GPGPU+CUDA

Cui Tao
tcui@lsec.cc.ac.cn

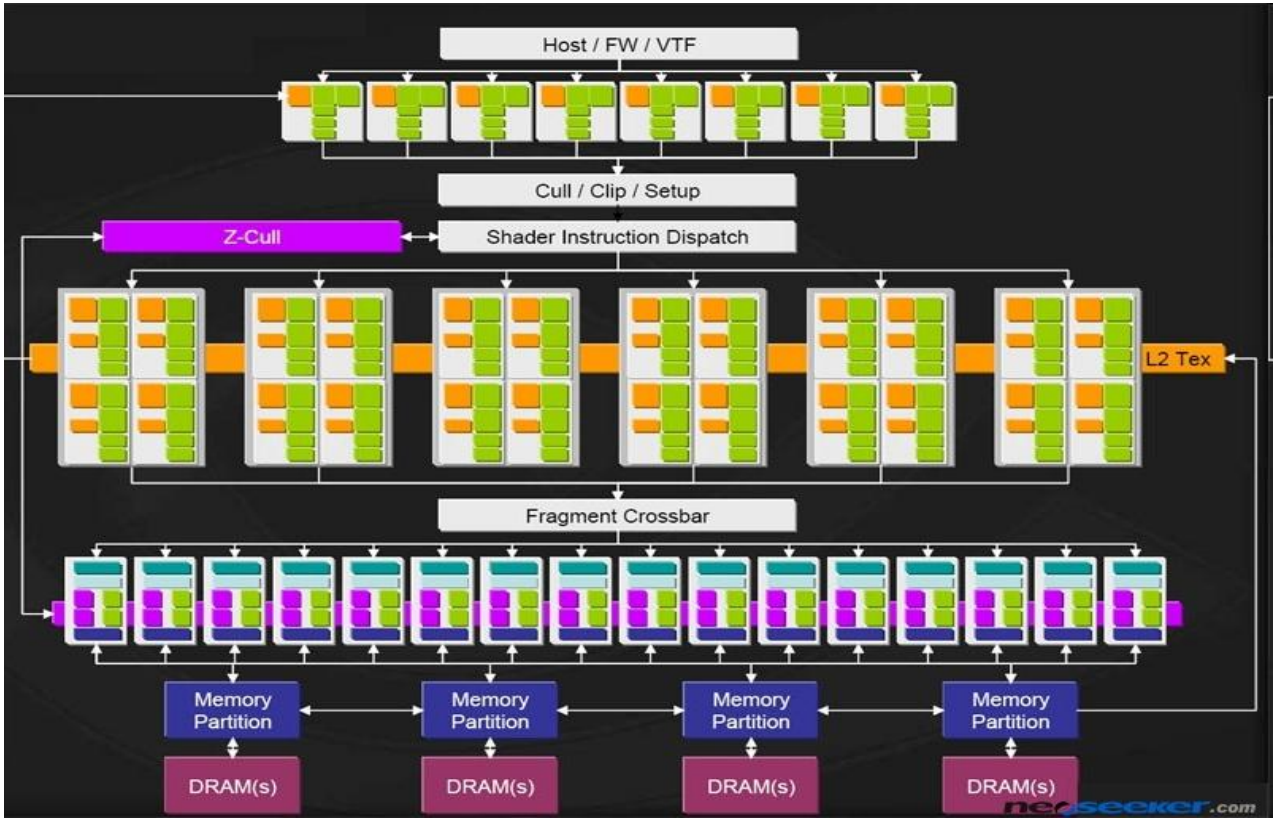
GPU是什么

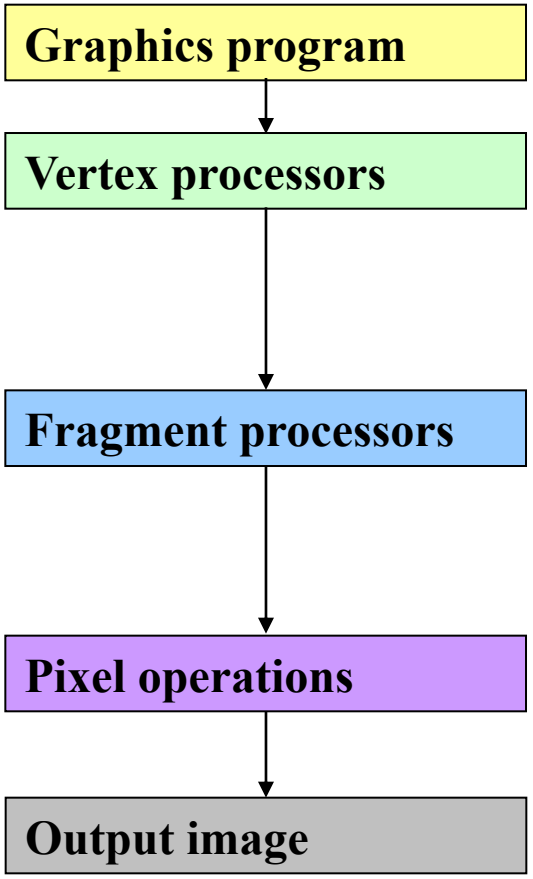
- Graphic Processing Unit：图形处理器，显卡的处理核心
- NVIDIA公司在1999年发布Geforce 256图形处理芯片时首先提出GPU的概念，随后大量复杂的应用需求促使整个产业蓬勃发展至今。
- Geforce 256之所以被称作GPU原因就在于Geforce 256划时代的在图形芯片内部集成了T&L（几何光照转换）功能，使得GPU拥有初步的几何处理能力，彻底解决了当时众多游戏瓶颈发生在CPU几何吞吐量不够的瓶颈。

GPU是什么

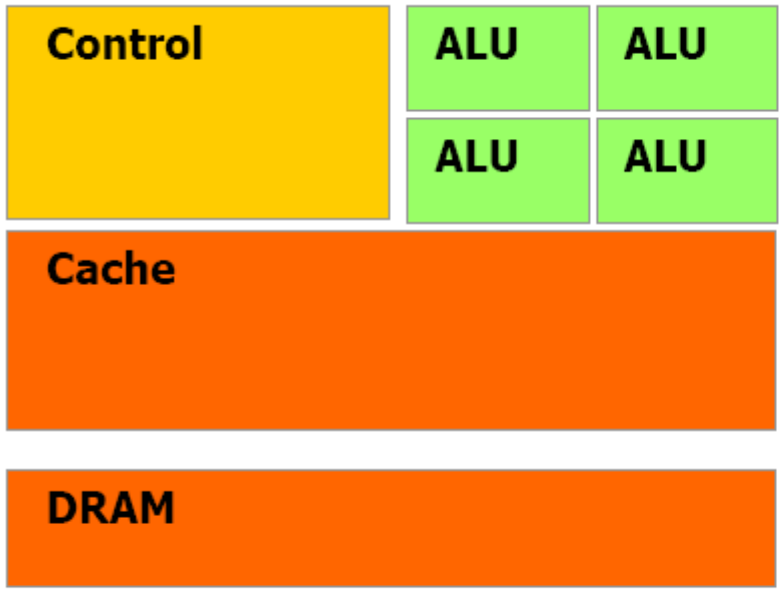
- 第一代GPU(1999年之前)：
  - 部分功能从CPU分离，实现硬件加速，
  - GE(Geometry Engine)为代表，只起3D图像处理的加速作用，不具有软件编程特性
- 第二代GPU(1999-2002年)：
  - 进一步硬件加速和有限的编程性
  - 1999年NVIDIA GeForce 256将T&L(Transform and Lighting)等功能从 CPU分离出来，实现了快速变换
  - 2001年NVIDIA和ATI分别推出的GeForce3和Radeon 8500，图形硬件的流水线被定义为流处理器，出现了顶点级可编程性，同时像素级也具有有限的编程性，但GPU的编程性仍然比较有限
- 第三代GPU(2002年之后)：
  - 2006年NVIDIA推出了CUDA(Computer Unified Device Architecture，统一计算架构)编程环境
  - GPU通用计算编程的复杂性大幅度降低
  - GPU已演化为一个新型可编程高性能并行计算资源，全面开启面向通用计算的新时代

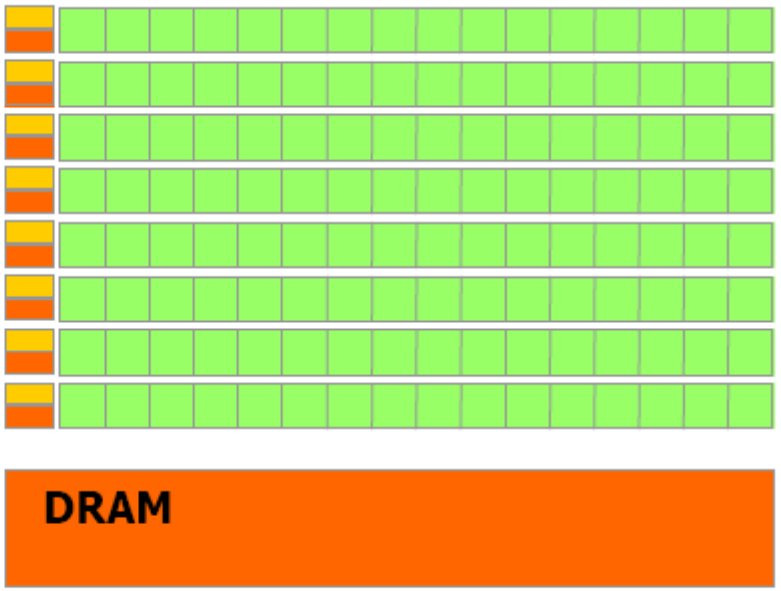
传统GPU架构


Host / FW / VTF
Cull / Clip / Setup
Z-Cull
Shader Instruction Dispatch
L2 Tex
Fragment Crossbar
Memory Partition
Memory Partition
Memory Partition
Memory Partition
DRAM(s)
DRAM(s)
DRAM(s)
DRAM(s)


Graphics program
Vertex processors
Fragment processors
Pixel operations
Output image

