CUDA编程(1)

周　斌 @ NVIDIA & USTC 2015年7月

致谢

- 某些幻灯片来自 David Kirk 和 Wen-mei Hwu’s UIUC 课件
- 大部分幻灯片来自 Patrick Cozzi University of Pennsylvania CIS 565

GPU 架构概览

- GPU 特别适用于
  - 密集计算，高度可并行计算
  - 图形学
- 晶体管主要被用于:
  - 执行计算
  - 而不是:
    - 缓存数据
    - 控制指令流

GPU 架构概览

晶体管用途

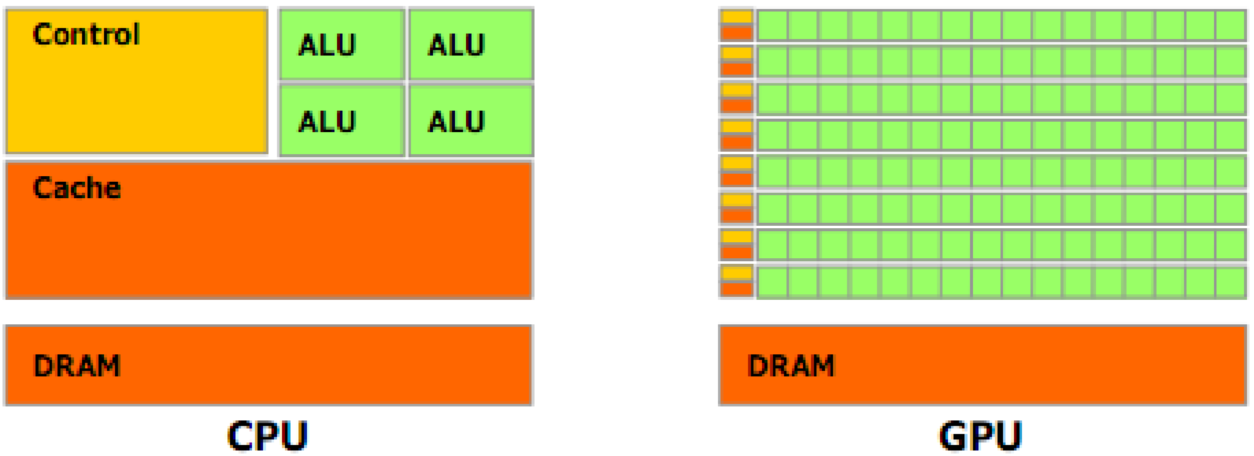


Figure 1-2. The GPU Devotes More Transistors to Data Processing

让我们开始用CUDA来编程!

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

CUDA 的一些信息

- 层次化线程集合 A hierarchy of thread groups
- 共享存储Shared memories
- 同步Barrier synchronization

CUDA 术语

- *Host* – 即主机端 通常指 CPU
  - 采用ANSI标准C语言编程
- *Device* – 即设备端 通常指 GPU (数据可并行)
  - 采用ANSI标准C的扩展语言编程
- Host 和 Device 拥有各自的存储器
- CUDA 编程
  - 包括主机端和设备端两部分代码

CUDA 术语

- *Kernel* – 数据并行处理函数
  - 通过调用kernel函数在设备端创建轻量级线程
    - 线程由硬件负责创建并调度

- 类似于OpenGL的 *shader*?

