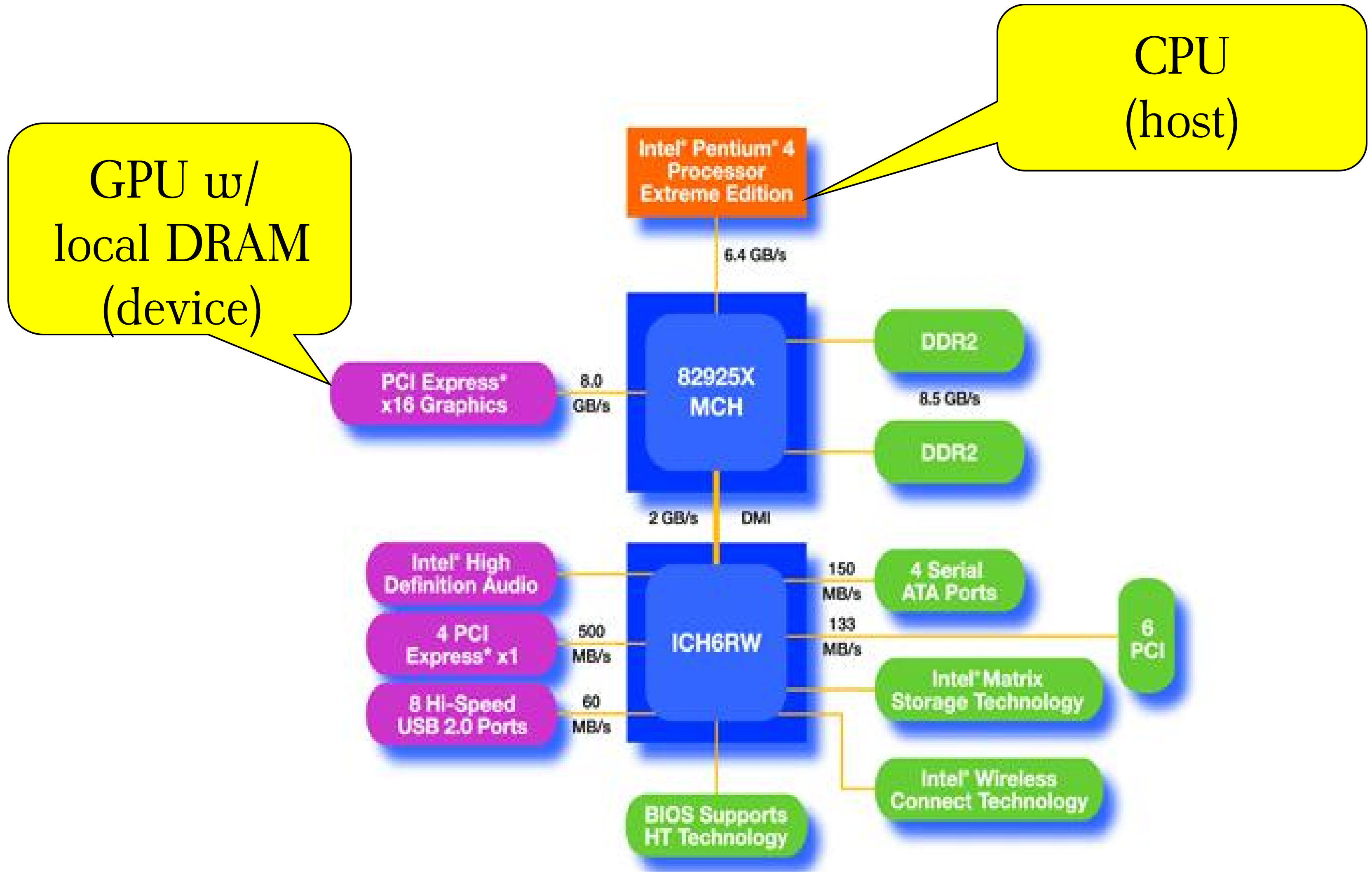
实验1C：CUDA编程模型

CUDA

- 计算统一设备架构（Compute Unified Device Architecture）的缩写
- 一种通用（General purpose）编程模型
  - 用户可以在GPU上启动海量线程
  - GPU = 专用的超线程、大规模数据并行协处理器
- 有针对性的软件栈
  - 计算导向的驱动、语言和工具
- 驱动设计将计算程序载入GPU
  - 标准驱动-优化计算
  - 设计计算接口-无图形API
  - 确保最高的下载/回读速率
  - 显式的GPU内存管理