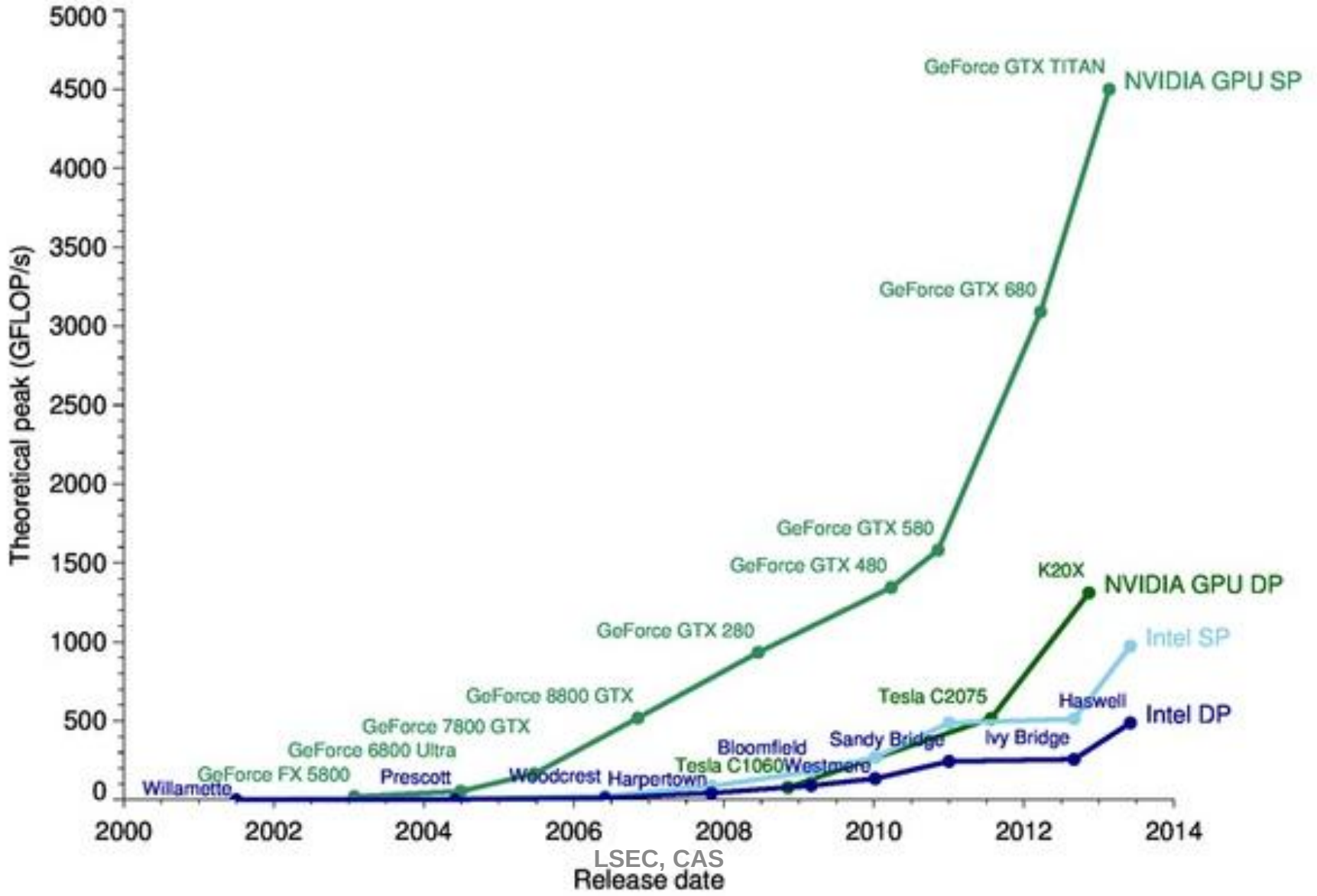
Graphic Processing Unit *V.S.* Central Processing Unit



