高性能计算与云计算

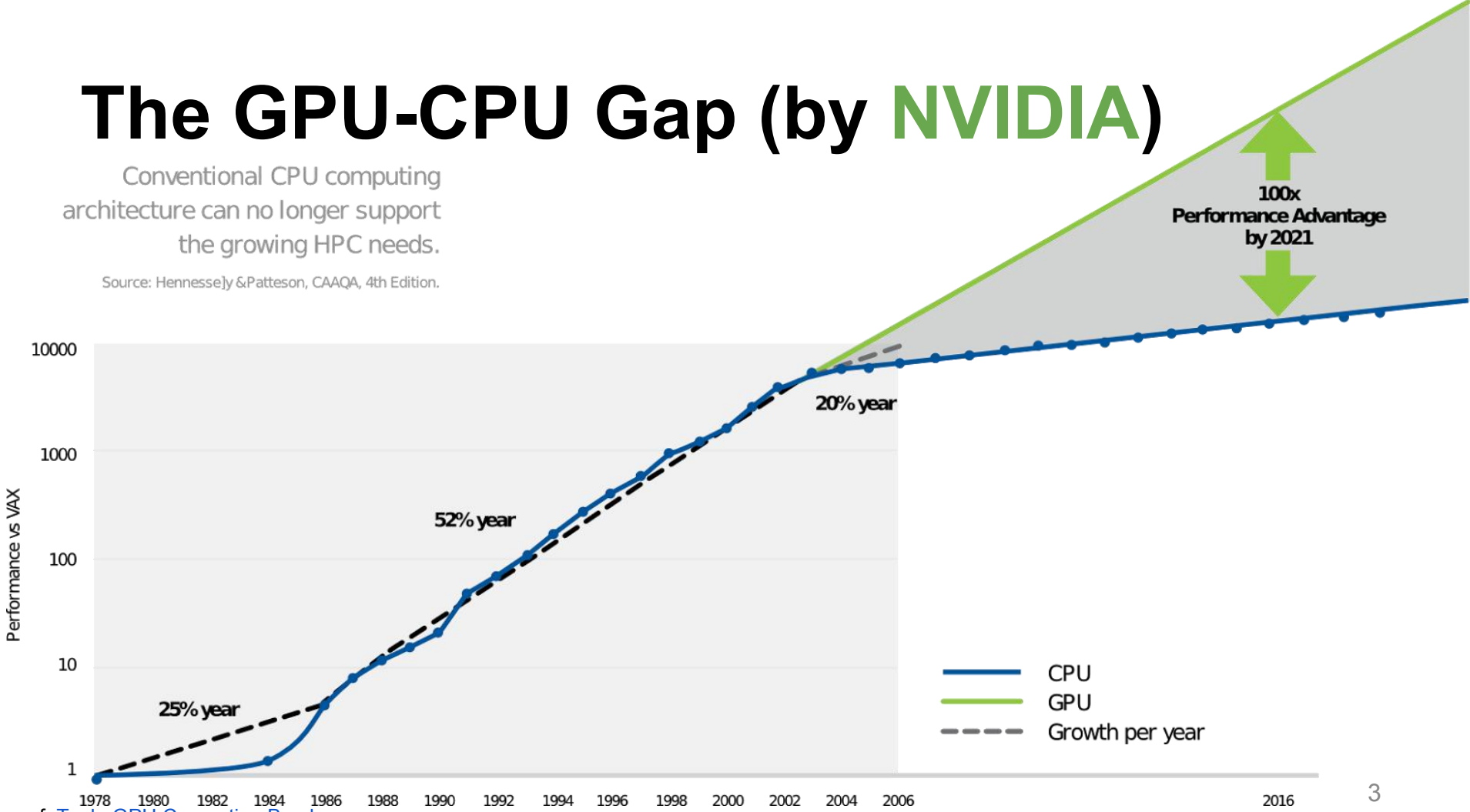
第九讲 CUDA编程

目录

The GPU-CPU Gap (by NVIDIA)

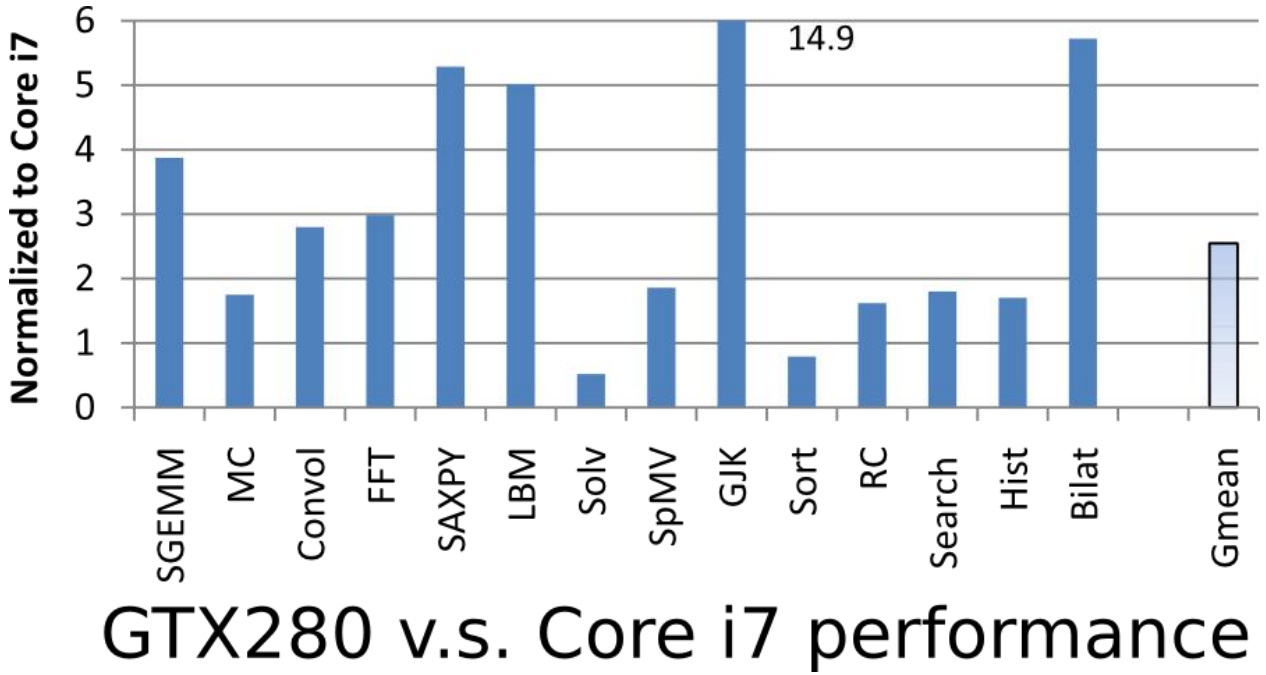
Conventional CPU computing architecture can no longer support the growing HPC needs.

Source: Hennessey &Patteson, CAAQA, 4th Edition.



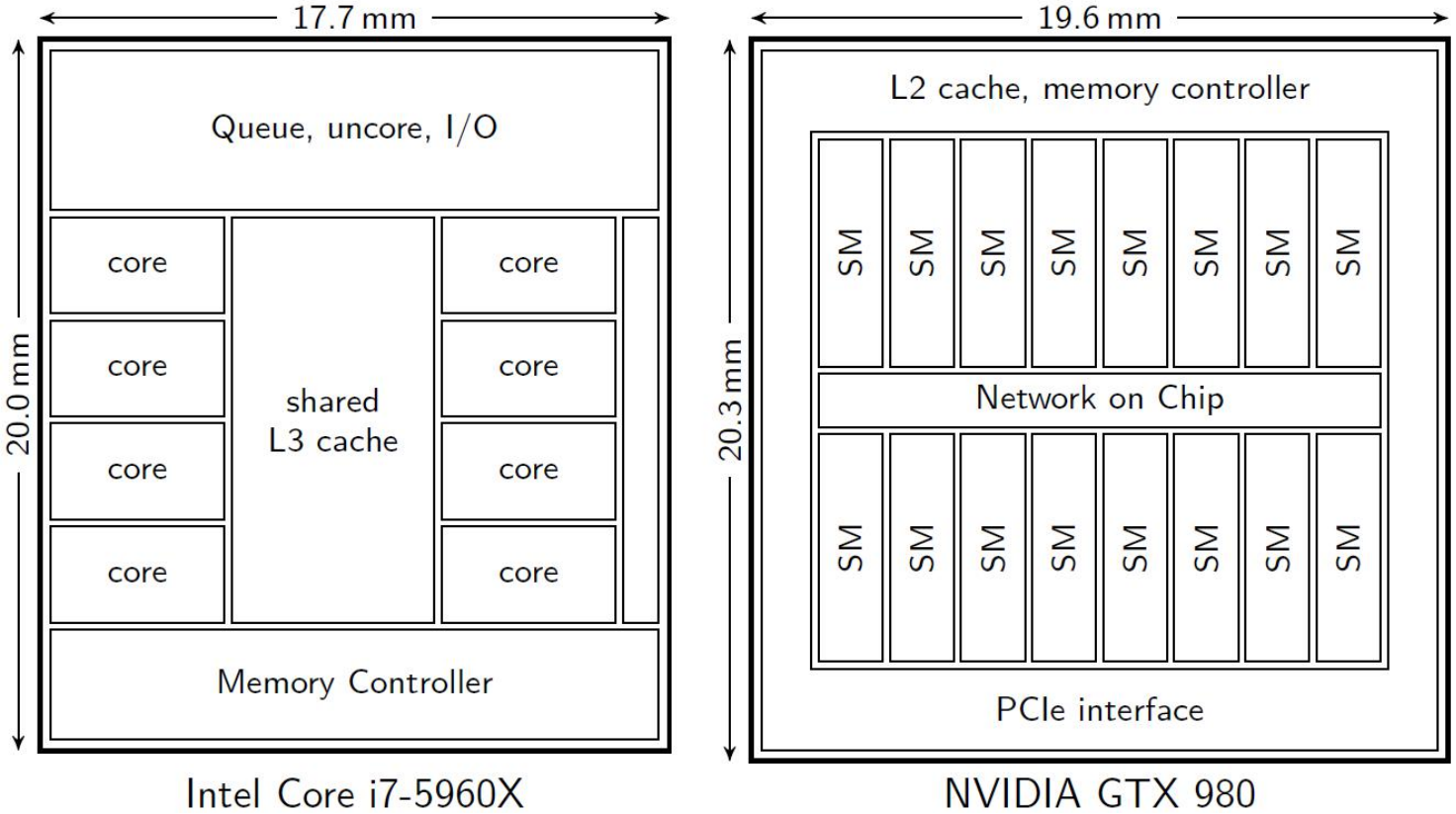
ref: Tesla GPU Computing Brochure

The GPU-CPU Gap (by Intel, 2010)



ref: "Debunking the 100X GPU vs. CPU myth", http://dx.doi.org/10.1145/1815961.1816021

CPU vs. GPU – 芯片面积



ref: "Improving GPU Performance - Reducing Memory Conflicts and Latency", Gert-Jan v.d. Braak - http://repository.tue.nl/ffb3603c-a702-42bd-9a67-56d5c2b65603

CPU与GPU的比较

- 传统CPU
  - ✓ 将芯片的大部分区域用于片上缓存；
  - ✓ CPU core比较重，用来处理非常复杂的控制逻辑，以优化串行程序执行。
- GPU
  - ✓ 将芯片上的大部分区域用于计算逻辑；
  - ✓ GPU Core比较轻，用于优化具有简单控制逻辑的数据并行任务，注重并行程序的吞吐量。

- 多核（multi-core）vs. 众核（many-core）

  多核：CPU ；　众核：GPU

目录

GPU的应用

➢历史上，GPU最初被设计用来专门处理并行图形计算问题，是图形加速器。

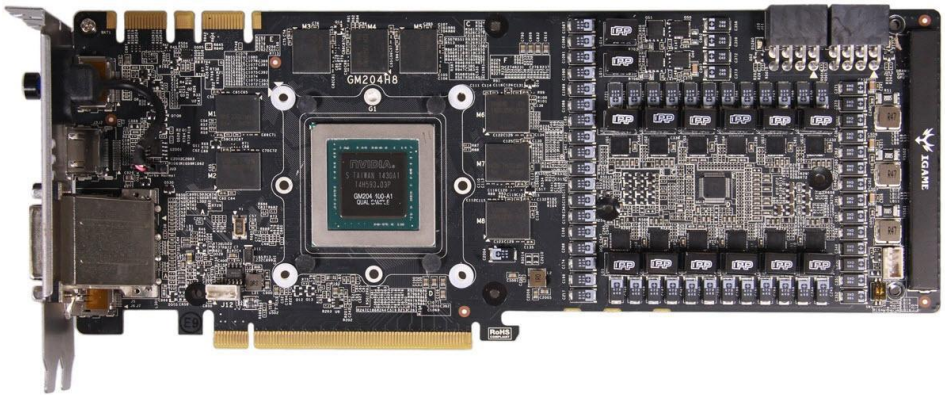
- ✓ GPU使显卡减少了对CPU的依赖，并进行部分原本CPU的工作，尤其是在3D图形处理时

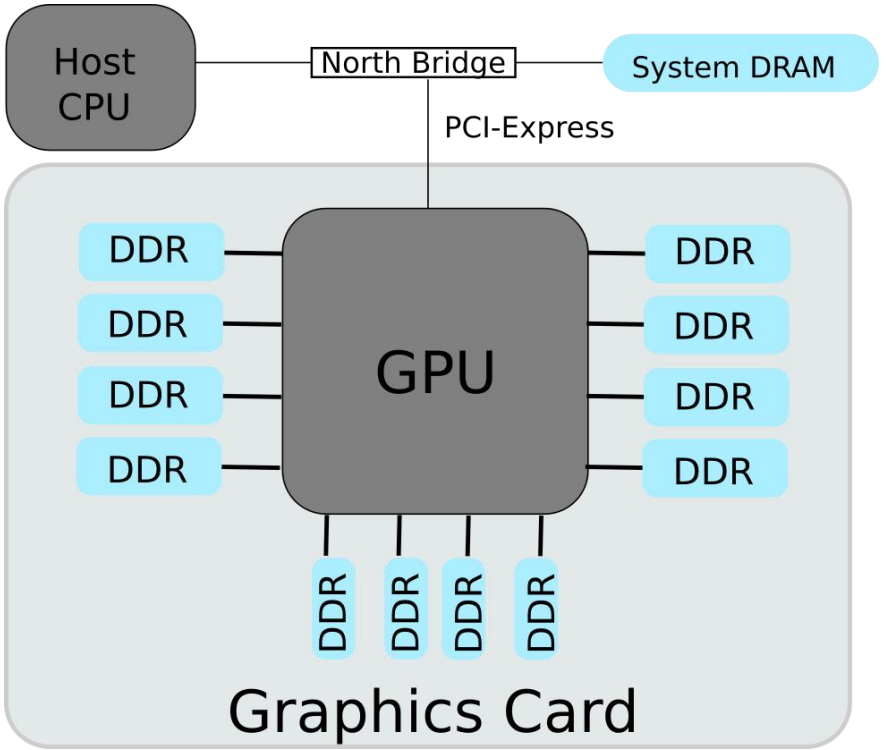
➢目前，GPU已演化成一个强大的、多用途的、完全可编程的，以及任务和数据并行的处理器。

➢异构架构：GPU作为CPU的协处理器，GPU通过PCIe总线与CPU相连。

显卡中的GPU

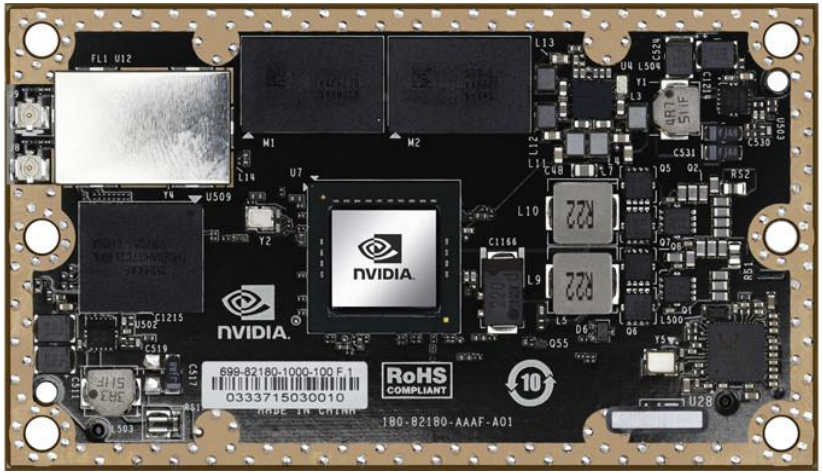
Image: Nvidia GTX 980

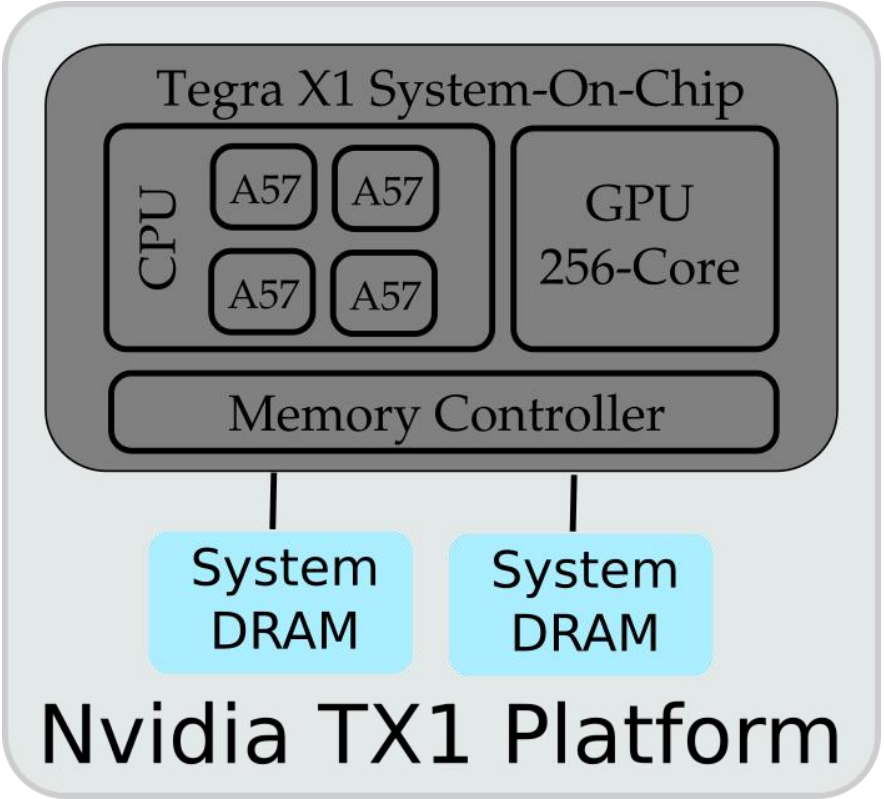




移动处理器中的GPU

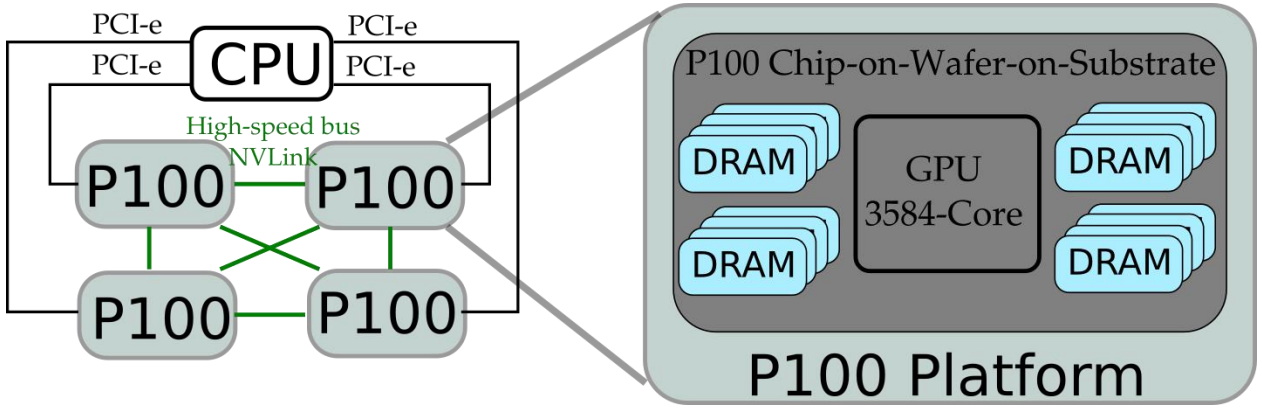
Image: Nvidia Jetson TX1 (Tegra X1 SOC)