CUDA后面的物理实现

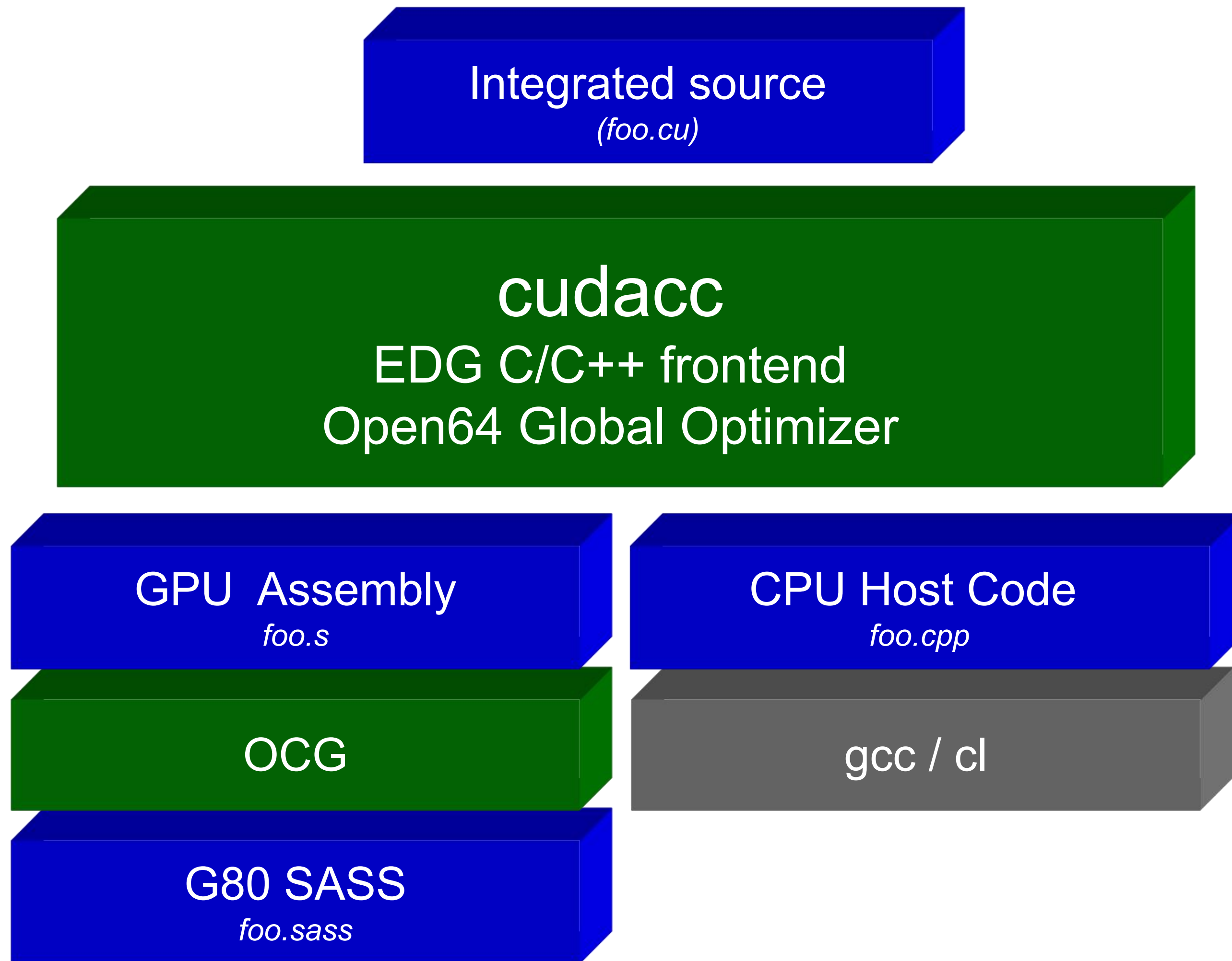


扩展 C 编程

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

扩展C程序



CUDA总览

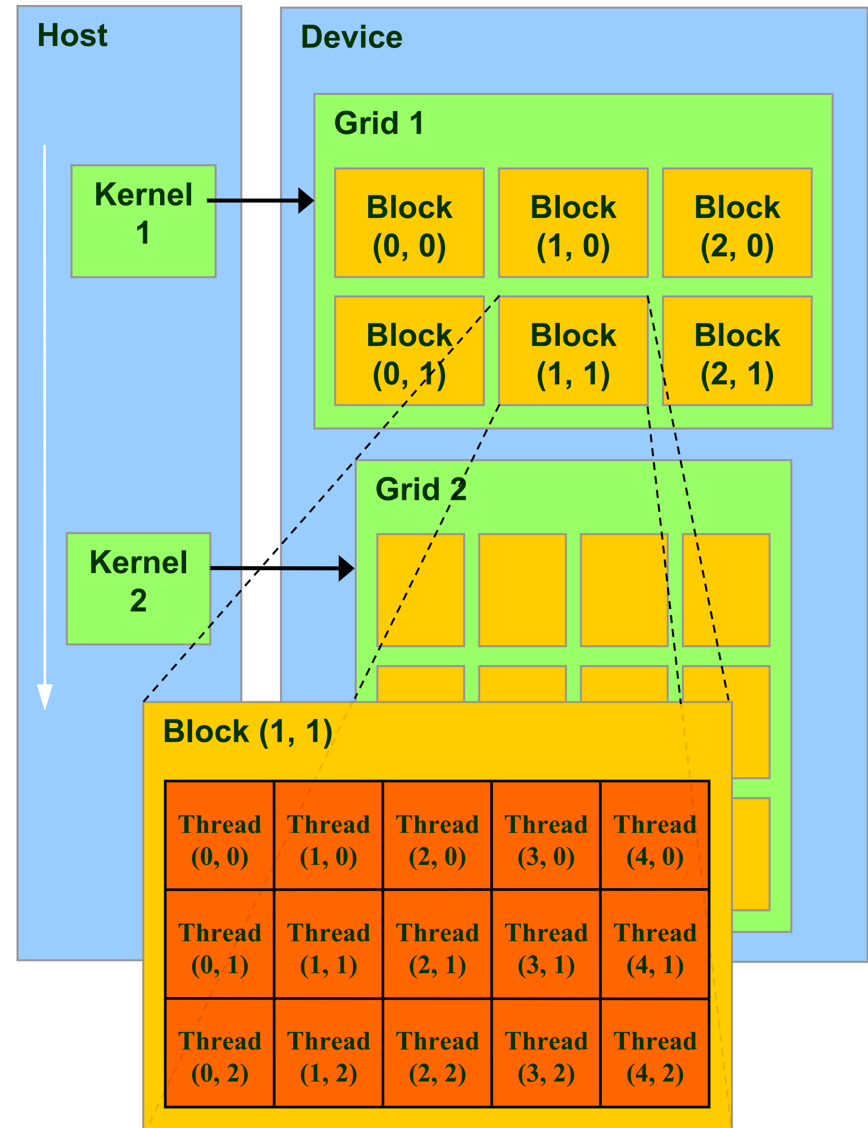
- CUDA编程模型 - 基本概念和数据类型
- CUDA应用编程接口 - 基本
- 简单程序案例
- 性能特征将在稍后介绍