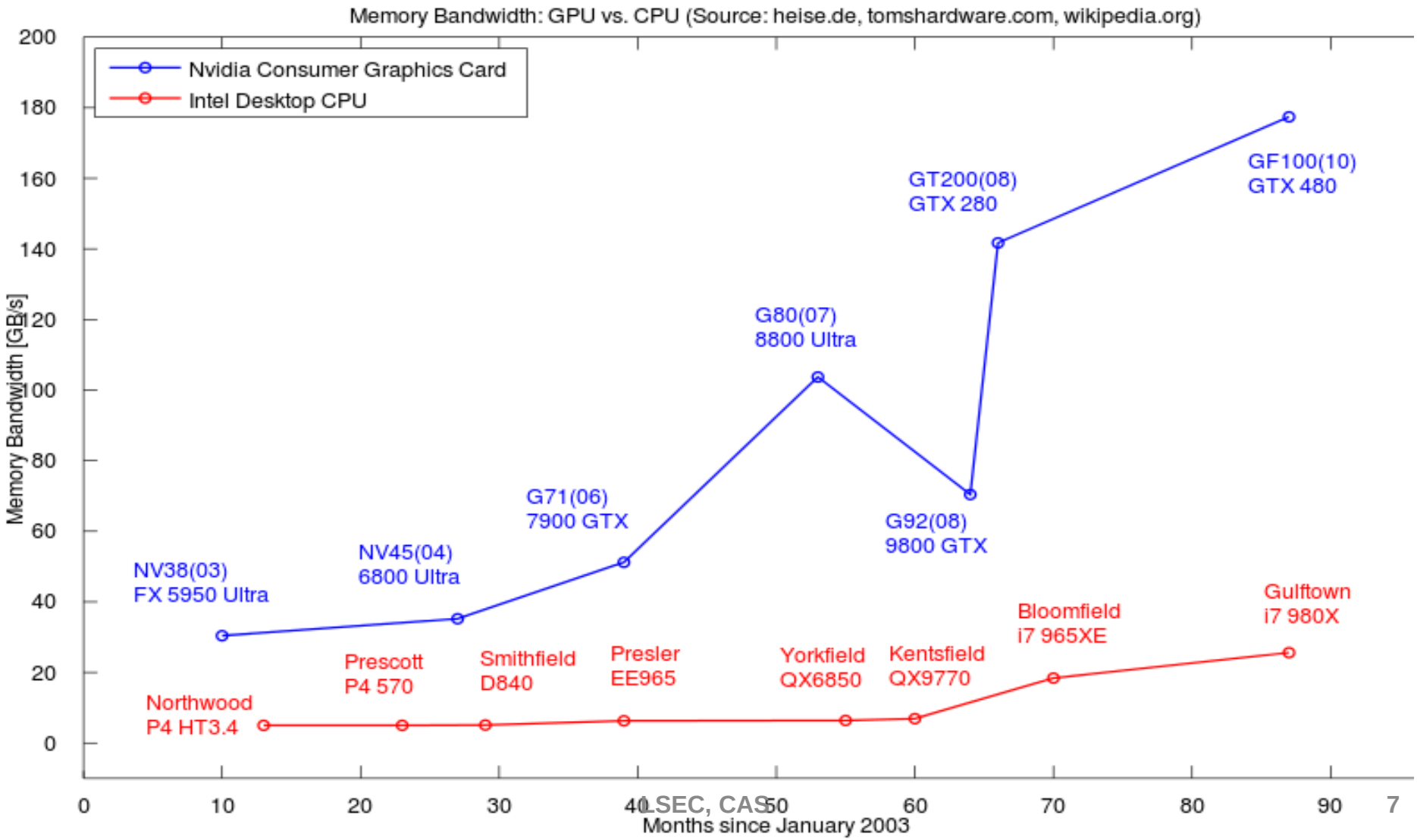


GPU: 计算密集型，高度并行化，可容纳上千个没有依赖关系的数值计算线程
CPU: 处理负责指令调度、分支、逻辑判断等，无法完成上百个线程的并行计算

Graphic Processing Unit *V.S.* Central Processing Unit

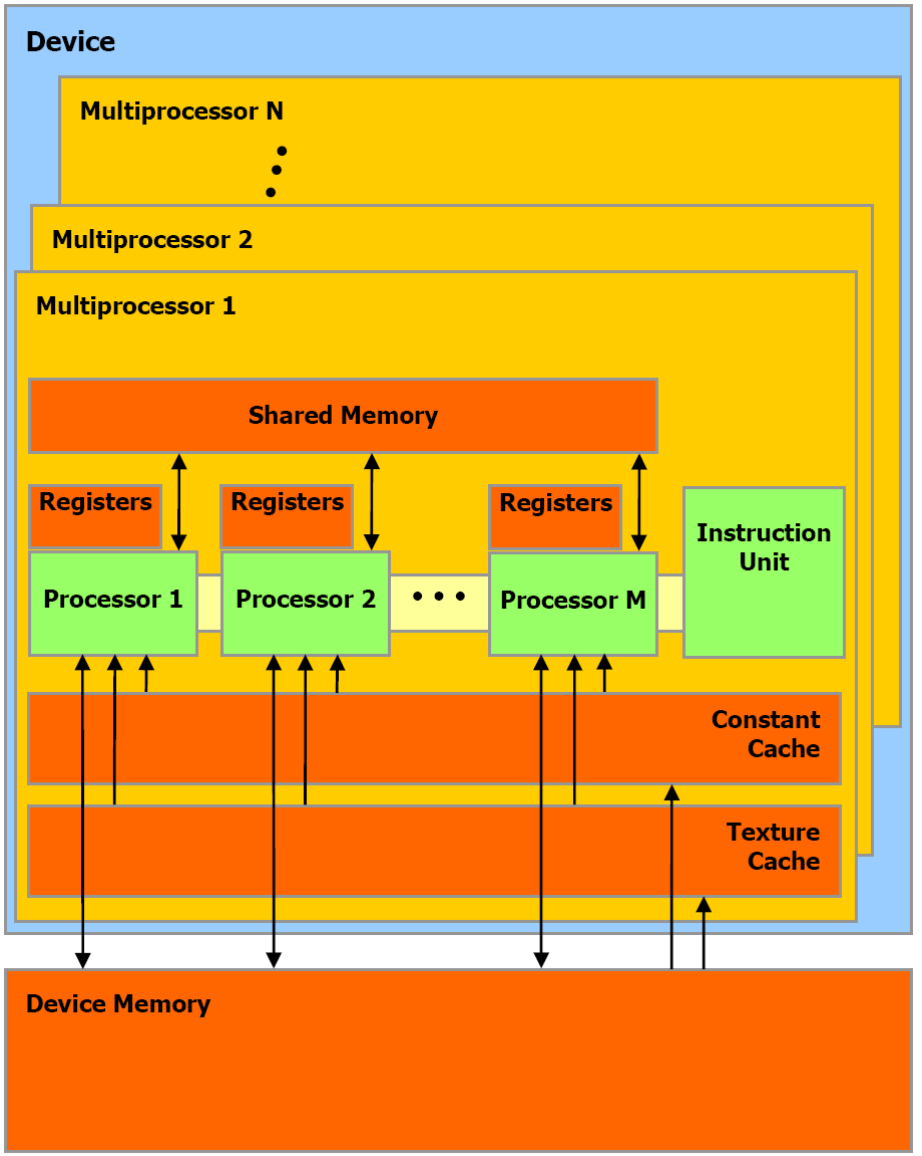


Graphic Processing Unit *V.S.* Central Processing Unit



GPU的整体架构

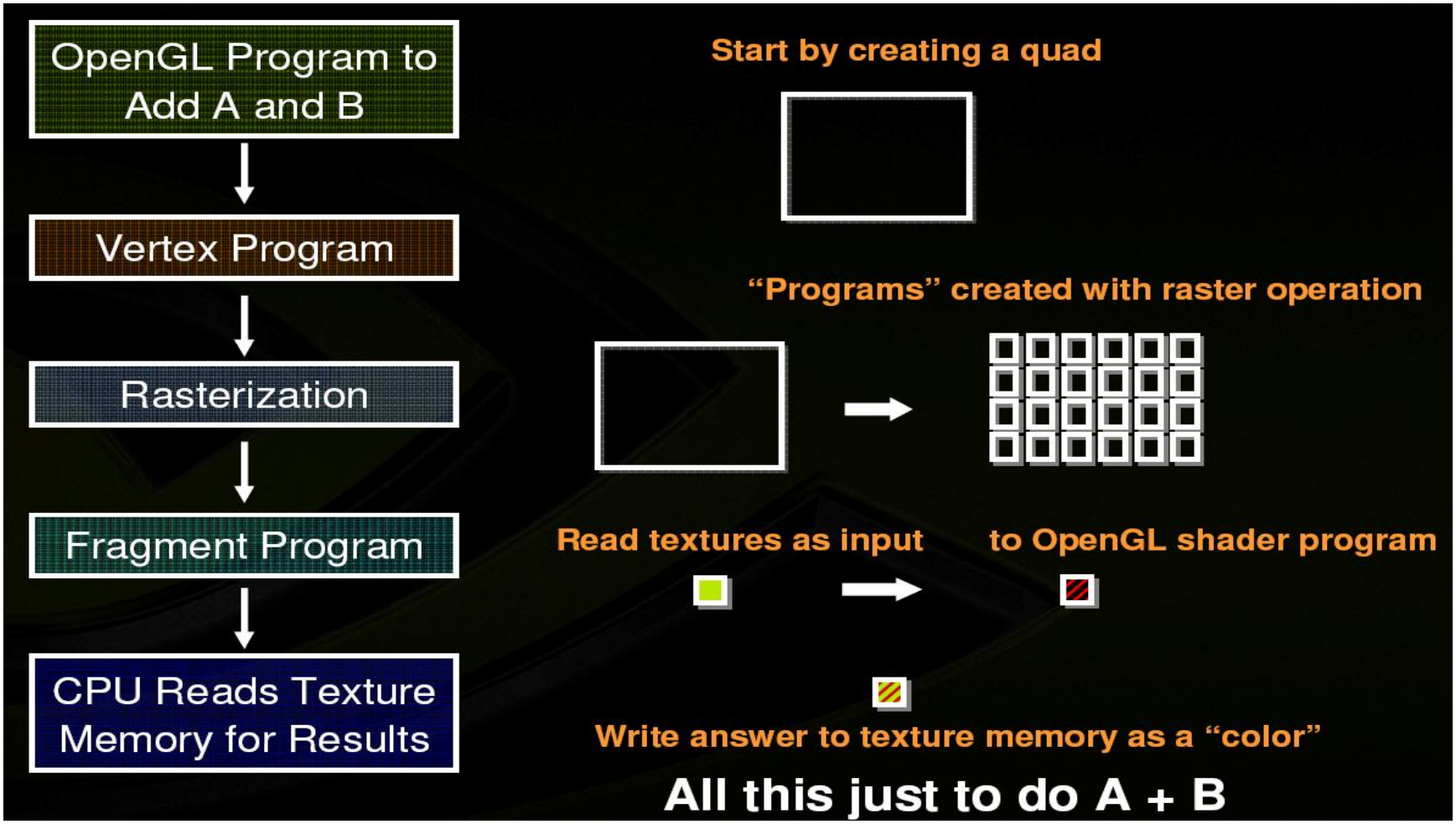
- 基本硬件架构：
  - 流多处理器（Stream Multiprocessor）
  - 流处理器（Stream Processor）
  - 共享内存（Shared Memory）
  - 板载显存（Device Memory）



GPU的三个重要抽象概念

- **GPGPU 概念1: 数组 = 纹理**
  - 数组的索引，而在GPU中，我们需要的是纹理坐标
- **GPGPU 概念 2: 内核 = 着色器**
  - 面向循环的CPU运算 vs. 面向内核的GPU数据并行运算
- **GPGPU 概念 3: 运算 = 渲染**

General Purpose Computing on GPU (GPGPU)



GPGPU

核心思想:

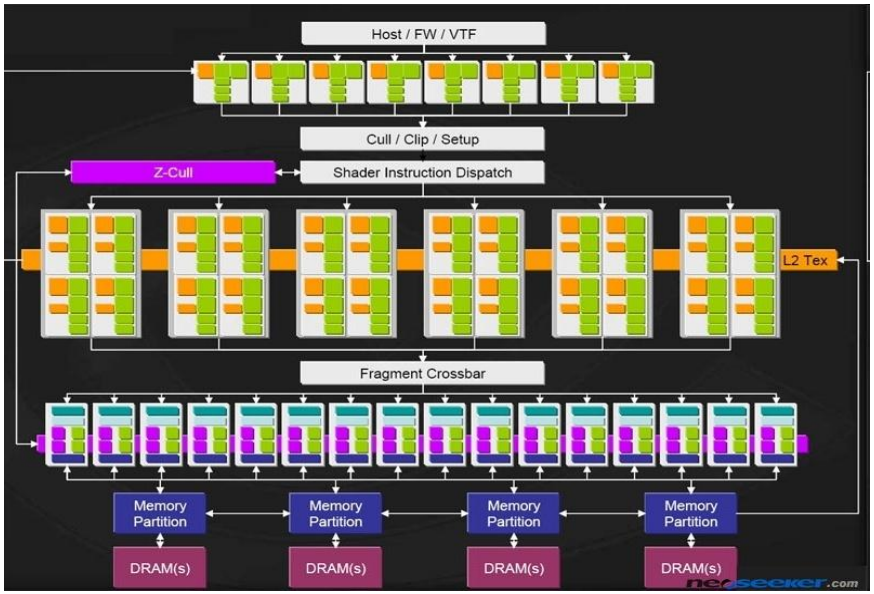
- 用图形语言描述通用计算问题
- 把数据映射到vertex或者fragment处理器

好处:

- 显示芯片通常具有更大的内存带宽。
- 示芯片具有更大量的执行单元。
- 和高价 CPU 相比，显卡的价格较为低廉。

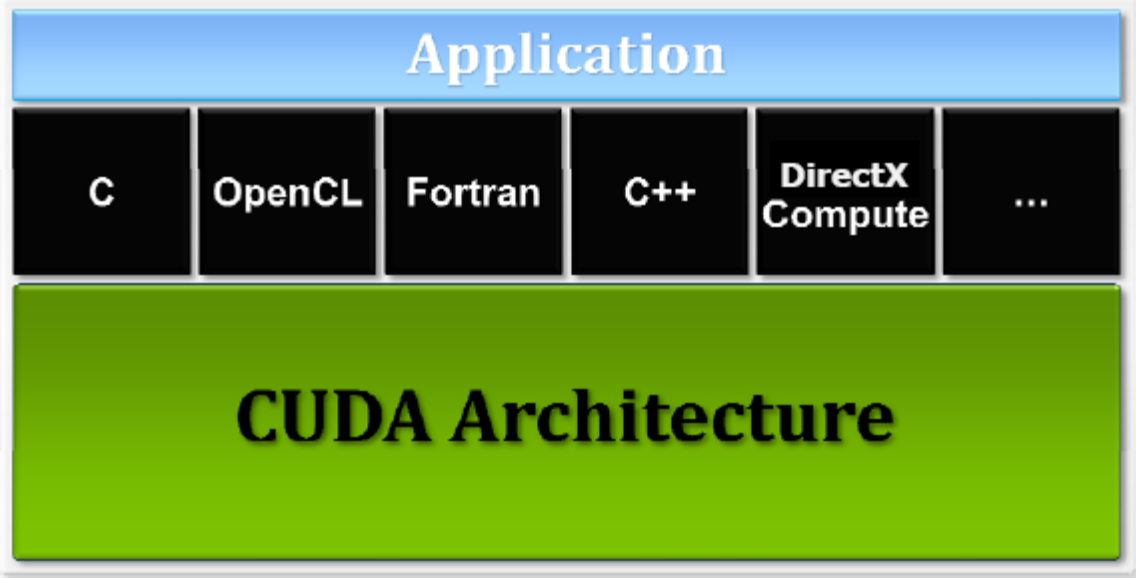
但是:

- 硬件资源使用不充分
- 存储器访问方式严重受限
- 难以调试和查错
- 高度图形处理和编程技巧
- 需要计算高度并行化
- 不具有分支预测等复杂的流程控制单元