CUDA 核函数（Kernels）

- 在N个不同的CUDA线程上并行执行

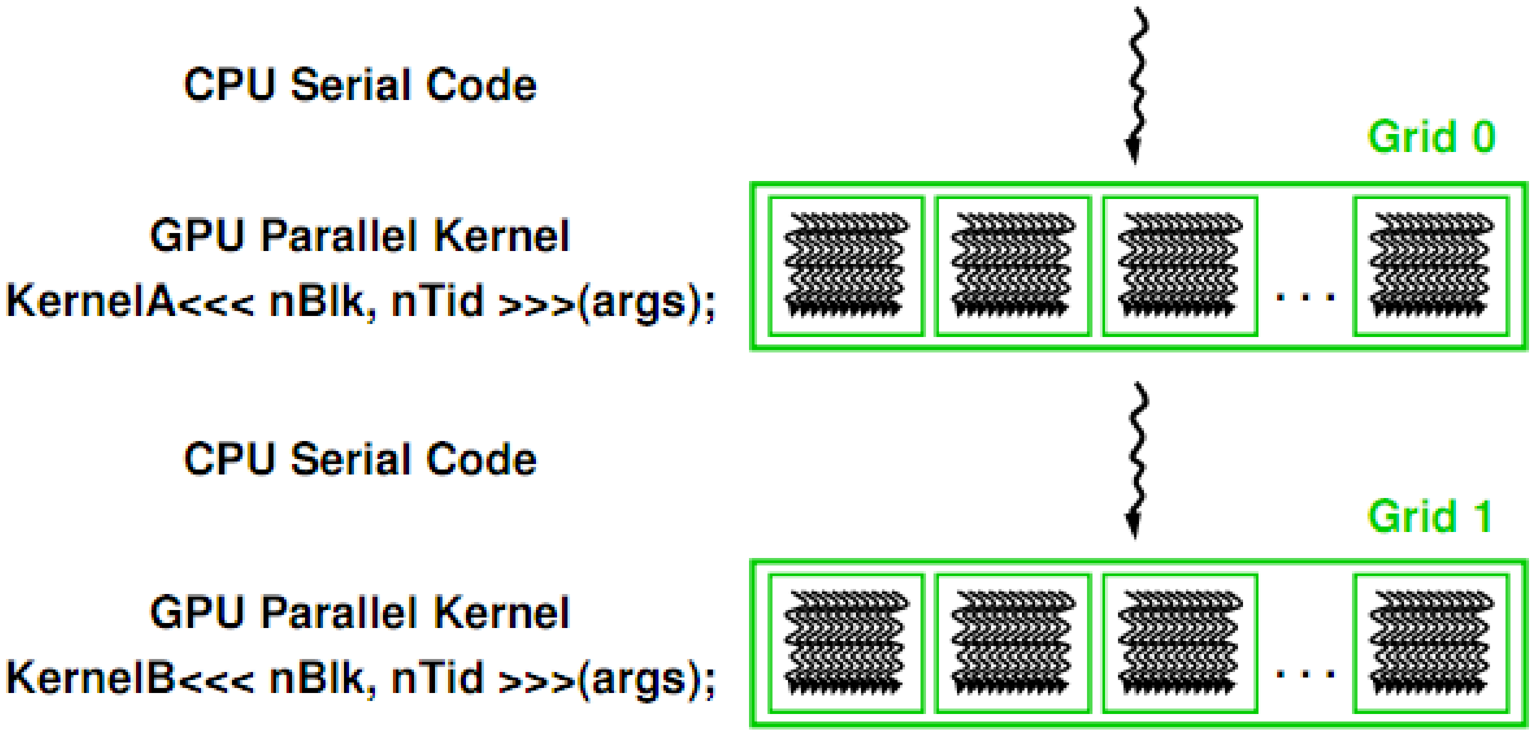
```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

Declaration Specifier

Thread ID

Execution Configuration

CUDA 程序的执行

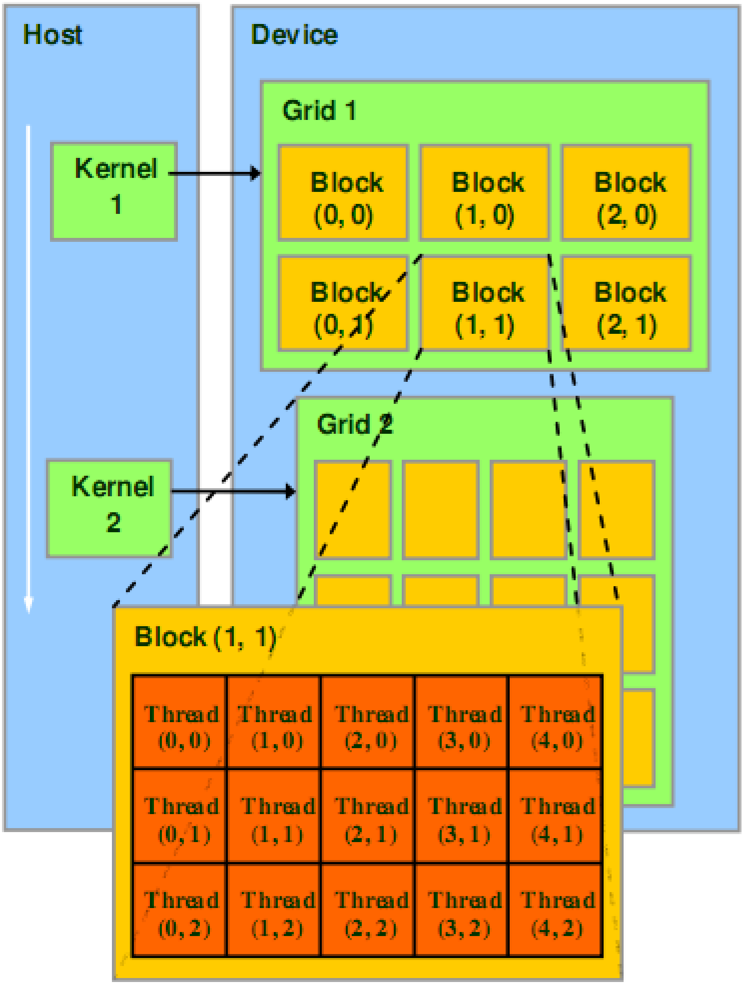

CPU Serial Code
Grid 0
GPU Parallel Kernel
KernelA<<< nBlk, nTid >>>(args);
. . .
CPU Serial Code
Grid 1
GPU Parallel Kernel
KernelB<<< nBlk, nTid >>>(args);
. . .

线程层次 Thread Hierarchies

- *Grid* – 一维或多维线程块(block)
  - 一维 二维 或 三维
- *Block* – 一组线程
  - 一维，二维或三维
  - 一个Grid里面的每个Block的线程数是一样的
  - block内部的每个线程可以：
    - 同步 synchronize
    - 访问共享存储器 shared memory

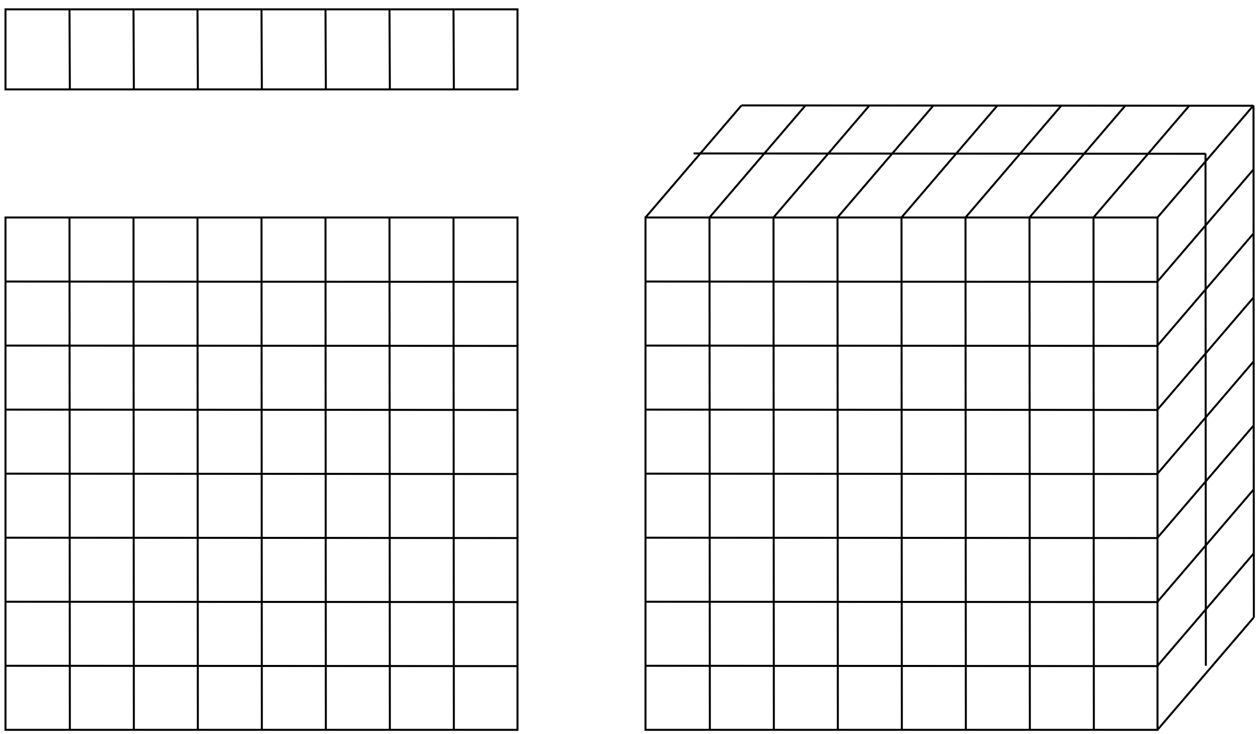
