NVIDIA CUDA 超大规模并行程序设计训练课程

邓仰东
dengyd@tsinghua.edu.cn
清华大学微电子学研究所





日程安排

- Day 1: CUDA概论
- Day 2: 编程模型
- Day 3: 多线程和存储器硬件
- Day 4: 性能提升





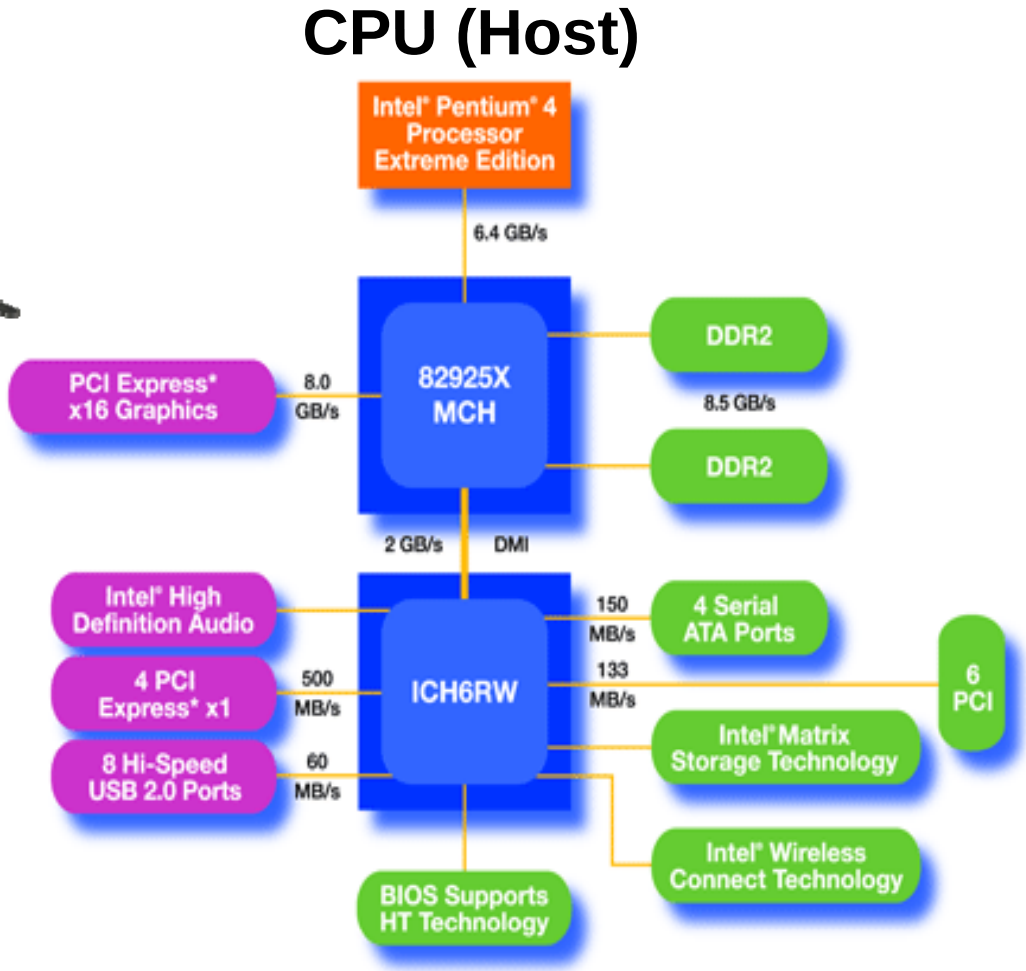
提纲

- 综述
- 并行线程组织
- 存储器模型和内存分配
- CUDA变量和函数
- CUDA程序设计实例
- CUDA libraries和程序设计工具

