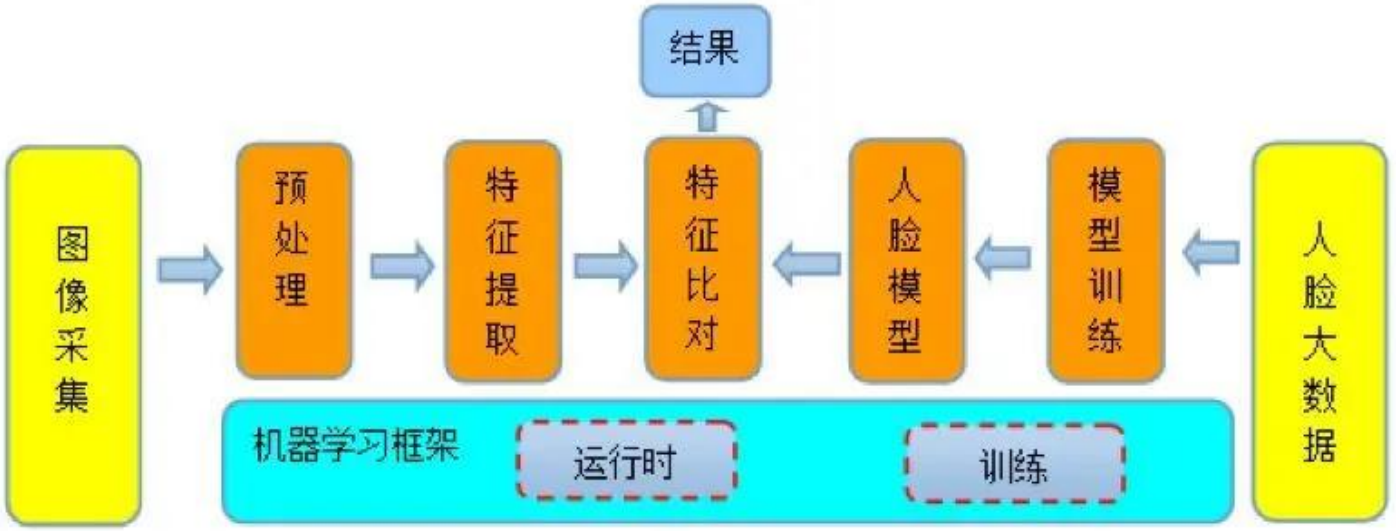
2. 漏洞的原因是GPU从设计理念上就缺少传统CPU的内存隔离和进程隔离机制;

3. 如何利用该漏洞获得LLM上的模型参数，可进一步参考相关论文。

内容概要

- GPU微处理器架构
- GPU微结构安全风险
- AI学习框架安全风险
- 总结

为什么要关注框架安全?