线程层次 Thread Hierarchies

- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



线程层次 Thread Hierarchies

- *Block* – 一维，二维或三维
  - 例如：索引数组，矩阵，体



线程层次 Thread Hierarchies

- *Thread ID*: Scalar thread identifier
- 线程索引: threadIdx

- 一维Block: Thread ID == Thread Index
- 二维Block (D_x, D_y)
  - Thread ID of index $(x, y) == x + y\,D_y$
- 三维Block (D_x, D_y, D_z)
  - Thread ID of index $(x, y, z) == x + y\,D_y + z\,D_x D_y$

线程层次 Thread Hierarchies

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

2D Index

1 Thread Block

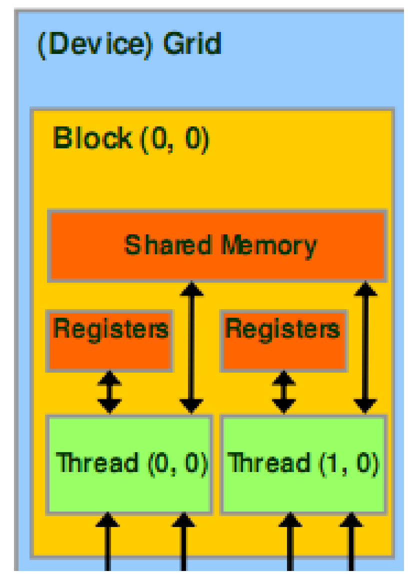
2D Block

线程层次 Thread Hierarchies

- Thread Block线程块
  - 线程的集合
    - G80 and GT200: 多达512 个线程