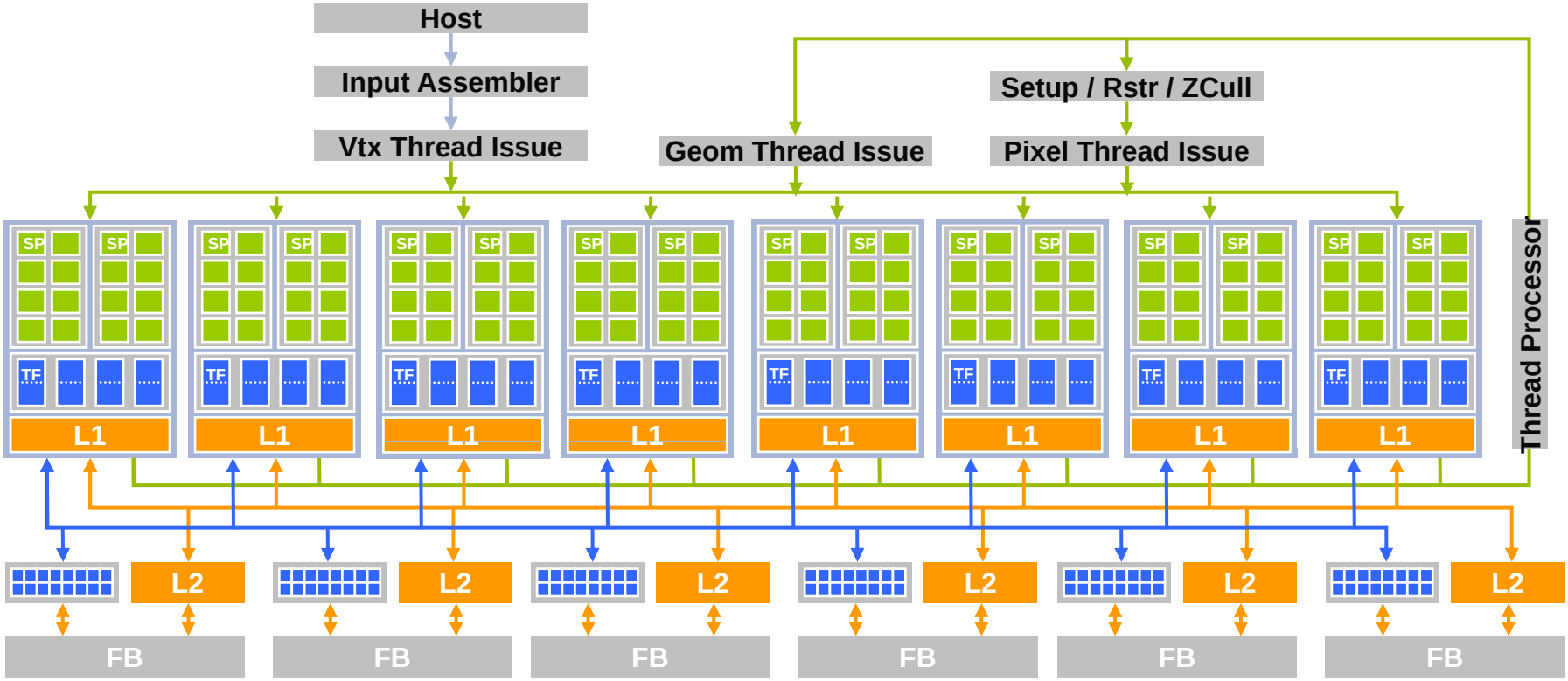
硬件架构



GPU w/
video DRAM
(Device)



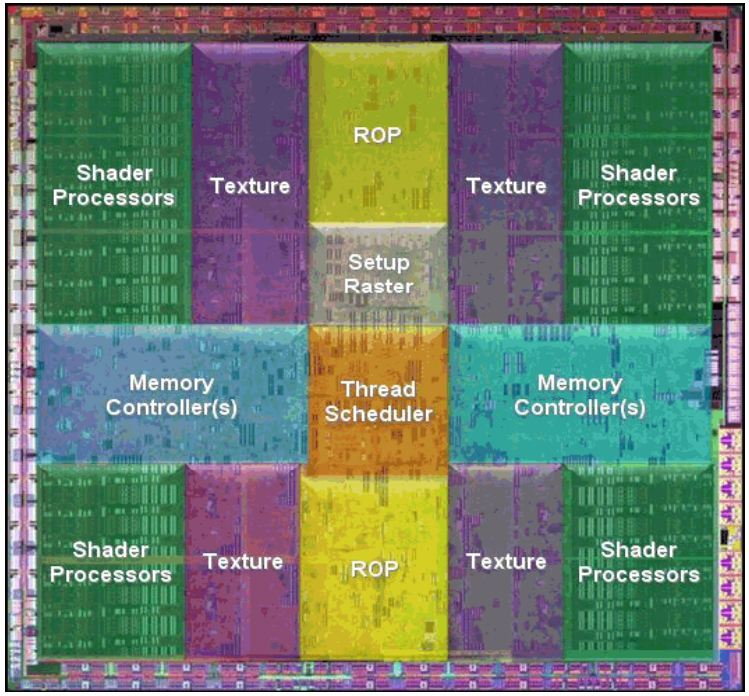
G80架构


Host
Input Assembler
Vtx Thread Issue
Geom Thread Issue
Setup / Rstr / ZCull
Pixel Thread Issue
SP
TF
L1
L2
FB
Thread Processor