高度多线程协处理器

- GPU是计算设备（device）
  - 是CPU或主机（host）的协处理器（coprocessor）
  - 具有自身的内存DRAM (device memory)
  - 并行执行许多线程（threads in parallel）
- 应用程序的数据可并行部分在设备上作为核函数（kernels）在诸多线程上并行执行。
- CPU与GPU线程的差异
  - GPU线程是相当轻量级的（lightweight）
    - 创建线程的开销极小
  - 要充分实现效率，GPU需要成千上万个线程
    - 而多核CPU只要很少几个线程即可

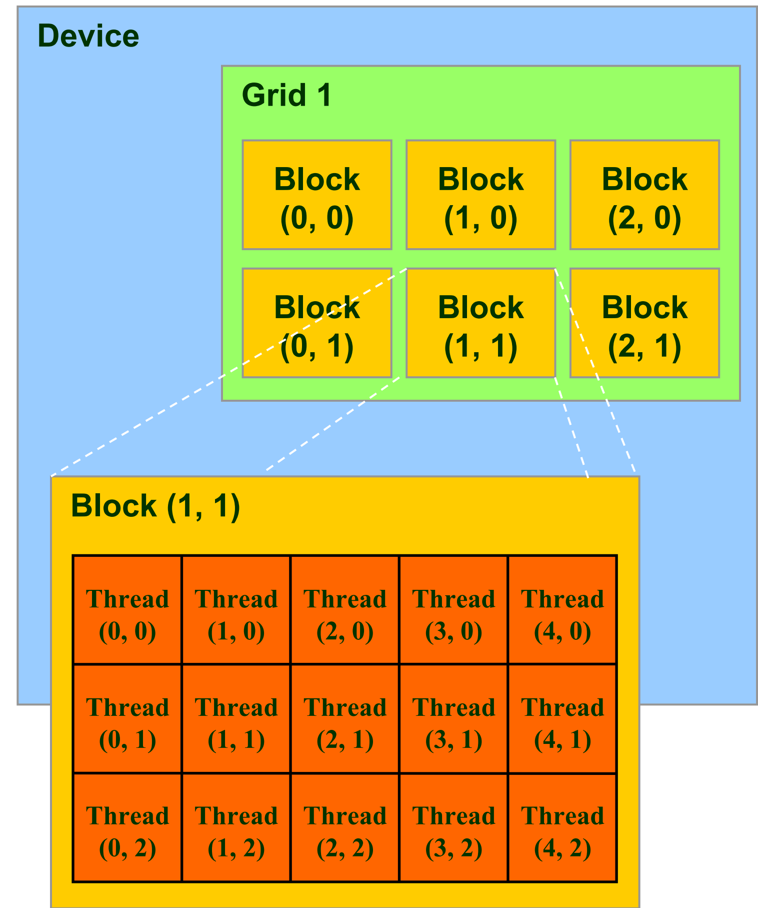
批量线程（Thread）：格点（Grids）与块（ Blocks）

- 核函数在一个线程块网格上执行（a grid of thread blocks）
  - 所有线程共享数据内存空间
- 一个线程块（thread block）是一个批量的线程，同一个线程块内部的线程之间通过
  - 同步执行（synchronization）
  - 通过低延迟的共享内存（ shared memory）高效共享数据
- 来自不同线程块（blocks）的两个线程无法协作



