GPU与CUDA简介

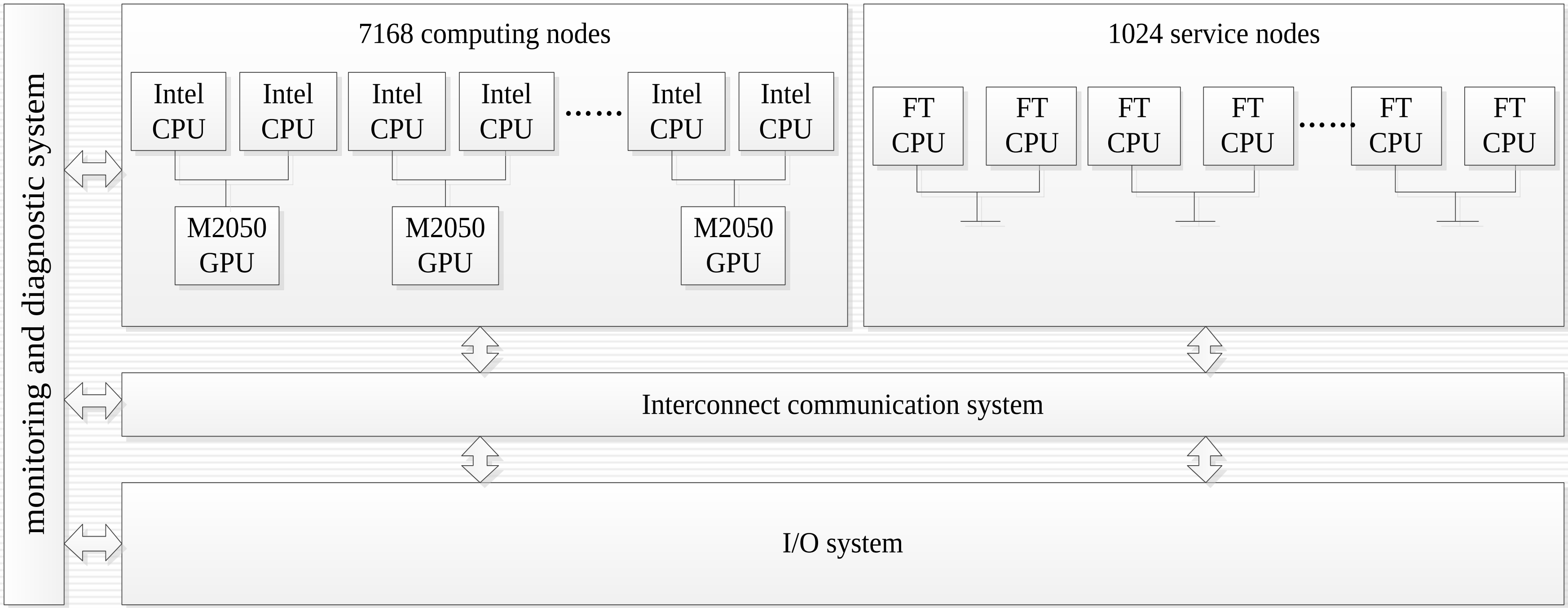
赵洋 工程师

zhaoyang@nscc-tj.gov.cn

国 家 超 级 计 算 天 津 中 心

内容提要

- "天河一号"架构
- GPU简介
- CUDA硬件模型和编程模型
- CUDA Fortran & CUDA C
- CUDA程序优化策略


monitoring and diagnostic system
7168 computing nodes
Intel CPU
Intel CPU
Intel CPU
Intel CPU
……
Intel CPU
Intel CPU
M2050 GPU
M2050 GPU
M2050 GPU
1024 service nodes
FT CPU
FT CPU
FT CPU
FT CPU
……
FT CPU
FT CPU
Interconnect communication system
I/O system

TH-1A 系统架构

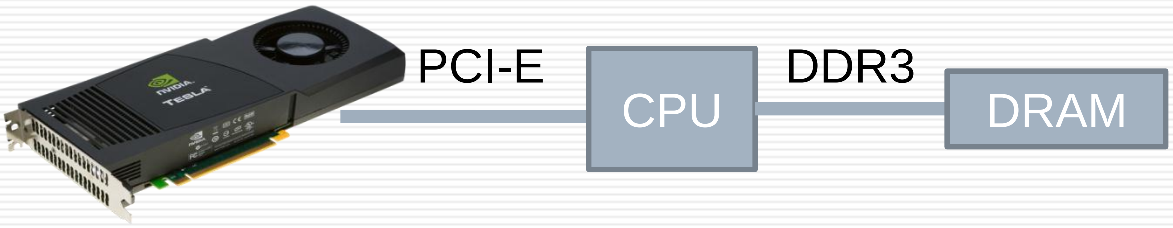
天河一号上GPU编译环境

CUDA编译环境：

◆CUDA安装主路径：/vol-th/software/cuda

/vol6/software/cuda

GPU简介



天河

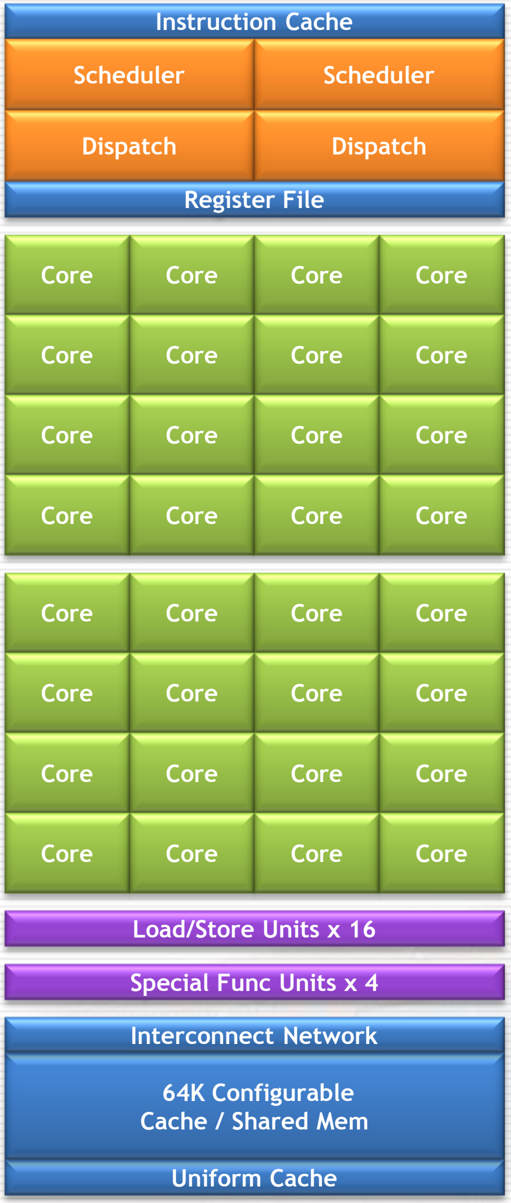
GPU简介

Specification	Tesla M2050
Processor	1 x Tesla T20
CUDA cores	448
Core clock	1.15 GHz
on-board memory	3 GB
Memory bandwidth	148 GB/s peak
Single/double precision floating performance	1030/515 GFlops
System I/O	PCI-E x 16 Gen2
Board power	<= 225W

CUDA硬件模型

Fermi架构下的SM:

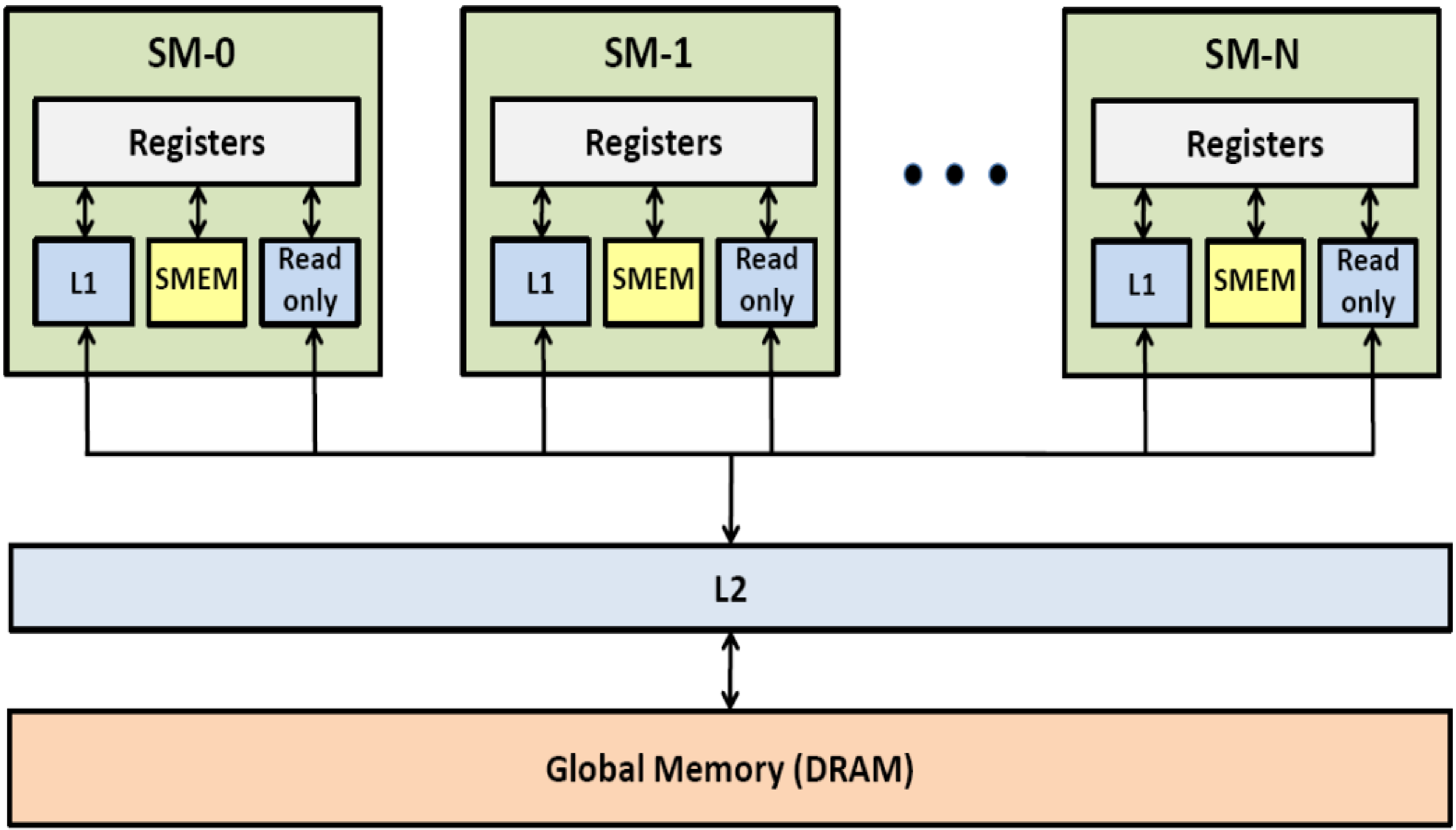
- 控制单元
  - 两个指令发射和调度器
- 运算单元
  - fp32, fp64, int32, SFU, LFU
- 存储单元
  - 48K 可配置的shared memory/L1
  - 32K 32-bit 寄存器
  - texture/constant cache



Fermi存储模型

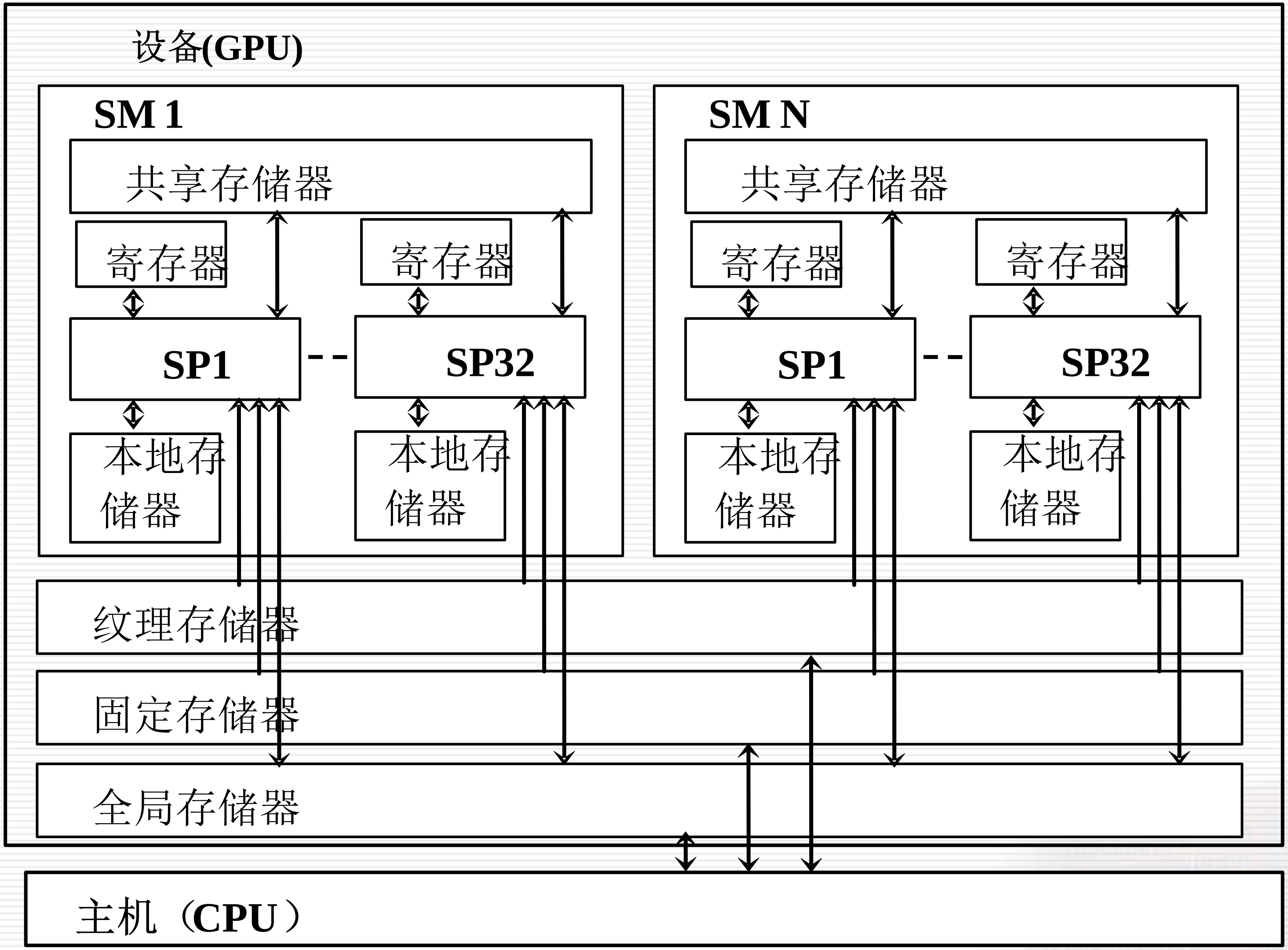
- 三个层次，与CPU存储模型类似
- DRAM/global memory
- Caches
  - First level (On SM)
    - L1
    - Shared memory
    - Texture cache
    - Constant cache
  - L2
- Register