    - Fermi: 多达1024个线程
  - 位于相同的处理器核(SM)
  - 共享所在核的存储器

线程层次 Thread Hierarchies

- Thread Block 线程块
  - 线程的集合
    - G80 and GT200: 多达512 个线程
    - Fermi: 多达1024个线程
  - 位于相同的处理器核心（SM）
  - 共享所在核心的存储器



线程层次 Thread Hierarchies

- 块索引: blockIdx
- 维度: blockDim
  - 一维或二维或三维

线程层次 Thread Hierarchies

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

16x16
Threads per block

2D Thread Block

线程层次 Thread Hierarchies

- 例如: N = 32
  - 每个块有 16x16 个线程 (跟N无关)
    - threadIdx ([0, 15], [0, 15])
  - Grid里面有 2x2 个线程块Block
    - blockIdx ([0, 1], [0, 1])
    - blockDim = 16

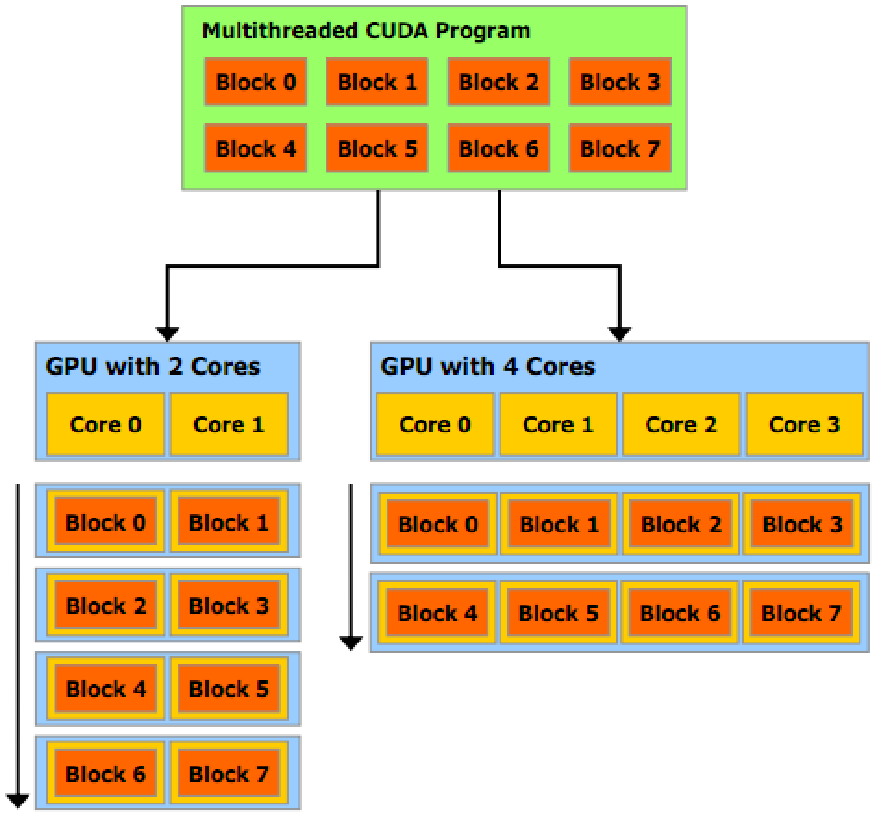
```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