块ID与线程ID

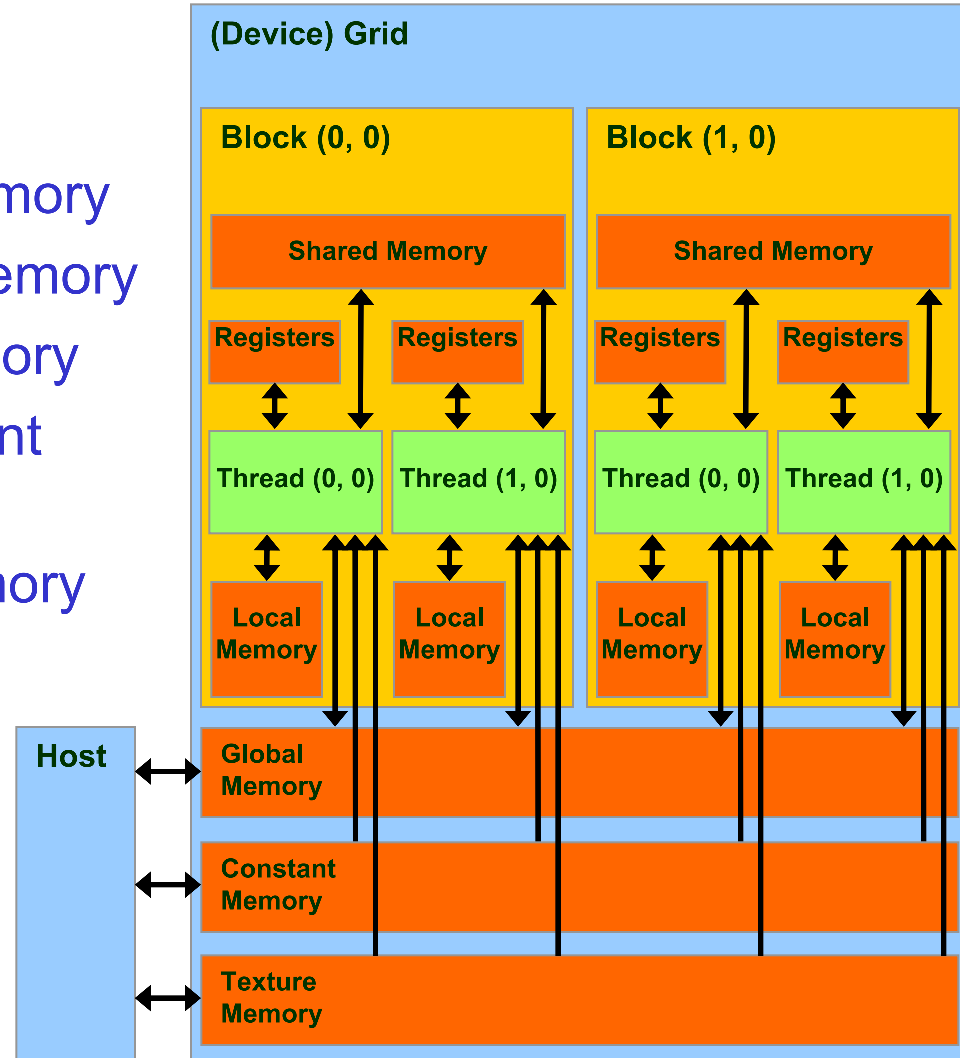
- 每个线程和线程块有唯一的ID
  - 于是每个线程可以确定要处理的数据
  - 块Block的ID: 1D 或 2D
  - 线程Thread的ID: 1D, 2D, 或 3D
- 处理多维数据时可以简化内存编址
  - 图像处理（Image processing）
  - 求解大型偏微分方程
  - …



Courtesy: NDVIA

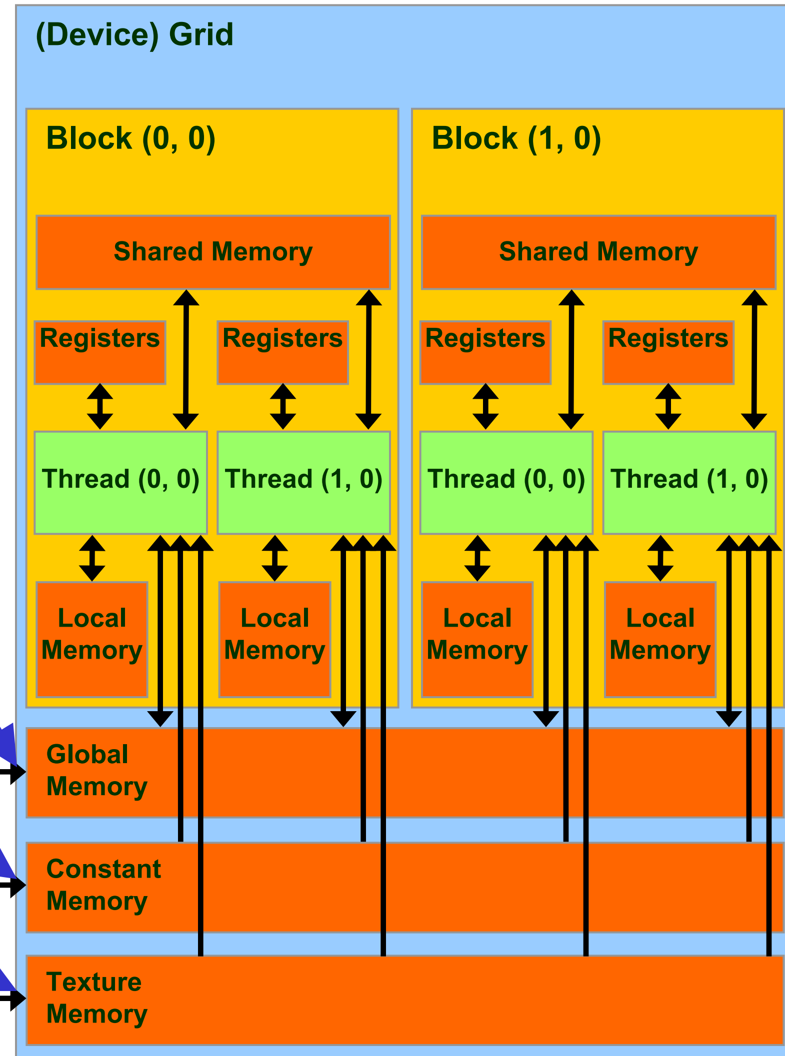
CUDA设备内存：概要

- 每一个线程均可
  - 读/写 per-thread registers
  - 读/写 per-thread local memory
  - 读/写 per-block shared memory
  - 读/写 per-grid global memory
  - 读/写 only per-grid constant memory
  - 只读 per-grid texture memory
- 而CPU主机只能读/写global, constant, 以及texture内存