GPU硬件—GT200

- 1.4 billion transistors
- TSMC 65nm process
- 583.2 mm^2
- 30 SM's
  - 240 SP's

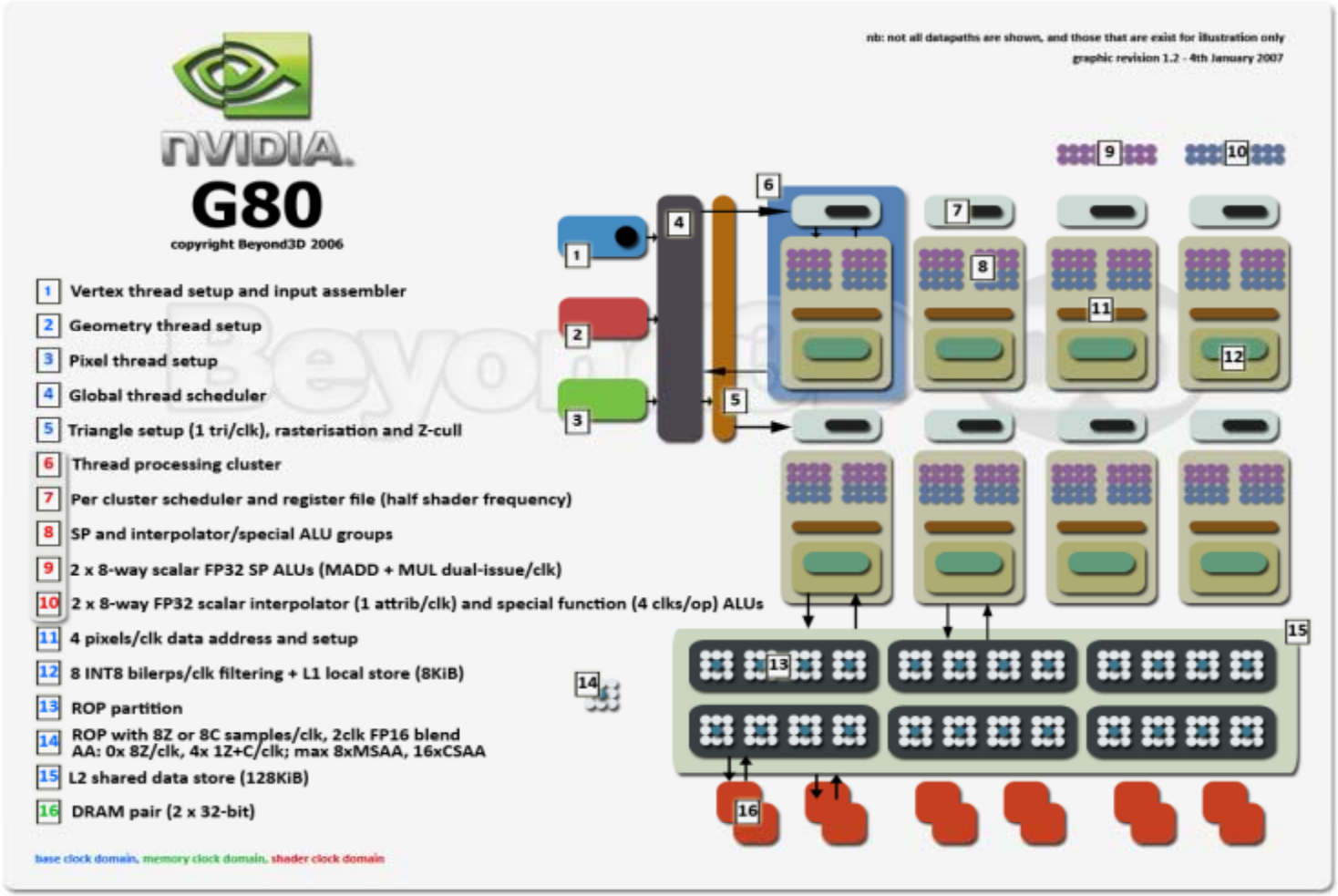




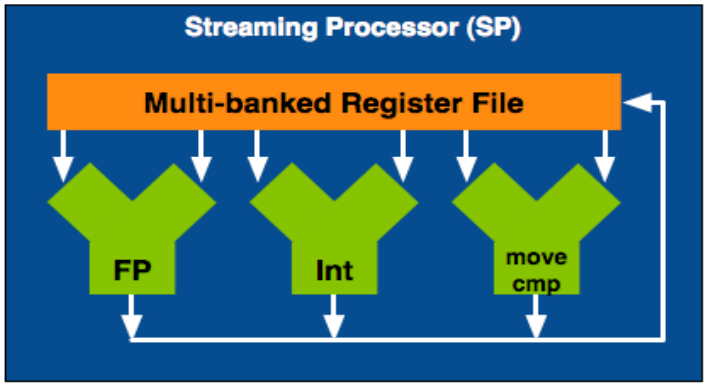
NVIDIA

清華大學 Tsinghua University

G80架构 - 图型处理角度

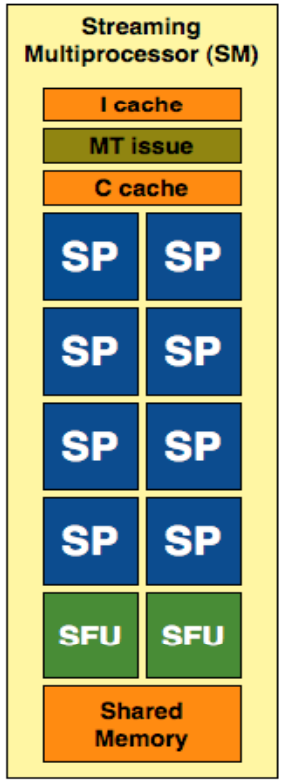