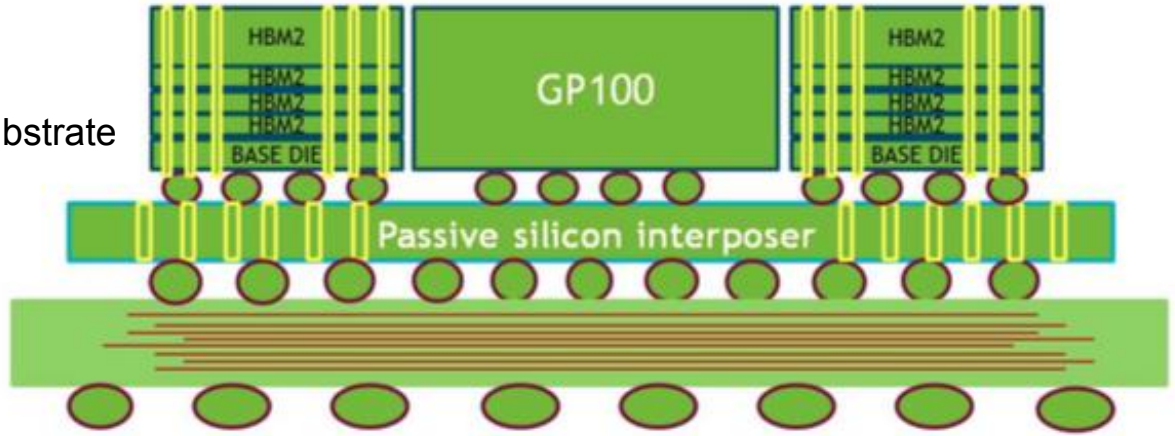


高性能计算机中的GPU

Image: Nvidia P100
(Pascal Architecture)



Chip-on-Wafer-on-Substrate



高性能计算机中的GPU

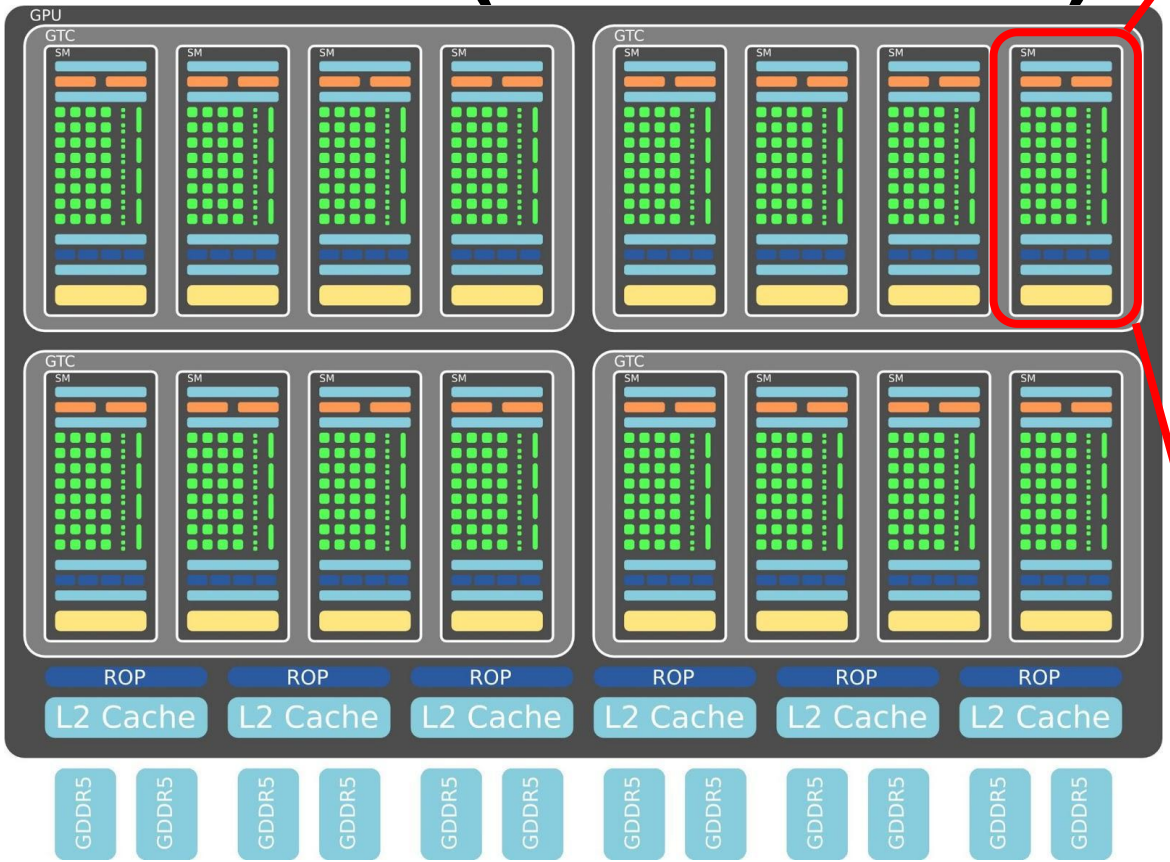
TOP500 supercomputer list in Jun. 2019. (http://www.top500.org)

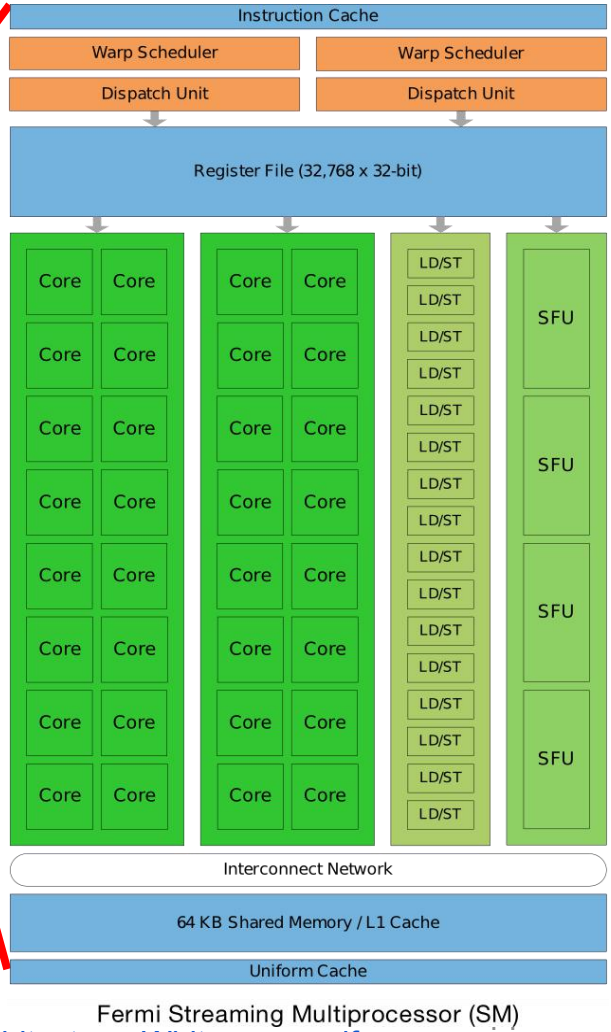
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100 Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,397,824	143,500.0	200,794.9	9,783
2	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100 Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371

目录

- GPU微处理器简介
  - ✓GPU与CPU的比较
  - ✓GPU应用
  - ✓GPU举例
- CUDA编程

NVIDIA (Fermi 架构)





Fermi Streaming Multiprocessor (SM)

ref: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

晶体管数目

ref: http://en.wikipedia.org/wiki/Transistor_count

SOC

处理器	晶体管	工艺技术
61-Core Xeon Phi	5,000,000,000	22nm
22-core Xeon Broadwell-E5	7,200,000,000	14nm
32-Core Sparc M7	10,000,000,000+	20nm

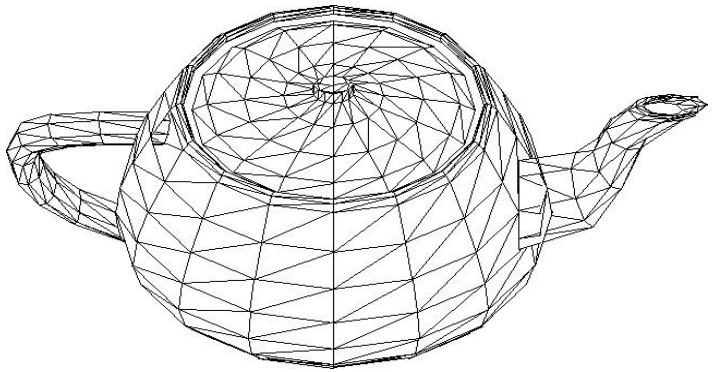
GPU

处理器	晶体管	工艺技术
Nvidia GP100 Pascal	15,300,000,000	16nm

FPGA

FPGA	晶体管	工艺技术
Virtex-Ultrascale XCVU440	20,000,000,000+	20nm
Stratix 10 10GX5500	30,000,000,000+	14nm

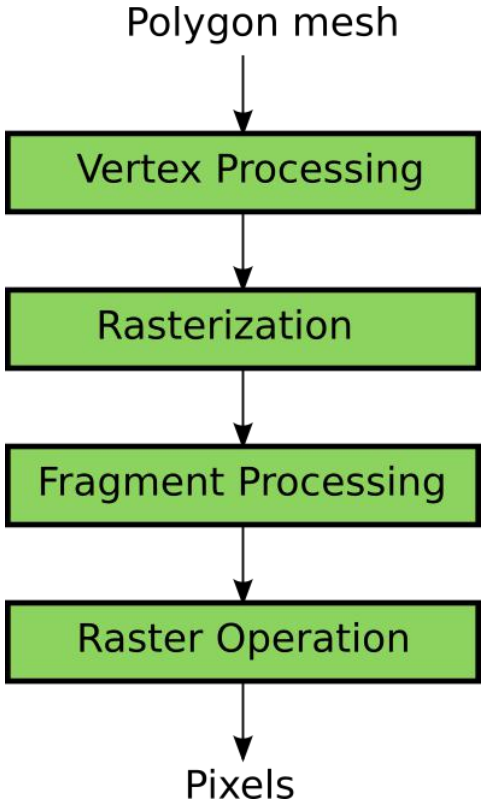
举例：153亿个晶体管的作用



Render triangles.
Billions of triangles per second.

ref: "How GPUs Work", http://dx.doi.org/10.1109/MC.2007.59

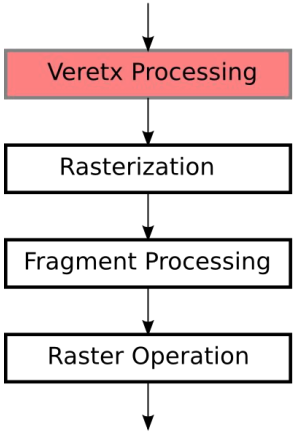
图形流水线（The Graphics Pipeline）

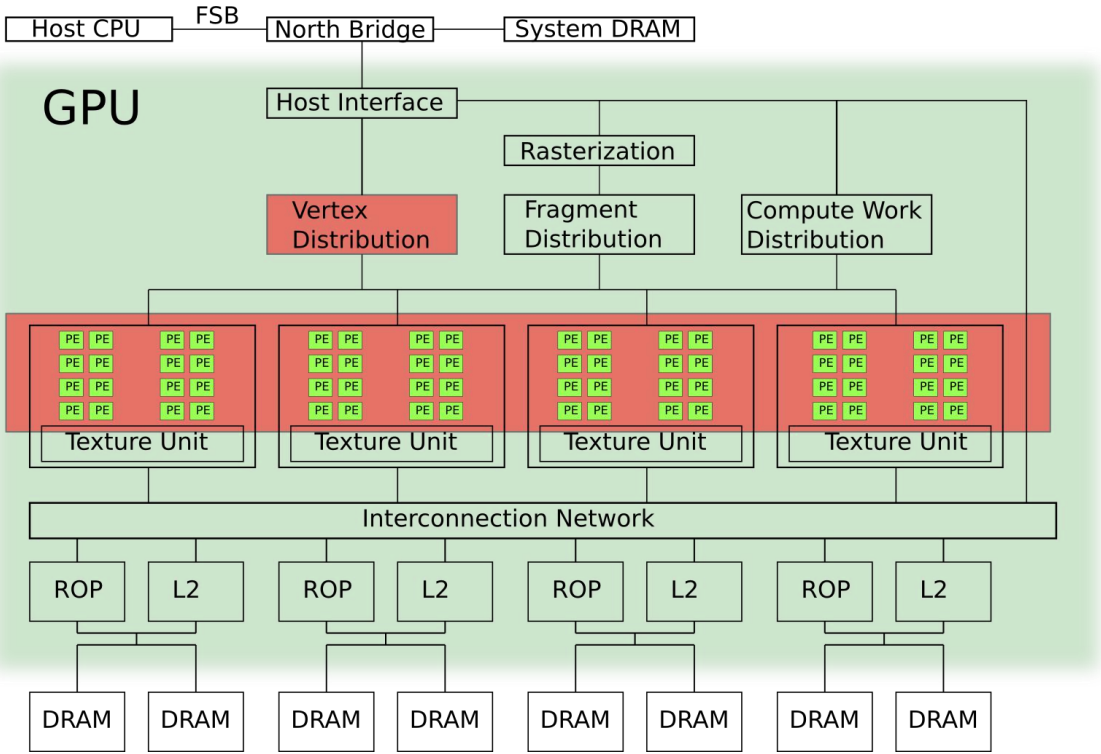


- 图形流水线将3D模型表示为多边形网格，并根据当前帧的视图渲染模型。
- 基本步骤：定点处理，光栅化，片元处理，光栅操作
- 现代图形流水线更加复杂。

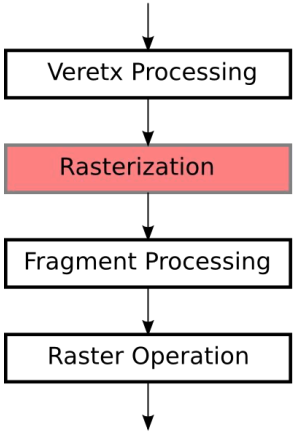
K. Fatahalian, et al. "GPUs: a Closer Look", ACM Queue 2008, http://doi.acm.org/10.1145/1365490.1365498

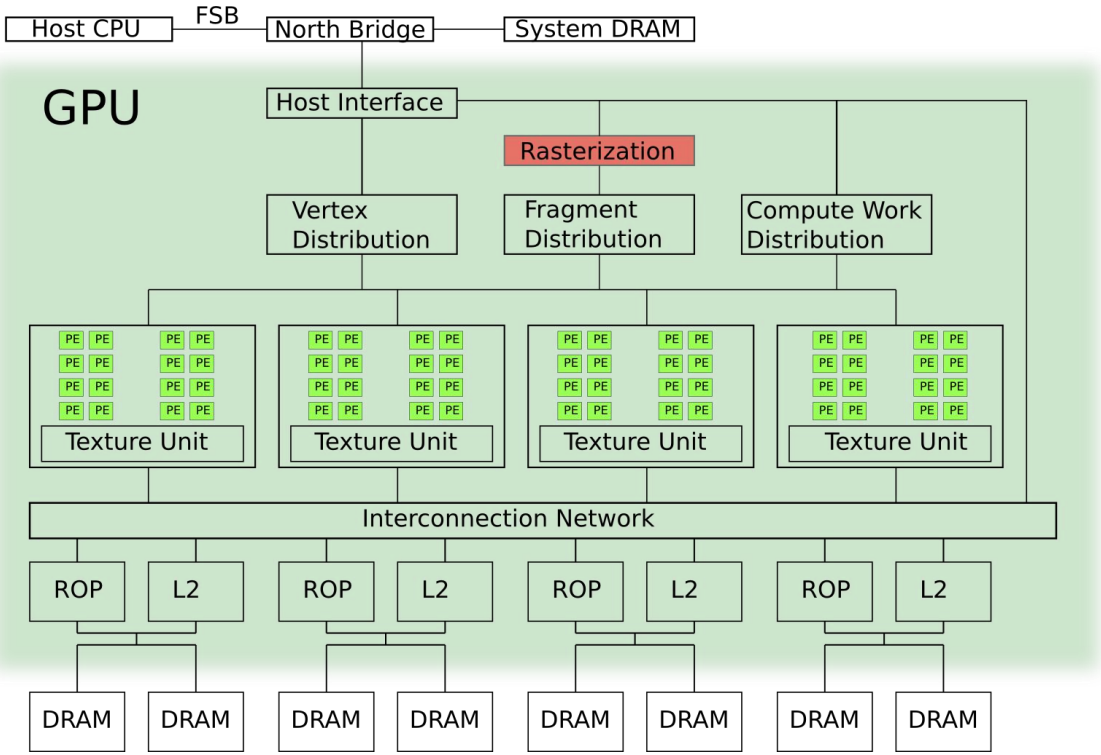
GPU的图形流水线（1）


Veretx Processing
Rasterization
Fragment Processing
Raster Operation

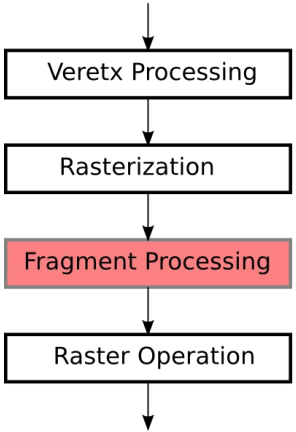

Host CPU
FSB
North Bridge
System DRAM
GPU
Host Interface
Rasterization
Vertex Distribution
Fragment Distribution
Compute Work Distribution
PE
Texture Unit
Interconnection Network
ROP
L2
DRAM

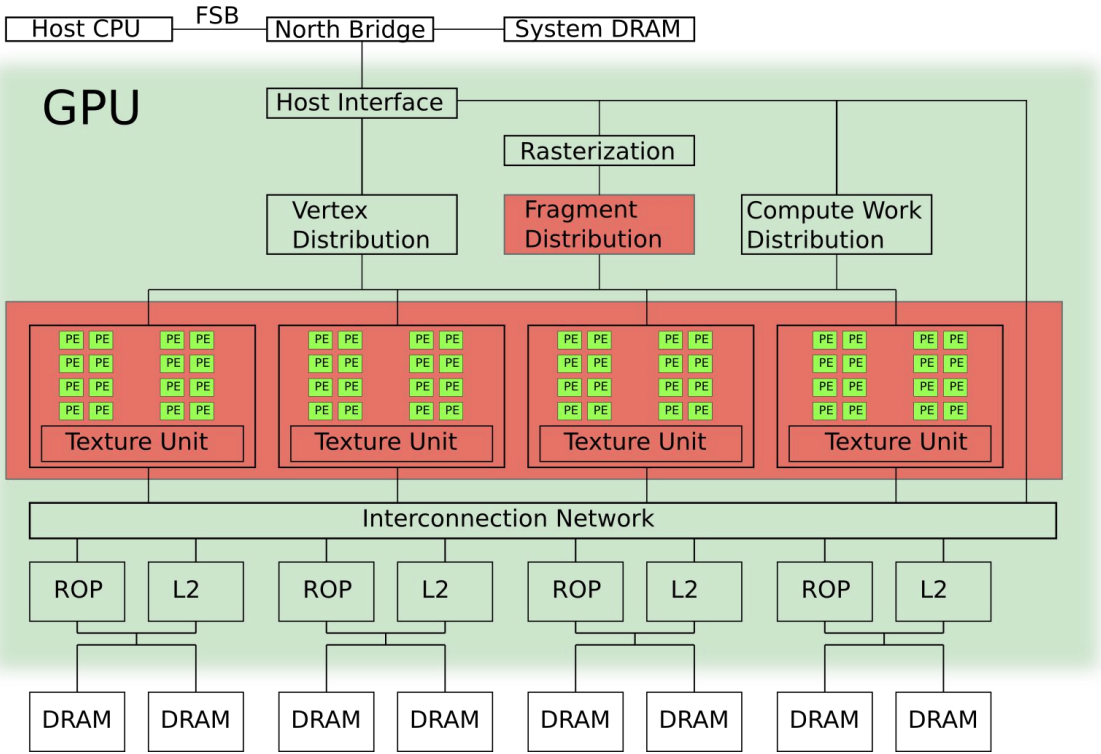
GPU的图形流水线（2）


Veretx Processing
Rasterization
Fragment Processing
Raster Operation

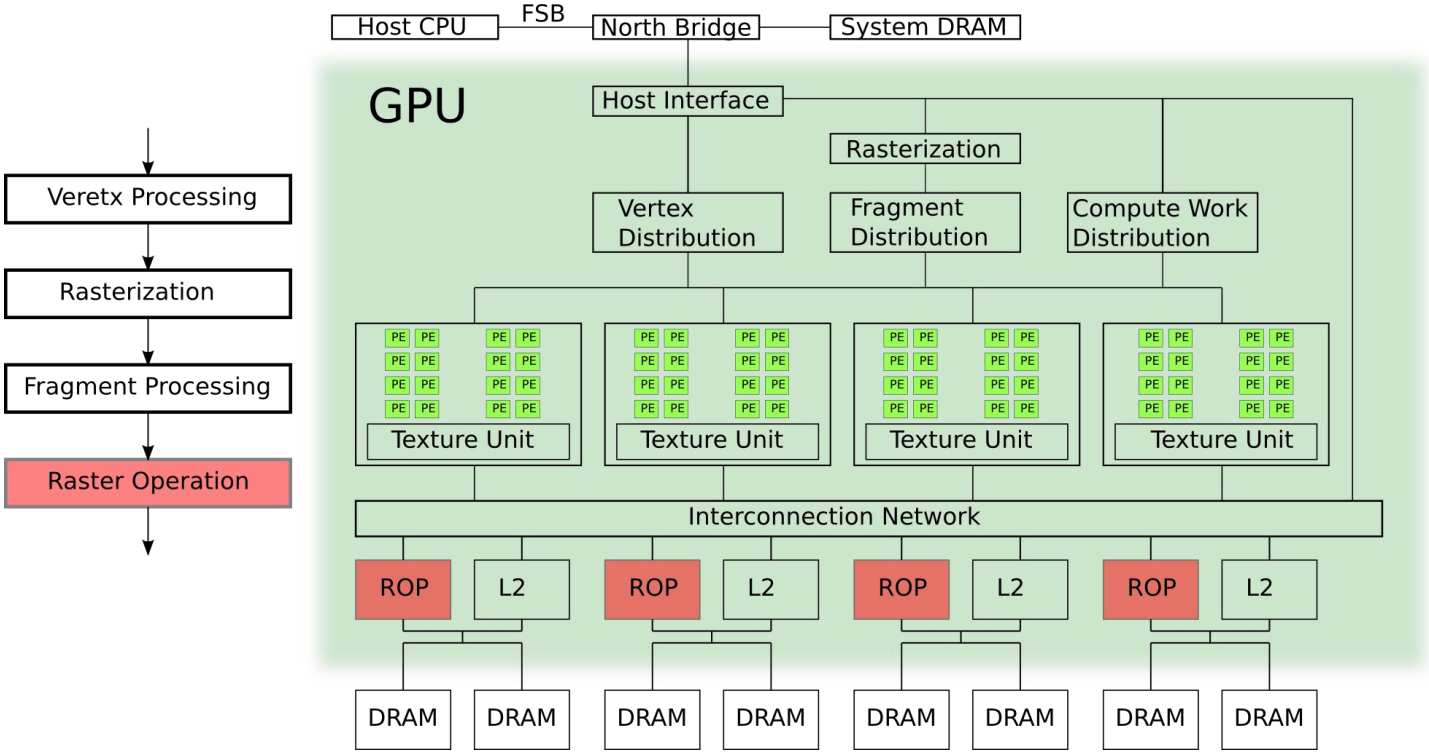

Host CPU
FSB
North Bridge
System DRAM
GPU
Host Interface
Rasterization
Vertex Distribution
Fragment Distribution
Compute Work Distribution
PE
Texture Unit
Interconnection Network
ROP
L2
DRAM