全局（Global）/常量（Constant）/纹理（Texture）内存：(长延迟访问)

- 全局（Global）内存
  - 是主机（Host）与设备（device）之间交换读/写数据的主要方式
  - 数据对所有线程可见
- 纹理（Texture）与常量（Constant）内存
  - 常量由host初始化
  - 内存对所有线程开放访问



Courtesy: NDVIA

CUDA – API

CUDA API

- 是对ANSI C的扩展

  → Low learning curve

- 硬件设计的目的是运行轻量级的运行与驱动

  → High performance

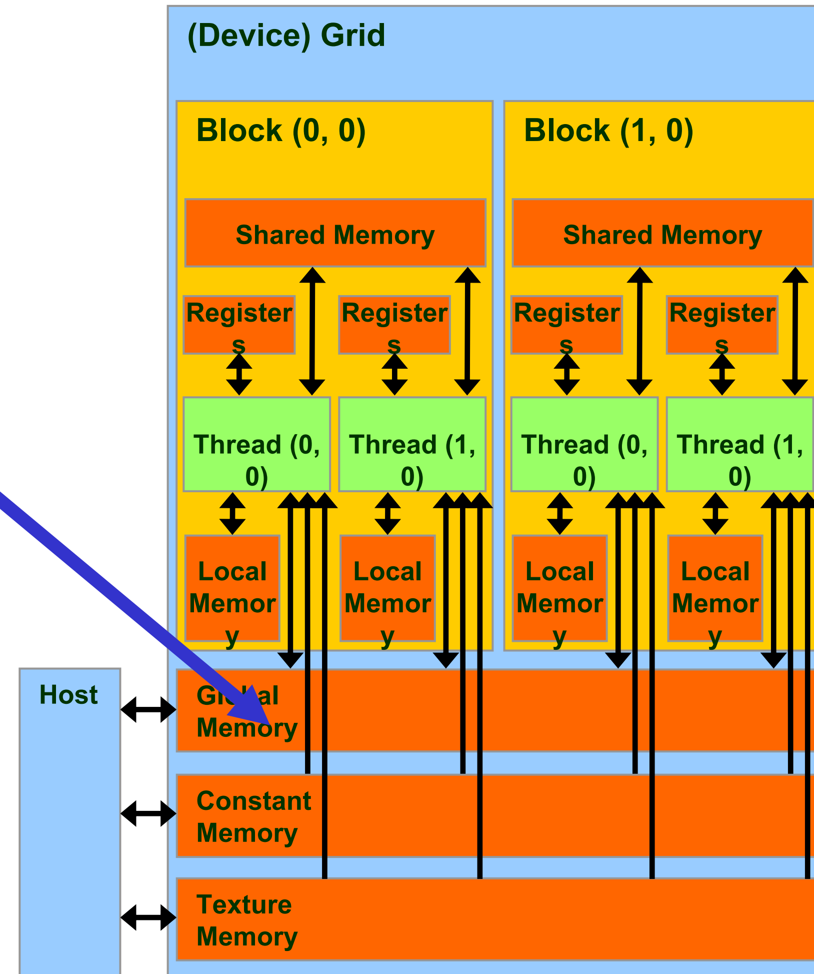
Matrix数据类型

- 并非CUDA的组成部分
- 常常用在许多例程中
  - 2D 矩阵
  - 单精度浮点数元素
  - width * height 个元素
  - pitch 仅当矩阵是另一个矩阵的子阵方有意义
  - 数据元素的分配，并组成 elements

```
typedef struct {
    int width;
    int height;
    int pitch;
    float* elements;
} Matrix;
```

CUDA设备的内存分配

- cudaMalloc()
  - 为设备中的对象分配全局内存（Global Memory）
  - 需要两个参数
    - **指向分配对象的指针地址**
    - **分配对象的大小（Size）**
- cudaFree()
  - 从设备的全局内存中释放对象
    - **参数：欲释放对象的指针**