Streaming Processor (SP)



- A fully pipelined, single-issue, in-order microprocessor
  - 2 ALUs and a FPU
  - Register file
  - 32-bit scalar processing
  - No instruction fetch and scheduling
  - No cache

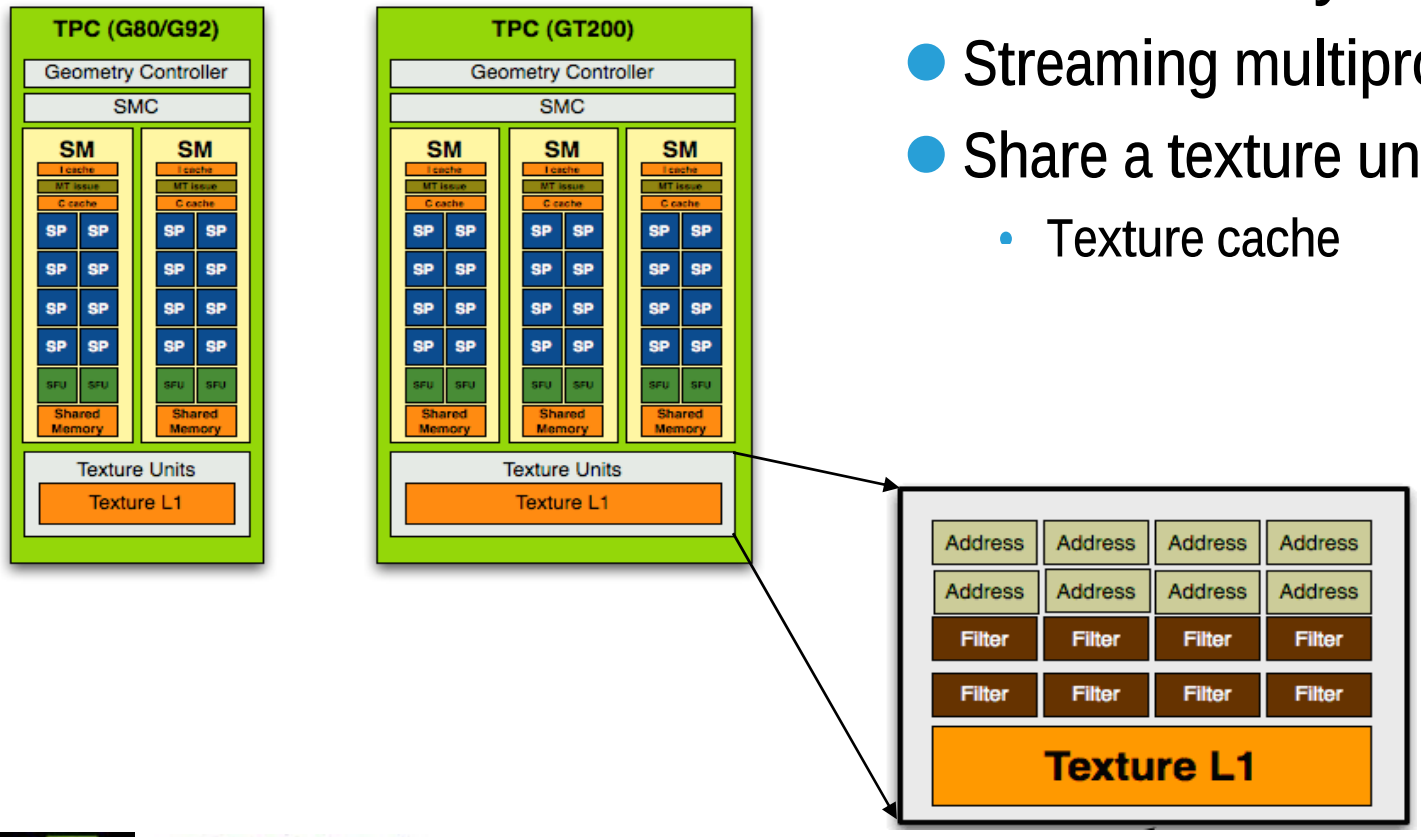
Streaming Multiprocessor (SM)