GPU的图形流水线（3）


Veretx Processing
Rasterization
Fragment Processing
Raster Operation


Host CPU
FSB
North Bridge
System DRAM
GPU
Host Interface
Rasterization
Vertex Distribution
Fragment Distribution
Compute Work Distribution
PE
Texture Unit
Interconnection Network
ROP
L2
DRAM

GPU的图形流水线（4）


Veretx Processing
Rasterization
Fragment Processing
Raster Operation
Host CPU
FSB
North Bridge
System DRAM
GPU
Host Interface
Rasterization
Vertex Distribution
Fragment Distribution
Compute Work Distribution
PE
Texture Unit
Interconnection Network
ROP
L2
DRAM

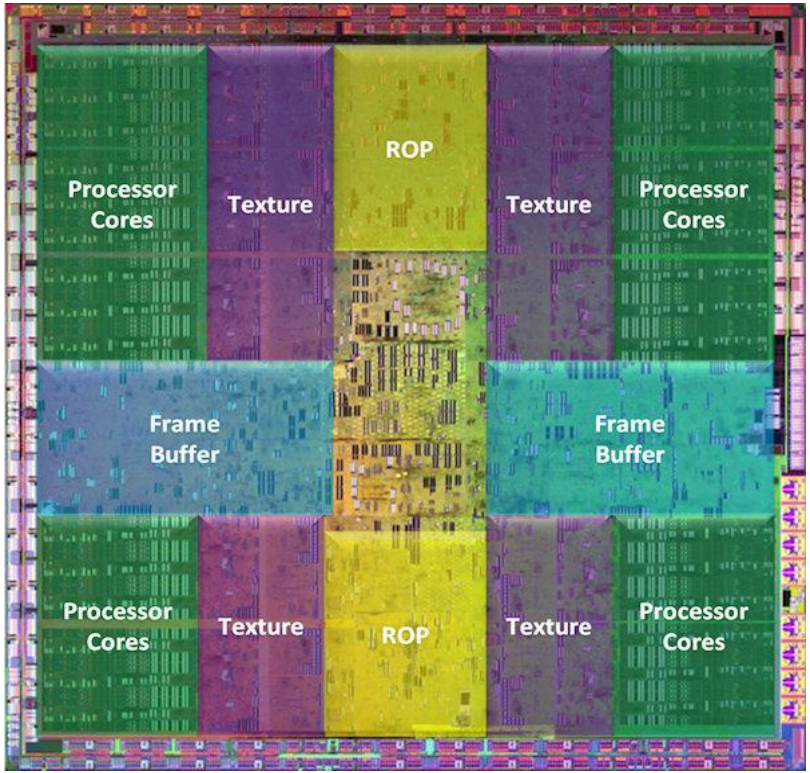
GPUs内部区域

GPUs are designed to match the workload of 3D graphics.

Die photo of GeForce GTX 280 (source: NVIDIA)

Texture:
For fragment processing.

ROP & Frame Buffer:
For raster operation.



J. Roca, et al. "Workload Characterization of 3D Games", IISWC 2006, link
T. Mitra, et al. "Dynamic 3D Graphics Workload Characterization and the Architectural Implications", Micro 1999, link

目录

- GPU微处理器简介
- CUDA编程
  - 引言：开发平台、异构计算
  - CUDA编程模型：线程、线程块、线程网格
  - CUDA执行模型：流多处理器和线程束
  - 内存层次结构
  - 编程实例

GPU计算的历史

- 2001/2002 - 研究人员把GPU当做数据并行协处理器
  - *GPGPU* 这个新领域从此诞生
- 2007 - NVIDIA 发布 CUDA
  - *CUDA* - 全称Compute Uniform Device Architecture 统一计算设备架构
  - GPGPU 发展成 *GPU Computing*
- 2008 - Khronos 发布 *OpenCL* 规范

GPU主流开发平台

- CUDA:

Compute Unified Device Architecture，2006年英伟达发布，将GPU计算由传统图形处理领域扩展到通用计算领域，支持c，c++，Fortran和Python等语言，提供2组API（一个底层，一个更高层），是实现更高层第三方API和库的基础。CUDA只针对英伟达硬件。

- OpenCL

Open computing Language开放计算语言，可以在如GPU、CPU、DSP和其他处理器等众多计算平台上进行编程的开放标准，英伟达和AMD都支持OpenCL。与CUDA编程模型相似。

GPU主流开发平台（2）

- OpenACC

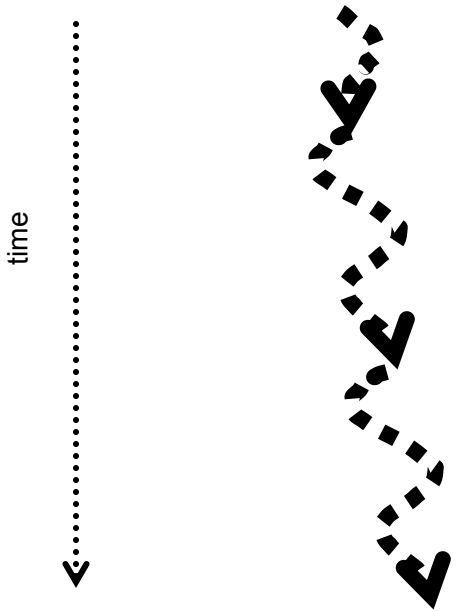
  是一个API的开放规范，允许使用编译制导语句（如#progma acc，风格与OpenMP类似），根据用户提示自动将计算映射到GPU或者多核芯片上。

- C++AMP

  是微软基于DiretX 11开发的技术，允许C++代码根据程序员提供的大量指令或者语言扩展在CPU或者GPU计算平台上透明执行。

- 回顾：**Sequential Execution Model**

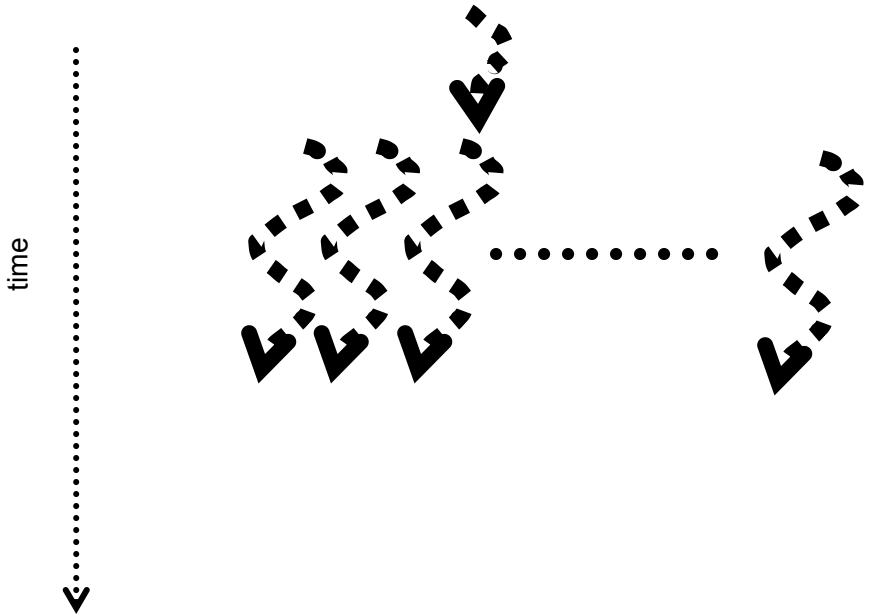
```
int a[N]; // N is large
for (i =0; i < N; i++)
  a[i] = a[i] * fade;
```



Flow of control / Thread
One instruction at the time
Optimizations possible at the machine level

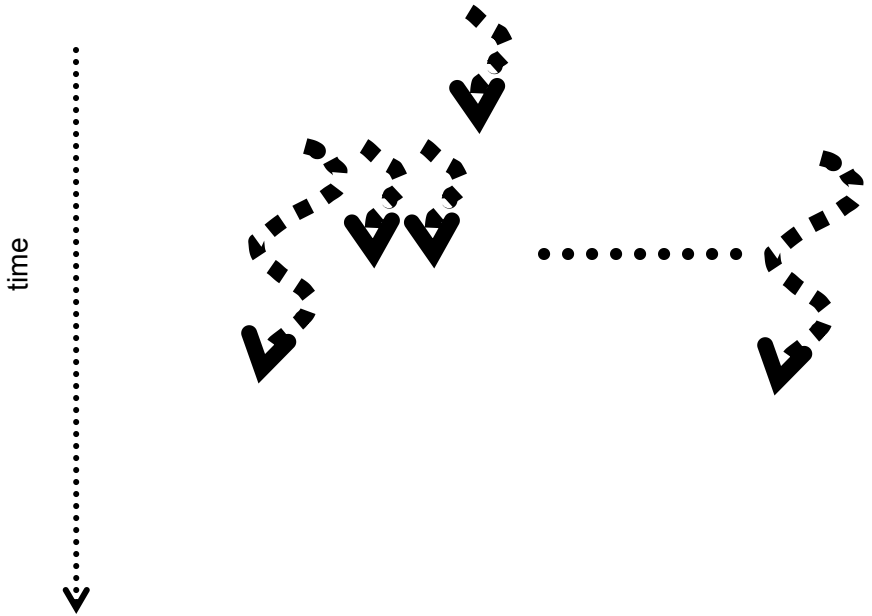
- 回顾：**Data Parallel Execution Model / SIMD**

```
int a[N]; // N is large
for all elements do in parallel
  a[index] = a[index] * fade;
```

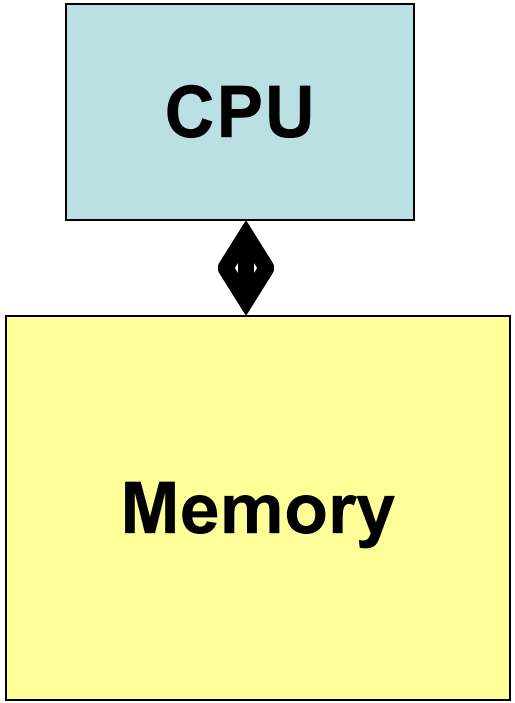


- 回顾：**Single Program Multiple Data / SPMD**

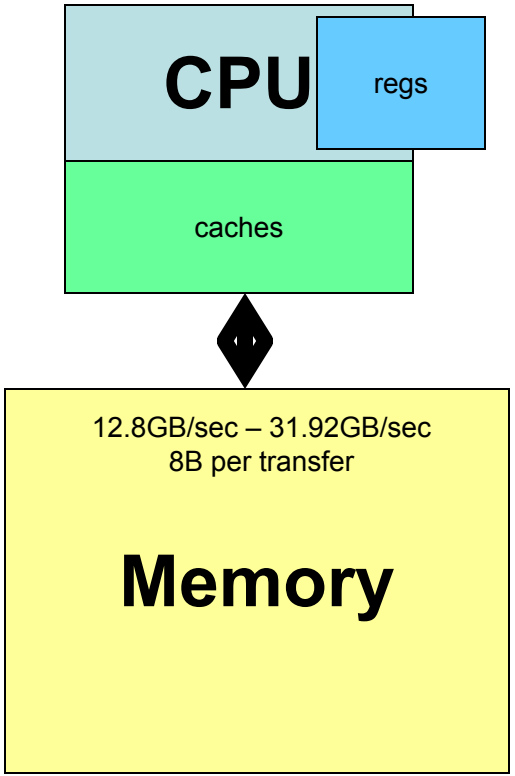
```
int a[N]; // N is large
for all elements do in parallel
  if (a[i] > threshold) a[i]*= fade;
```



- 回顾：程序员视角 – 典型系统

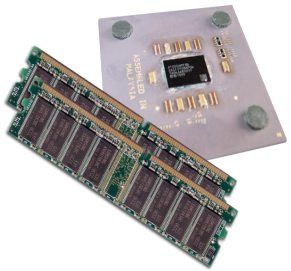


如特别关注性能



异构计算

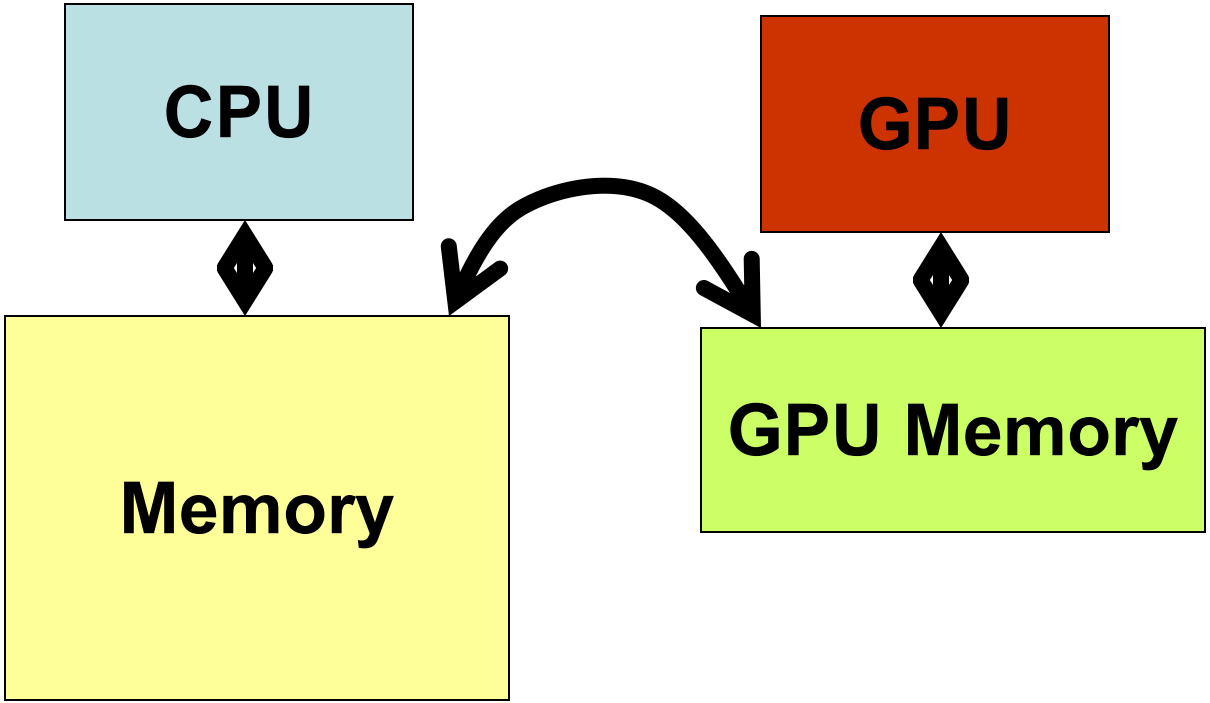
- 术语:
  - 主机(Host)　　CPU和它的内存 (主机内存）
  - 设备(Device)　　GPU和它的内存 (设备内存)



Host

Device

- 异构计算（2）
- 主机（Host） 设备（Device）



异构计算：全局串行局部并行

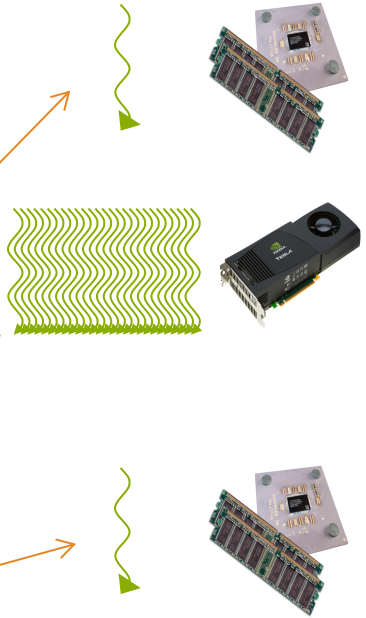
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex +
BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in +
RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