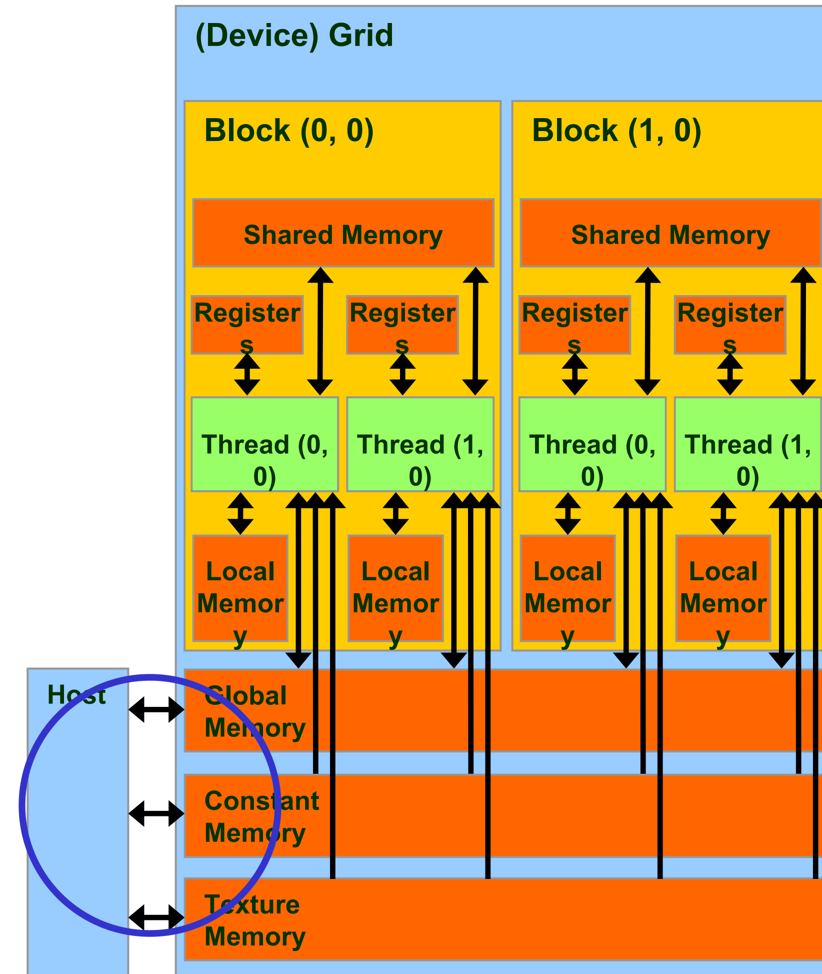
CUDA 设备内存分配 (cont.)

- 示例程序:
  - 分配一个 64 * 64 单精度浮点数组
  - 分配的存储绑定 Md.elements
  - “d” 常常用来表示设备数据结构

```
BLOCK_SIZE = 64;
Matrix Md
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

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

CUDA 的Host-Device数据传输

- cudaMemcpy()
  - 内存数据迁移
  - 需要四个参数
    - 源指针、目标指针、拷贝字节数
    - 传输类型
      - 从Host 到 Host
      - 从Host 到 Device
      - 从Device 到 Host
      - 从Device 到 Device
- 默认阻塞、同步



CUDA的Host-Device数据传输

- 例程
  - 传输一个 64 * 64 单精度浮点数组
  - M在主存中；Md在GPU显存中
  - cudaMemcpyHostToDevice与cudaMemcpyDeviceToHost均为字符常数

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

cudaMemcpy(M.elements, Md.elements, size,
           cudaMemcpyDeviceToHost);
```

CUDA函数声明

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- __global__ 定义核函数
  - 函数返回值必须为 void
- __device__ 与 __host__ 可以同时使用

CUDA函数的声明 (cont.)