- i = [0, 1] * 16 + [0, 15]

线程层次 Thread Hierarchies

- 线程块之间彼此独立执行
  - 任意顺序：并行或串行
  - 被任意数量的处理器以任意顺序调度
    - 处理器的数量具有可扩展性

线程层次 Thread Hierarchies

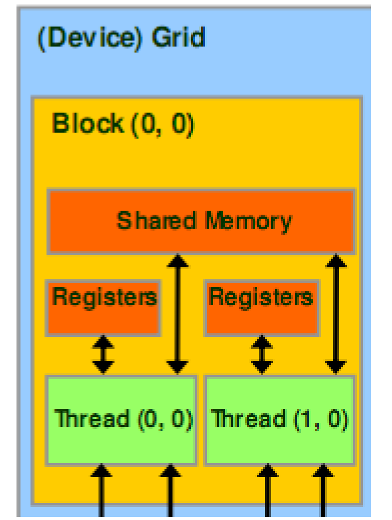


A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

Figure 1-4. Automatic Scalability

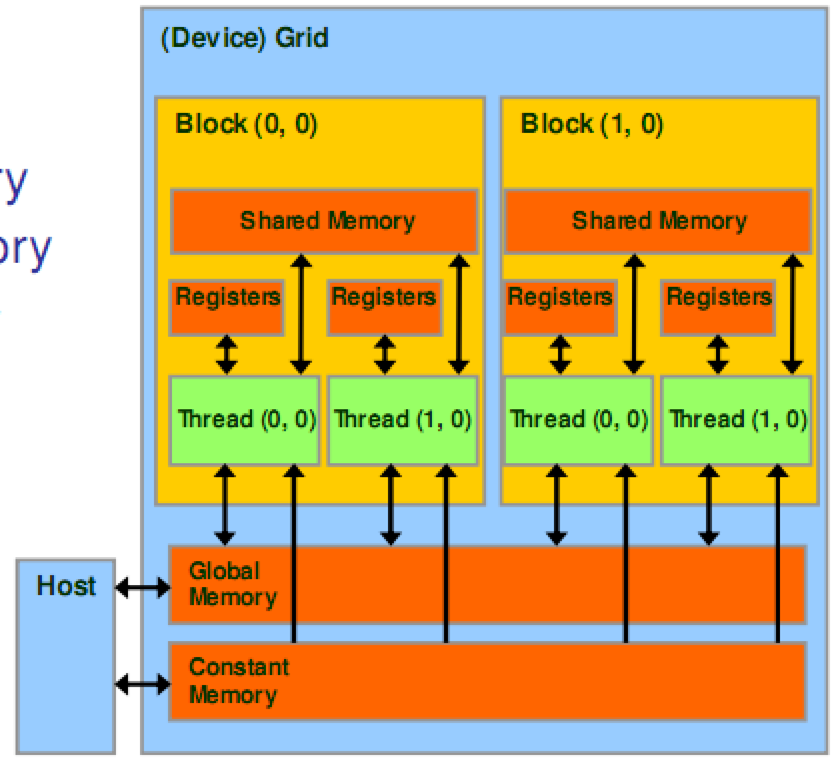
线程层次 Thread Hierarchies

- 一个块内部的线程
  - 共享容量有限的低延迟存储器
  - 同步执行
    - 合并访存
    - __syncThreads()
      - → Barrier – 块内线程一起等待所有线程执行都某处语句
      - → 轻量级



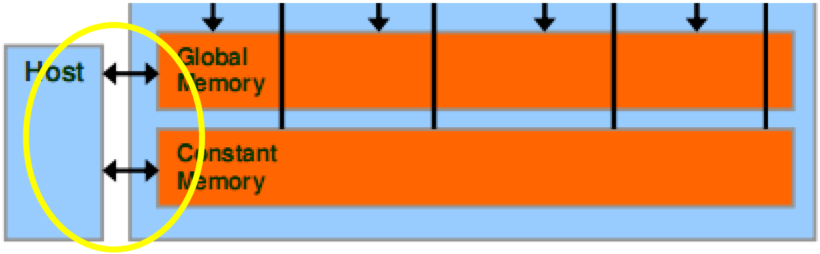
CUDA 内存传输

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - R/W per grid global and constant memories



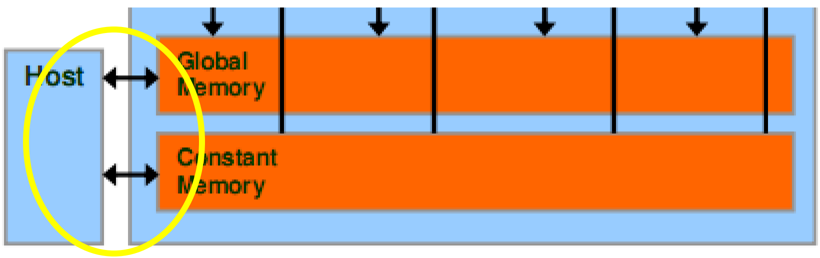
CUDA 内存传输

- Host 可以从 device 往返传输数据
  - *Global* memory 全局存储器
  - *Constant* memory 常量存储器



CUDA 内存传输

- `cudaMalloc()`
  - 在设备端分配 global memory
- `cudaFree()`
  - 释放存储空间



CUDA 内存传输

```
float *Md
int size = Width * Width * sizeof(float);

cudaMalloc((void**)&Md, size);
…
cudaFree(Md);
```

CUDA 内存传输

```
float *Md
int size = Width * Width * sizeof(float);

cudaMalloc((void**)&Md, size);
…
cudaFree(Md);
```

Pointer to device memory

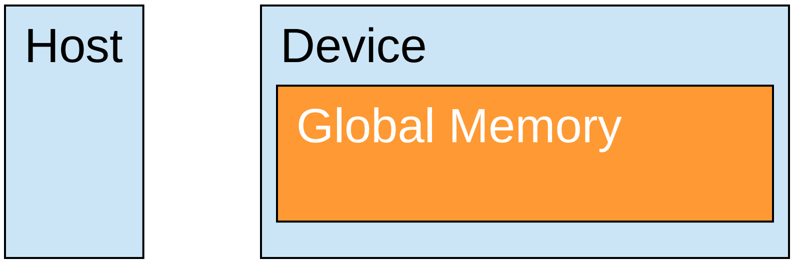
CUDA 内存传输

```
float *Md
int size = Width * Width * sizeof(float);

cudaMalloc((void**)&Md, size);
…
cudaFree(Md);
```