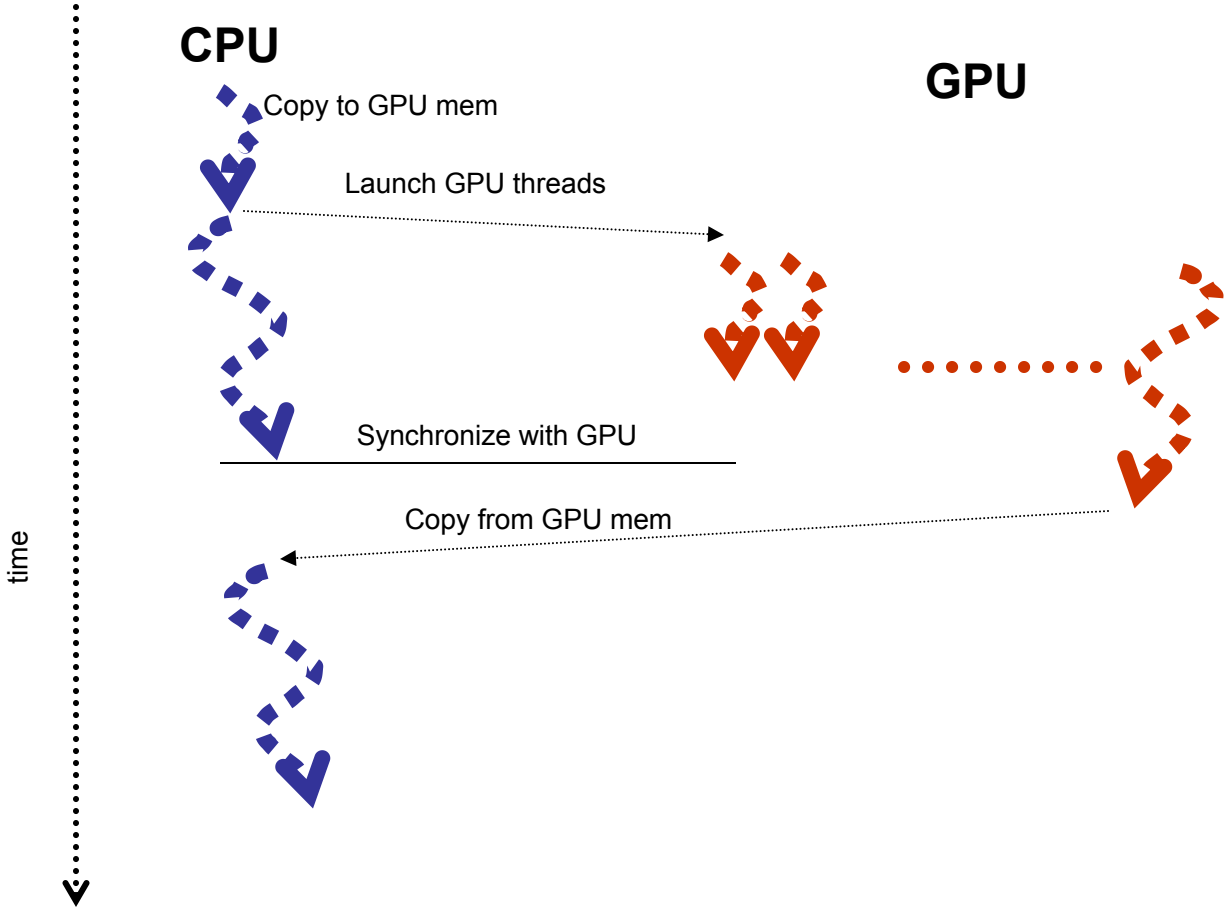
serial code

parallel code

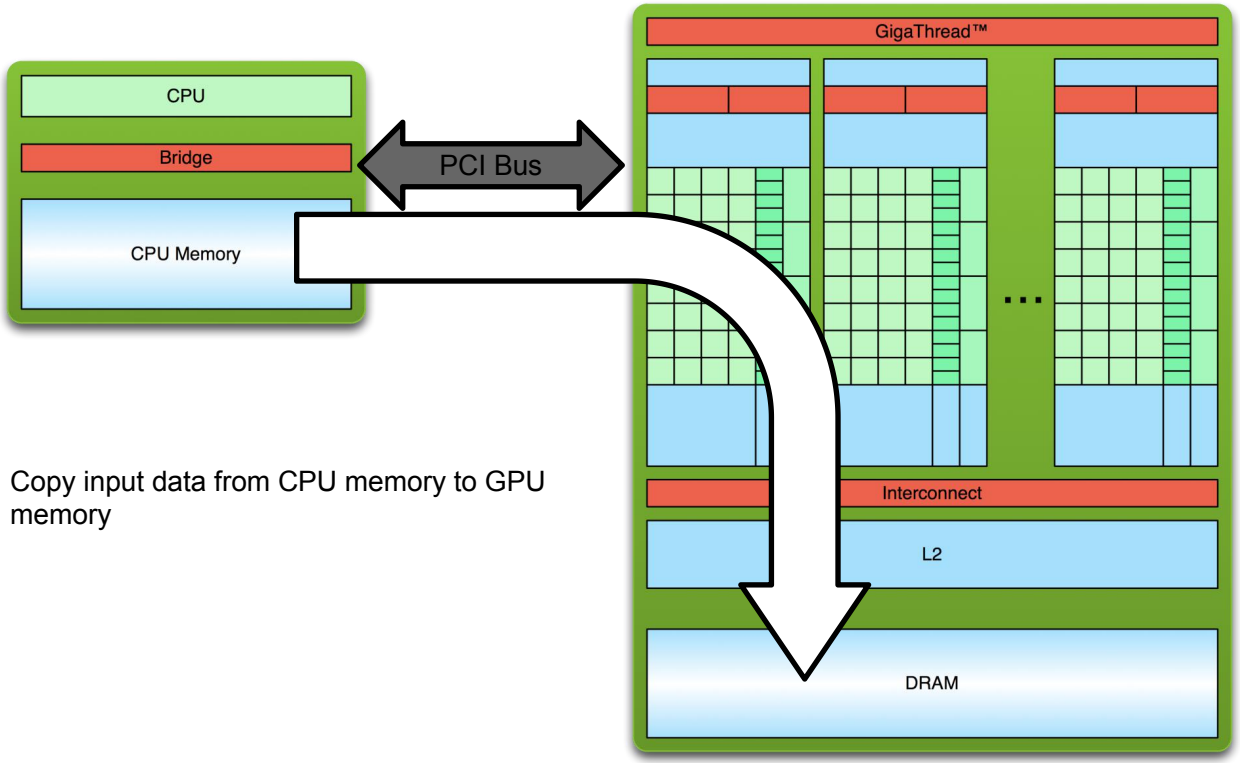
serial code



- 异构计算：全局串行局部并行（2）

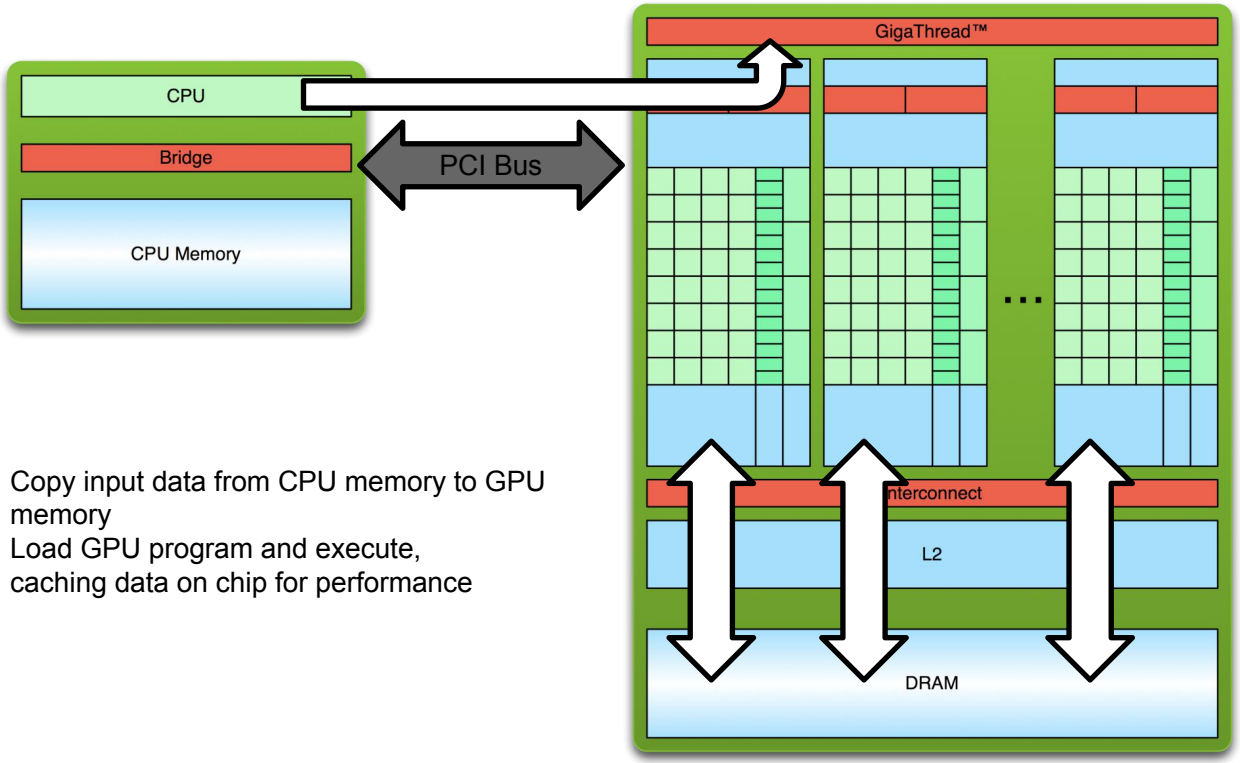


简单处理流程



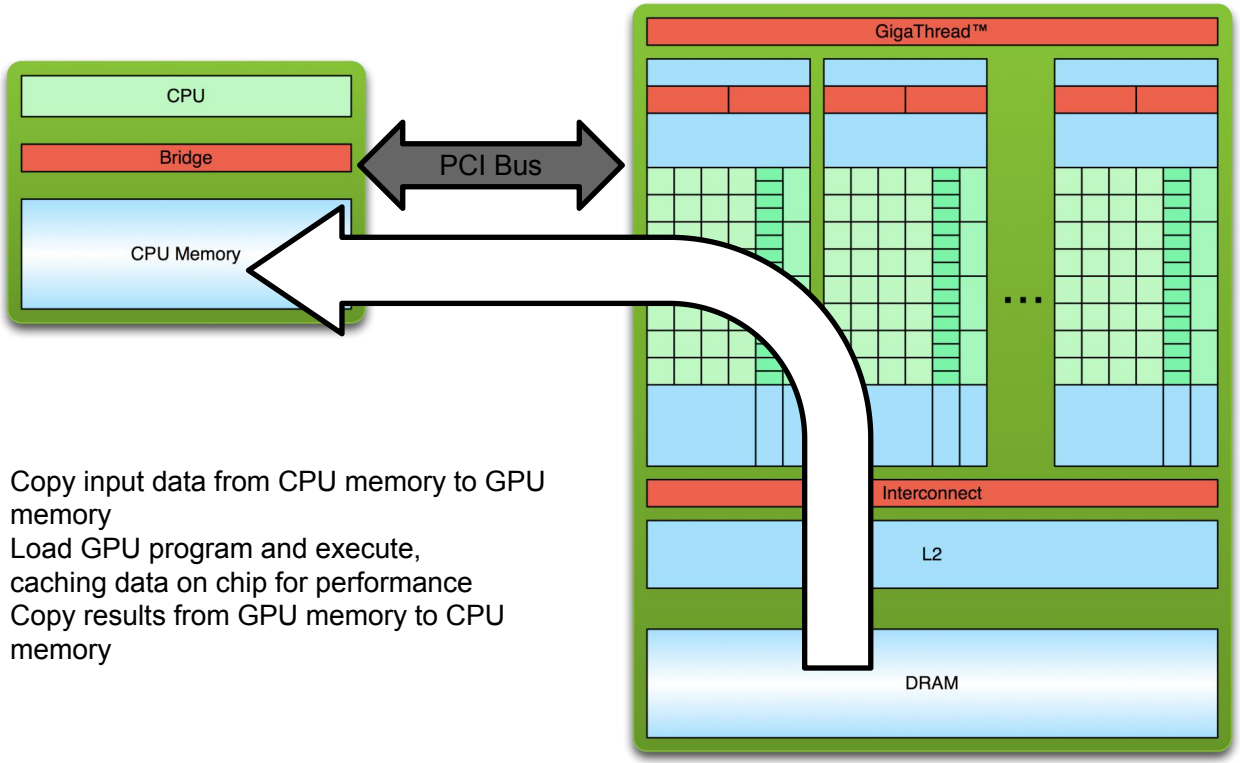
1. Copy input data from CPU memory to GPU memory

简单处理流程（2）



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

简单处理流程（3）



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

目录

- GPU微处理器简介
- CUDA编程
  - 引言：开发平台、异构计算
  - **CUDA编程模型：线程、线程块、线程网格**
  - CUDA执行模型：流多处理器和线程束
  - 内存层次结构
  - 编程实例

Hello World

```
#include <stdio.h>
__global__ void helloFromGPU (void)
{
    printf("Hello World from GPU!\n");
}

int main(void)
{
// hello from cpu
printf("Hello World from CPU!\n");

helloFromGPU <<<1, 10>>>();

cudaDeviceReset();

return 0;
}
```

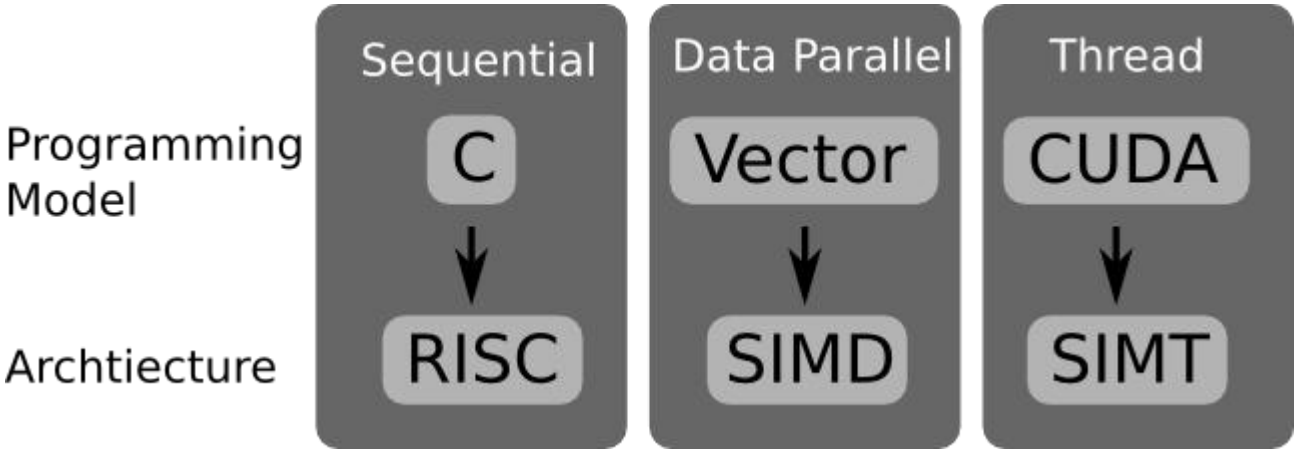
```
$nvcc –arch sm_20 hello.cu –o hello
$ ./hello
```

```
Hello World from CPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
```

程序结构的5个步骤：

1. 分配GPU空间;
2. 将数据从CPU端复制到GPU端;
3. 调用CUDA kernel来执行计算;
4. 计算完成后将数据从GPU拷贝回CPU;
5. 清理GPU内存空间。

举例：GPU编程—编程模型与架构



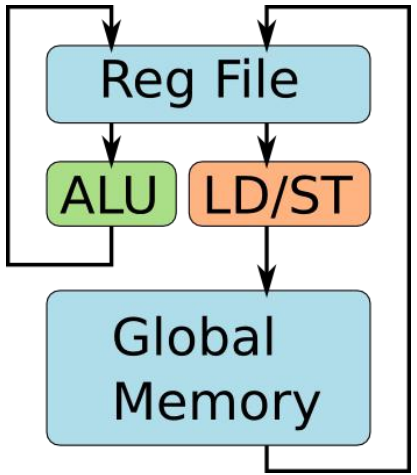
回顾：C语言和RISC架构

回顾：C 语言和 RISC架构

```
int A[2][4];
for(i=0;i<2;i++){
    for(j=0;j<4;j++){
        A[i][j]++;
    }
}
```

循环内的汇编代码

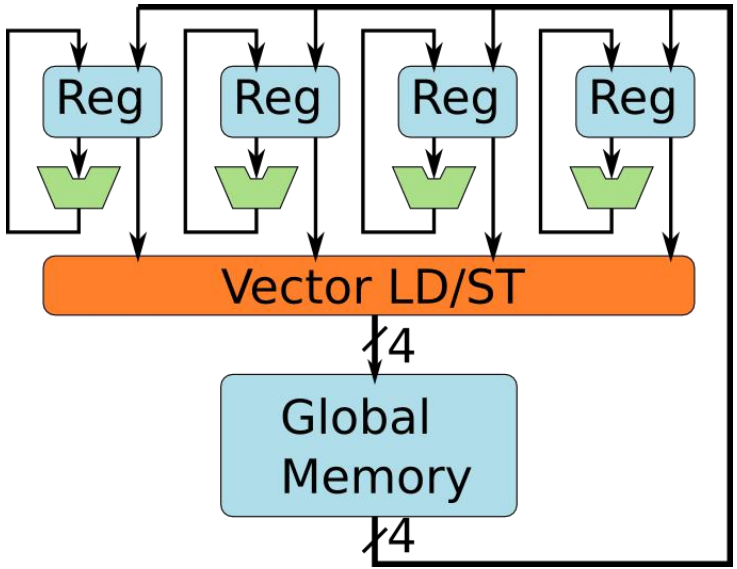
```
lw r0, 4(r1)
addi r0, r0, 1
sw r0, 4(r1)
```



Programmer's view of RISC

回顾：C 语言和 RISC架构（2）

许多 **CPUs具有vector（矢量）单指令多数据流（SIMD）单元**



- Programmer‘s view of a vector SIMD
- 如Intel Streaming SIMD Extension(SSE)

回顾：C 语言和 RISC架构（3）
矢量SIMD编程

内部循环展开成向量SIMD：

```
int A[2][4];
for(i=0;i<2;i++){
    for(j=0;j<4;j++){
        A[i][j]++;
    }
}
```

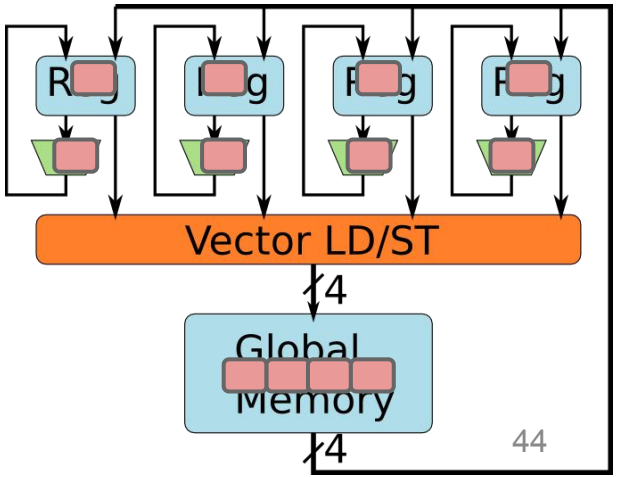
```
int A[2][4];
for(i=0;i<2;i++){
    movups  xmm0,       [ &A[i][0] ]  // load
    addps   xmm0,       xmm1          // add 1
    movups  [ &A[i][0] ], xmm0        // store
}
```

与前面的例子一样，但每个SSE指令在 4 ALUs上执行.

回顾：C 语言和 RISC架构（4）

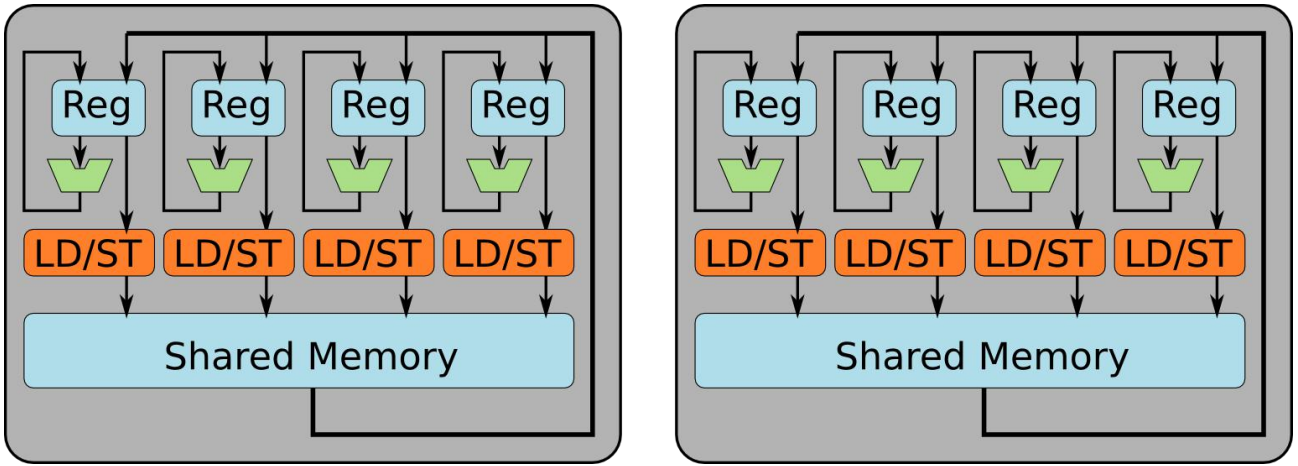
矢量程序的执行

```
int A[2][4];
for(i=0;i<2;i++){
    movups xmm0,       [ &A[i][0] ]  // load
    addps  xmm0,       xmm1          // add 1
    movups [ &A[i][0] ], xmm0        // store
}
```



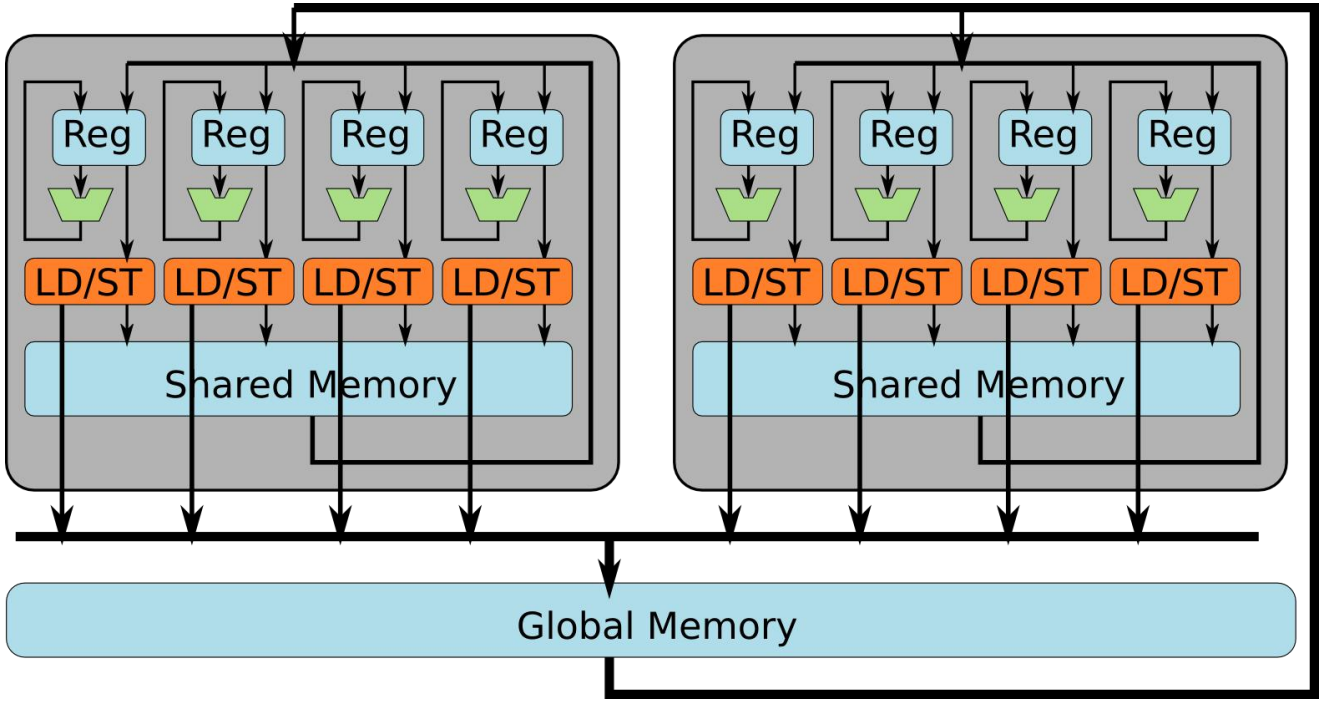
CUDA 程序员的GPUs视角

一个GPU具有多个SIMD单元.