- __device__ 函数无法获取其地址
- 对于设备执行的函数:
  - 不允许递归
  - 函数内部无静态变量声明
  - 参数数量固定

调用Kernel函数 – 创建线程

- 通过执行配置（execution configuration）调用核函数

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

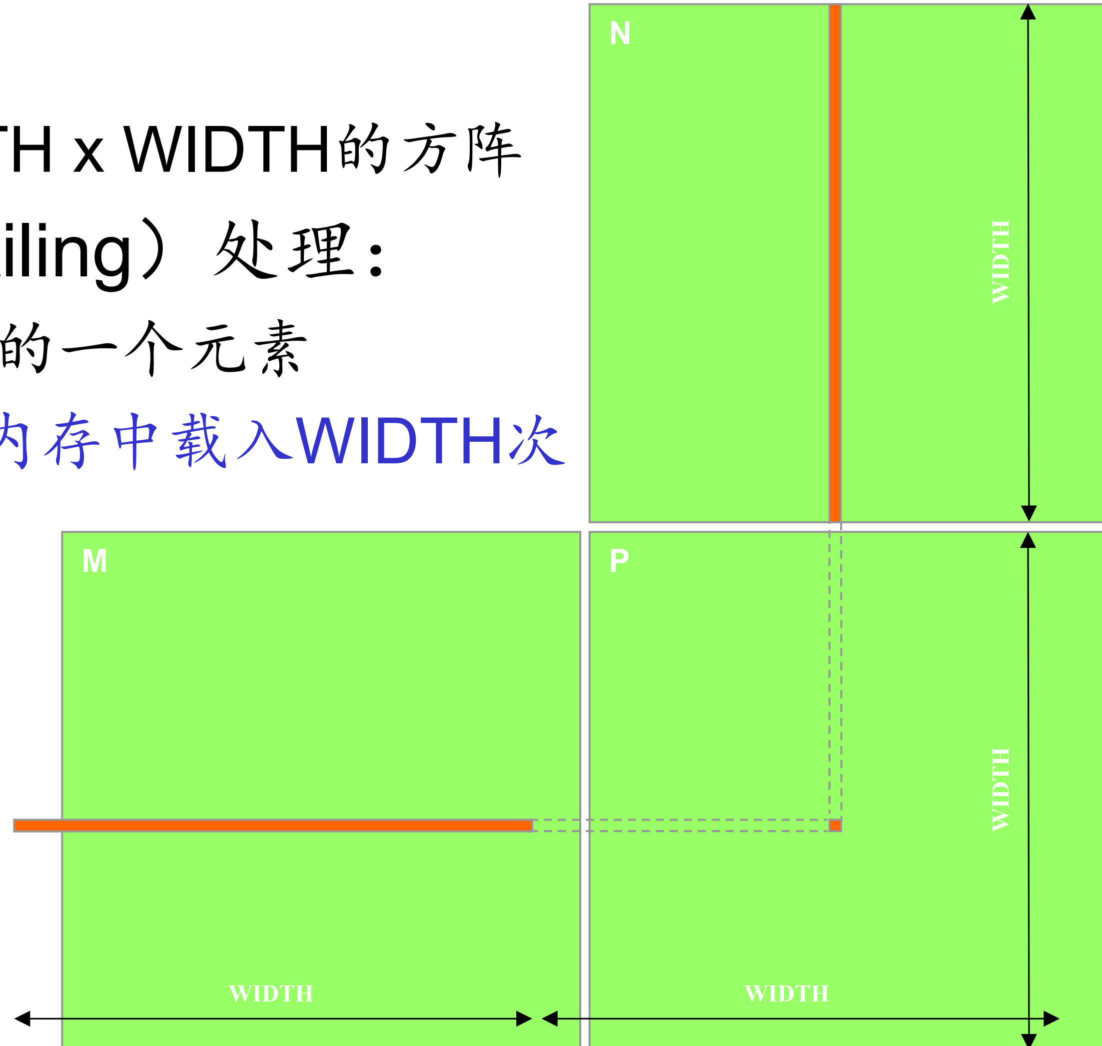
- 所有核函数的调用都是异步的。如果需要阻塞，需要显式同步

简单例程：矩阵相乘

- 我们常常用矩阵相乘这个例子来阐述CUDA程序内存基本特征和线程管理
  - 暂不介绍共享内存的使用
  - 局部内存和寄存器的使用
  - 线程 ID的使用
  - Host与Device之间内存数据传输的API

方阵相乘

- P = M * N
  - M和N都是WIDTH x WIDTH的方阵
- 不需平铺展开（tiling）处理：
  - 每个线程处理P的一个元素
  - M 和 N 从全局内存中载入WIDTH次



Step 1: 矩阵数据传输

```
// Allocate the device memory where we will copy M to
Matrix Md;
Md.width  = WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);

// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size,
   cudaMemcpyHostToDevice);

// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size,
   cudaMemcpyDeviceToHost);
...
// Free device memory
cudaFree(Md.elements);
```

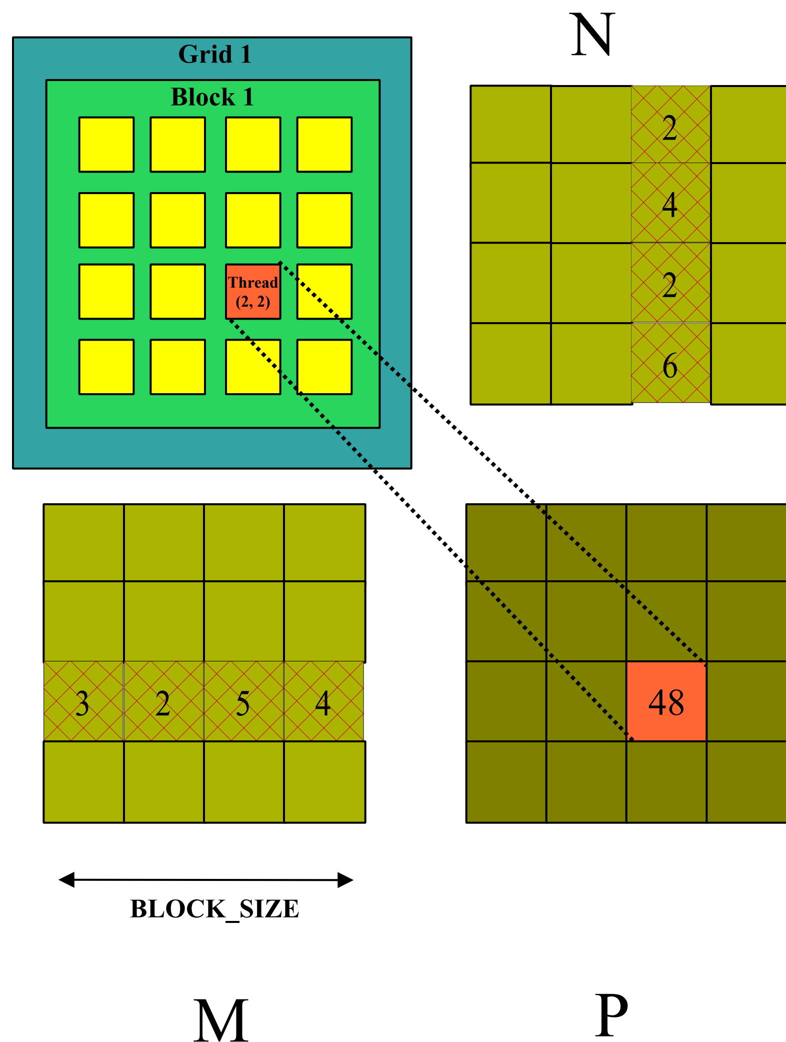
Step 2: 矩阵相乘（Host）

// Matrix multiplication on the (CPU) host in double precision
// for simplicity, we will assume that all dimensions are equal

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

单个线程块上的乘法计算

- 用一个线程块上的线程计算矩阵P
  – 每个线程计算P的一个元素
- 每个线程执行任务
  – 载入矩阵M的一行
  – 载入矩阵N的一列
  – 执行每一对M和N元素的相乘并相加
- 矩阵的大小受到线程块中线程数量的限制



Step 3: Host端main函数

```
int main(void) {
// Allocate and initialize the matrices
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

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

// Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);
return 0;
}
```

Step 3: Host端代码

```
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);
    CopyToDeviceMatrix(Pd, P); // Clear memory
```

Step 3: Host端代码 (cont.)