- An array of SPs
  - 8 streaming processor
  - 2 Special Function Units (SFU)
    - Each is also a fully pipelined, single-issue, in-order microprocessor
    - 4 FPU's
      - > Transcendental operations (e.g. sin, cosin) and interpolation
  - A 16KB read/write shared memory
    - Not a cache
    - But a software-managed data store
  - Multithreading issuing unit
    - Dispatch instructions
  - Instruction and constant cache

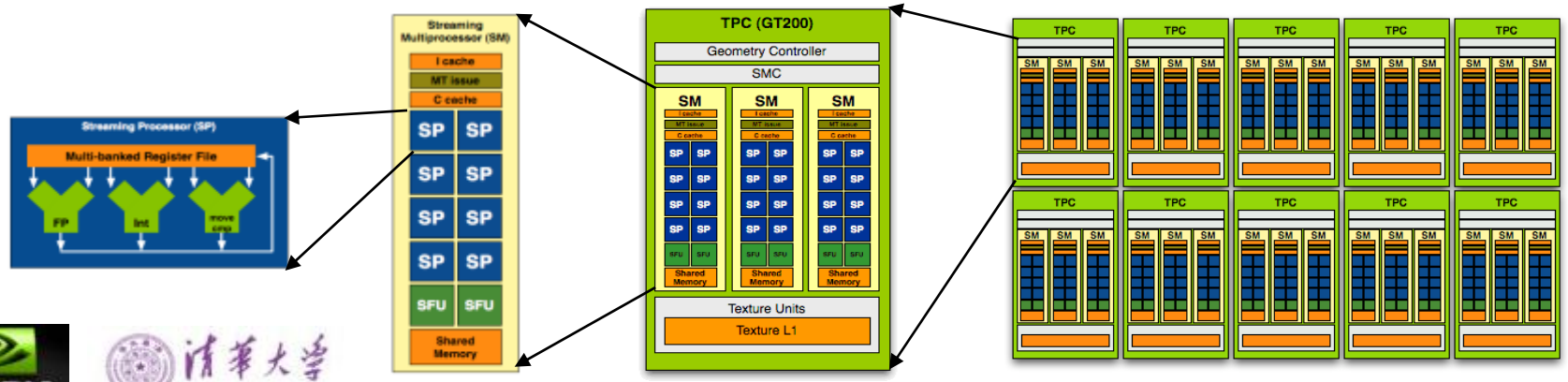
Texture/Processor Cluster (TPC)

- A modular array
  - Streaming multiprocessors
  - Share a texture units
    - Texture cache



Modern GPU = Streaming Processor Array (SPA)