CUDA硬件模型

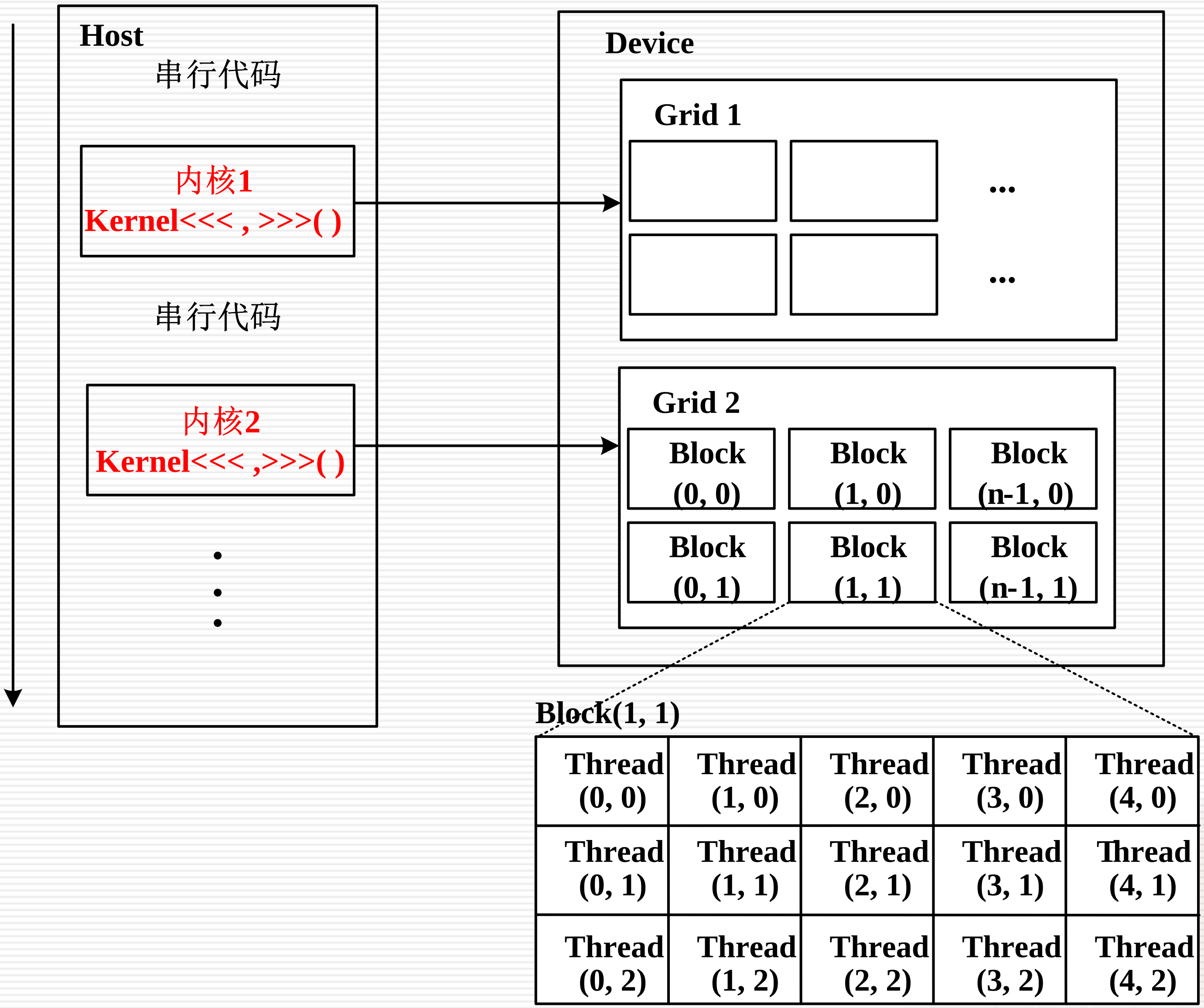


CUDA存储模型

CUDA硬件模型


设备(GPU)
SM 1
共享存储器
寄存器
寄存器
SP1
SP32
本地存储器
本地存储器
SM N
共享存储器
寄存器
寄存器
SP1
SP32
本地存储器
本地存储器
纹理存储器
固定存储器
全局存储器
主机（CPU）

CUDA编程模型


Host
串行代码
内核1
Kernel<<< , >>>( )
串行代码
内核2
Kernel<<< ,>>>( )
Device
Grid 1
...
...
Grid 2
Block (0, 0)
Block (1, 0)
Block (n-1, 0)
Block (0, 1)
Block (1, 1)
Block (n-1, 1)
Block(1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

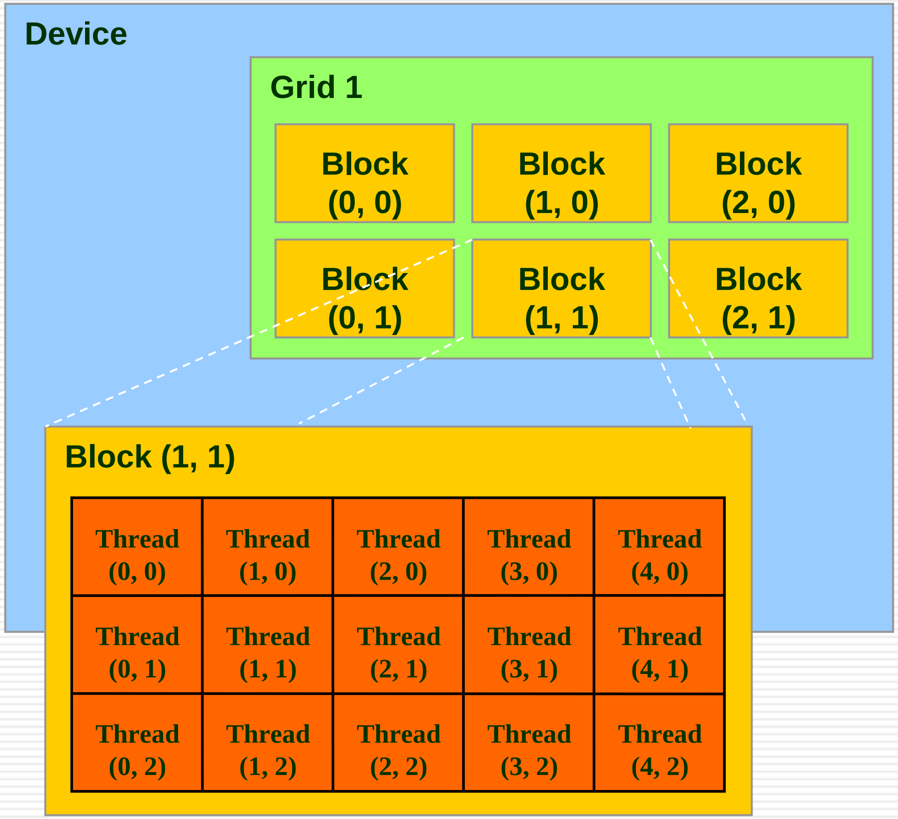
CUDA编程模型

两个层次的并行：

- **grid内多个block的并行**
- **block内多个thread的并行**

block内：

- 可以进行线程同步：__syncthreads()
- thread间通过shared memory进行通信



指定线程数目：

```
dim3 grid(1024,1,1), block(256,1,1);
kernel<<<grid, block>>>(…);
```

如何计算线程编号：

通过内置变量blockIdx, blockDim, threadIdx

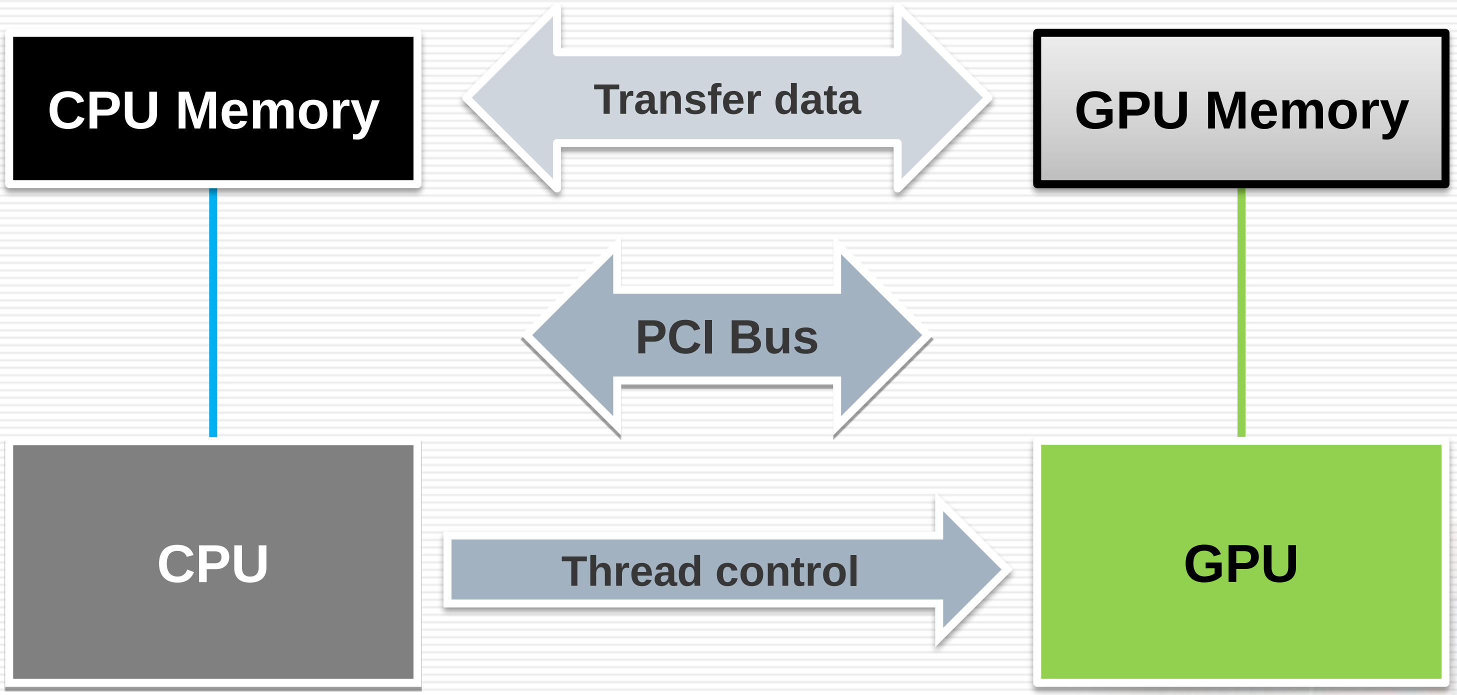
dim3 grid(4,1,1), block(4,1,1);

threadIdx.x:	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
blockIdx.x:	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
idx:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