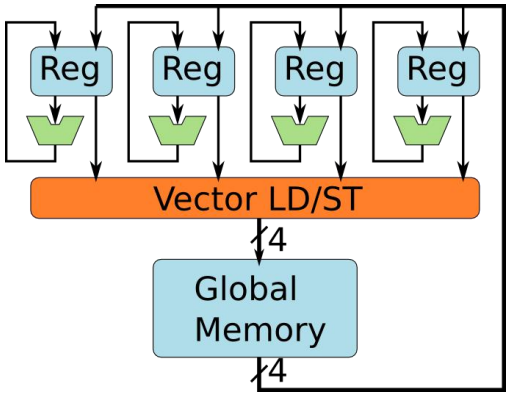
CUDA 程序员的GPUs视角（2）

A GPU contains multiple SIMD Units. 所有SIMD单元都可以访问全局内存.

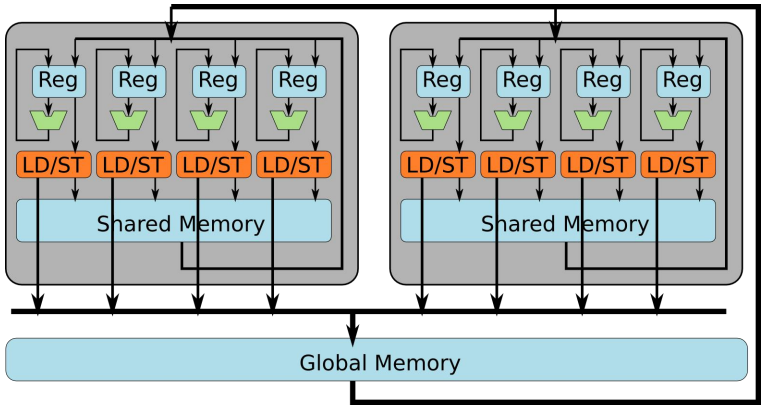


SIMD CPU(SSE)与GPU的区别

SSE



GPU

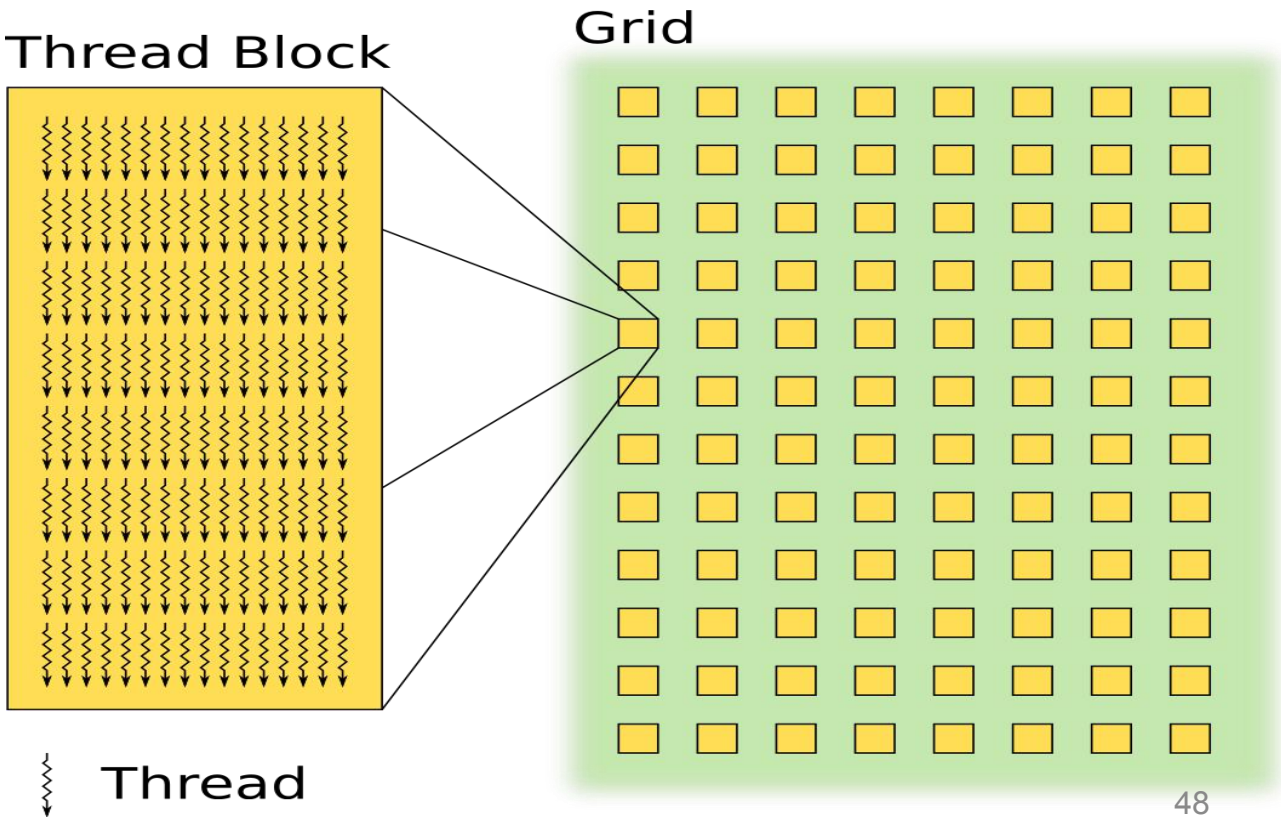


两个主要不同:

1. GPUs 使用线程（ threads）代替矢量（ vectors）
2. GPUs 具有共享内存 （Shared Memory）空间

CUDA的线程层次结构

- 线程网格(Grid)包含线程块(Thread Blocks)
- 线程块包含线程(Threads)

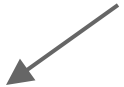


从C到CUDA C

```
int A[2][4];
for(i=0;i<2;i++)
    for(j=0;j<4;j++)
        A[i][j]++;
```

被线程网格中所有线程执行的函数，在CUDA术语中，该函数称为Kernel. 即核函数

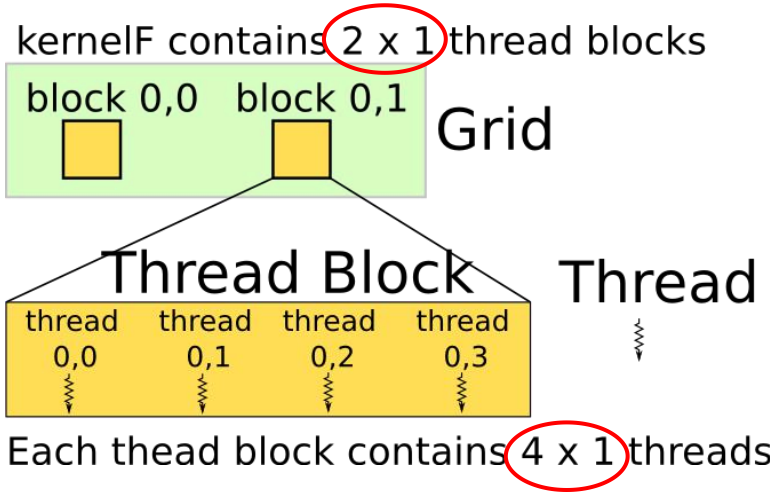
转换成 CUDA



```
int A[2][4];
kernelF<<<(2,1),(4,1)>>>(A);   // define 2x4=8 threads
__device__    kernelF(A){      // all threads run same kernel
    i = blockIdx.x;            // each thread block has its id
    j = threadIdx.x;           // each thread has its id
    A[i][j]++;                 // each thread has different i and j
}
```

线程层次结构

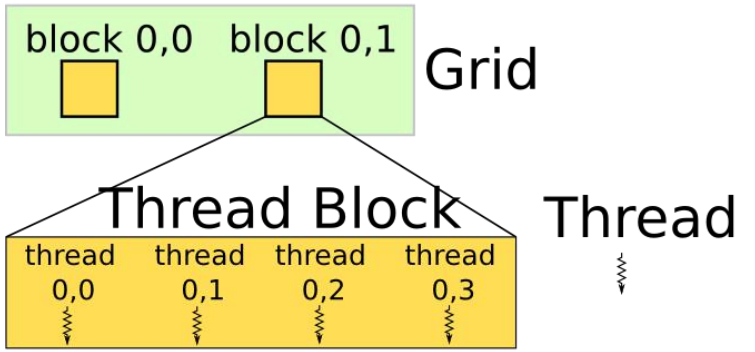
例如:
thread 3 of block 1 operates
on element A[1][3]



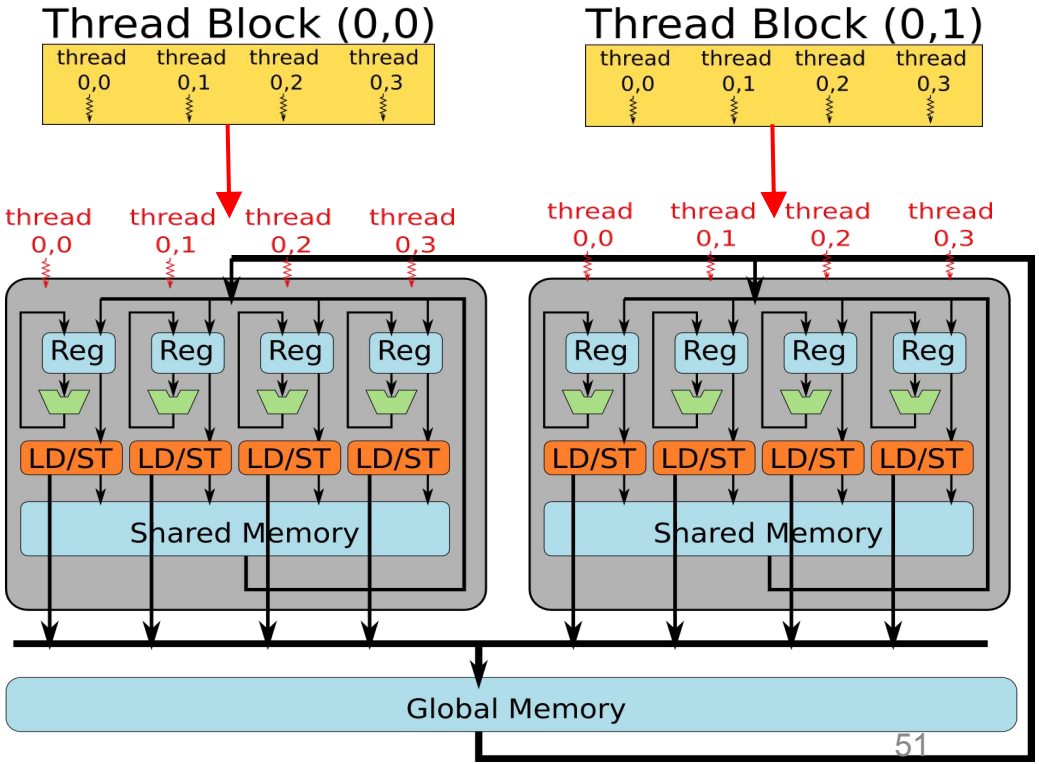

```
int A[2][4];
kernelF<<<(2,1),(4,1)>>>(A); // define 2x4=8 threads
__device__     kernelF(A){
    i = blockIdx.x;          // all threads run same kernel
    j = threadIdx.x;         // each thread block has its id
    A[i][j]++;               // each thread has its id
}                            // each thread has different i and j
```

线程分配

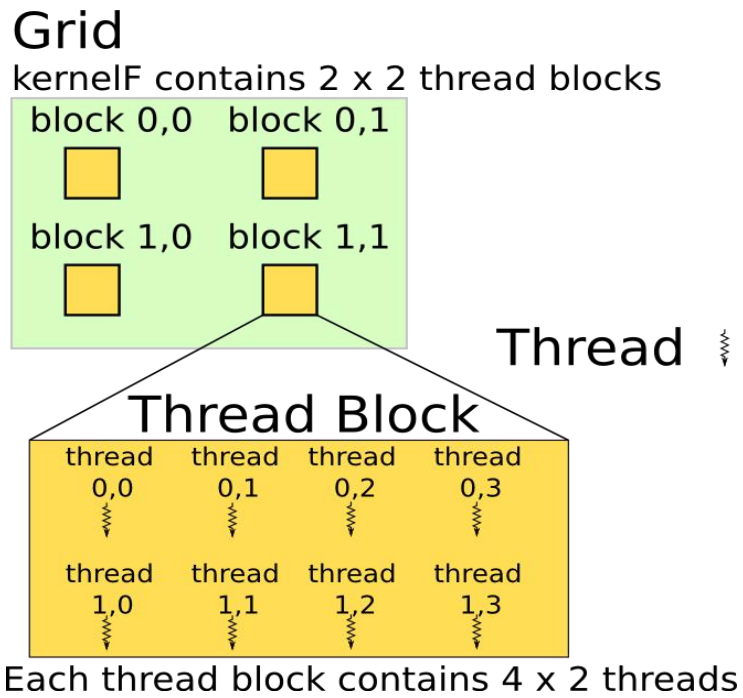
kernelF contains 2 x 1 thread blocks

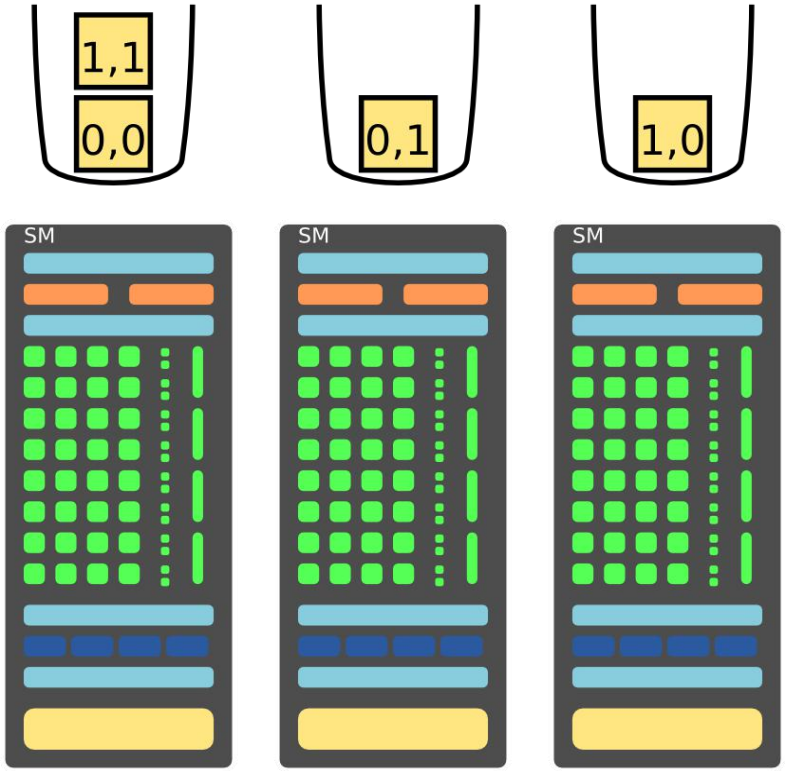


Each thead block contains 4 x 1 threads



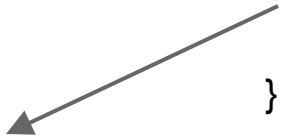
线程块动态分配到SIMD组中





线程执行

```
int A[2][4];
kernelF<<<(2,1),(4,1)>>>(A);
__device__ kernelF(A){
    i = blockIdx.x;
    j = threadIdx.x;
    A[i][j]++;
}
```



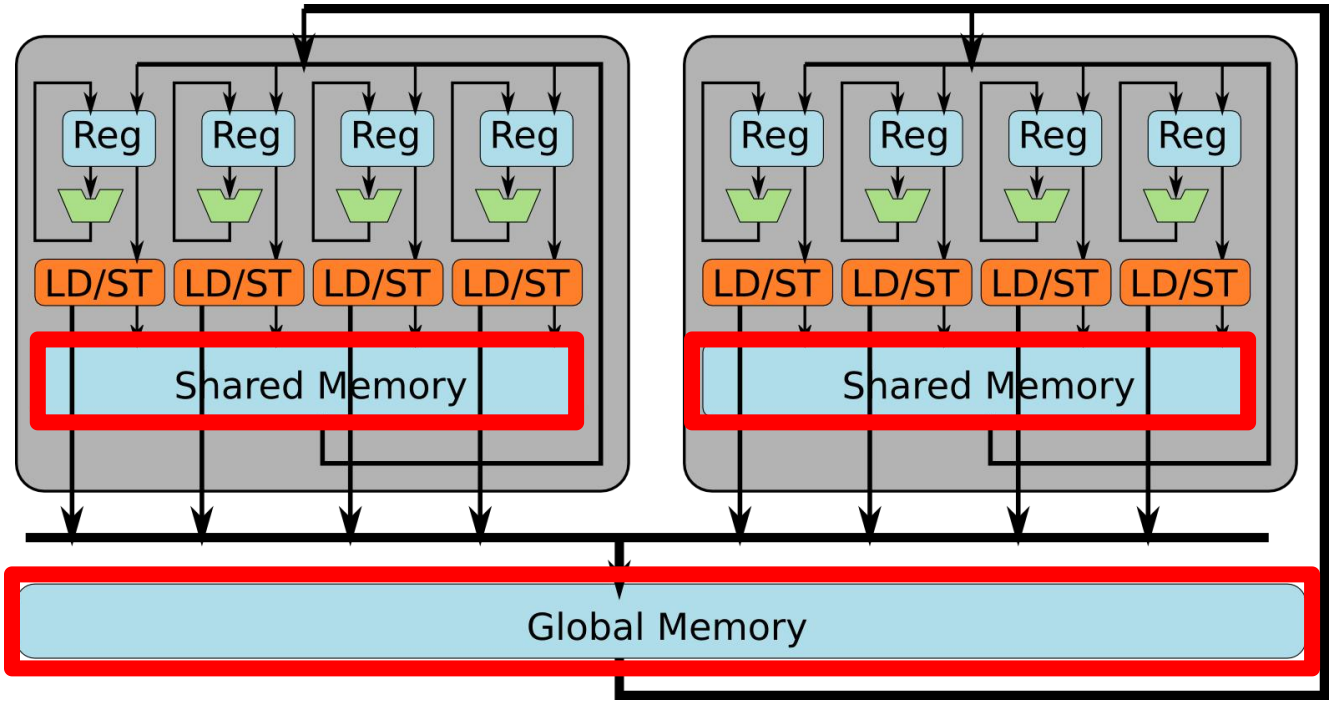
```
mv.u32          %r0, %ctaid.x          // r0 = i = blockIdx.x
mv.u32          %r1, %ntid.x           // r1 = "threads-per-block"
mv.u32          %r2, %tid.x            // r2 = j = threadIdx.x
mad.u32         %r3, %r2, %r1, %r0     // r3 = i * "threads-per-block" + j
ld.global.s32 %r4, [%r3]               // r4 = A[i][j]
add.s32         %r4, %r4, 1            // r4 = r4 + 1
st.global.s32 [%r3], %r4               // A[i][j] = r4
```

利用内存层次结构

内存访问
延迟

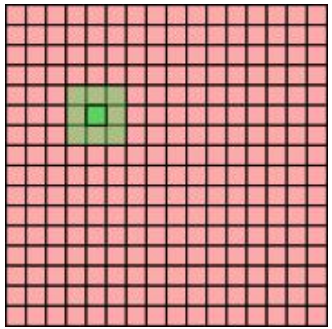
several cycles

100+ cycles



示例: 均值滤波

Average over a 3x3 window for a 16x16 array

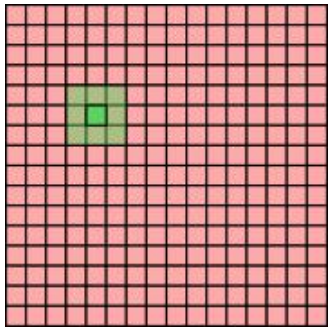


```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    i = threadIdx.y;
    j = threadIdx.x;
    tmp = (A[i-1][j-1] + A[i-1][j] +
              … + A[i+1][i+1] ) / 9;
          A[i][j] = tmp;
}
```

Each thread loads 9 elements from global memory.
It takes hundreds of cycles.

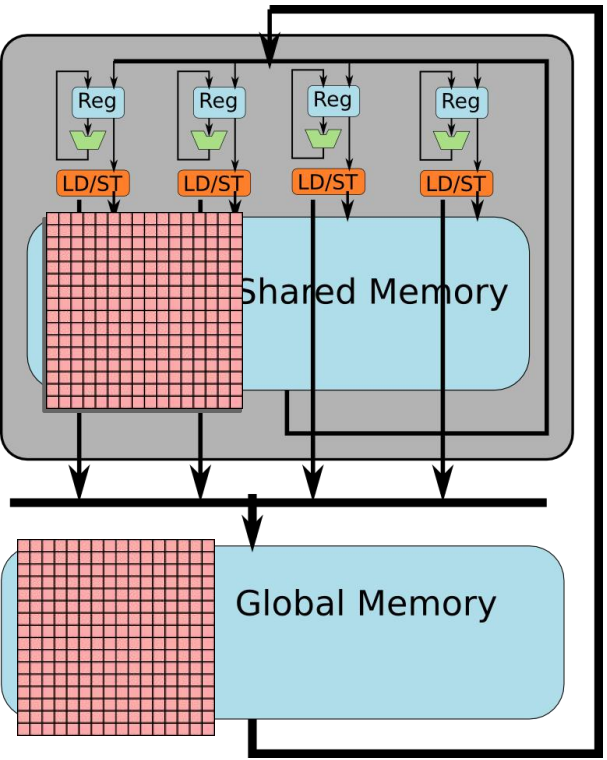
利用共享内存

Average over a 3x3 window for a 16x16 array



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ int smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                    … + smem[i+1][i+1] ) / 9;
}
```

利用共享内存（2）




```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

allocate shared mem

Each thread loads one element from global memory.