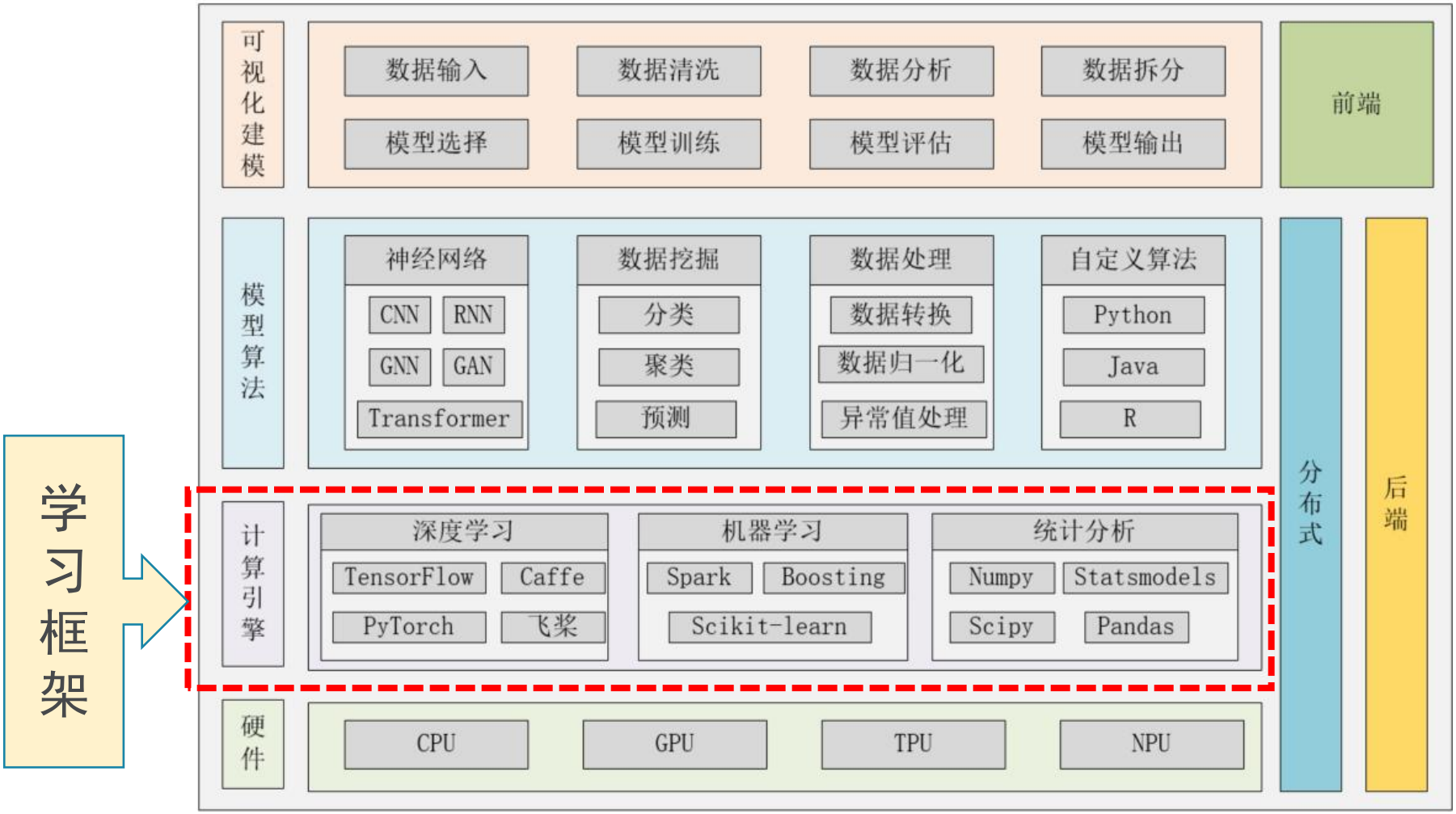
典型人脸识别系统框图

AI算法安全：算法模型是一个AI系统的核心，也是目前AI安全攻防对抗的焦点。目前AI算法安全的主要风险在于对抗样本(adversarial examples)攻击，通过输入恶意样本来欺骗AI算法，目前已发展出诸如生成对抗网络(GAN)这种技术。