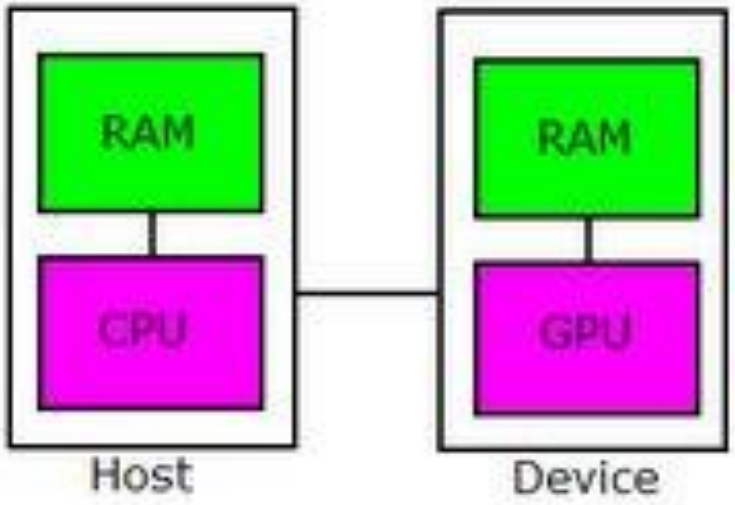


Compute Unified Device Architecture 介绍

- CUDA 是一种并行编程模型和软件环境
  - 提供标准C语言支持, 兼容性、可移植性
  - 单指令、多数据执行模式, 大量并行计算资源处理不同数据
  - 隐藏存储器延时
    - 提升计算 / 通信比例
    - 合并相邻地址的内存访问
    - 快速线程切换1 cycle@GPU vs. ~1000 cycles@CPU



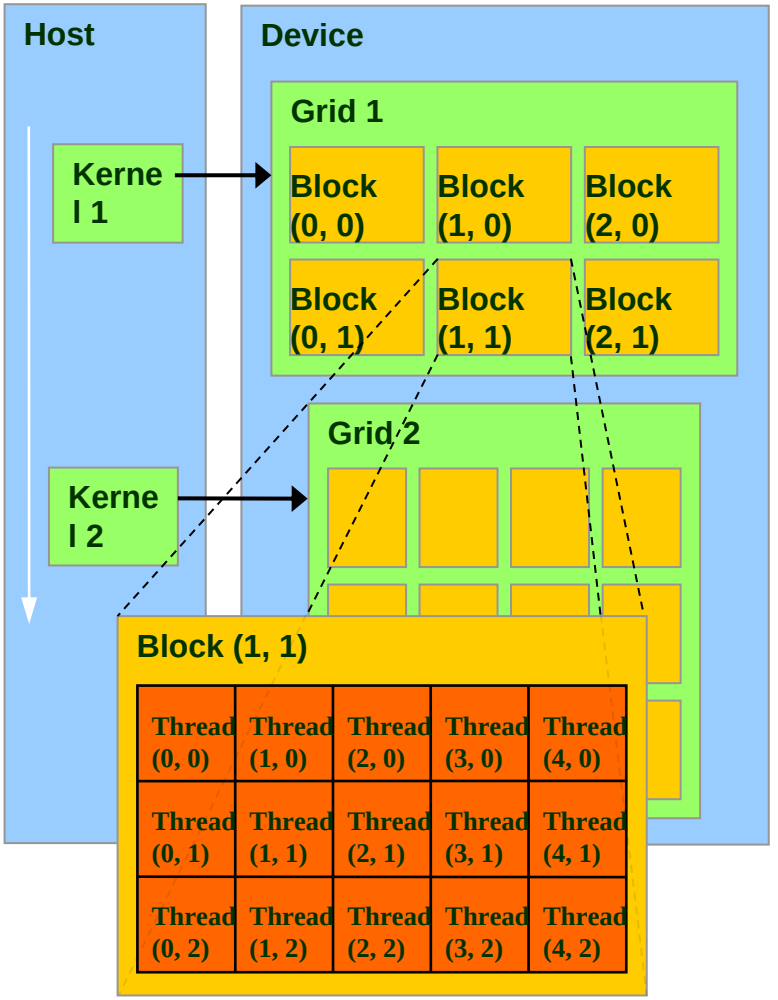
CUDA 架构



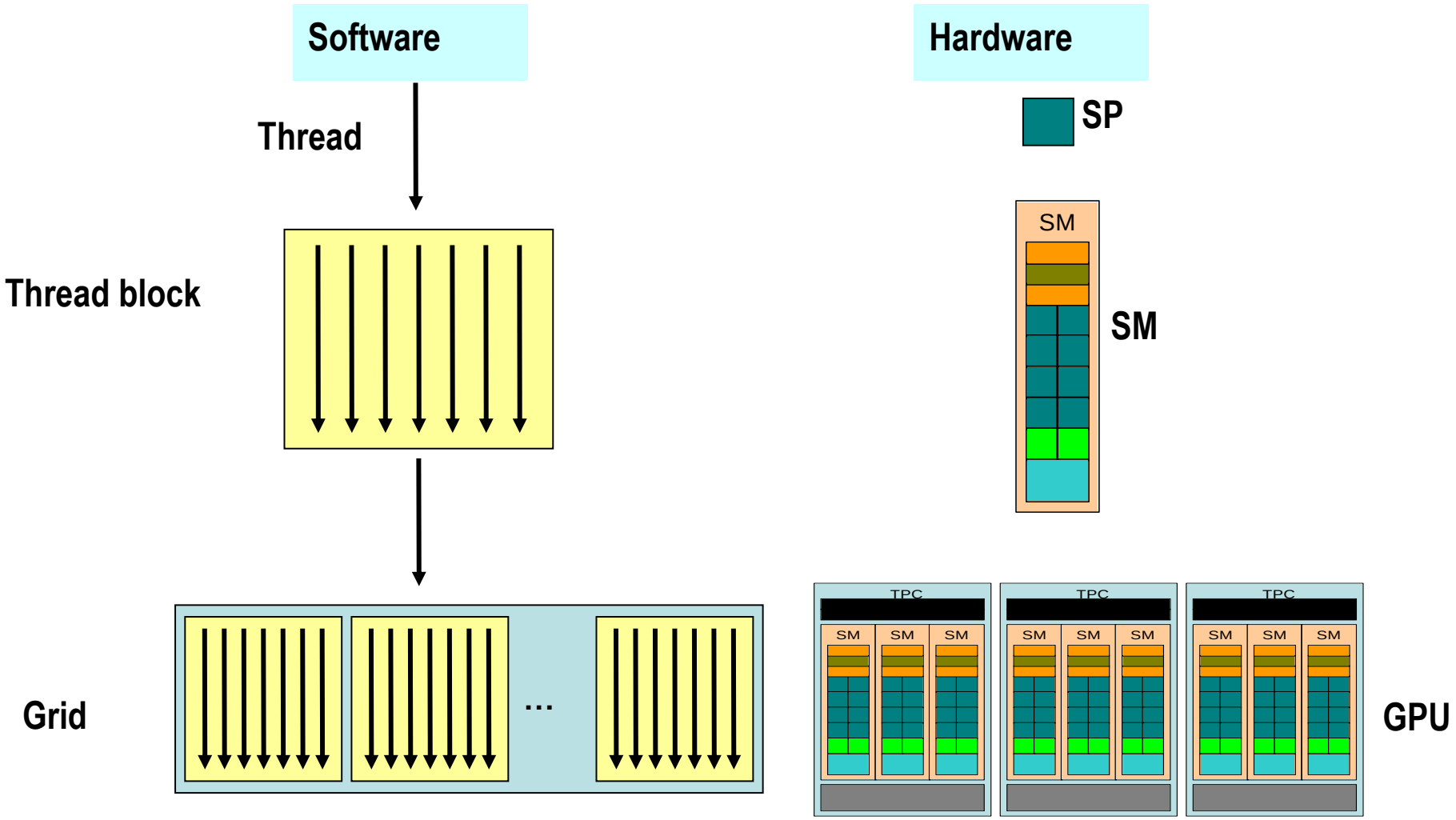
- Host端是指CPU
- Device端则是显示芯片,Device 端的程序又称为 kernel
- 在 CUDA 架构下，显示芯片执行时的最小单位是 thread。数个 thread 可以组成一个 block。一个 block 中的 thread 能存取同一块共享的内存，而且可以快速进行同步的动作。

并行线程层次结构

- **Thread**: 并行的基本单位
- **Thread block**: 互相合作的线程组
  - ✓ Cooperative Thread Array (CTA)
  - ✓ 允许彼此同步
  - ✓ 通过快速共享内存交换数据
  - ✓ 以1维、2维或3维组织
  - ✓ 最多包含512个线程
- **Grid**: 一组**thread block**
  - ✓ 以1维或2维组织
  - ✓ 共享全局内存
- **Kernel**: 在**GPU**上执行的核心程序
  - ✓ One **Kernel** ↔ one **Grid**
  - ✓ 不同的 **Grid** 则可以执行不同的程序