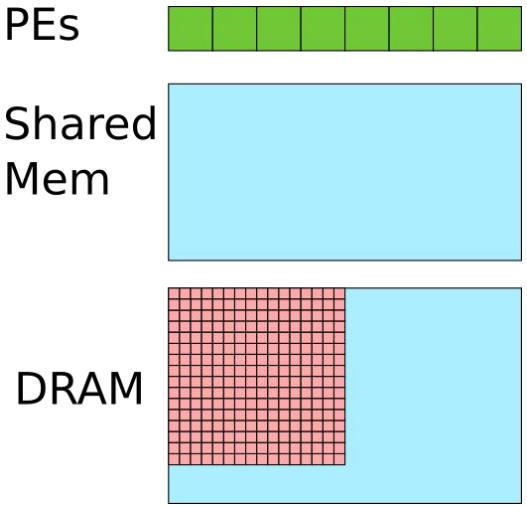
但是，程序出错了！

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

Hazards!

为什么会出错？

Assume 256 threads are scheduled on 8 PEs.

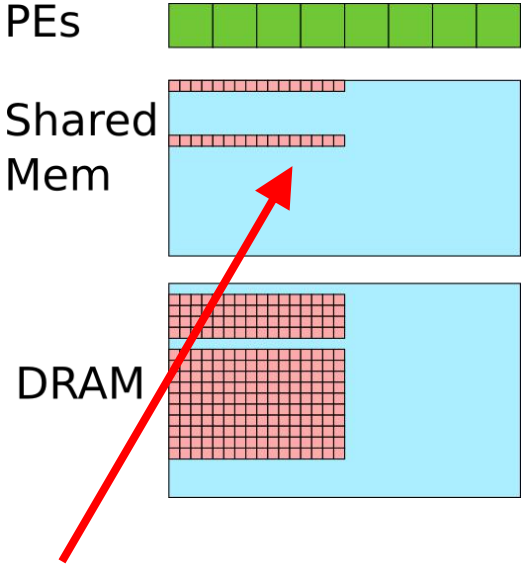



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

Before load instruction

为什么会出错？（2）

Assume 256 threads are scheduled on 8 PEs.

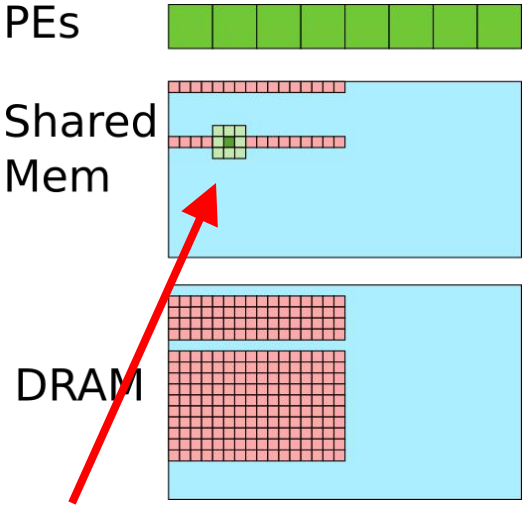


Some threads finish the load earlier than others.

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

为什么会出错？（3）

Assume 256 threads are scheduled on 8 PEs.

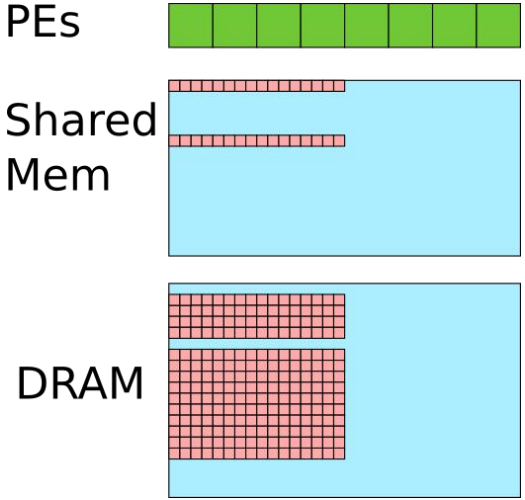


Some elements in the window are not yet loaded by other threads. Error!

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

解决办法？

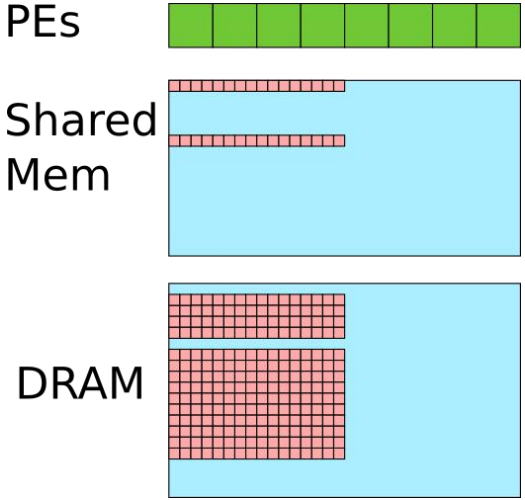
Assume 256 threads are scheduled on 8 PEs.




```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                    … + smem[i+1][i+1] ) / 9;
}
```

使用 "SYNC" 同步障

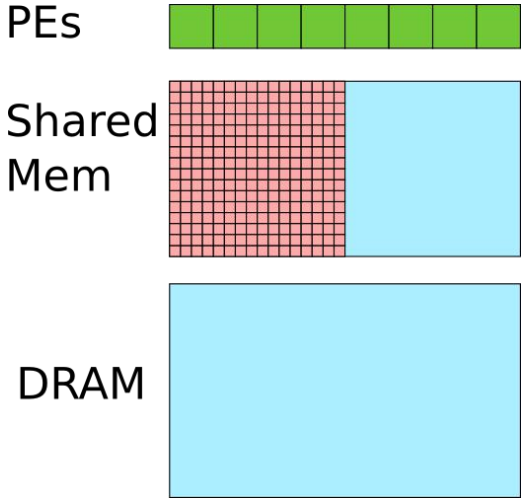
Assume 256 threads are scheduled on 8 PEs.




```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __SYNC();
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

使用 "SYNC" 同步障（2）

Assume 256 threads are scheduled on 8 PEs.

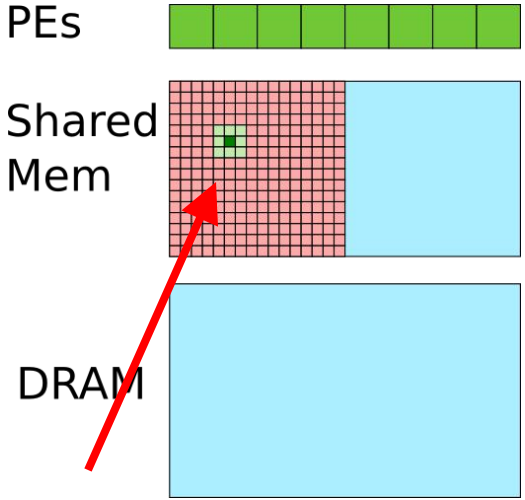



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __SYNC();
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

Wait until all threads hit barrier

使用 "SYNC" 同步障（3）

Assume 256 threads are scheduled on 8 PEs.



All elements in the window are loaded when each thread starts averaging.

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __SYNC();
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

目录

- GPU微处理器简介
- CUDA编程
  - 引言：开发平台、异构计算
  - CUDA编程模型：线程、线程块、线程网格
  - CUDA执行模型：流多处理器和线程束
  - 内存层次结构
  - 编程实例

CUDA执行模型：流多处理器

- SP：GPU内核上运行kernel函数时，相同的指令序列会被大量称为流处理器（SP）的处理单元同步执行。
  - 每个线程运行在一个SP上，SP称为内核（CUDA Core）

- SM：同一控制单元控制下执行的一组SP称为流多处理器（SM）。
  - GPU可以包含多个SM，每个运行自己的kernel函数。

CUDA执行模型：SM和warp

● **执行模型**会提供一个操作视图，说明如何在特定的计算架构上执行指令。

●CUDA采用单指令多线程（SIMT）架构来管理和执行线程，每32个（大小与硬件相关）线程为一组，被称为线程束（warp）。

- 线程束中的所有线程同时执行相同的指令。
- 每个SM都将分配给它的线程块分到线程束中，然后在可用的硬件资源上调度执行。

讨论：

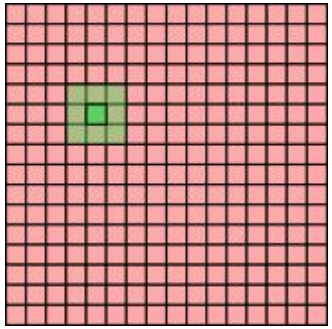
1. 单指令多线程 (SIMT)
2. 共享内存

Q: 显式管理内存的优缺点是什么？

Q: SIMT 与 vector SIMD编程模型的区别是什么？

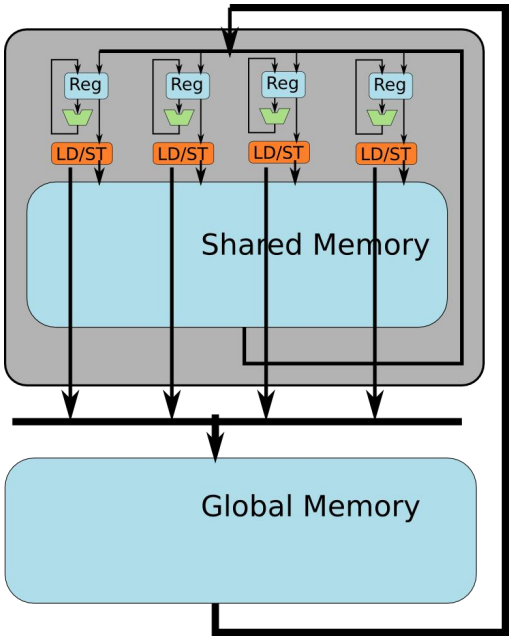
回顾前一个示例

Average over a 3x3 window for a 16x16 array



假设vector SIMD 和 SIMT 都有共享内存.

它们的区别?



Vector SIMD v.s. SIMT

```
int A[16][16]; // A in global memory
__shared__ int B[16][16]; // B in shared mem
for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){
      movups xmm0, [ &A[i][j] ]
      movups [ &B[i][j] ], xmm0 }}
for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){
      addps xmm1, [ &B[i-1][j-1] ]
      addps xmm1, [ &B[i-1][j] ]
      … divps xmm1, 9 }}
for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){
      addps [ &A[i][j] ], xmm1 }}
```

(1)

(2)

(3)

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
   __shared__ int smem[16][16];
   i = threadIdx.y;
   j = threadIdx.x;
   smem[i][j] = A[i][j]; // load to smem
   __sync(); // threads wait at barrier
   A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
      … + smem[i+1][i+1] ) / 9;
}
```

(1)

(2)

(3)

(1) load to shared mem
(2) compute
(3) store to global mem

Vector SIMD v.s. SIMT

```
int A[16][16]; // A in global memory
__shared__ int B[16][16]; // B in shared mem

for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){ <-- (a)
      movups xmm0, [ &A[i][j] ]
      movups [ &B[i][j] ], xmm0 }}
for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){
      addps xmm1, [ &B[i-1][j-1] ]
      addps xmm1, [ &B[i-1][j] ]
      … divps xmm1, 9 }}
for(i=0;i<16;i++){
   for(j=0;j<16;j+=4){
      addps [ &A[i][j] ], xmm1 }}
```

(c)