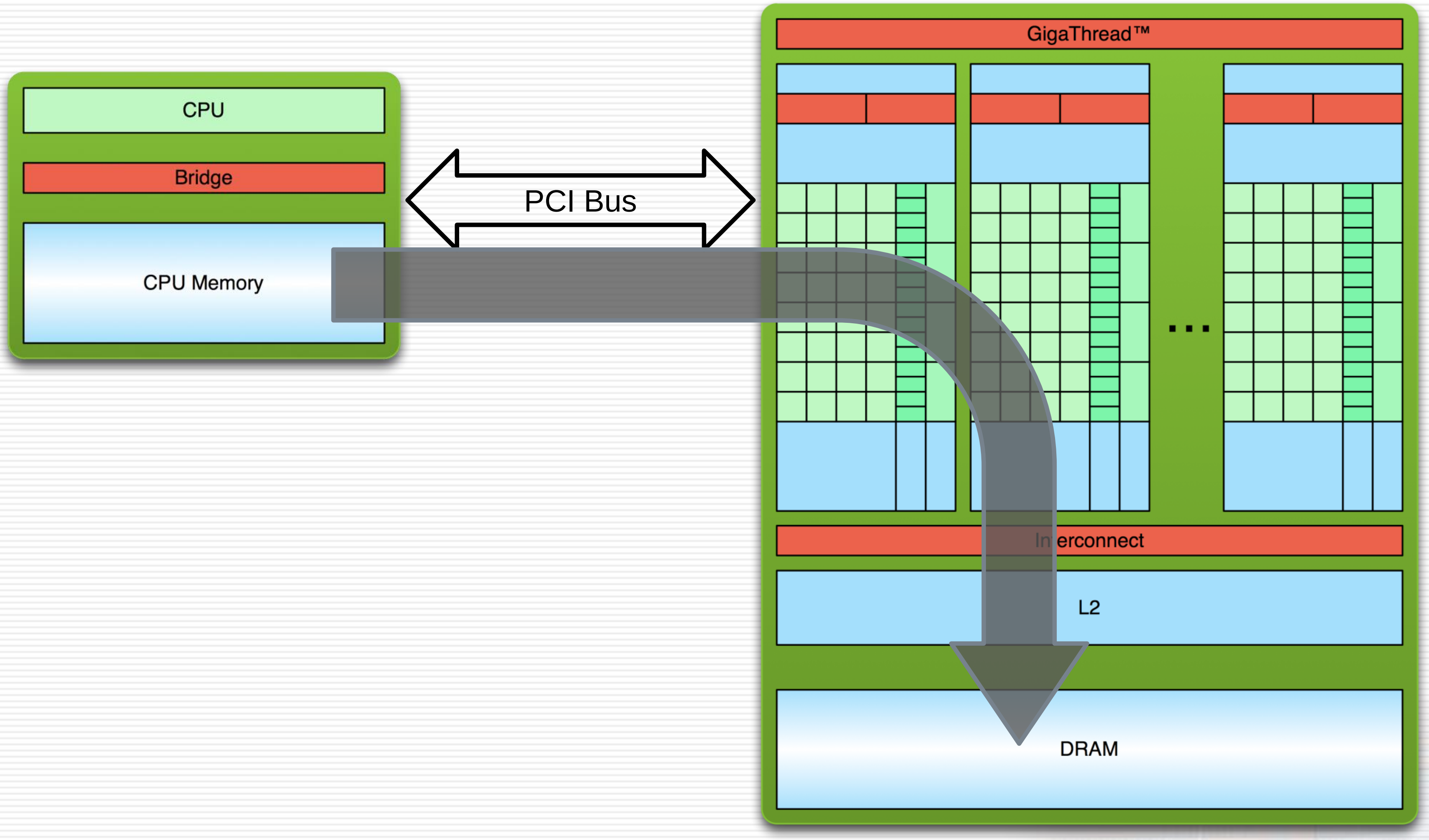
idx = blockIdx.x*blockDim.x + threadIdx.x;

使用CUDA来开发程序要做的两件事:

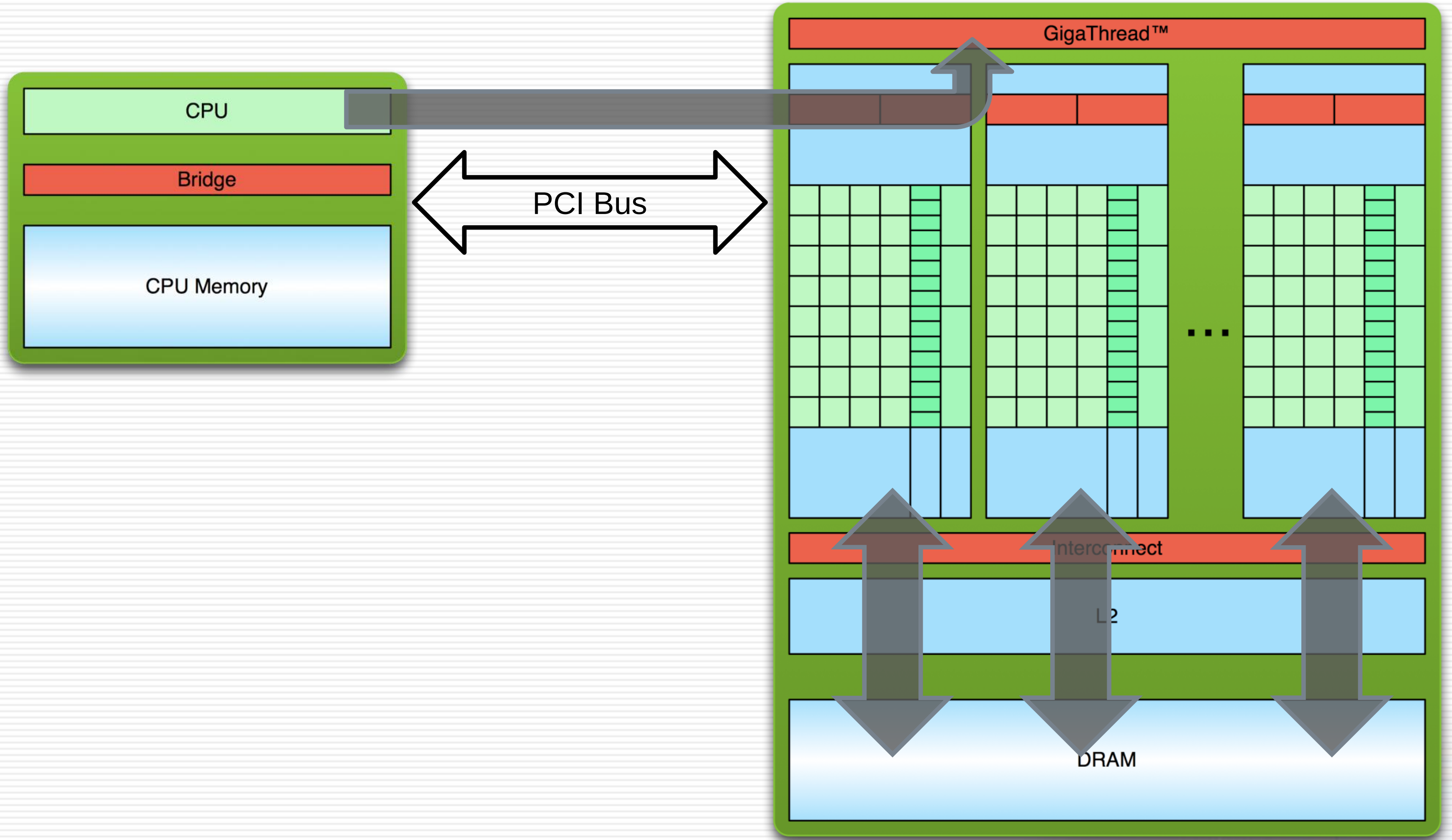
- **CPU-GPU间的数据传输**
- **GPU上的并行计算**



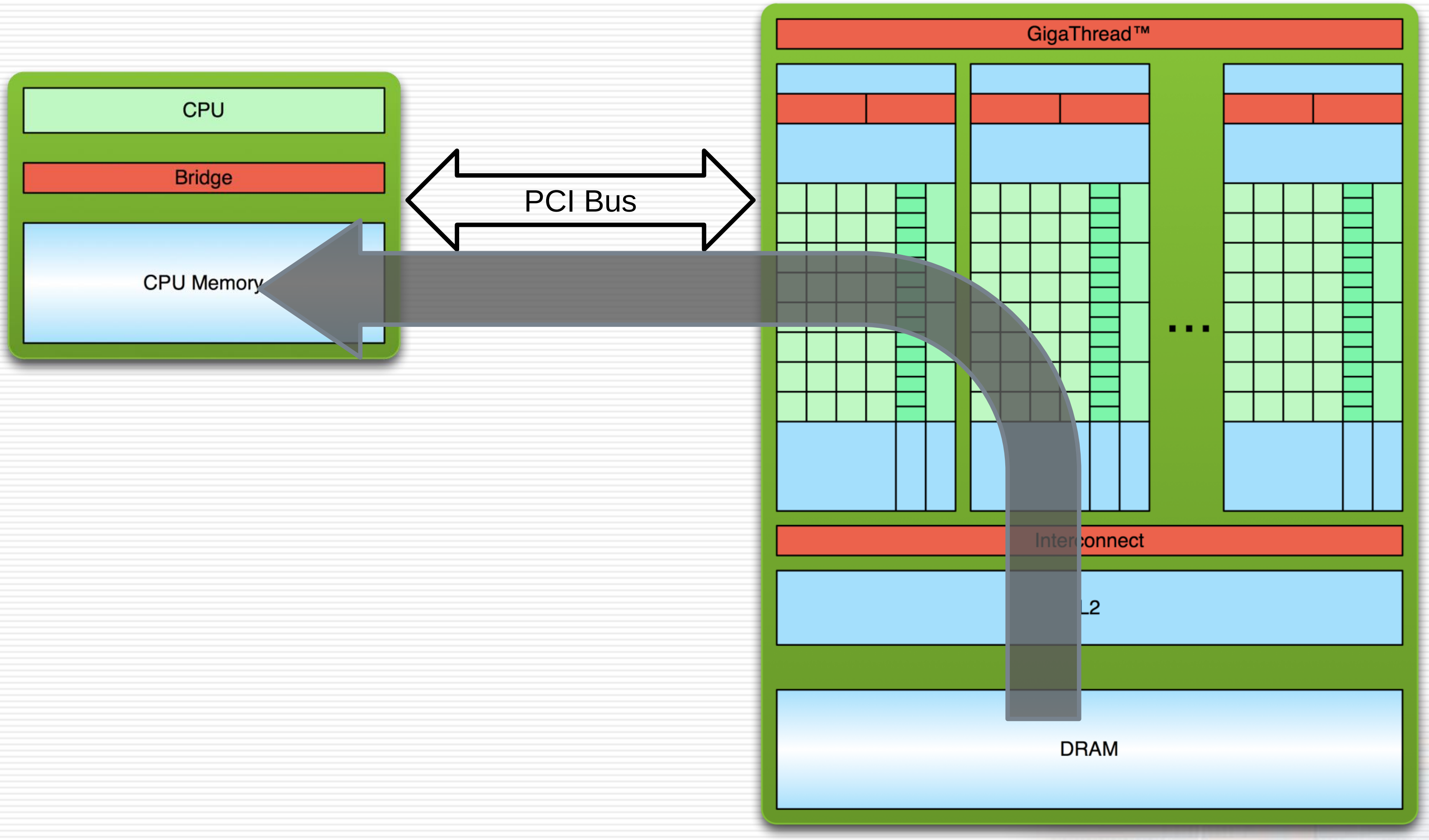
CUDA编程模型


CPU
Bridge
CPU Memory
PCI Bus
GigaThread™
...
Interconnect
L2
DRAM

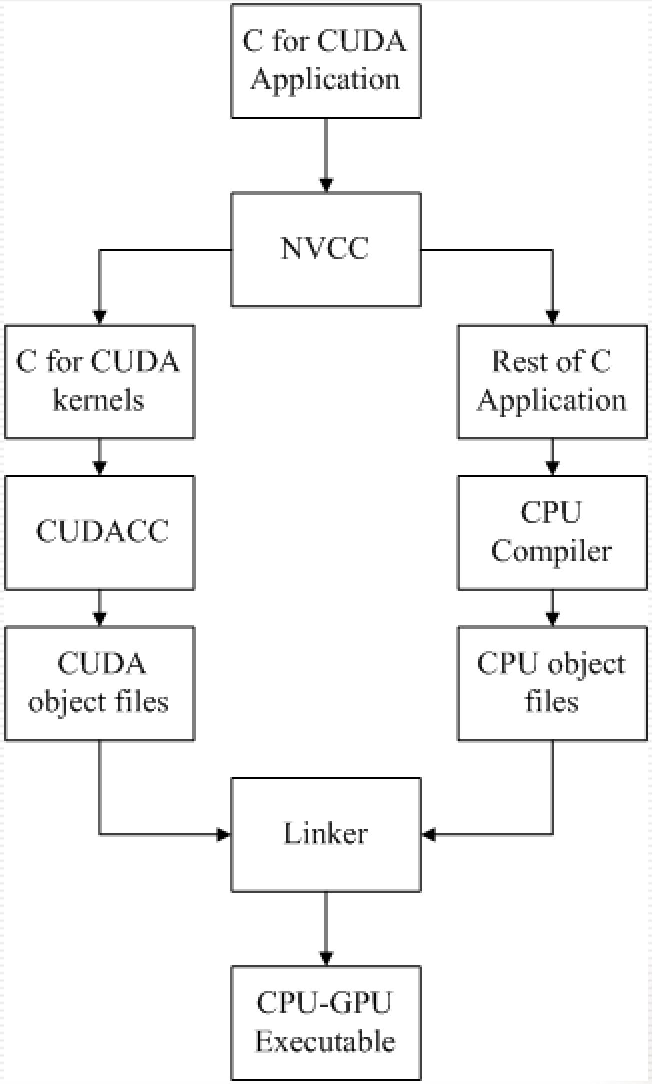
CUDA编程模型


CPU
Bridge
CPU Memory
PCI Bus
GigaThread™
...
Interconnect
L2
DRAM

CUDA编程模型


CPU
Bridge
CPU Memory
PCI Bus
GigaThread™
Interconnect
L2
DRAM

nvcc编译流程


C for CUDA Application
NVCC
C for CUDA kernels
Rest of C Application
CUDACC
CPU Compiler
CUDA object files
CPU object files
Linker
CPU-GPU Executable

CUDA Fortran

方法一：“cuf”指导语句

```
subroutine scale(a,n)
  integer :: i
  integer, value :: n
  real :: a(:)
!$cuf kernel do (1) <<<*,*>>>
  do i=1,n
      a(i) = a(i) * 2
  enddo
end subroutine
```

```
program scaleTest
  use cudafor
  …
  real, allocatable :: b(:)
  real, allocatable, device :: d_b(:)
  allocate(b(n))
  allocate(d_b(n))
  …
  d_b = b(1:n)
  call scale(d_b, n)
  b = d_b(1:n)
…
end program scaleTest
```

```
$ pgfortran –Mcuda scale.cuf
$./a.out
```

方法二：把subroutine改为kernel

```
subroutine scale(a,n)
  integer :: i
  integer, value :: n
  real :: a(:)
  do i=1,n
      a(i) = a(i) * 2
  enddo
end subroutine

real, allocatable :: b(:)
allocate(b(n))
…
call scale(b, n)
```

```
attributes(global) subroutine scale(a,n)
  integer :: i
  integer, value :: n
  real :: a(:)
  i = threadIdx%x + (blockIdx%x-1)*blockDim%x
  a(i)=a(i)*2
end subroutine

real, allocatable :: b(:)
real, allocatable, device :: d_b(:)
allocate(b(n))
allocate(d_b(n))
…
d_b = b(1:n)
call scale<<<n/128, 128>>>(d_b, n)
b = d_b(1:n)
```