AI框架安全：好比于计算机系统中的OS层，可以想象这里如果出现安全问题，影响将会巨大，**而框架的安全性目前并没有得到足够的关注。**

AI业务安全：好比于应用安全，不多赘述。

学习框架在平台中所处的层次



Tensorflow结构组成

前端系统：
提供编程模型，负责构造计算图

后端系统：
提供运行时环境，负责执行计算图

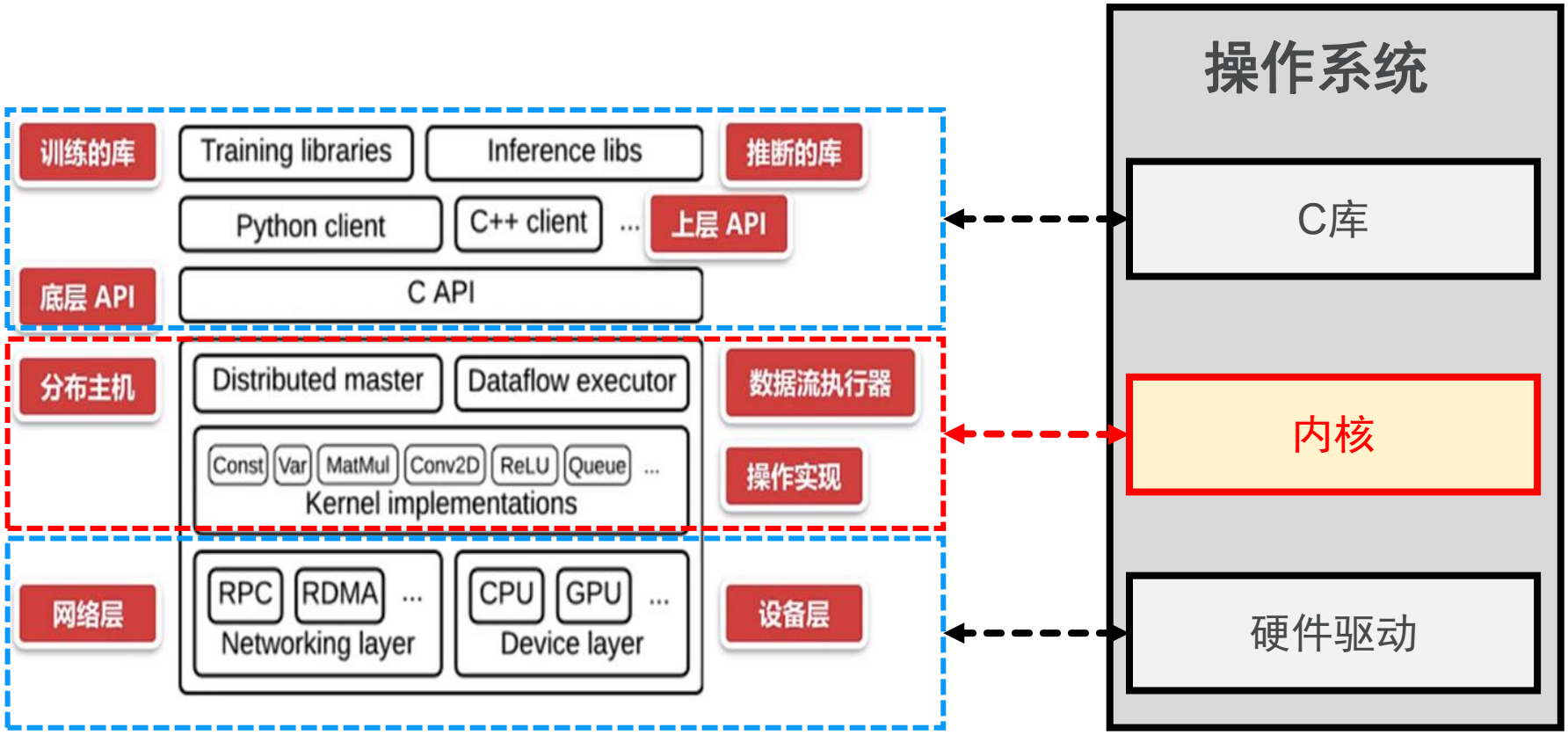
运行时：提供本地模式和分布式模式，共享大部分设计和实现

计算层：由各个 OP 的 Kernel 实现组成；在运行时，Kernel 实现执行 OP 的具体数学运算

通信层：基于gRPC 实现组件间的数据交换，在支持 IB 网络的节点间实现 RDMA 通信

设备层：计算设备是 OP 执行的主要载体，TensorFlow 支持多种异构的计算设备类型

- Tensorflow在机器学习平台中的地位相当于操作系统在计算机中地位。



内容概要

- GPU微处理器架构
- GPU微结构安全风险
- AI学习框架安全风险
  - CUDA溢出漏洞
  - CUDA内存泄漏漏洞
- 总结

CUDA栈溢出漏洞

```
1   typedef unsigned long(*pFdummy)(void);
2  __device__ __noinline__ unsigned long normal1() {
3   printf("Normal\n");
4   return 0;
5  }
6  __device__ __noinline__ unsigned long malicious() {
7   printf("Attack!\n");
8   return 0;
9  }
10 __device__ int overf[100];
11 //============================================================
12  for(int i = 0; i < length; i++) {overf[i] = input[i];}
13  unsigned int buf[16];
14  pFdummy fp[8];
15  fp[0]=normal1; fp[1]=normal2; fp[2]=normal3; fp[3]=normal4;
16  fp[4]=normal5; fp[5]=normal6; fp[6]=normal7; fp[7]=normal8;
17  for(int i = 0; i < length; i++) {buf[i] = overf[i];}
18  fp[5];
```