Parallel Program Organization in CUDA

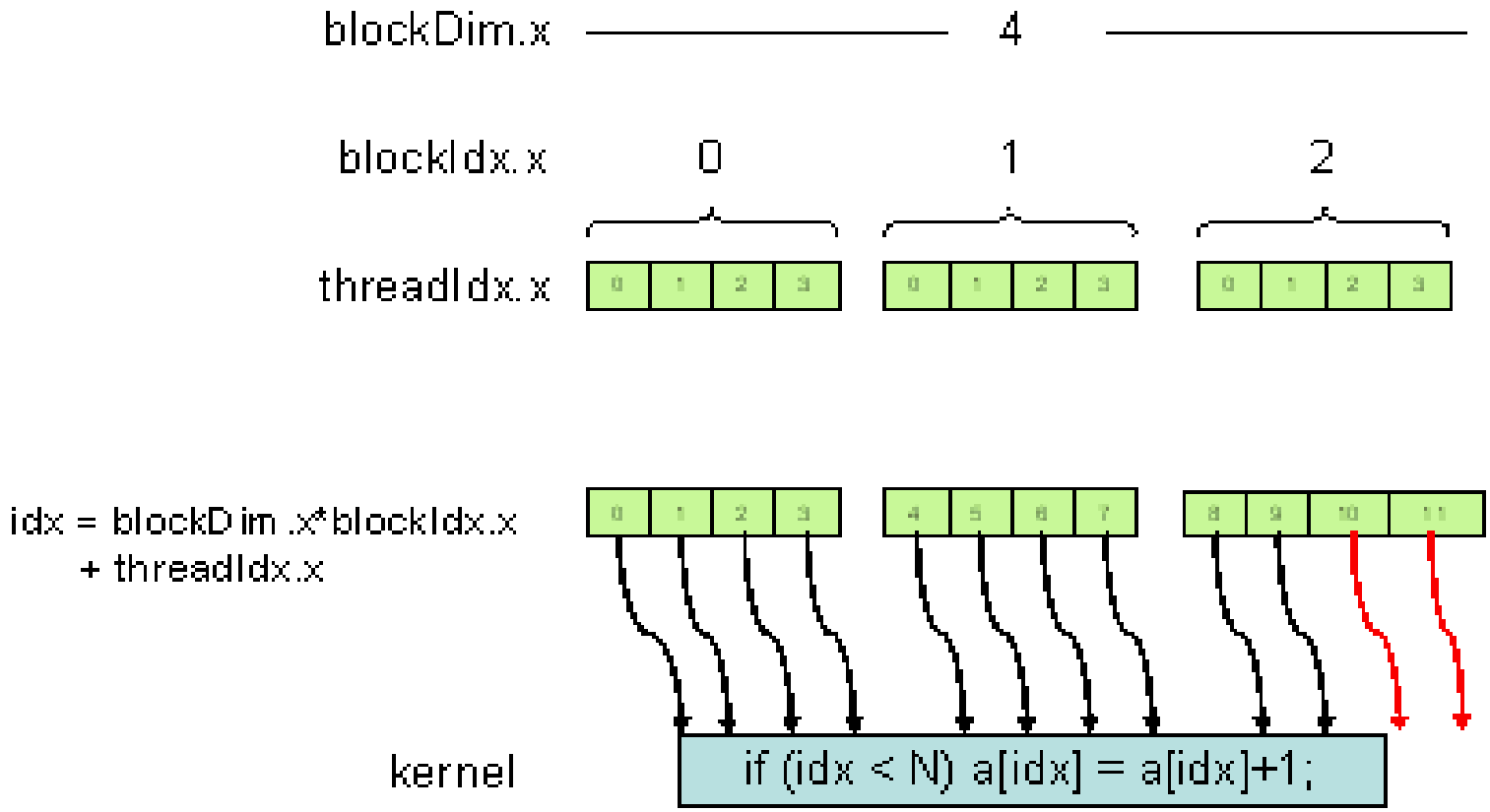


并行线程执行

- 调用kernel function 需要指定执行配置

```
__global__ void kernel(...);
dim3   DimGrid(3, 2);    // 6 thread blocks
dim3   DimBlock(16, 16);   // 256 threads per block
kernel<<< DimGrid, DimBlock>>> (...);
```

- Threads和blocks具有IDs
  - threadIdx: 1D, 2D, or 3D
  - blockIdx: 1D, or 2D
  - 由此决定相应处理数据


blockDim.x
4
blockIdx.x
0
1
2
threadIdx.x
0 1 2 3
0 1 2 3
0 1 2 3
idx = blockDim.x*blockIdx.x
+ threadIdx.x
0 1 2 3 4 5 6 7 8 9 10 11
kernel
if (idx < N) a[idx] = a[idx]+1;

实例1: Element-Wise Addition

```
//CPU program
//sum of two vectors a and b
void add_cpu(float *a, float *b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] += b[idx];
}


void main()
{
    .....
    fun_add(a, b, N);
}
```

```
//CUDA program
//sum of two vectors a and b
__global__ void add_gpu(float *a, float *b, int N)
{
    Int idx =blockIdx.x* blockDim.x+ threadIdx.x;
    if (idx < N)
        a[idx] += b[idx];
}

void main()
{
    …..
    dim3 dimBlock (256);
    dim3 dimGrid( ceil( N / 256 );
    fun_add<<<dimGrid, dimBlock>>>(a, b, N);
}
```

执行配置

- __global__ void func(void *)
- func <<<Dg, Db, Ns, s>>> (parameter)
- Dg 的类型为 dim3，指定网格的维度和大小，Dg.x * Dg.y 等于所启动的块数量，Dg.z 无用；
- Db 的类型为 dim3，指定各块的维度和大小，Db.x * Db.y * Db.z 等于各块的线程数量；
- Ns 的类型为 size_t，指定各块为此调用动态分配的共享存储器（除静态分配的存储器之外）；Ns 是一个可选参数，默认值为 0；
- S 的类型为 cudaStream_t，指定相关流；S 是一个可选参数，默认值为 0。

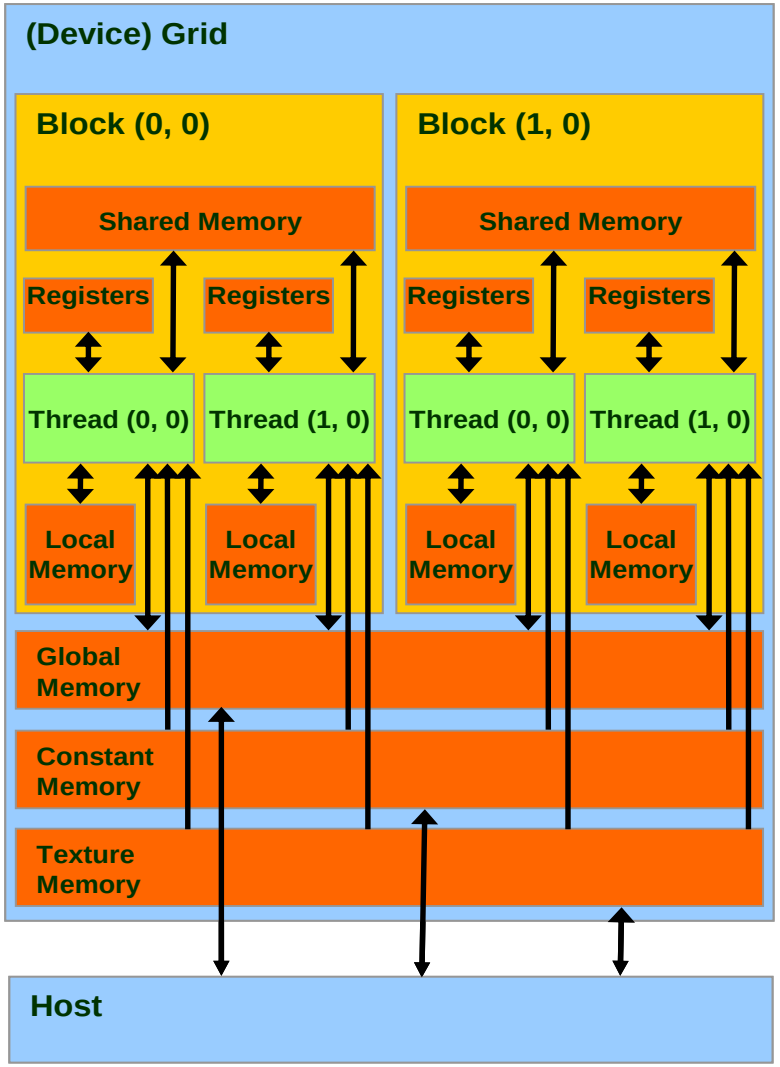
CUDA扩展语言结构

- Declspecs
  - global, device,
  - shared, local, constant
- Keywords
  - threadIdx, blockIdx
  - threadDim, blockDim
- 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
foo<<<100, 10>>> (parameters);
```

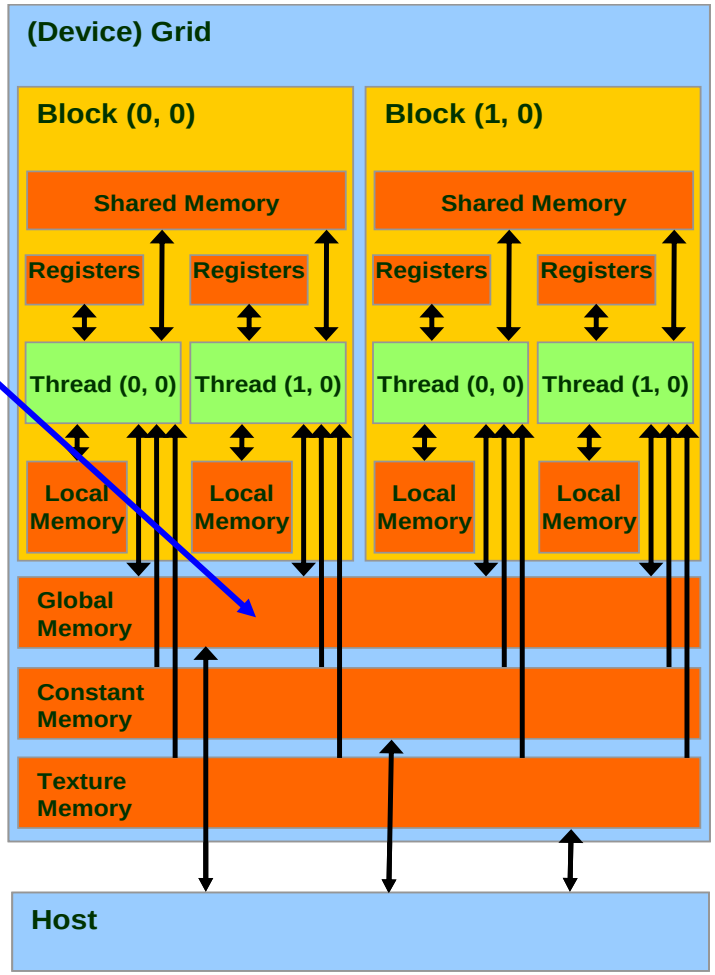
存储器空间

- R/W per-thread
  registers
  - ✓ 1-cycle latency
- R/W per-thread
  local memory
  - ✓ Slow – register spilling to global memory
- R/W per-block
  shared memory
  - ✓ 1-cycle latency
  - ✓ “__shared__”
  - ✓ But bank conflicts may drag down
- R/W per-grid
  global memory
  - ✓ ~500-cycle latency
  - ✓ “__device__”
  - ✓ But coalescing accessing could hide latency
- Read only per-grid
  constant and texture memories
  - ✓ ~500-cycle latency, but cached



GPU Global Memory分配

- cudaMalloc()
  - 分配显存中的global memory
  - 两个参数
    - 对象数组指针和数组尺寸
- cudaFree()
  - 释放显存中的global memory
    - 对象数组指针

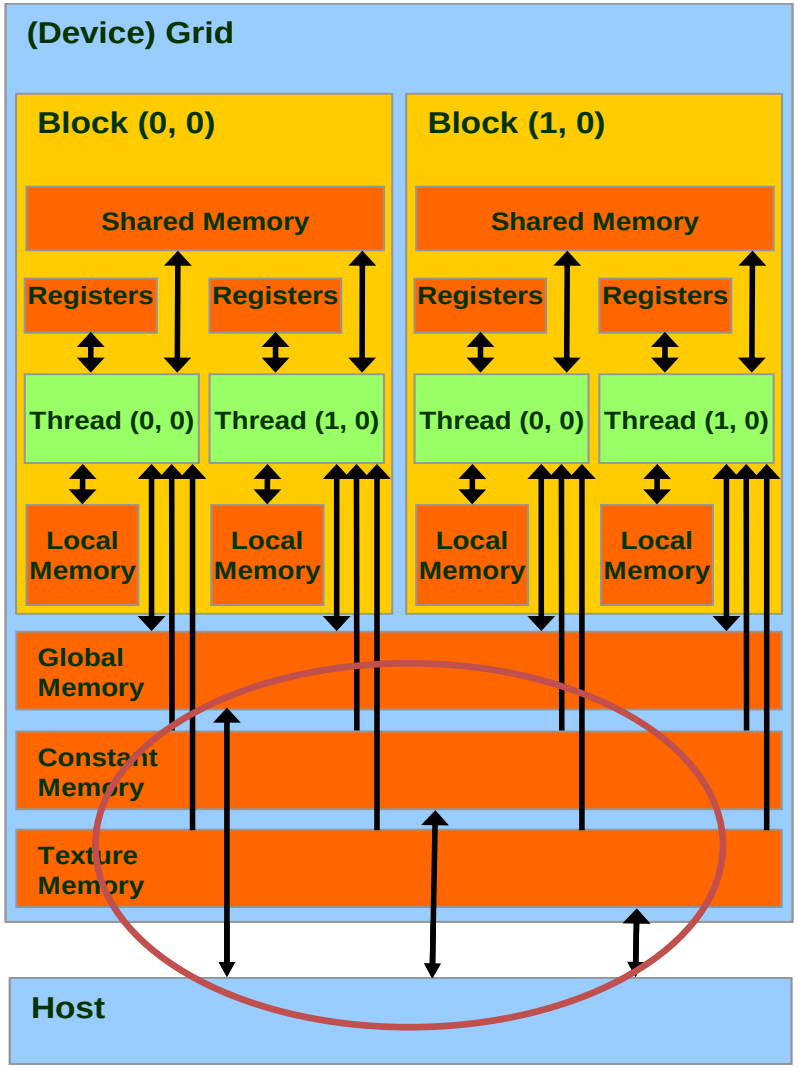