CUDA C

把function改为kernel

```
void scale(a,n)
{
  int n, i;
  float *a;
  for (i=0; i<n; i++)
      a[i] = a[i] * 2;
}

float *b;
b = (float*) malloc (n*sizeof(float));
…
scale(b, n);
```

```
__global__ void scale(a,n)
{
  int n, i;
  float *a;
  i = threadIdx.x + blockIdx.x*blockDim.x;
  a[i]=a[i]*2;
}

float *b, *d_b;
b = (float*) malloc (n*sizeof(float));
cudaMalloc((void **) &d_b, n*sizeof(float));
…
cudaMemcpy(d_b, b, n*sizeof(float),
                    cudaMemcpyHostToDevice);
scale<<<n/128, 128>>>(d_b, n);
cudaMemcpy(b, d_b, n*sizeof(float),
                    cudaMemcpyDeviceToHost);
```

```
$ nvcc scale.cu
$./a.out
```

CUDA程序优化策略

- 存储器访问优化
  - 主机-设备端通信优化
    - 降低传输次数
    - 使用页锁定数据传输
    - 异步数据传输
  - 设备端存储器访问优化
    - 全局内存的合并访问优化
    - 共享存储器访问优化
    - 纹理存储器访问优化
- 指令流优化
- 执行配置优化

Memory bandwidth bound

Instruction throughput bound

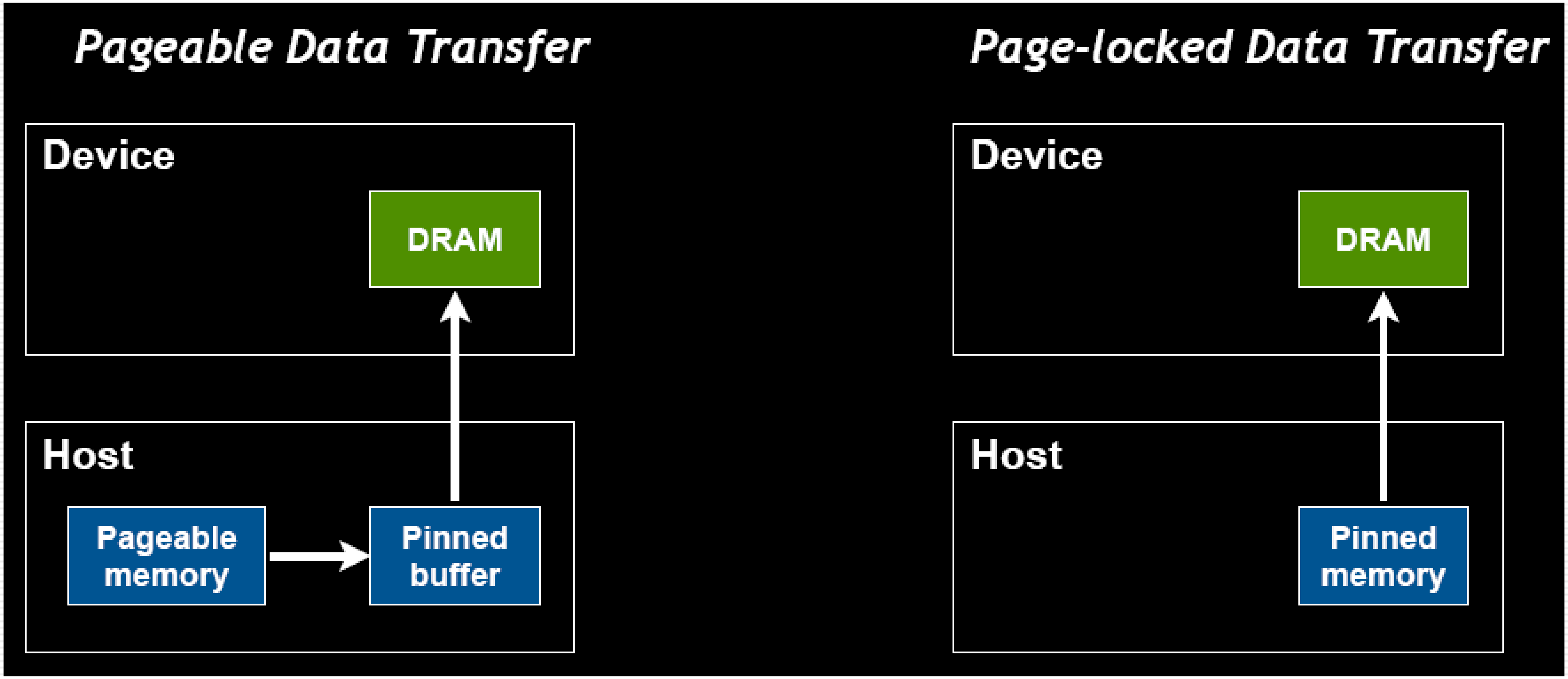
Latency bound

天河

页锁定数据传输

- 主机-设备端通信优化
  - 主机到设备端的带宽远远低于设备端内部带宽
    - 8 GB/s peak (PCIe × 16 Gen 2) vs. 148 GB/s peak (Tesla M2050)
    - 1.54GHz*(384/8)Byte*2=148GB/s
- 降低主机-设备端通信次数，分次传输合为一次
  - PCI-E延迟10us
- 合理使用page-locked(or pinned) memory
  - 正常malloc可能会分配到低速的虚拟内存中，页锁定内存能够保证在物理内存，始终不会分配到虚拟内存。

页锁定数据传输



页锁定数据传输示意图

页锁定数据传输

```
real, allocatable, pinned :: b(:)
real, allocatable, device :: d_b(:)
allocate(b(n), STAT=istat, PINNED=pinnedFlag)
allocate(d_b(n))
…
d_b = b(1:n)
call scale<<<n/128, 128>>>(d_b, n)
b = d_b(1:n)
```

Tesla M2050

- Pageable:~3.5 GB/s
- Pinned: ~6 GB/s

异步数据传输

流：一系列必须串行执行的内核函数和数据传输指令集合。流之间的执行顺序是并发的。

- **kernel的启动是异步的，常规数据拷贝是阻塞式的**

```
d_b = b(1:n)
call scale<<<n/128, 128>>>(d_b, n)
b = d_b(1:n)
```

```
cudaMemcpy(d_b, b, n*sizeof(float), cudaMemcpyHostToDevice);
scale<<<n/128, 128>>>(d_b, n);
cudaMemcpy(b, d_b, n*sizeof(float), cudaMemcpyDeviceToHost);
```

- **异步数据传输函数是非阻塞式，可直接与CPU串行计算交叠；通过多流技术可使数据传输与GPU的kernel执行交叠**

异步数据传输

要求：主机端必须为页锁定存储

- 数据传输与CPU计算交叠

```
istat = cudaMemcpyAsync(d_b, b, n, 0)
call scale<<<n/128, 128>>>(d_b, n)
call cpuFunction(c)
```

交叠

如果主机端要使用内核函数计算出的结果，应加入同步操作。cudaThreadSyhchronize();

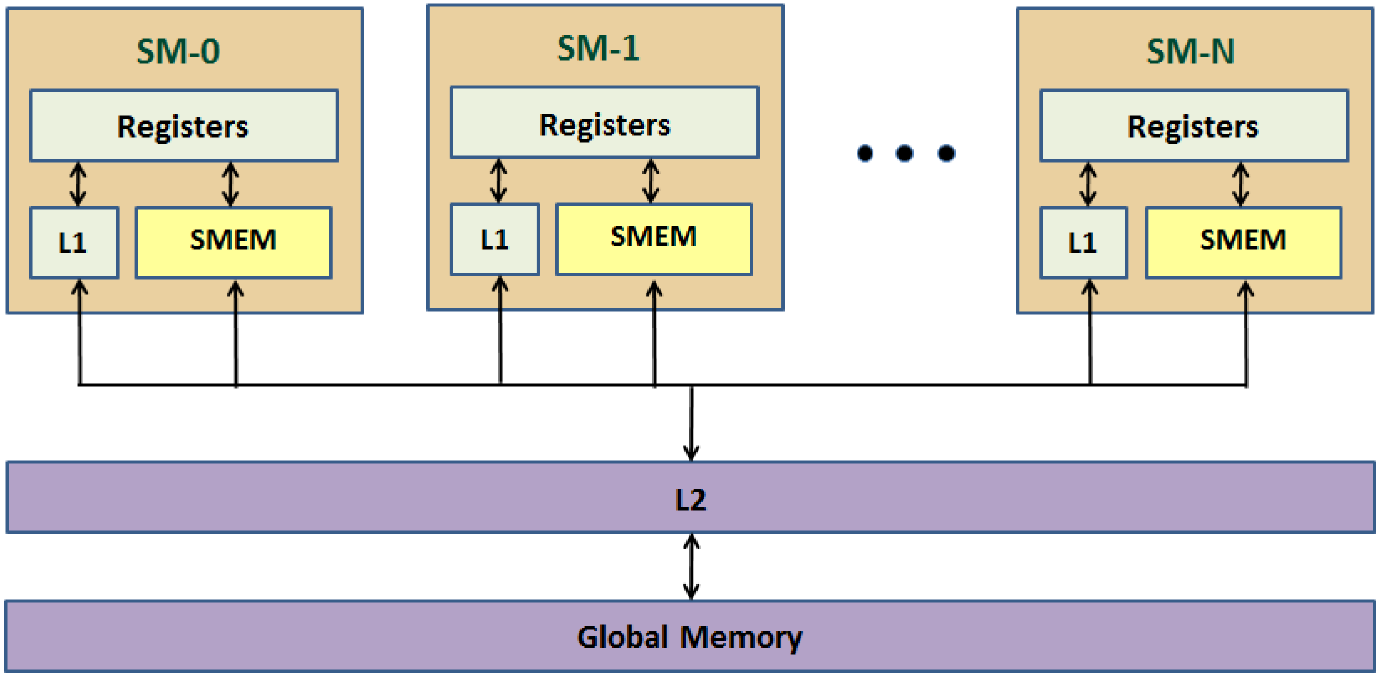
- 数据传输与kernel交叠

```
integer (kind=cuda_stream_kind) :: stream1, stream2
istat = cudaStreamCreate(stream1)
istat = cudaStreamCreate(stream2)
istat = cudaMemcpyAsync(d_b, b, n, stream1)
call scale<<<n/128, 128, 0, stream2>>>(d_c, n)
call cpuFunction(c)
```