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
   __shared__ smem[16][16];
   i = threadIdx.y;
   j = threadIdx.x;
   smem[i][j] = A[i][j]; // load to smem
   __sync(); // threads wait at barrier <-- (d)
   A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
             … + smem[i+1][i+1] ) / 9;
}
```

(b)

(a) HW vector width **explicit** to programmer
(b) HW vector width **transparent** to programmers
(c) each vector executed by all PEs in **lock step**
(d) threads executed **out of order**, need explicit sync

SIMT vs. SIMD

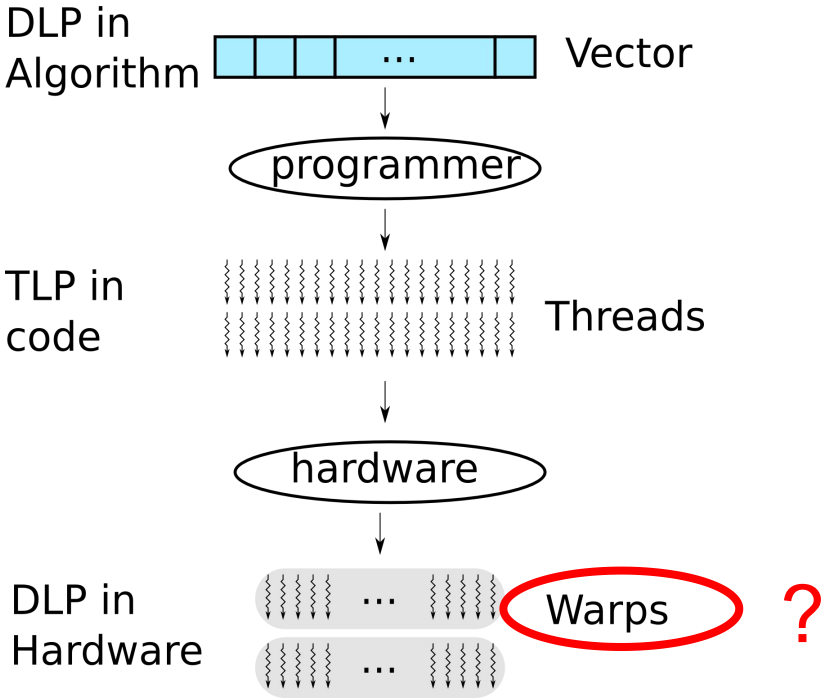
- 相同点：两者都是将相同的指令广播给多个执行单元来实现并行。
- 不同点：
  - SIMD要求同一个向量中的所有元素要在一个统一的同步组中一起执行；
  - SIMT允许属于同一线程束的多个线程独立执行。

- SIMT包含3个SIMD不具备的关键特征：
  - 每个线程都有自己的指令地址计数器
  - 每个线程都有自己的寄存器状态
  - 每个线程可以有一个独立的执行路径

SIMT ： SM和线程块

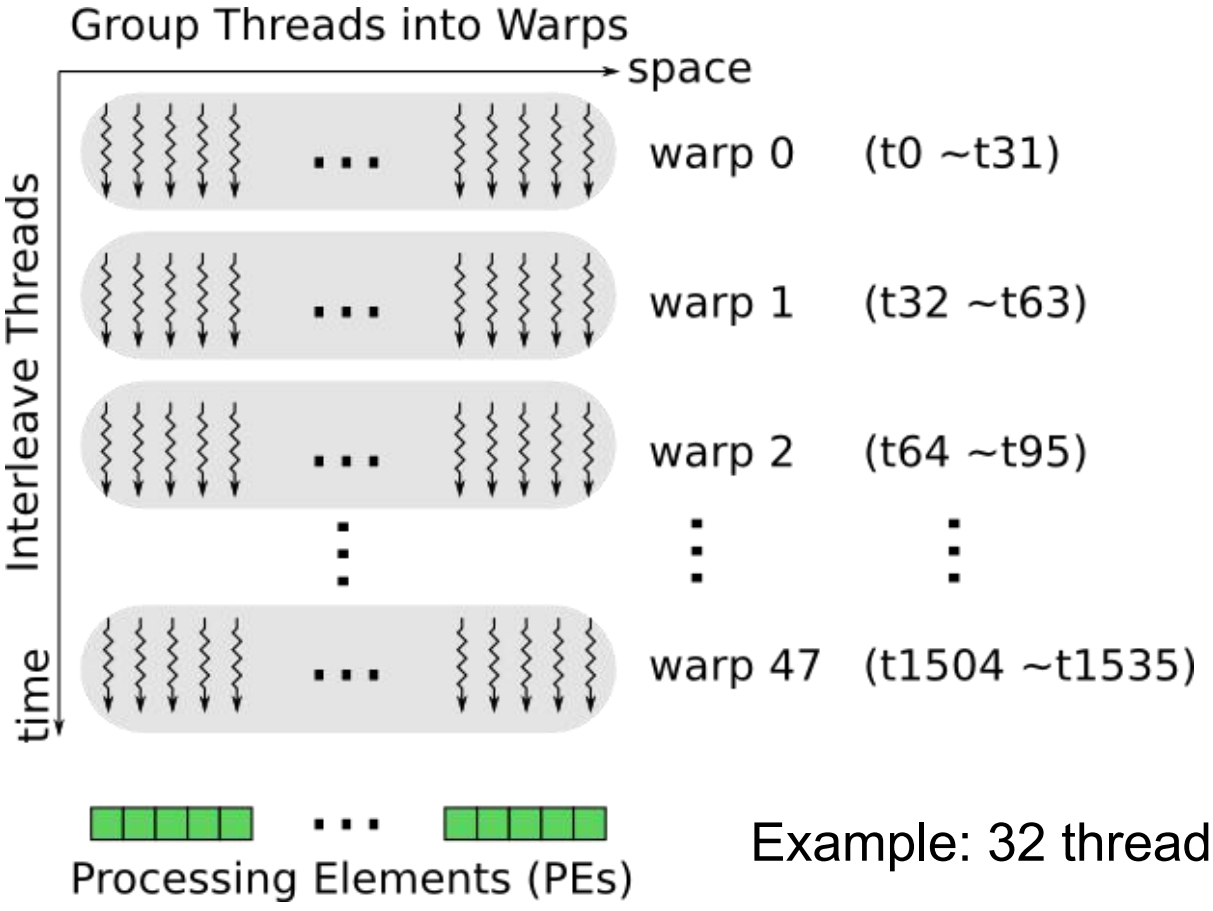
- 一个线程块只能在一个SM上被调度
- 一个SM可以容纳多个线程块
- 寄存器和共享内存是SM中的稀缺资源
- 线程块中的不同线程可能会以不同的速度前进。

回顾：数据级并行到线程级并行



Programmers convert data level parallelism (DLP) into thread level parallelism (TLP).

硬件（HW）将线程分为不同的线程束（ Warp）



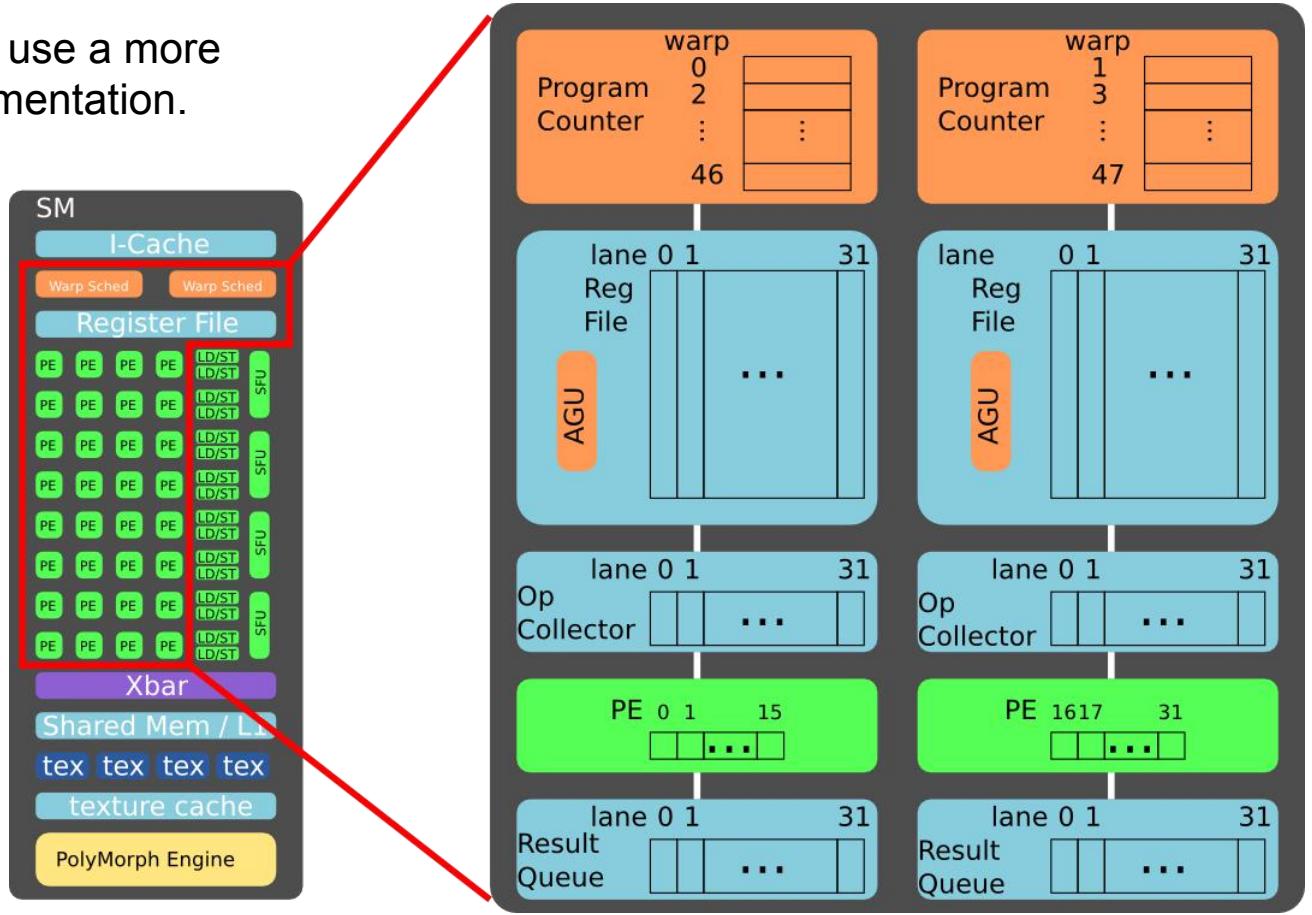
Example: 32 threads per warp

线程和线程束的执行

举例

举例：硬件结构

Note: NVIDIA may use a more complicated implementation.

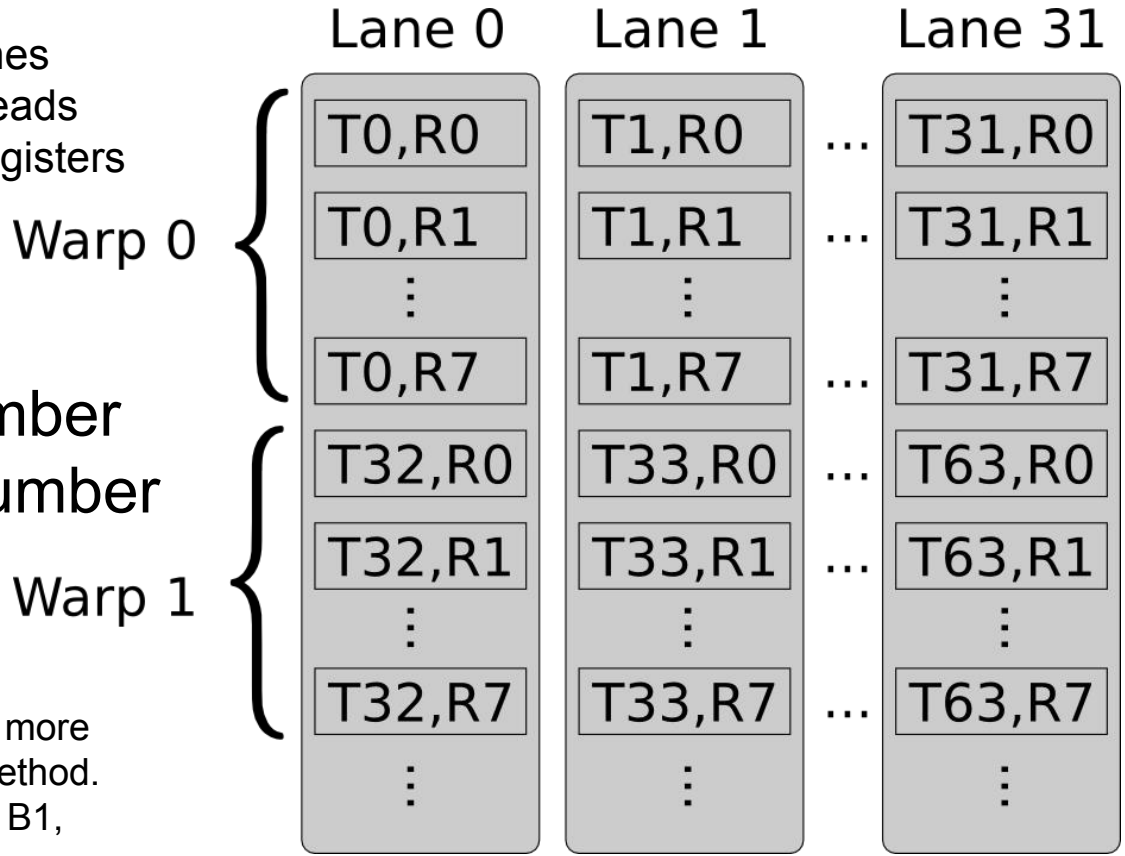


举例：寄存器分配

Assumption:
register file has 32 lanes
each warp has 32 threads
each thread uses 8 registers

Acronym:
“T”: thread number
“R”: register number

Note: NVIDIA may use a more complicated allocation method. See patent: US 7634621 B1, “Register file allocation”.



举例：线程束

Address : Instruction

0x0004 : add r0, r1, r2

0x0008 : sub r3, r4, r5

假设:

两个数据通道

warp 0 在左边的数据通道

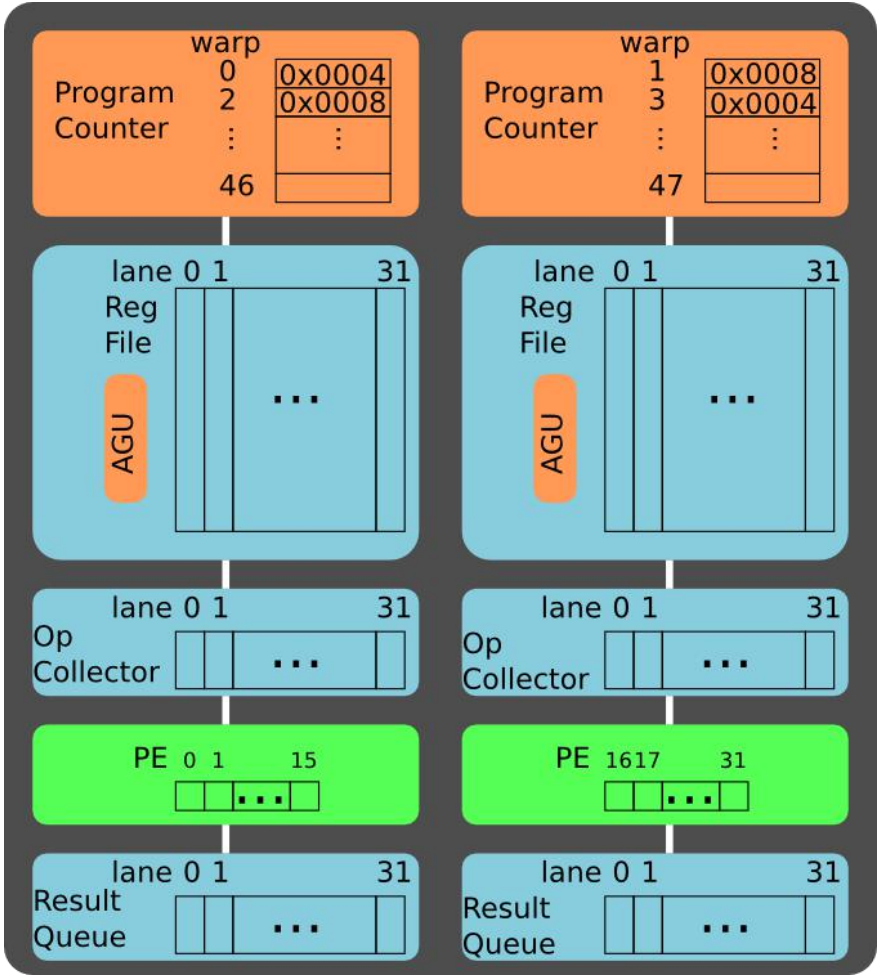
warp 1 在右边的数据通道

术语:

“**AGU**”: address generation unit

“**r**”: register in a thread

“**w**”: warp number



Read Src Op 1

Address : Instruction

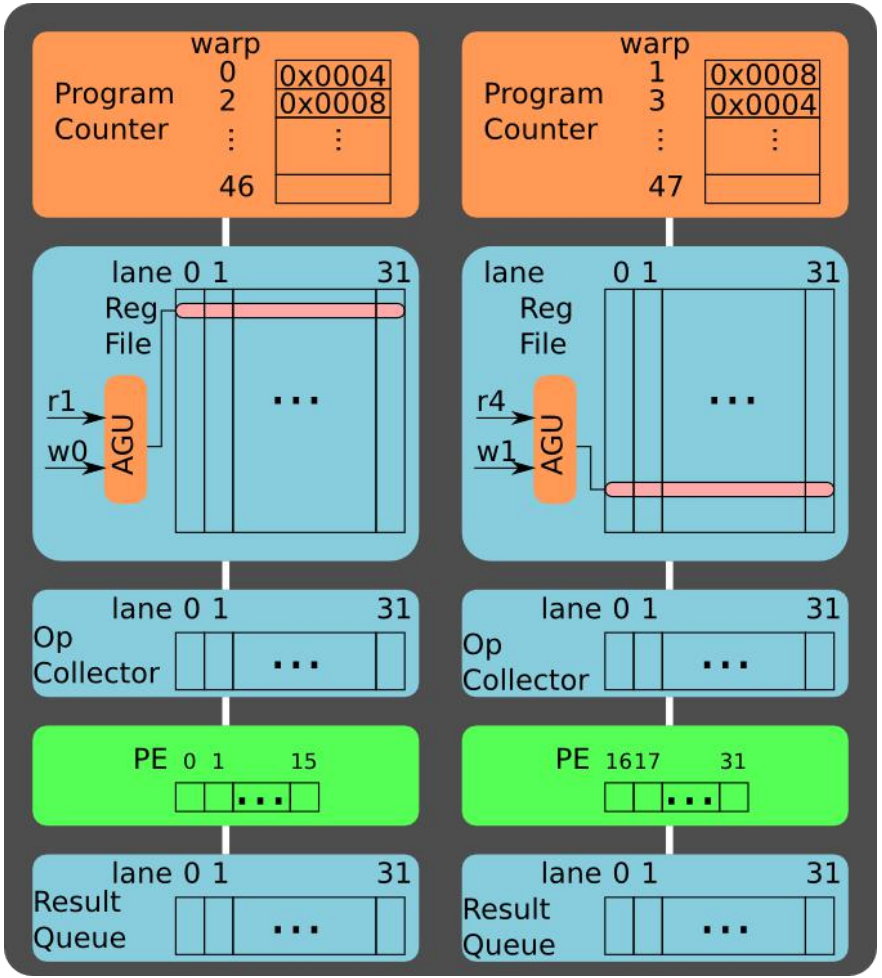
0x0004 : add r0, r1, r2

0x0008 : sub r3, r4, r5

Read source operands:

r1 for warp 0

r4 for warp 1



Buffer Src Op 1

Address : Instruction

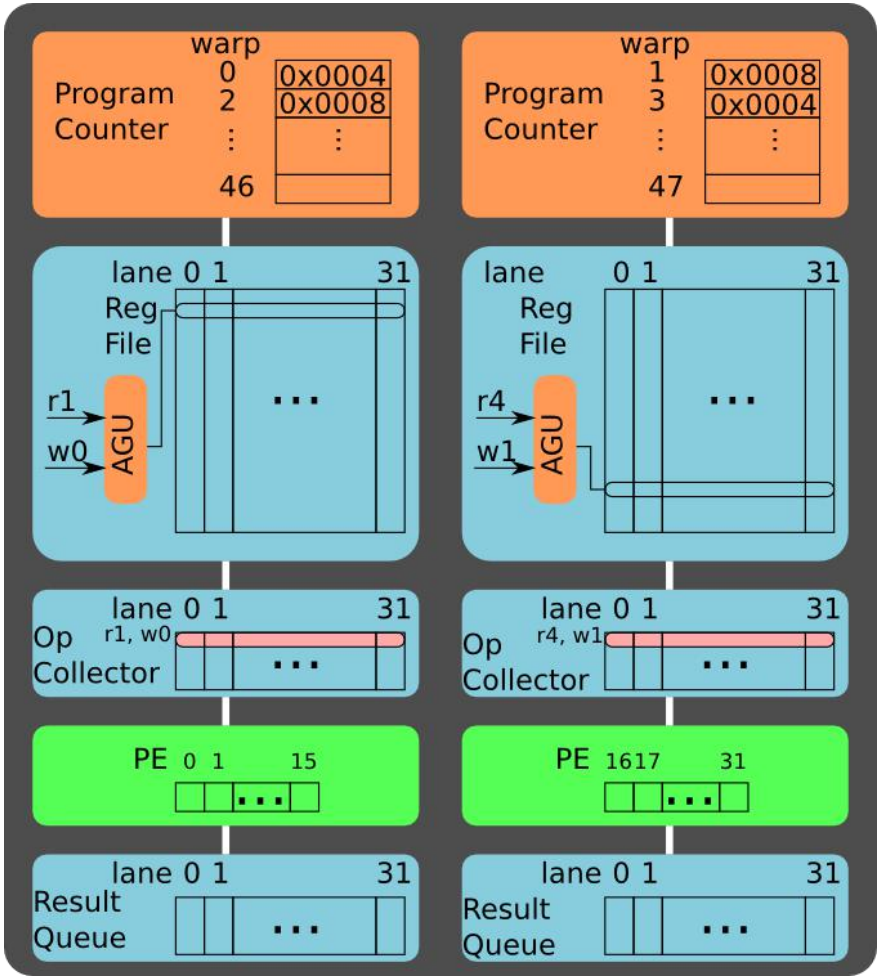
0x0004 : add r0, r1, r2

0x0008 : sub r3, r4, r5

Push ops to op collector:

r1 for warp 0

r4 for warp 1



Read Src Op 2

Address : Instruction

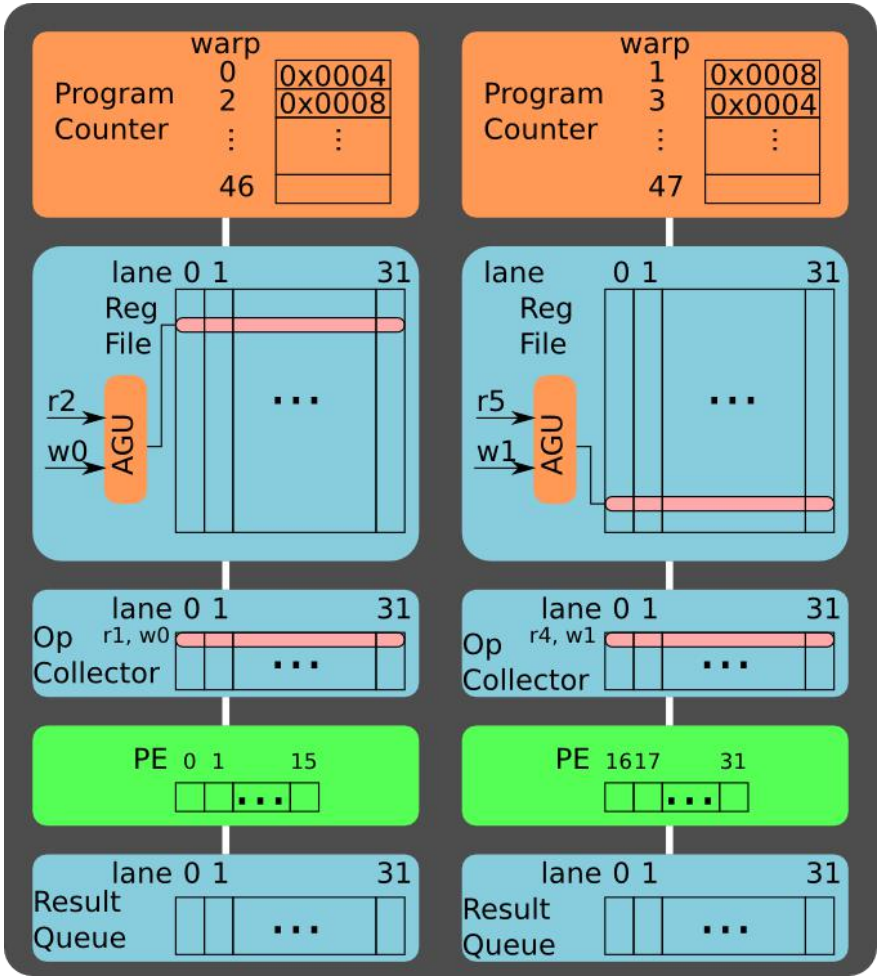
0x0004 : add r0, r1, r2

0x0008 : sub r3, r4, r5

Read source operands:

r2 for warp 0

r5 for warp 1



Buffer Src Op 2

Address : Instruction

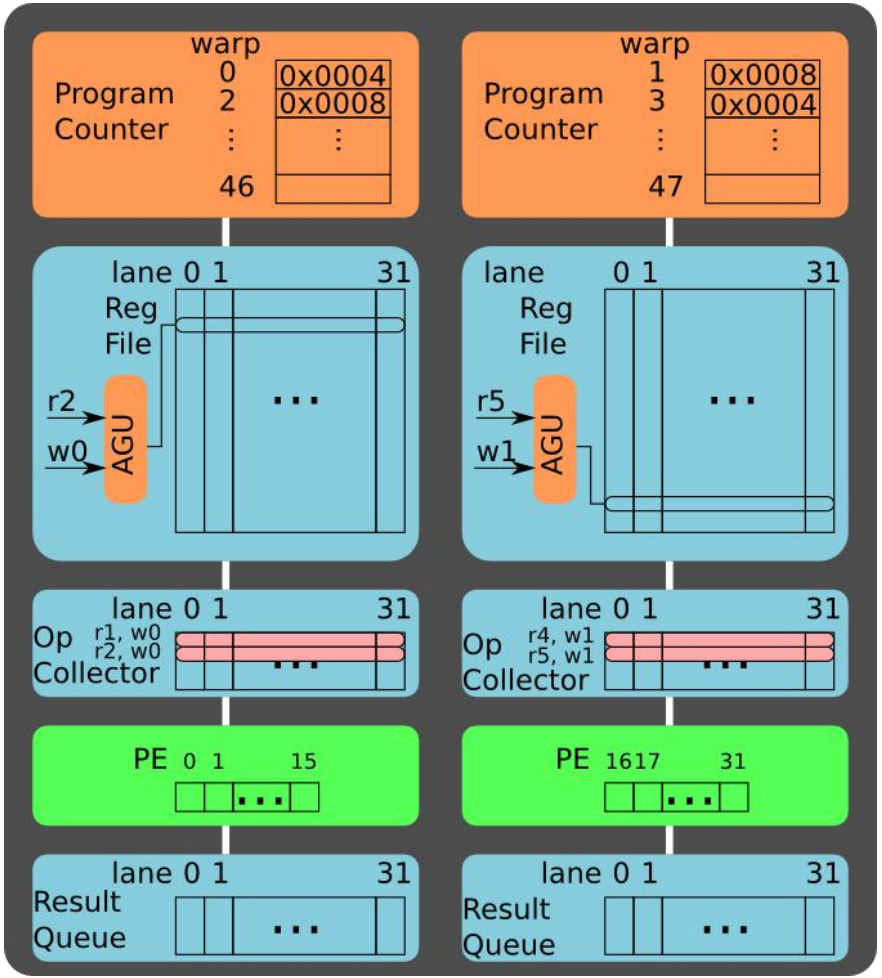
0x0004 : add r0, r1, r2

0x0008 : sub r3, r4, r5

Push ops to op collector:

r2 for warp 0

r5 for warp 1

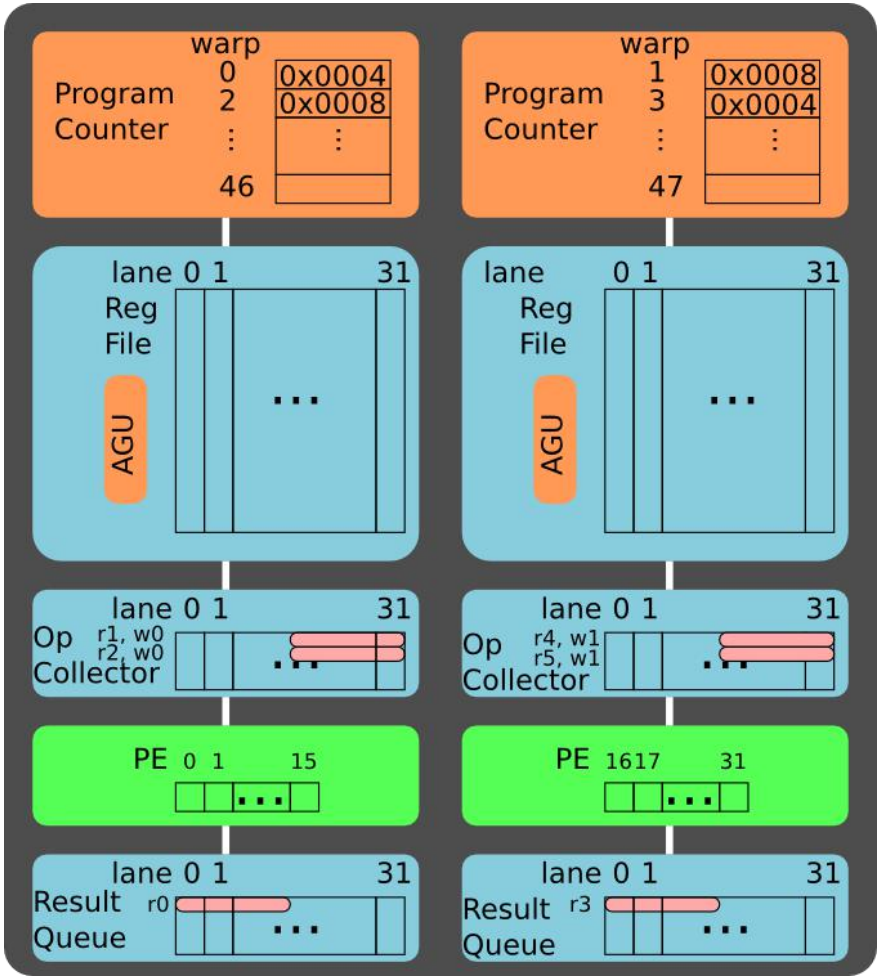


Execute Stage 1

Address : Instruction

0x0004 : add r0, r1, r2

0x0008 : sub r3, r4, r5



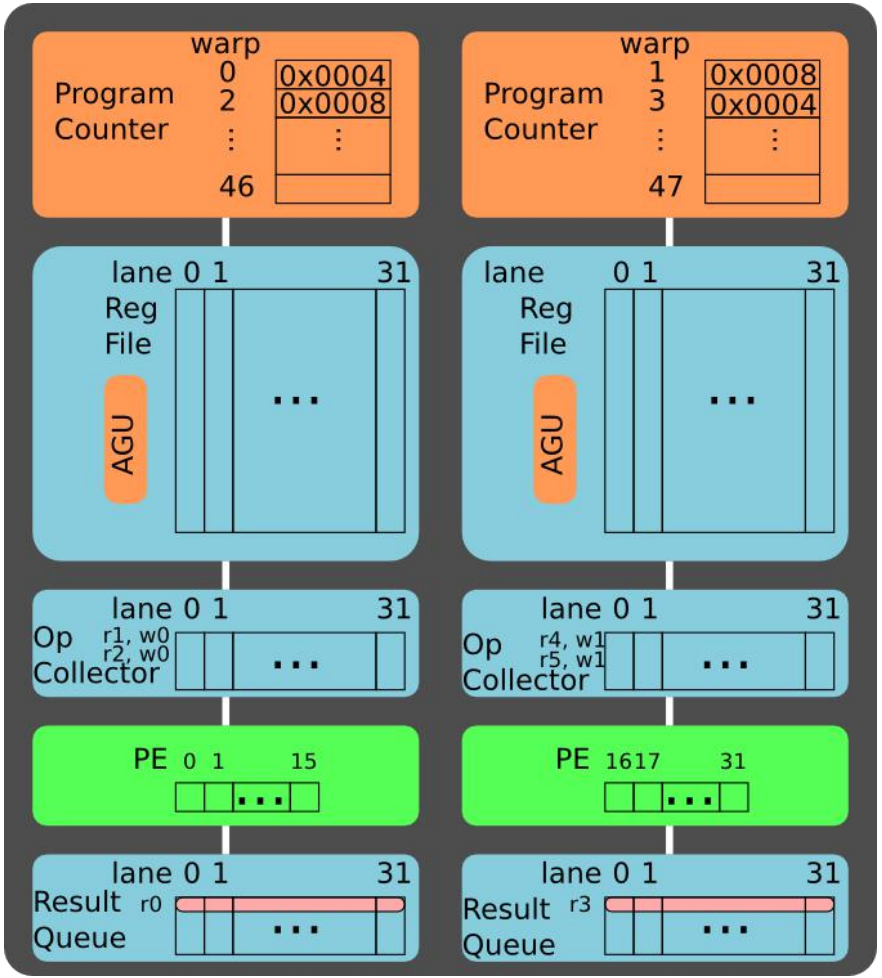
Compute the first 16 threads in the warp.

Execute Stage 2

Address : Instruction

0x0004 : add r0, r1, r2

0x0008 : sub r3, r4, r5



Compute the last 16 threads in the warp.

Write Back

Address : Instruction

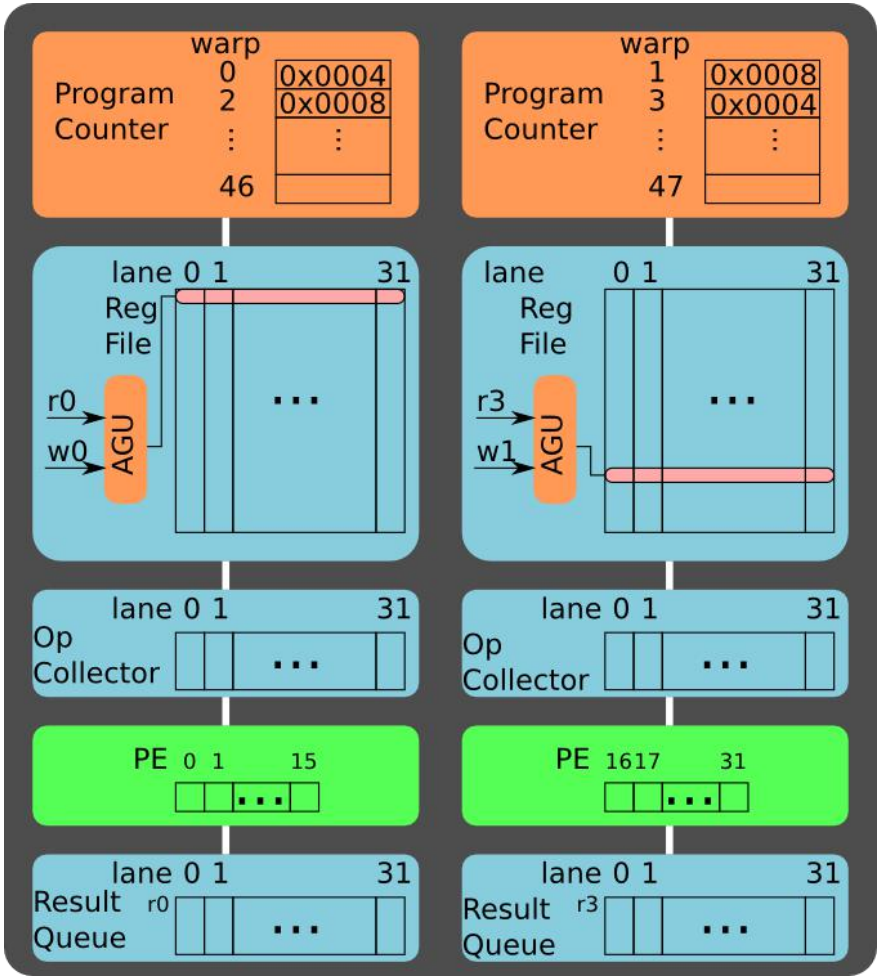
0x0004 : add r0, r1, r2

0x0008 : sub r3, r4, r5

Write back:

r0 for warp 0

r3 for warp 1



目录

- GPU微处理器简介
- CUDA编程
  - 引言：开发平台、异构计算
  - CUDA编程模型：线程、线程块、线程网格
  - CUDA执行模型：流多处理器和线程束
  - 内存层次结构
  - 编程实例

CUDA存储模型

- 对程序员来说，一般有两类存储器：
  - 可编程的：显式控制哪些数据存放在可编程内存中；
  - 不可编程的：程序员不能决定数据的存放位置，程序将自动生成存放位置以获得良好性能。（如CPU缓存）

- CUDA内存模型有多种可编程内存
  - 寄存器
  - 共享内存
  - 本地内存
  - 常量内存
  - 纹理内存
  - 全局内存

CUDA存储模型-寄存器

- 寄存器是GPU上运行速度最快的存储空间。
- 寄存器变量对于每个线程都是私有的。
- 寄存器变量与核函数的生命周期相同。
- Fermi GPU每个线程最多有63个寄存器，Kepler GPU每个线程最多255个寄存器
- 一个核函数使用了超过硬件限制数量的寄存器，则会用本地内存代替。

CUDA存储模型-本地内存、共享内存

- 本地内存：
  - 核函数中的线程都有自己私有的本地内存。

- 共享内存：
  - 线程块有自己的共享内存，对同一线程块中的所有线程都可见，是线程间相互通信的基本方式。
  - 共享内存是片上内存，类似CPU一级缓存，但可编程。
  - SM中的一级缓存和共享内存共同使用64KB的片上内存，可静态划分或运行时动态配置。

CUDA存储模型-常量内存、纹理内存、全局内存

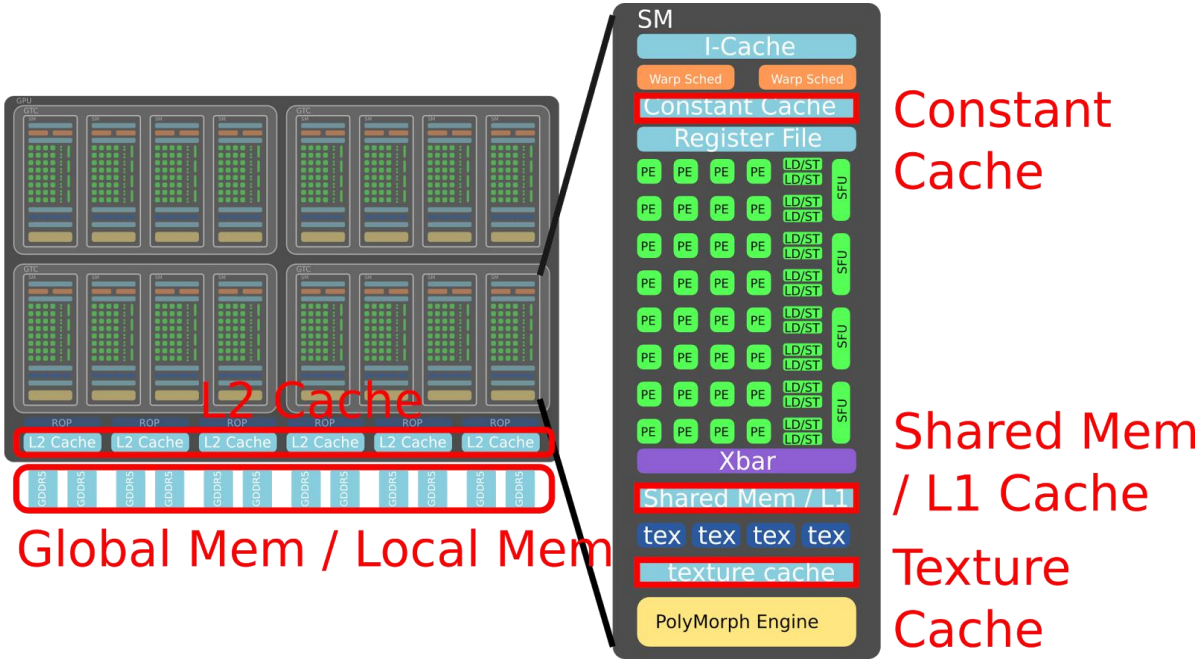
- 常量内存：所有线程只读访问
  - 驻留在设备内存中，并在每个SM专用的常量缓存中缓存。
  - 常量内存是静态声明的，并对同一编译单元中的所有核函数可见。
- 纹理内存：所有线程只读访问
  - 驻留在设备内存中，并在每个SM的只读缓存中缓存。
  - 是一种通过指定的只读缓存访问的全局内存。
- 全局内存：所有线程都可以访问
  - 是GPU中最大、延迟最高并且最常使用的内存。
  - 全局内存变量可以被静态声明或动态声明。
  - 全局内存常驻于设备内存中，可通过内存事务进行访问。

CUDA存储模型-GPU缓存

- GPU缓存是不可编程的内存。
  - 一级缓存（每个SM有一个）
  - 二级缓存（所有SM共享一个）
  - 只读常量缓存
  - 只读纹理缓存
- 一级和二级缓存存储本地内存、全局内存和寄存器溢出的部分。

内存层次结构

Name	Cache?	Latency (cycle)	Read-only?
全局内存	L1/L2	200~400 (cache miss)	R/W
共享内存	No	1~3	R/W
常量内存	Yes	1~3	Read-only
纹理内存	Yes	~100	Read-only
本地内存	L1/L2	200~400 (cache miss)	R/W



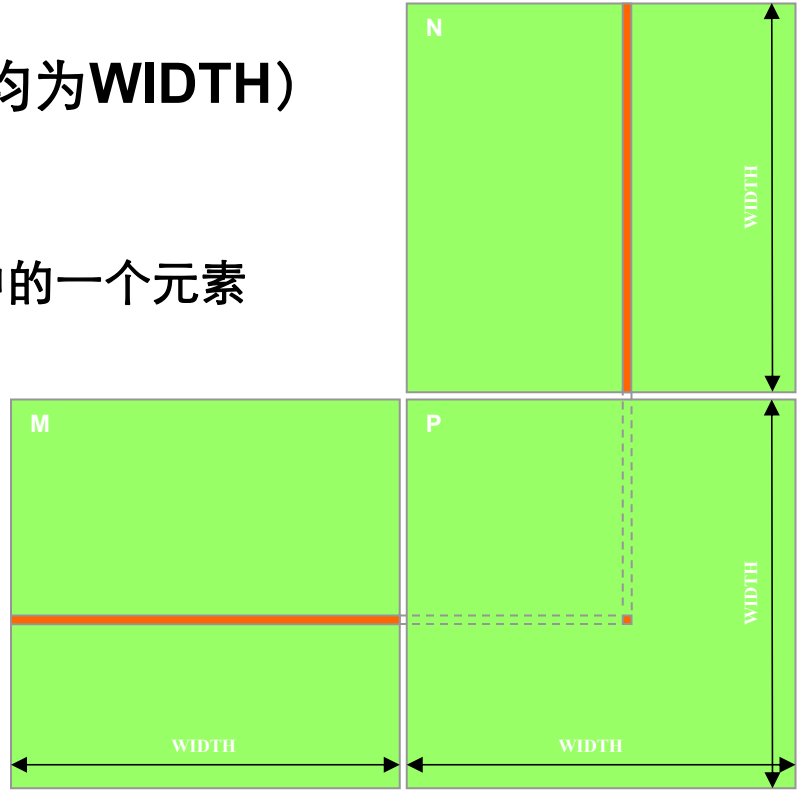
目录

- GPU微处理器简介
- CUDA编程
  - 引言：开发平台、异构计算
  - CUDA编程模型：线程、线程块、线程网格
  - CUDA执行模型：流多处理器和线程束
  - 内存层次结构
  - 编程实例

CUDA程序设计实例——矩阵相乘

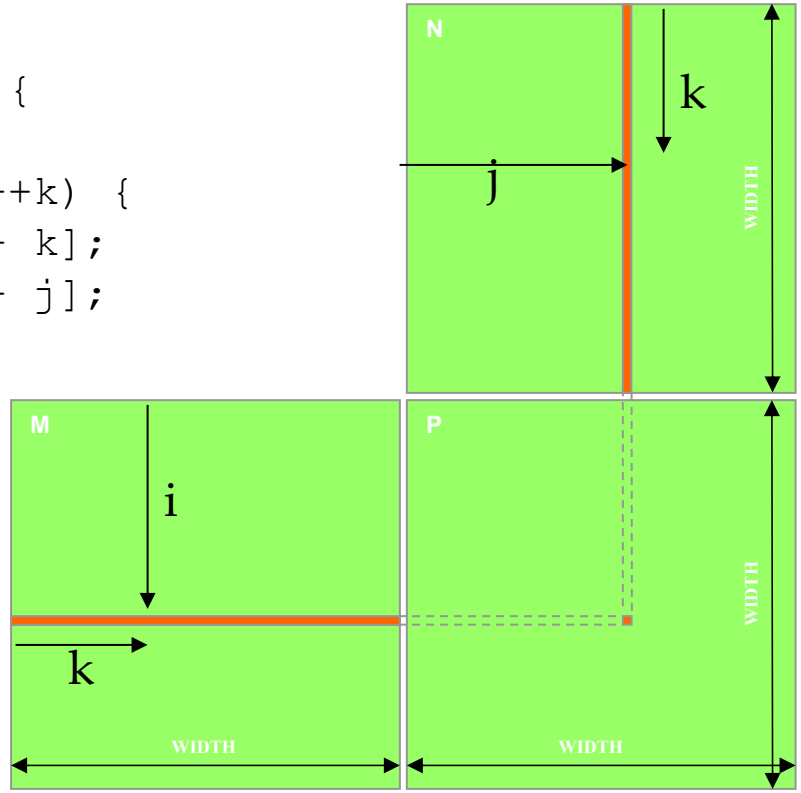
- P = M * N （长宽均为WIDTH）
- 算法

  每个线程计算矩阵P中的一个元素



第一步：CPU实现