```
int blk_sz = 64;
float* Md;
int size = blk_sz * blk_sz * sizeof(float);

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

Host – Device数据交换

- **cudaMemcpy**()
  - ➢ Memory data transfer
  - ➢ Requires four parameters
    - ✓ Pointer to destination
    - ✓ Pointer to source
    - ✓ Number of bytes copied
    - ✓ Type of transfer
      - – Host to Host
      - – Host to Device
      - – Device to Host
      - – Device to Device

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

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



CUDA函数定义

	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kernel函数
  - ✓ 必须返回void
- **__device__** 函数
  - ✓ 不能用&运算符取地址, 不支持递归调用, 不支持静态变量(static variable), 不支持可变长度参数函数调用

CUDA数学函数

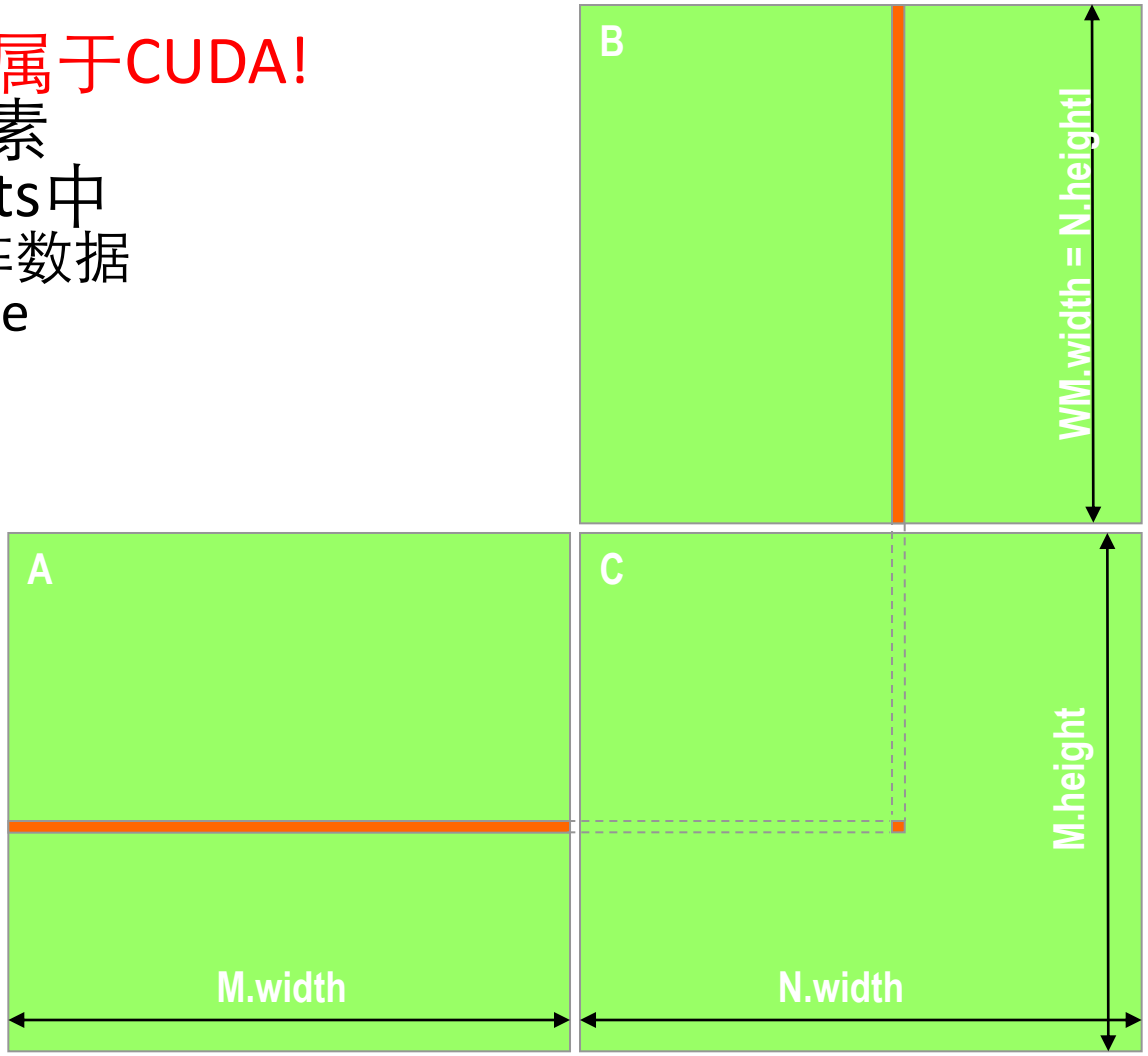
pow, sqrt, cbrt, hypot, exp, exp2, expm1, log, log2, log10, log1p, sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, asinh, acosh, atanh, ceil, floor, trunc, round, etc.

- 只支持标量运算
- 许多函数有一个快速、较不精确的对应版本
  - ✓以“__”为前缀，如__sin()
  - ✓编译开关**-use_fast_math**强制生成该版本的目标码

实例2: 矩阵相乘

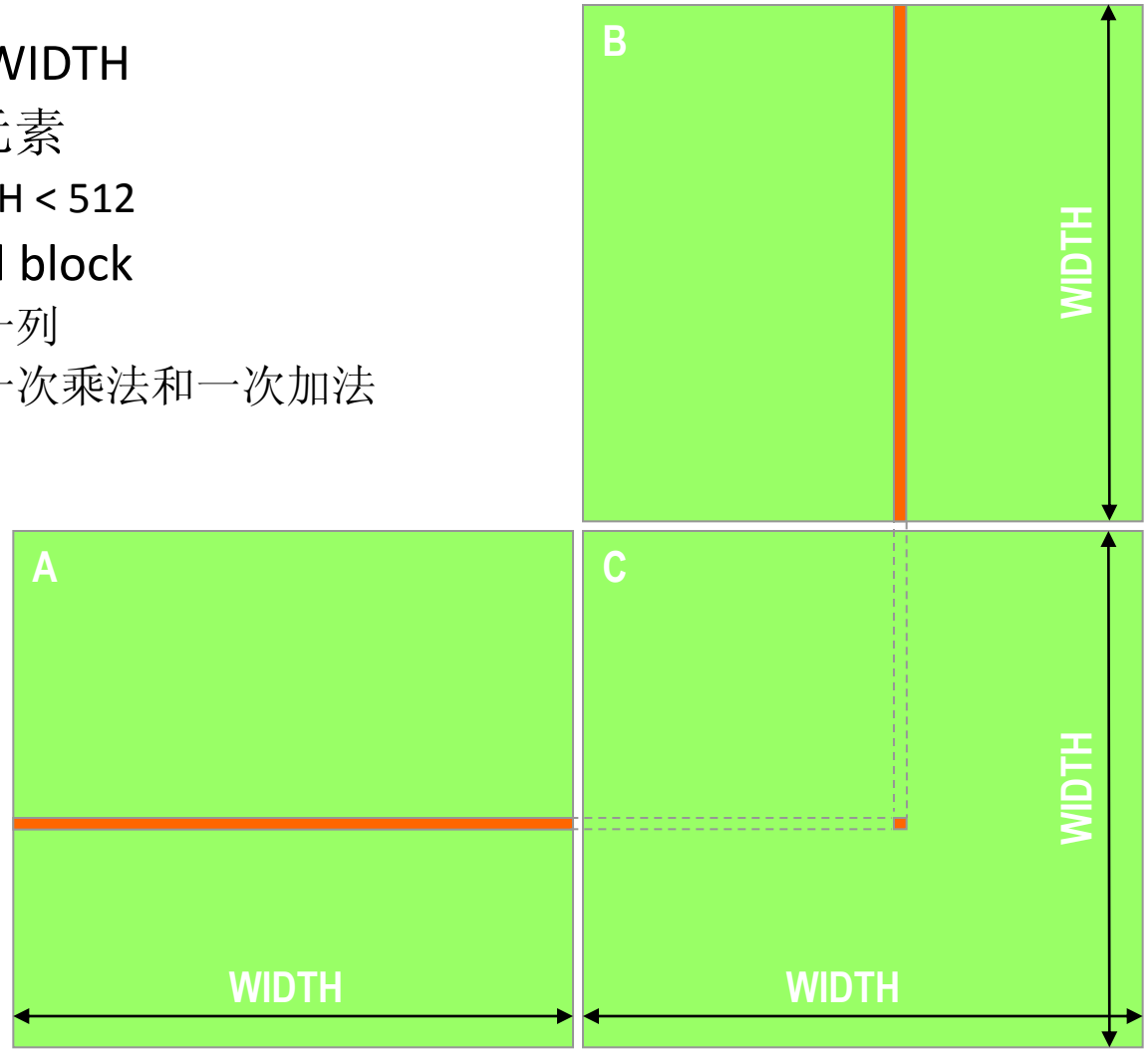
- 矩阵数据类型 – 不属于CUDA!
- width × height个元素
- 矩阵元素在elements中
  - ✓1-D数组存放矩阵数据
  - ✓Row-major storage