交叠

设备端存储器访问优化

- 全局内存的合并访问优化
- 共享存储器访问优化
- 纹理存储器访问优化



全局内存的合并访问优化

L1缓存：可被配置为16KB或48KB，缓存线大小为128B

L2缓存：768KB，缓存线大小可动态调整：32B、64B、96B、128B.
所有对全局内存的访问都经过L2缓存

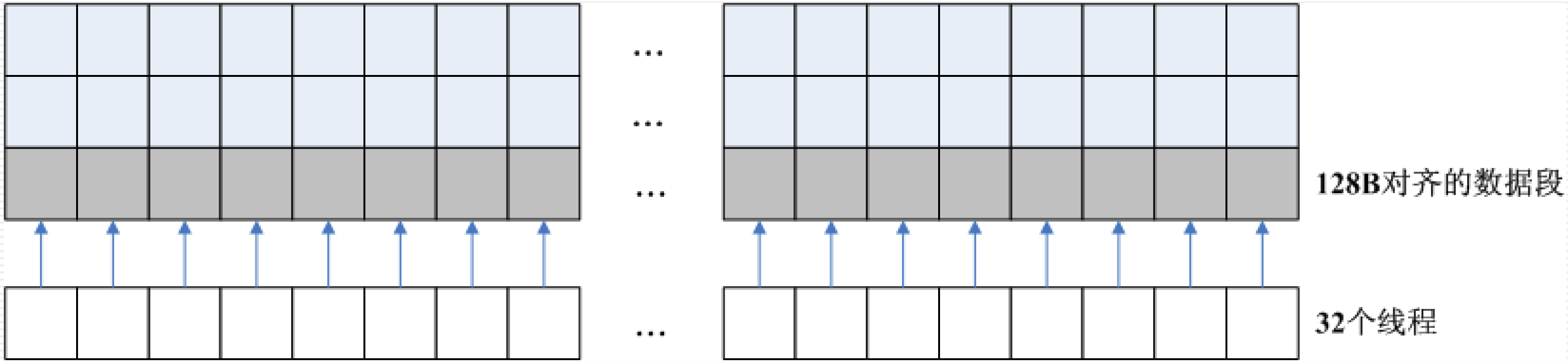
Fermi架构下，对全局内存是以128B的L1缓存线为基本单位进行访问

优化准则：每个warp内的32个线程访问的数据尽可能涵盖少的L1缓存线

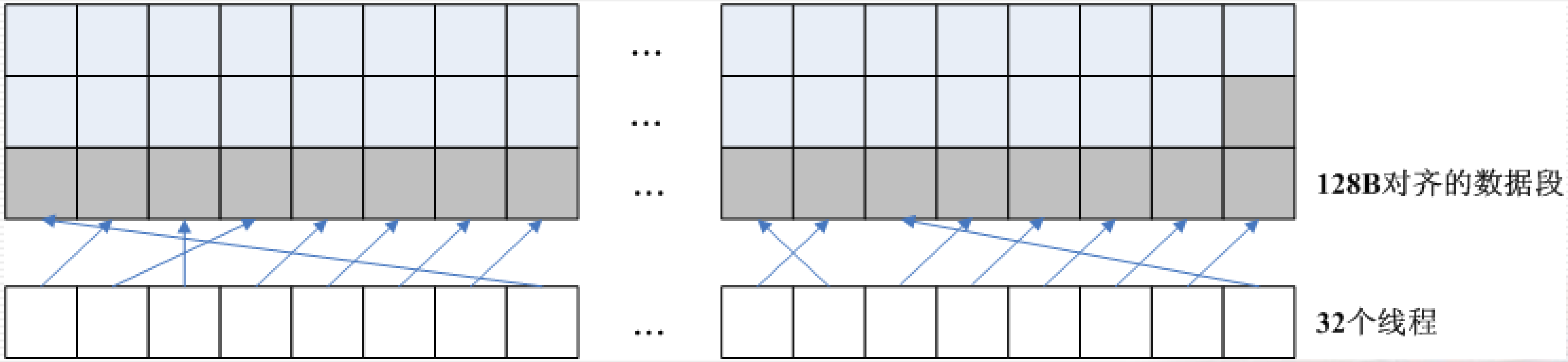
- 访问数据段对齐
- 访问间隔

```
__global__ void offsetCopy(float* src, float* dst, int len, int offset)
{
    int xid = offset + threadIdx.x + blockIdx.x*blockDim.x;
    int index = xid%len;
    dst[index] = src[index];
}
```