Size in bytes

CUDA 内存传输

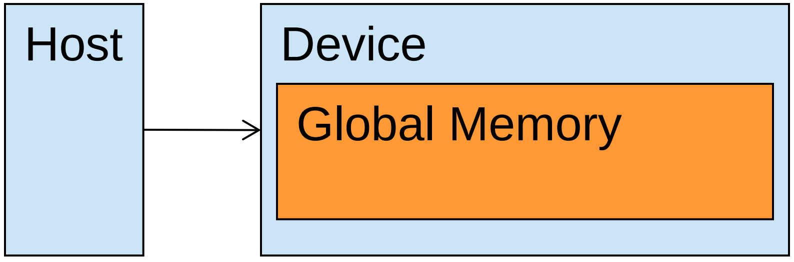
- cudaMemcpy()
  - 内存传输
    - Host to host
    - Host to device
    - Device to host
    - Device to device



CUDA 内存传输

Host to device

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

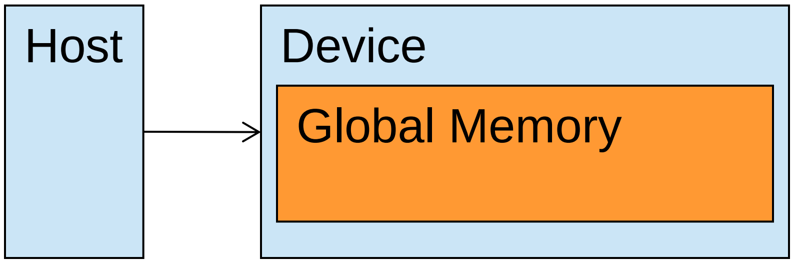


CUDA 内存传输

Destination (device)

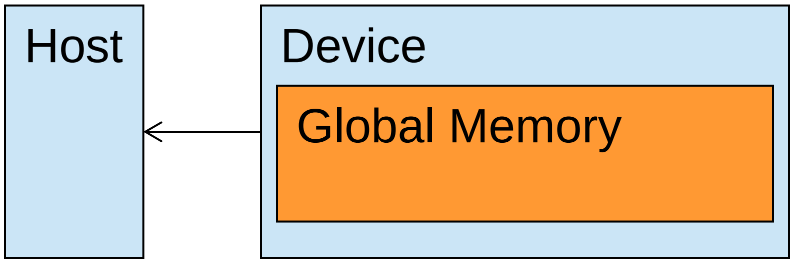
Source (host)