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



实例2: 矩阵相乘

- C = A × B of size WIDTH x WIDTH
- 一个线程处理一个矩阵元素
  - 简化: 假定 WIDTH x WIDTH < 512
    - 只需要一个thread block
  - 线程载入A的一行和B的一列
  - A和B的一对相应元素作一次乘法和一次加法



CUDA Implementation – Host Side

```
// Matrix multiplication on the device
void Mul(const Matrix A, const Matrix B, Matrix C)
{
    int size = A.width × A.width × sizeof(float);
    // Load M and N to the device
    float *Ad, *Bd, *Cd;
    cudaMalloc((void**)&Ad, size);      //matrix stored in linear order
    cudaMemcpy(Ad, A.elements, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B.elements, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    cudaMalloc((void**)&Cd, size);
```

CUDA Implementation – Host Side

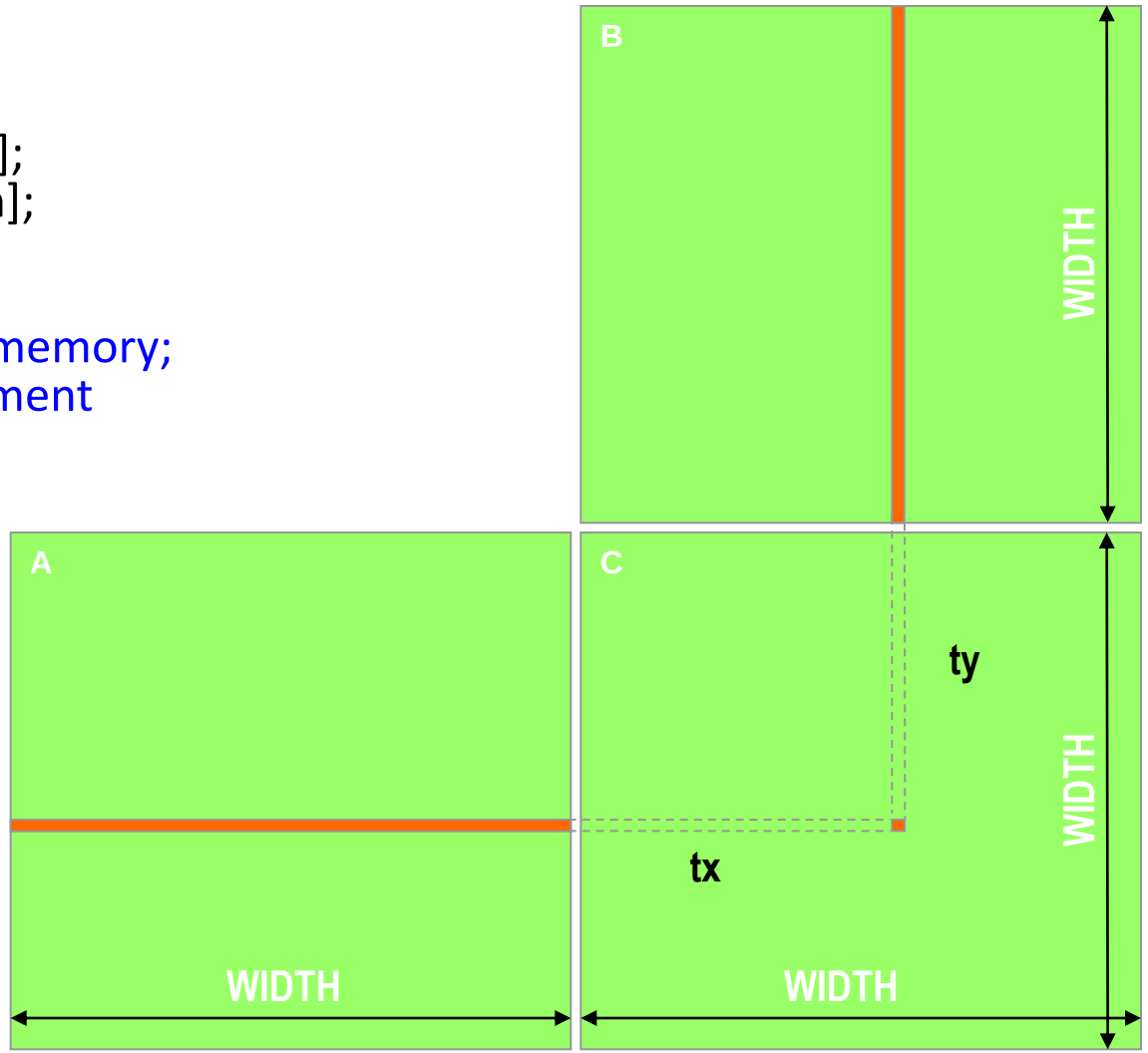
```
// Launch the device computation threads!
   dim3 dimGrid(1);
   dim3 dimBlock(M.width, M.width);
   Muld<<<dimGrid, dimBlock>>>(Ad, Bd, Cd, M.width);

   // Read P from the device
   copyFromDeviceMatrix(C.elements, Cd);
   cudaMemCopy(C, Cd, N * size, cudaMemcpyDeviceToHost);

   // Free device matrices
   cudaFree(Ad);
   cudaFree(Bd);
   cudaFree(Cd);
}
```

CUDA Implementation – Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void Muld (float* Ad, float* Bd, float* Cd, int width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

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

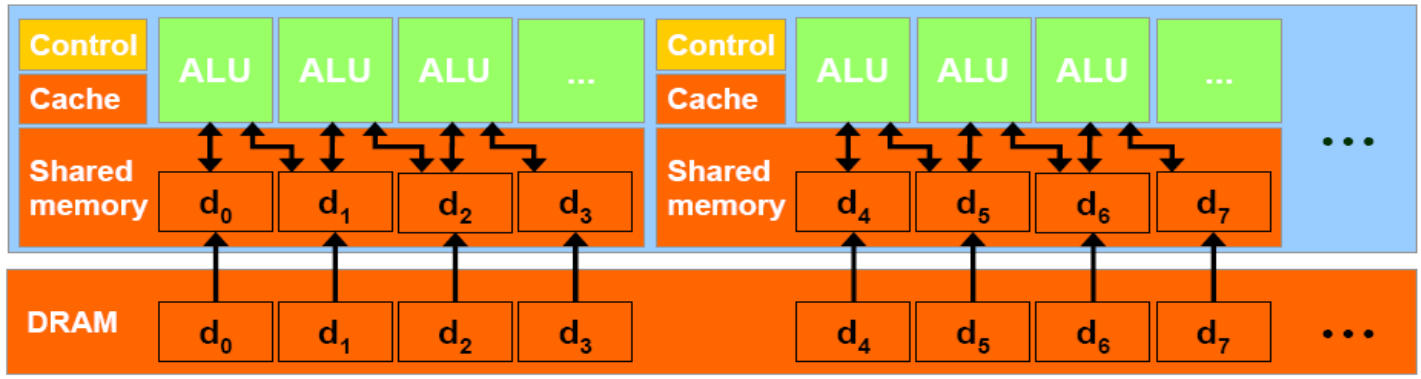
CUDA Implementation – Kernel

```
for (int k = 0; k < width; ++k)
  {
      float ae = Ad[ty * width + k];
      float be = Bd [tx + k * width];
      cvalue += ae * be;
  }
  // Write the matrix to device memory;
  // each thread writes one element
  Cd[ty * width + tx] = cvalue;
}
```



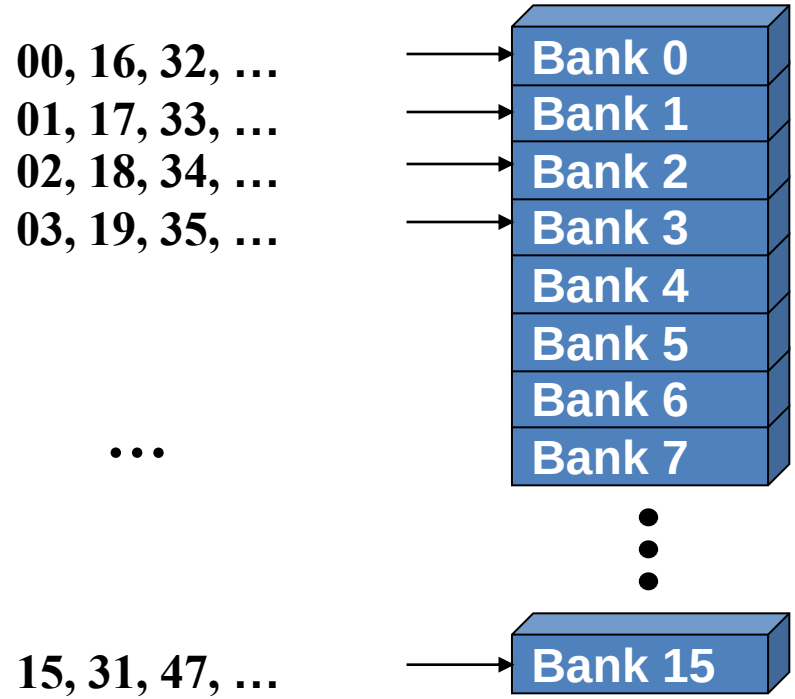
共享存储器(Shared Memory)