定义函数指针类型

非内联的设备上运行的正常函数，1~8

非内联的设备上运行的恶意函数

全局变量，存在Global memory中

恶意控制的变量Length和input，修改全局变量

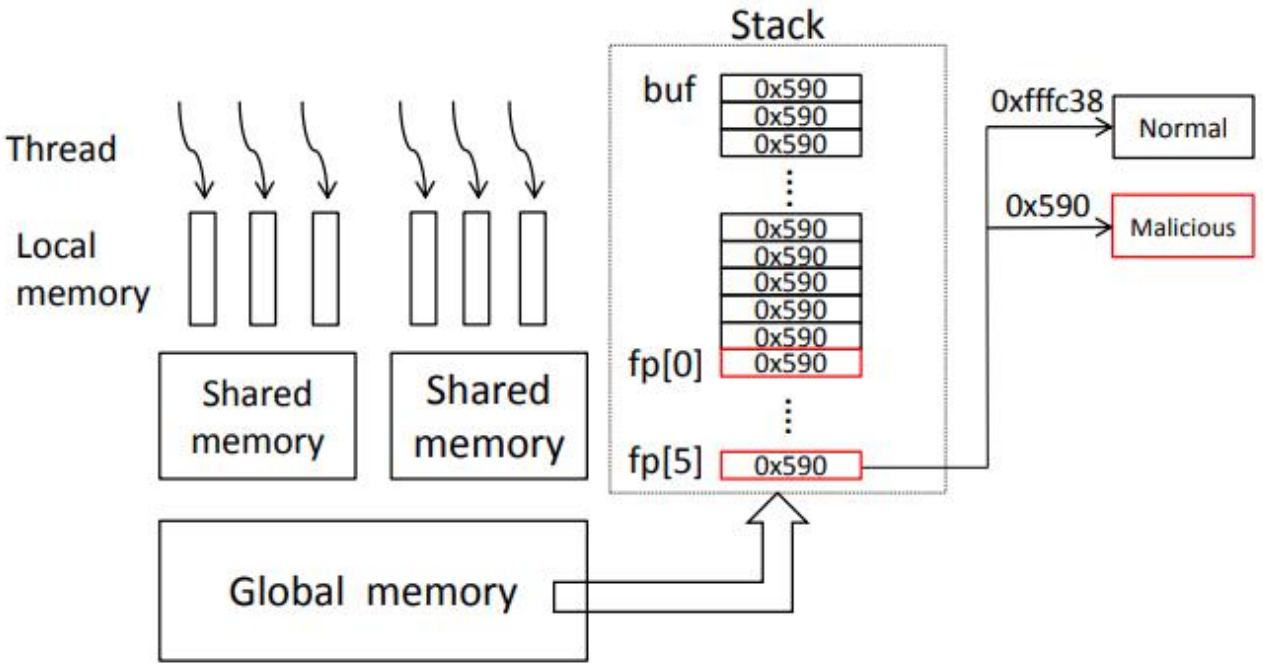
对buf数组进行溢出

fp[5]跳转到恶意函数

正常函数申请栈中连续变量

CUDA栈溢出漏洞

CUDA栈溢出示意：



思考：如果overf声明为共享变量，情况会怎样？如果是本地变量呢？

CUDA堆溢出漏洞

```
1  // ============== a virtual table of the device code ================
2  class Vtable {
3  public:
4    __device__ virtual unsigned long v1() {printf("Normal\n");return 0;}
5    __device__ virtual unsigned long v2() {printf("Normal\n");return 0;}
6    __device__ virtual unsigned long v3() {printf("Normal\n");return 0;}
7    __device__ virtual unsigned long v4() {printf("Normal\n");return 0;}
8  };
9  //====================== malicious function ========================
10 __device__ __noinline__ unsigned long malicious() {
11   printf("Attack!\n");
12   return 0;
13 }
14 //============= a snippet code of memory isolation of heap ===========
15   __shared__ unsigned long *buf;
16   if(threadIdx.x == 0)
17   buf = (unsigned long *) malloc(sizeof(unsigned long) * 8);
18   Vtable *fp = new Vtable;
19   if(threadIdx.x == 0)
20     for(int i = 0; i < length; i++) {buf[i] = input[i];}
21   if(threadIdx.x == 1)
22     printf("%lx", buf[0]);
23 //============ a snippet code of exploiting of 'global' heap =========
24   unsigned long *buf;
25   buf = (unsigned long *) malloc(sizeof(unsigned long) * 8);
26   Vtable *fp = new Vtable;
27   printf("malicious %p\n", malicious);
28   for(int i = 0; i < length; i++) {buf[i] = input[i];}
29   res=fp->v1(); res=fp->v2(); res=fp->v3(); res=fp->v4();
```