全局内存的合并访问优化

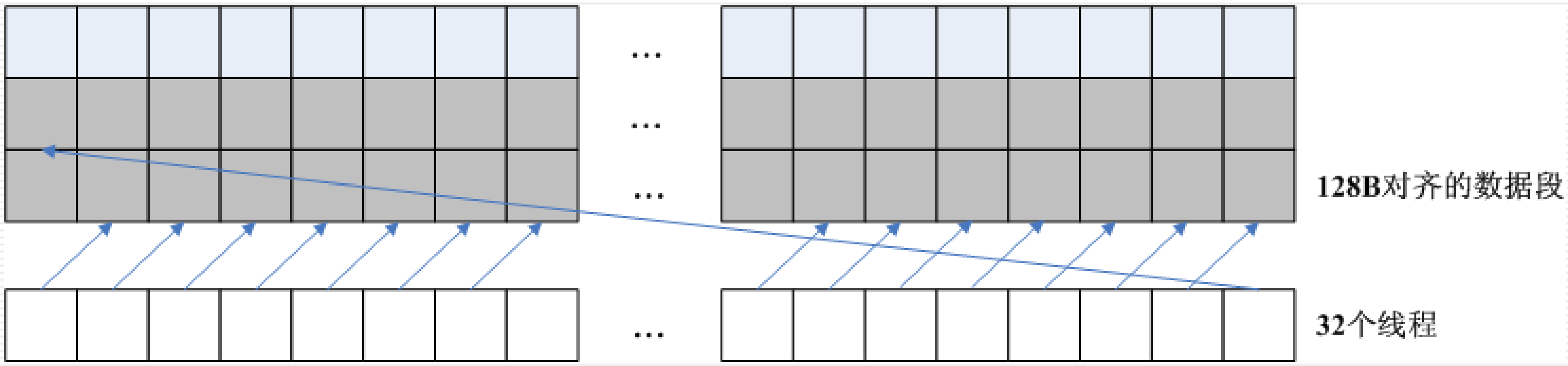


对齐访问1



对齐访问2

全局内存的合并访问优化



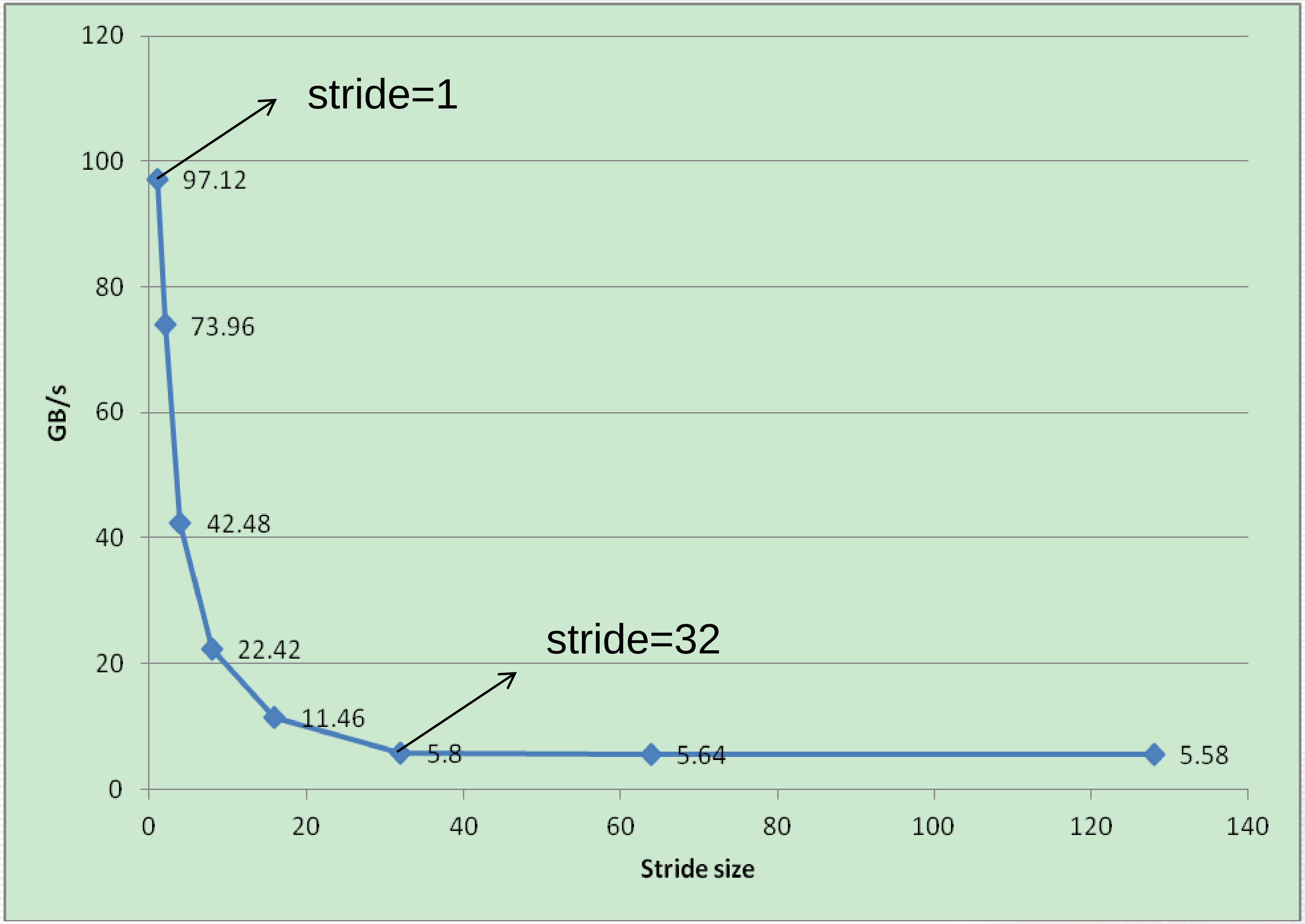
非对齐访问(offset=1)

全局内存的合并访问优化

访问间隔：

```
__global__ void kernel1D_stride(float* src, float* dst, int len, int stride)
{
    int dstx = threadIdx.x + blockIdx.x*blockDim.x;
    int srcx = (dstx*STRIDE)%len;
    if(dstx < len)
        dst[dstx] = src[srcx];
}
```

全局内存的合并访问优化



全局内存的合并访问优化

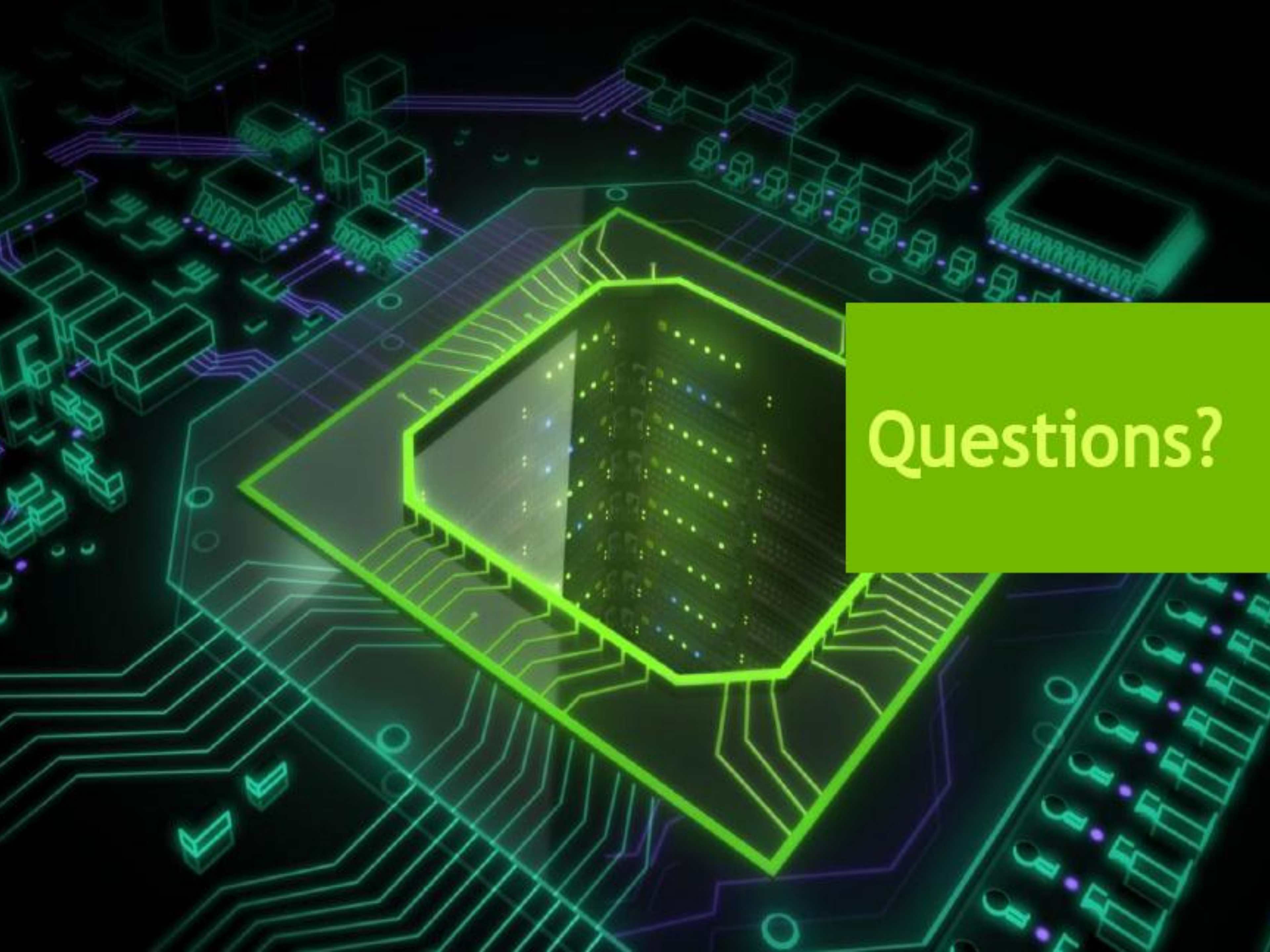
关闭L1缓存：数据只在L2得到缓存。

nvcc -Xptxas –dlcm=cg

对全局内存是以32B为基本单位进行访问

使用情况：

- ◆内存访问非常不合并，没有太多重用
- ◆寄存器使用过多，内核函数有寄存器溢出


Questions?

全局内存的合并访问优化

Specification	Tesla K20xm
Processor	1 x Tesla T20
CUDA cores	2688
on-board memory	6 GB
Memory bandwidth	249.6 GB/s peak
Single/double precision floating performance	3935.23/1311.74 GFlops
System I/O	PCI-E x 16 Gen3
Board power	<= 235W

全局内存的合并访问优化

--ptxas-options=-v
输出资源使用信息；然后将其填写入
occupation calculator