```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



第二步：将矩阵数据传给设备（GPU）内存

```
void MatrixMulOnDevice(float* M, float* N, float* P, int
Width)
{
     int size = Width * Width * sizeof(float);
     float* Md, Nd, Pd;
     …
1. // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(&Pd, size);
```

第三步：将计算结果传回主机（CPU）内存

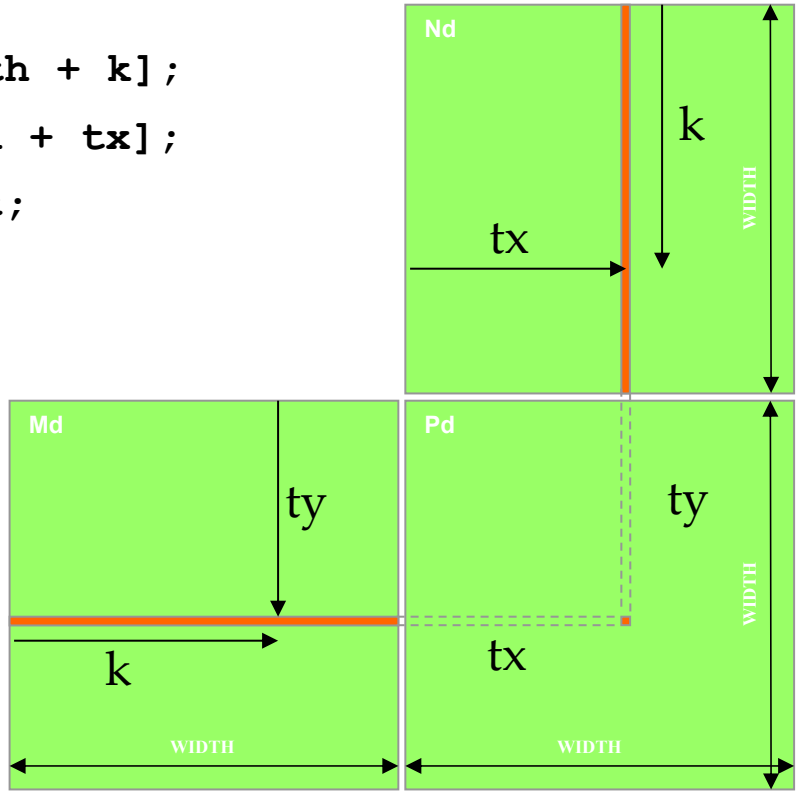
```
2.  // Kernel invocation code – to be shown later
     …

3.  // Read P from the device
     cudaMemcpy(P, Pd, size,
cudaMemcpyDeviceToHost);

     // Free device matrices
     cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

第四步：kernel函数

```
// Matrix multiplication kernel – per thread code


__global__ void MatrixMulKernel(float* Md, float* Nd,
float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;


    // Pvalue is used to store the element of the
matrix
    // that is computed by the thread
    float Pvalue = 0;
```

第四步：kernel函数（续）

```
    for (int k = 0; k < Width; ++k)  {
          float Melement = Md[ty * Width + k];
          float Nelement = Nd[k * Width + tx];
          Pvalue += Melement * Nelement;
    }
    Pd[ty * Width + tx] = Pvalue;
}
```



第五步：调用kernel函数

```
2.  // Kernel invocation code
    // Setup the execution configuration
    dim3 dimBlock(Width, Width);
    dim3 dimGrid(1, 1);

    // Launch the device computation threads!
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd,
Pd);
```

实例总结：

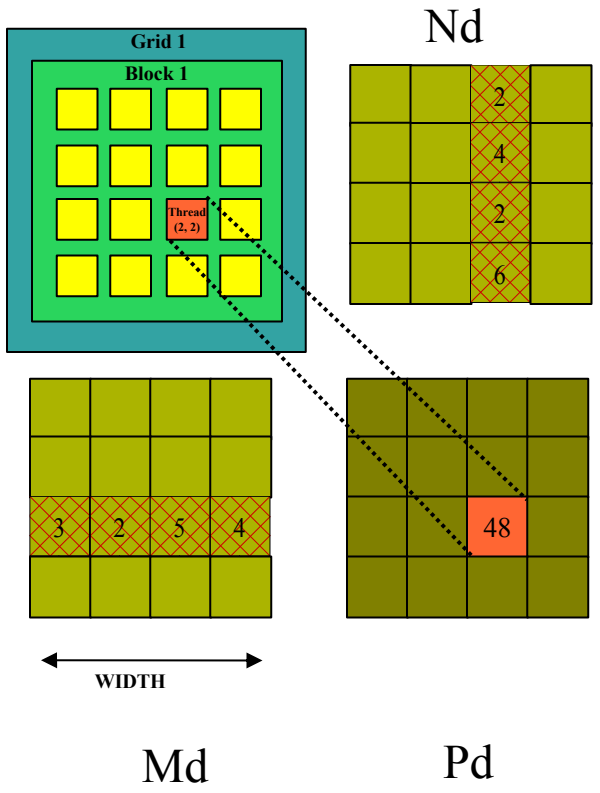
- **一个线程block计算Pd**
  - 每个线程计算Pd的一个元素
- **每个线程：**

  读Md矩阵的一行

  读Nd矩阵的一列

  为每对Md和Nd元素执行一次乘法和加法；

  计算次数和片外访存次数比率接近1:1（不是很高）
- **矩阵规模**受限于每个block允许的thread数目



课程小结

- GPU微处理器简介

- CUDA编程
  - CUDA编程模型：线程、线程块、线程网格
  - CUDA执行模型：流多处理器和线程束
  - 内存层次结构

推荐网站和读物

- Gerassimos Barlas著，张云泉等译，多核与GPU编程 工具、方法及实践，机械工业出版社，2017
- John Cheng等著，颜成钢等译，CUDA C编程权威指南，机械工业出版社，2017

下一讲

- 大规模数据处理及Map/Reduce编程