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```



CUDA 内存传输

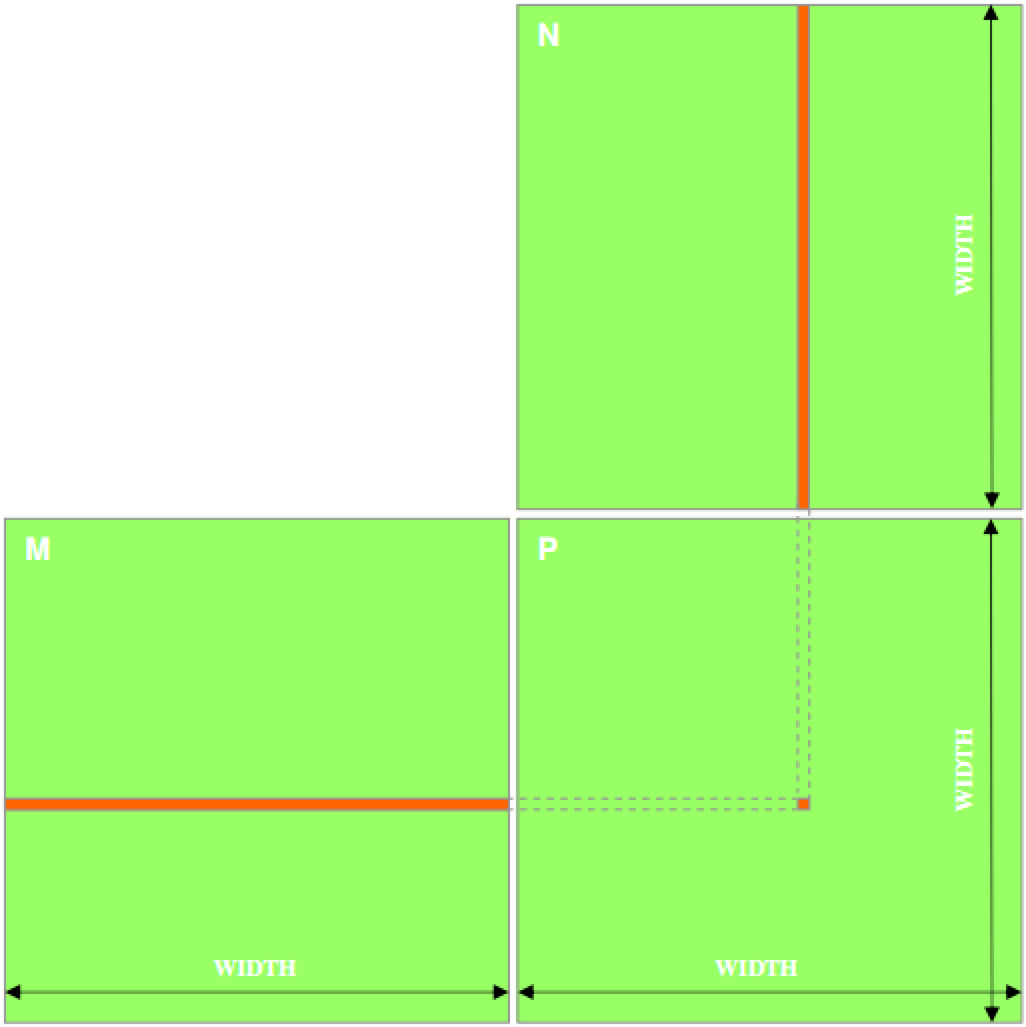
```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```



Matrix Multiply 矩阵相乘算法提示

- 向量
- 点乘
- 行优先或列优先?
- 每次点乘结果输出一个元素

Matrix Multiply矩阵相乘样例



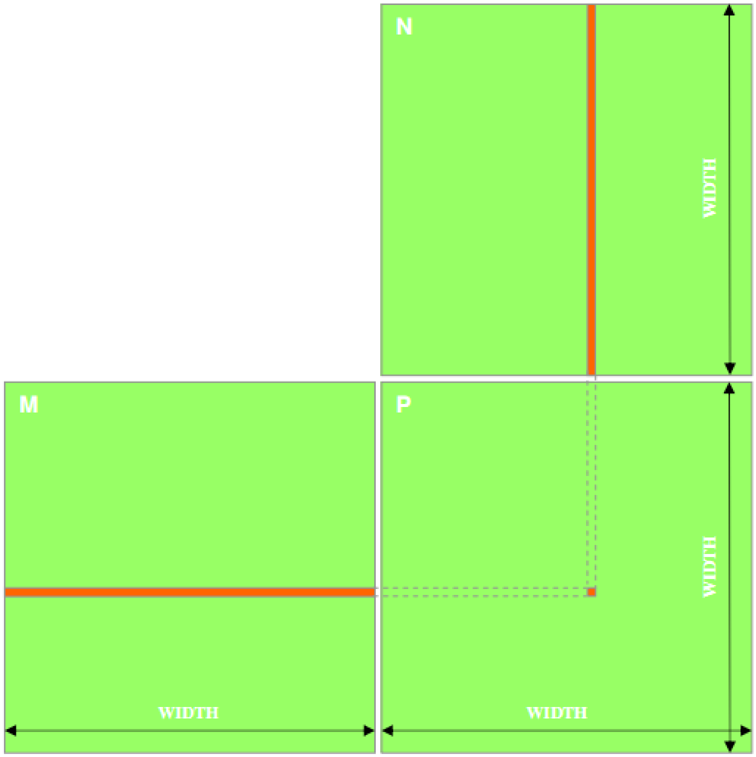
- P = M * N
- 假定 M and N 是方阵

Matrix Multiply矩阵相乘样例

- 1,000 x 1,000矩阵
  - 1,000,000 点乘
    - Each 1,000 multiples and 1,000 adds

Matrix Multiply: CPU 实现

```
void MatrixMulOnHost(float* M, float* N, float* P, int width)
{
  for (int i = 0; i < width; ++i)
    for (int j = 0; j < width; ++j)
    {
      float sum = 0;
      for (int k = 0; k < width; ++k
      {
        float a = M[i * width + k];
        float b = N[k * width + j];
        sum += a * b;
      }
      P[i * width + j] = sum;
    }
}
```



Matrix Multiply: CUDA 算法框架

```
int main(void) {
1.   // Allocate and initialize the matrices M, N, P
     // I/O to read the input matrices M and N
....

2.   // M * N on the device
     MatrixMulOnDevice(M, N, P, width);


3. // I/O to write the output matrix P
   // Free matrices M, N, P
...
return 0;
}
```

Matrix Multiply: CUDA 算法框架

```
int main(void) {
1.    // Allocate and initialize the matrices M, N, P
      // I/O to read the input matrices M and N
....

2.     // M * N on the device
       MatrixMulOnDevice(M, N, P, width);


3.  // I/O to write the output matrix P
    // Free matrices M, N, P
...
return 0;
}
```

Matrix Multiply: CUDA 算法框架

```
int main(void) {
1.    // Allocate and initialize the matrices M, N, P
      // I/O to read the input matrices M and N
....