定义“虚函数表”类，4个虚函数组成

非内联的设备上运行的恶意函数

共享变量，存在shared memory中

线程0从堆中申请一段空间

如果线程0对堆中内容进行了修改

线程1可以访问被修改的内容

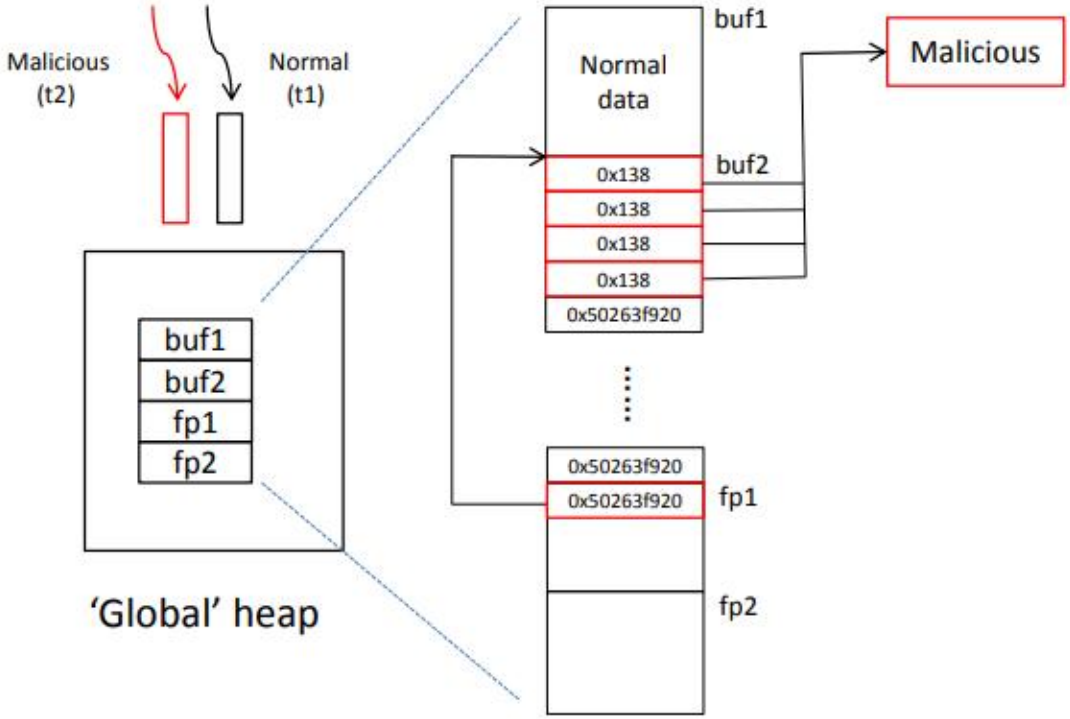
线程私有变量，存在local memory中

线程各自独立从堆中申请buf和Vtable空间

如果线程2有恶意，将其buf边界溢出直至覆盖其后的线程1申请的Vtable值

CUDA栈溢出漏洞

CUDA堆溢出示意：



CUDA的溢出风险小结

1. 线程的栈空间属于线程的局部空间，但缺少保护机制，存在栈溢出风险。（栈溢出例子）

2. 堆空间属于全局空间，线程块所属的线程共享堆空间，A线程申请的空间，只要没被释放，B线程仍可访问。（堆溢出例子中的中间代码片段）

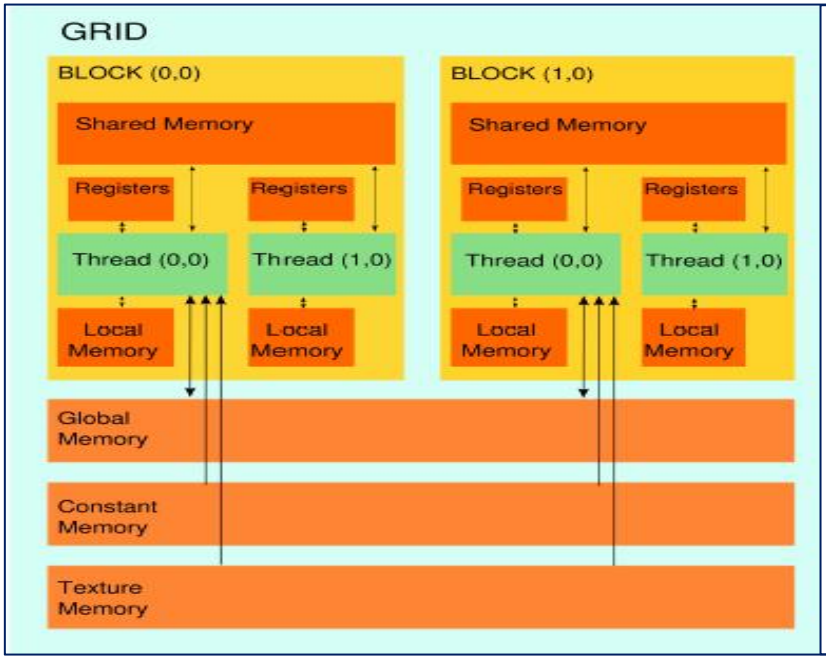
3. 堆缺少保护机制，如隔离，A线程申请的堆空间，B线程即使不知道地址入口，也能通过自身堆空间进行溢出，从而访问A线程申请的堆空间。