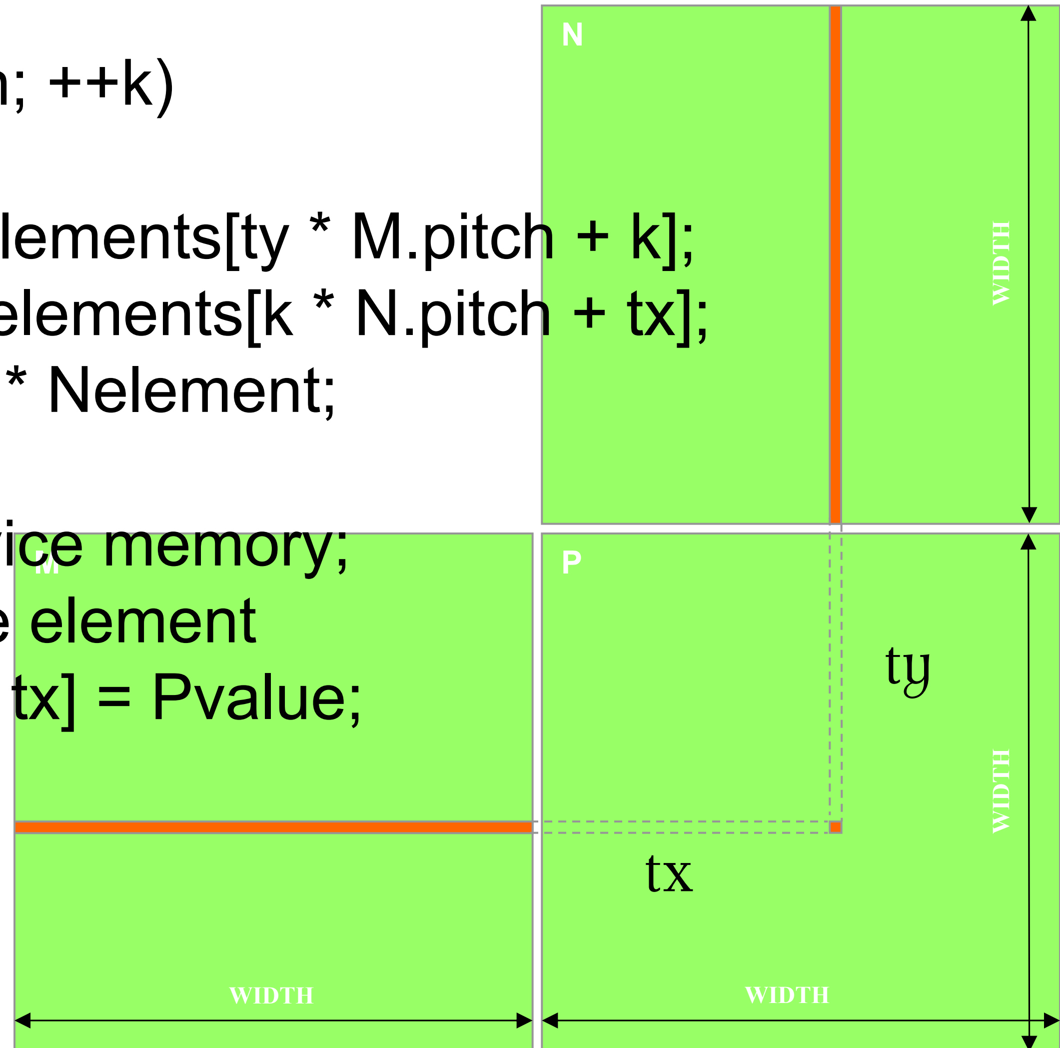
```
// Setup the execution configuration
    dim3 dimBlock(WIDTH, WIDTH);
    dim3 dimGrid(1, 1);

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

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

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

Step 4: Device端的核函数

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 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;
```

Step 4: Device端核函数（cont.)

```
for (int k = 0; k < M.width; ++k)
{
    float Melement = M.elements[ty * M.pitch + k];
    float Nelement = Nd.elements[k * N.pitch + tx];
    Pvalue += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element
P.elements[ty * P.pitch + tx] = Pvalue;
}
```



Step 5: 其他函数

```
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M)
{
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
```

Step 5: 其他函数 (cont.)

```
// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost)
{
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size,
            cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice)
{
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size,
            cudaMemcpyDeviceToHost);
}
```

Step 6: 处理任意大小的矩阵

- Have each 2D thread block to compute a $(BLOCK_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - Each has $(BLOCK_WIDTH)^2$ threads
- Generate a 2D Grid of $(WIDTH/BLOCK_WIDTH)^2$ blocks

You still need to put a loop around the kernel call for cases where WIDTH is greater than Max grid size!