2.    // M * N on the device
      MatrixMulOnDevice(M, N, P, width);


3.  // I/O to write the output matrix P
    // Free matrices M, N, P
...
return 0;
}
```

Matrix Multiply: CUDA 算法框架

- 第1步
  - 在算法框架中添加 *CUDA memory transfers*

Matrix Multiply：数据传输

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);

1. // Load M and 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);

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);
}
```

分配输入

Matrix Multiply : 数据传输

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);

1. // Load M and 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);
```

分配输出

```
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);
   }
```

Matrix Multiply : 数据传输

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);

1. // Load M and 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);

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);
}
```

Matrix Multiply：数据传输

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);

1. // Load M and 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);

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);
}
```

从device读回

Matrix Multiply : 数据传输

#include “cuda.h”

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);

1. // Load M and 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);

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);
}
```

Matrix Multiply矩阵相乘样例

- 第2步
  - CUDA C编程实现 *kernel*

Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

     for (int k = 0; k < Width; ++k)
     {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
     }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

访问一个matrix, 所以采用二维block

Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

     for (int k = 0; k < Width; ++k)
     {
         float Mdelement = Md[ty * Md.width + k];
         float Ndelement = Nd[k * Nd.width + tx];
         Pvalue += Mdelement * Ndelement;
     }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

每个 kernel 线程计算一个输出

Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Md.width + k];
        float Ndelement = Nd[k * Nd.width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

CPU 版本的2层外循环哪去了?

Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

     for (int k = 0; k < Width; ++k)
     {
         float Mdelement = Md[ty * Md.width + k];
         float Ndelement = Nd[k * Nd.width + tx];
         Pvalue += Mdelement * Ndelement;
     }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```

不需要锁或同步，为什么?

Matrix Multiply矩阵相乘样例

- 第3步
  - CUDA C编程调用 *kernel*

矩阵相乘: 调用 Kernel

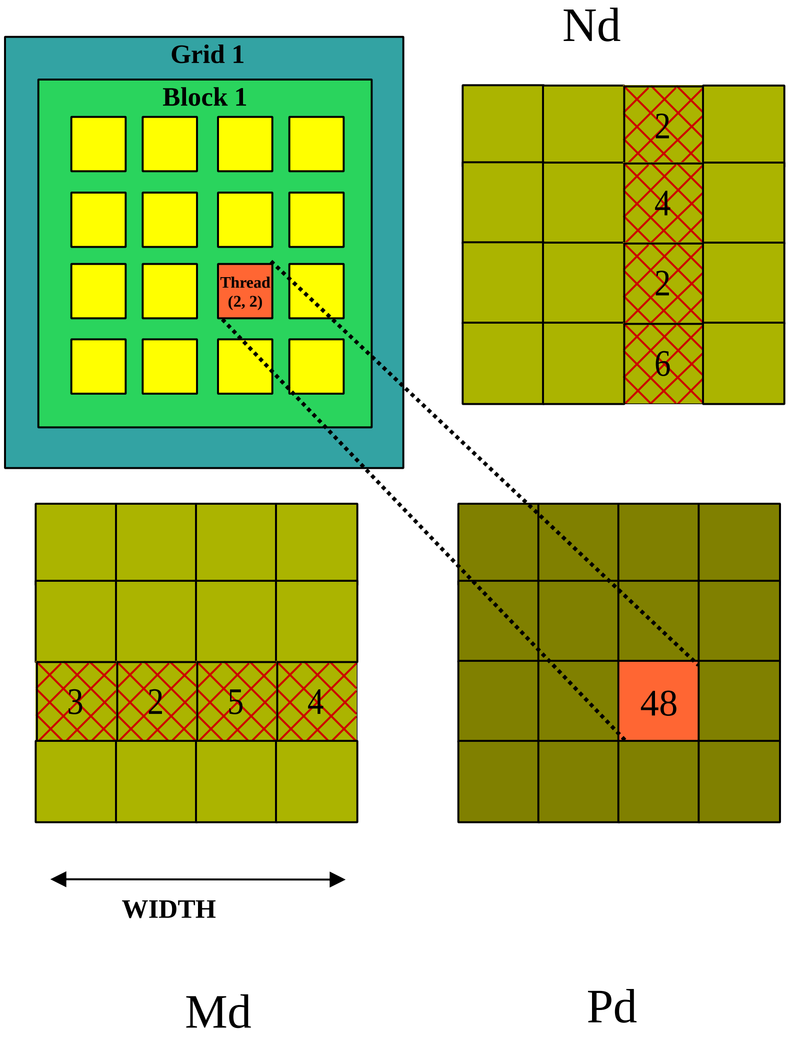
```
// Setup the execution configuration
dim3 dimBlock(WIDTH, WIDTH);
dim3 dimGrid(1, 1);
```

1个block 含 width * width 个线程

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

Matrix Multiply矩阵相乘样例

- 一个线程block计算Pd
  - 每个线程计算Pd的一个元素
- 每个线程
  - 读入矩阵Md的一行
  - 读入矩阵Nd的一列
  - 为每对Md和Nd元素执行一次乘法和加法
  - (not very high) 计算次数和片外访存次数比率接近1:1（不是很高）
- 矩阵的长度受限于一个线程块允许的线程数目



Matrix Multiply矩阵相乘样例 问题?

- 在算法实现中最主要的性能问题是什么?
- 主要的限制是什么?