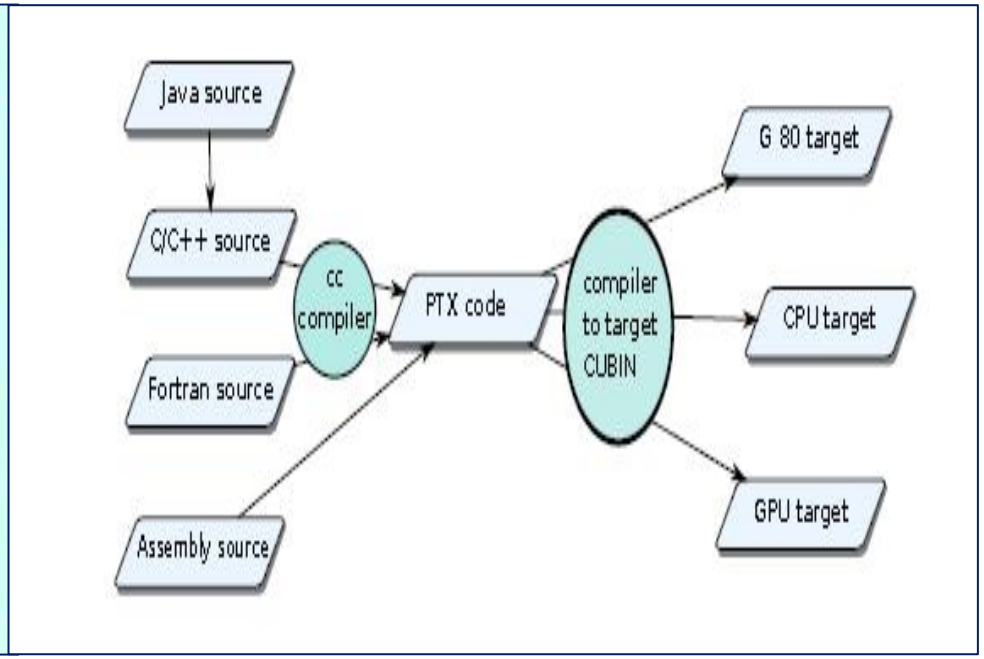
内容概要

- GPU微处理器架构
- GPU微结构安全风险
- AI学习框架安全风险
  - CUDA溢出漏洞
  - CUDA内存泄漏漏洞
- 总结

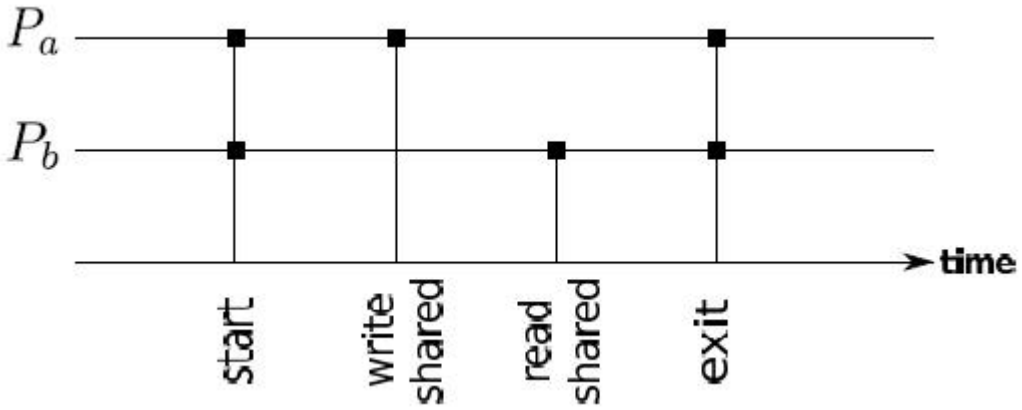
CUDA内存泄漏漏洞

- “CUDA Leaks:Information Leakage in GPU Architectures” （ACM Transactions on Embedded Computing Systems Vol.15 Issue 1）
  - **主要方法：利用GPU存储隔离缺陷，进行信息泄露**
  - **CUDA架构+访存调度获得：**
    - 基于shared memory的信息泄露利用
    - 基于global memory的信息泄露利用
    - 基于寄存器的信息泄露利用





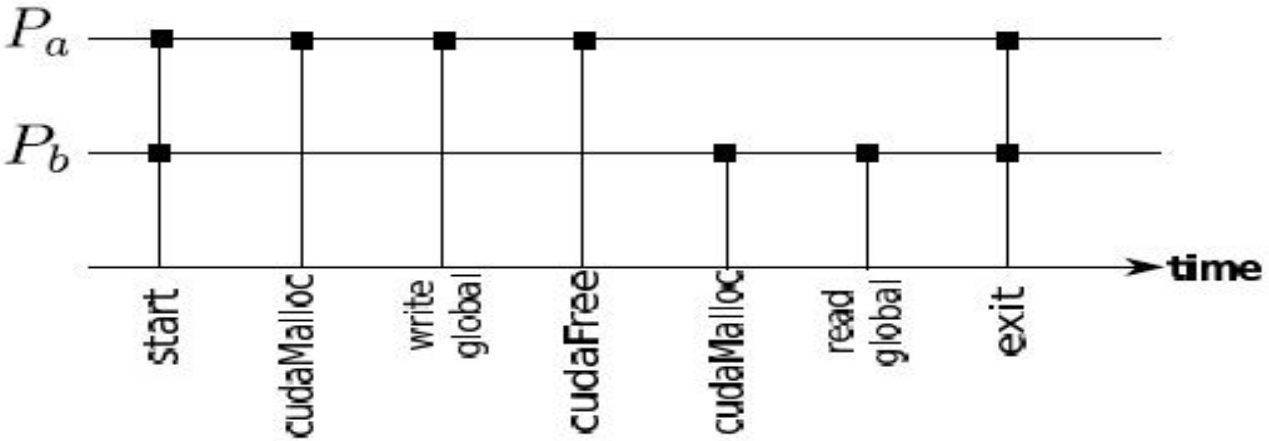
基于shared memory的信息泄露利用



原因：由于shared memory清除不及时,进程切换时，上一个进程信息泄露利用流程。

1. 创建两个独立的进程Pa,Pb。Pa为受害者程序，Pb为攻击者程序。创建等同于SM数量的block，每个block线程数设置为warp-size大小。
2. Pa执行K次从global memory到shared memory写操作的核函数。
3. Pb执行K次从shared memory中读取数据核函数。由于cuda调度机制未知，通过在Pa执行的核函数的PTX代码中嵌入asm(“mov.u32 %0, \%smid;” : “=r”(ret))来读取当前block运行所在的SM的id，通过确定Smid来确定写入数据的顺序，从而确定读取数据的顺序。
4. 由于cuda runtime在线程执行exit()前会对shared memory进行清零操作，因此在读取受害程序Pa的信息时，Pa必须还在运行中。