- 设置于streaming multiprocessor内部
- 由一个线程块内部全部线程共享
  - ✓完全由软件控制
  - ✓访问一个地址只需要1个时钟周期



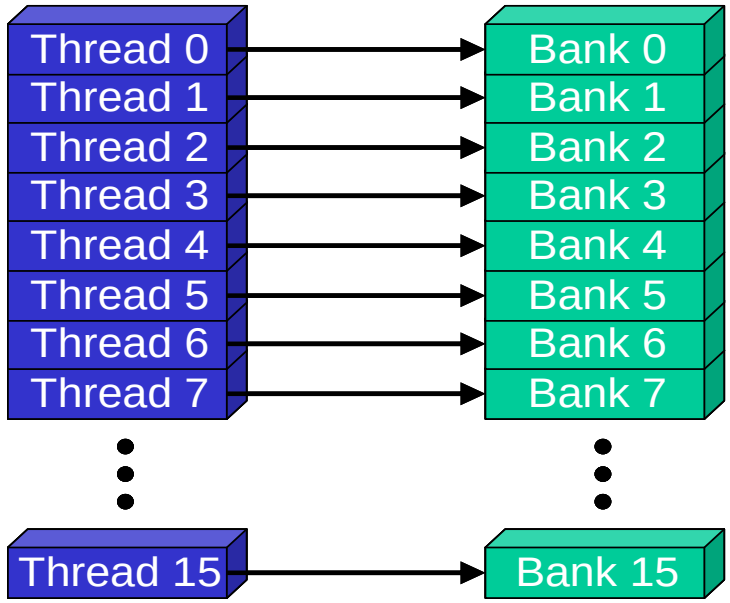
共享存储器结构

- G80的共享存储器组织为16 banks
  - ✓Addressed in 4 bytes
  - ✓Bank ID = 4-byte address % 16
  - ✓相邻4-byte地址映射相邻banks
  - ✓每一bank的带宽为4 bytes per clock cycle
- 对同一bank的同时访问导致bank conflict
  - ✓只能顺序处理
  - ✓仅限于同一线程块内的线程

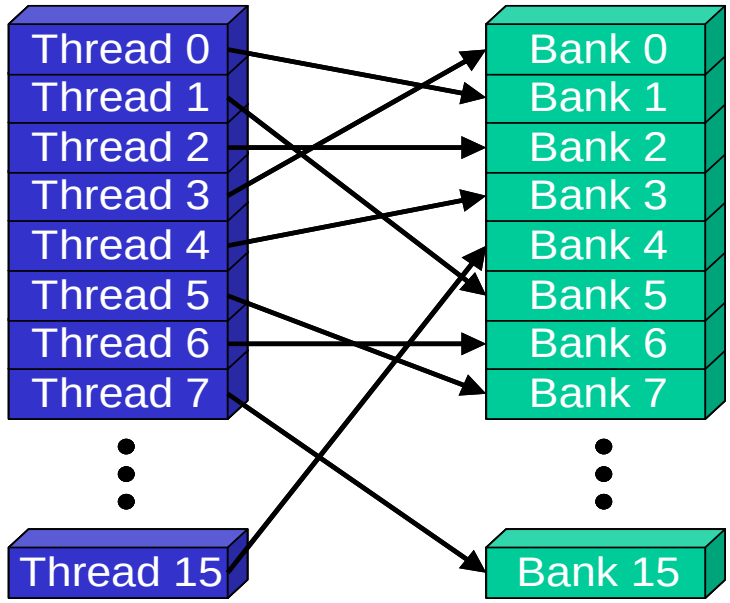


Bank Addressing实例

- No Bank Conflicts
  - Linear addressing
    stride == 1 (s=1)



- No Bank Conflicts
  - Random 1:1 Permutation

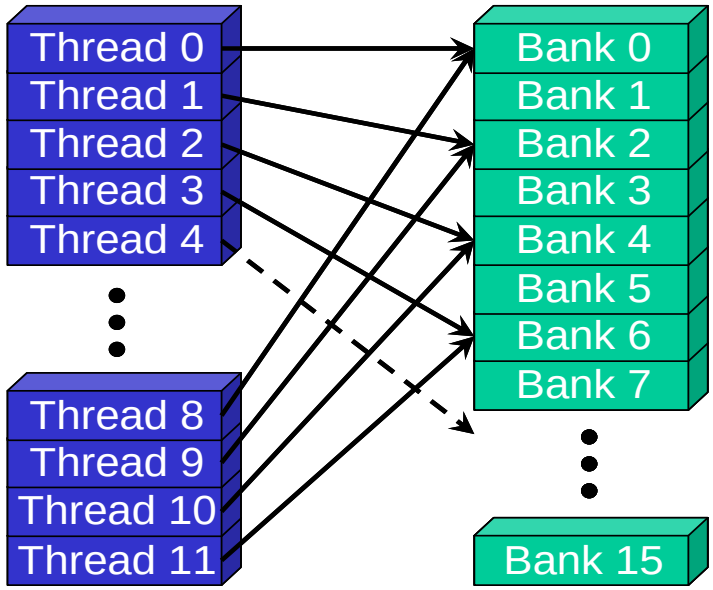


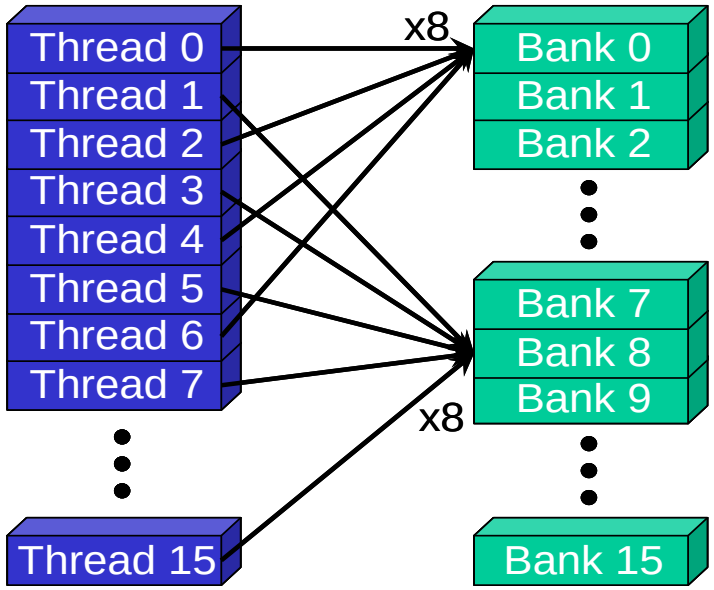

```
__shared__ float shared[256];
float foo = shared[threadIdx.x];
```

Bank Addressing实例

- 2-way bank conflicts
  - ✓ Linear addressing stride == 2 (s=2)

- 8-way bank conflicts
  - ✓ Linear addressing stride == 8 (s=8)



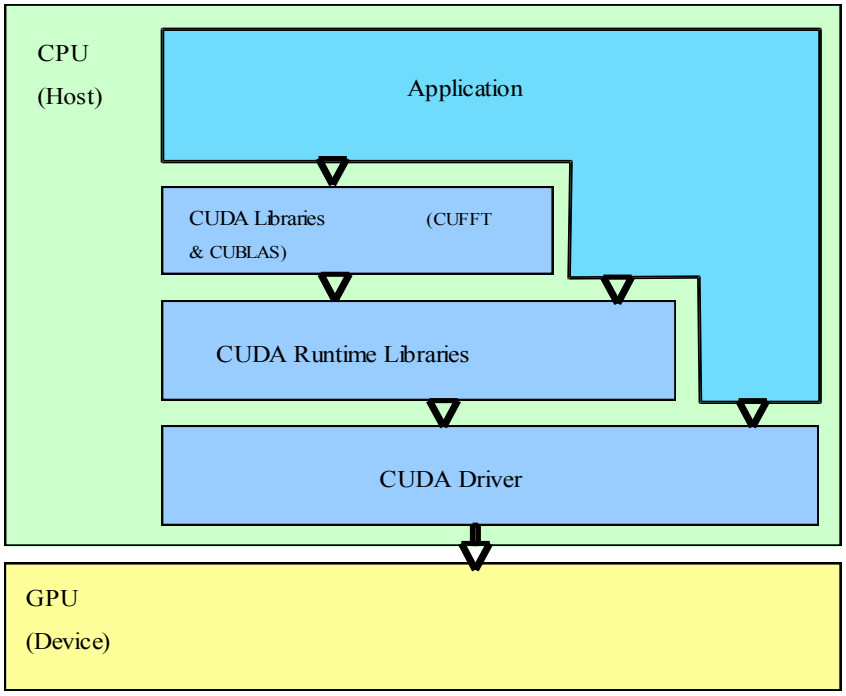



```
__shared__ float shared[256];
float foo = shared[2 * threadIdx.x];
```

```
__shared__ float shared[256];
float foo = shared[8 * threadIdx.x];
```

下载CUDA软件

- http://www.nvidia.cn/object/cuda_get_cn.html
- CUDA driver
  - ✓ 硬件驱动
- CUDA toolkit
  - ✓ 工具包
- CUDA SDK
  - ✓ 程序范例及动态链接库
- CUDA Visual Profiler
  - ✓ 程序剖析工具



CUDA程序的编译(compile)

- CUDA源文件被nvcc处理
  - ✓ nvcc is a compiler driver
- nvcc输出：
  - ✓ PTX (Parallel Thread eXecution)
    - ➢ Virtual ISA for multiple GPU hardware
    - ➢ Just-In-Time compilation by CUDA runtime
  - ✓ GPU binary
    - ➢ Device-specific binary object
  - ✓ Standard C code
    - ➢ With explicit parallelism