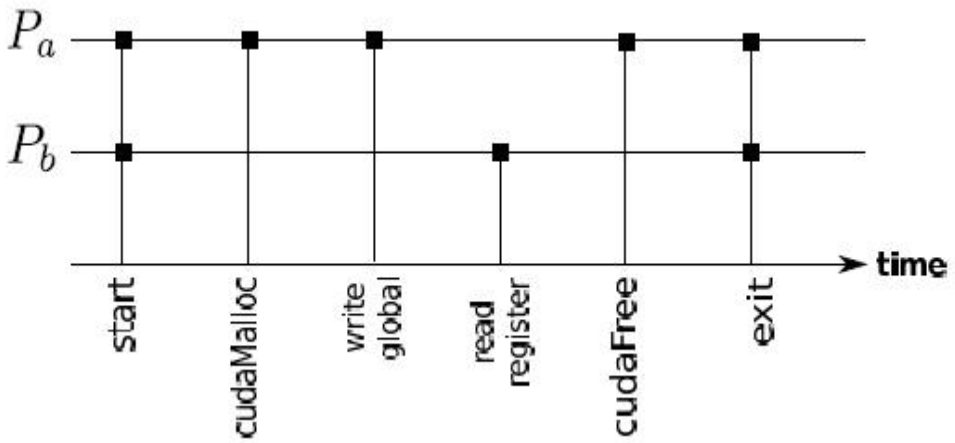
基于global memory的信息泄露利用



原因：由于global memoy清除不及时,context切换时，上一个context信息泄露利用流程。

1. 创建两个独立的进程Pa,Pb。Pa为受害者程序，Pb为攻击者程序
2. Pa使用cudaMalloc()函数在global memory上创建V1,V2,V3,V4四个向量，从V1，V2拷贝数据到V3，V4,然后退出。
3.Pb同样创建相同的四个向量，并将V3，V4的内容读取到主机内存，发现读取结果和Pa写入的内容一样。
4.该流程要求攻击者Pb程序分配了和受害者Pa一样大小的global memory才能获取受害者完整的数据信息。

基于寄存器的信息泄露利用




```
__device__
void get_reg32bit(uint32_t *regs32) {

    # declaration of 8300 registers
    asm(".reg .u32 r<8300>;\n\t");
    # move the content of register r0 into
    # the position 0 of regs32[]
    asm("mov.u32 %0, r0;" : "=r"(regs32[0]));
    asm("mov.u32 %0, r1;" : "=r"(regs32[1]));
    ...
    asm("mov.u32 %0, r8191;" : "=r"(regs32[8191]));

}
```

原因：在PTX转CUBIN过程中，当分配寄存器数量超过物理寄存器数量时，编译器会将global memory作为替代分配给执行程序。

利用流程

1.创建两个独立的进程Pa,Pb。Pa为受害者程序，Pb为攻击者程序

2. Pa对global memory进行多次写操作

3. Pb通过分配不同数量的寄存器试图绕过内存隔离机制读取Pa写到global memory的内容。由于GPU driver和cuda Runtime中实现的动态内存机制，Pb申请A数量的寄存器执行B次，能从global memory中获取到A*B数量的信息。

4. 该流程获取的信息多少取决于申请的寄存器数量和迭代的次数，且可以绕过cudaMalloc()机制，不受任何限制。

此种信息泄露的安全方案讨论

如何设计更好的内存管理机制以及及时的数据清除措施？

1. Shared memory信息泄露的利用窗口在核函数执行完成到主机程序退出中，在核函数中执行内存清零操作可以解决shared memory信息泄露问题。

2.在CUDA Runtime中设计在接受请求进程访问前进行global memory内存清除操作，该操作会带来部分的性能开销。

3. 在Nvidia driver中设计在使用global memory代替寄存器之前需要对相应的global memory进行清零的操作，也不能分配被占用的global memory。

内容概要

- GPU微处理器架构
- GPU微结构安全风险
- AI学习框架安全风险
- 总结

内容概要

- 介绍了GPU微结构及其安全风险
- 介绍了学习框架安全风险
- 结论：机器学习平台底层的结构风险大，且为引起学术和产业界足够重视


Q&A