NVIDIA Architecture Comparison	G80	G92	GT200
Streaming Processors per TPC	16	16	24
Texture Address Units per TPC	4	8	8
Texture Filtering Units per TPC	8	8	8
Total SPs	128	128	240
Total Texture Address Units	32	64	80
Total Texture Filtering Units	64	64	80



C扩展

- 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);
```

提纲

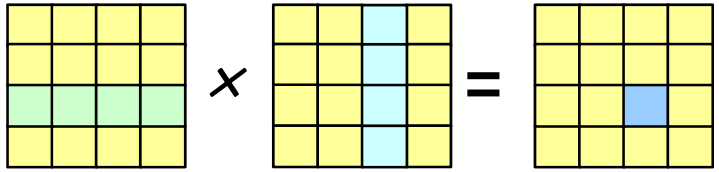
- 综述
- 并行线程组织
- 存储器模型和内存分配
- CUDA变量和函数
- CUDA程序设计实例
- CUDA libraries和程序设计工具

并行性的维度

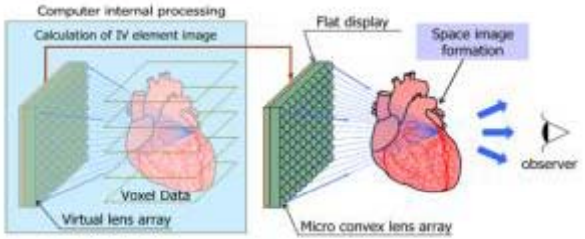
- 1维
  - *y = a + b //y, a, b vectors*

a[0]	*a[1]*	…	*a[n]*
+	+		+
b[0]	*b[1]*	…	*b[n]*
=	=		=
y[0]	*y[1]*	…	*y[n]*

- 2维
  - *P = M × N //P, M, N matrices*

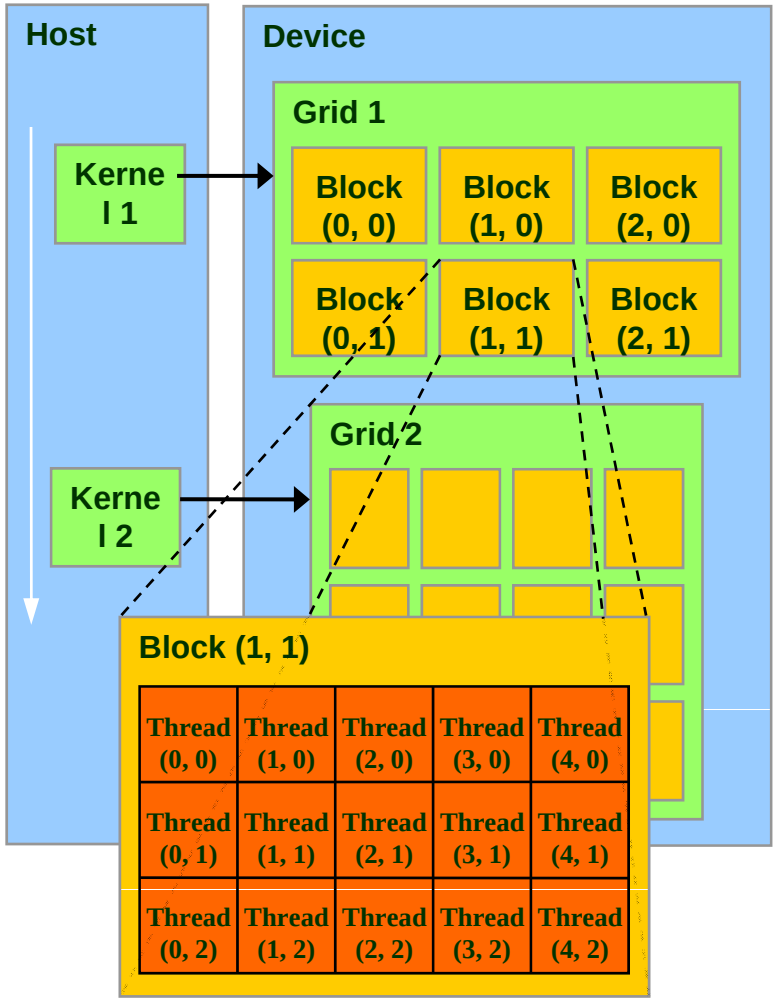


- 3维
  - CT or MRI imaging

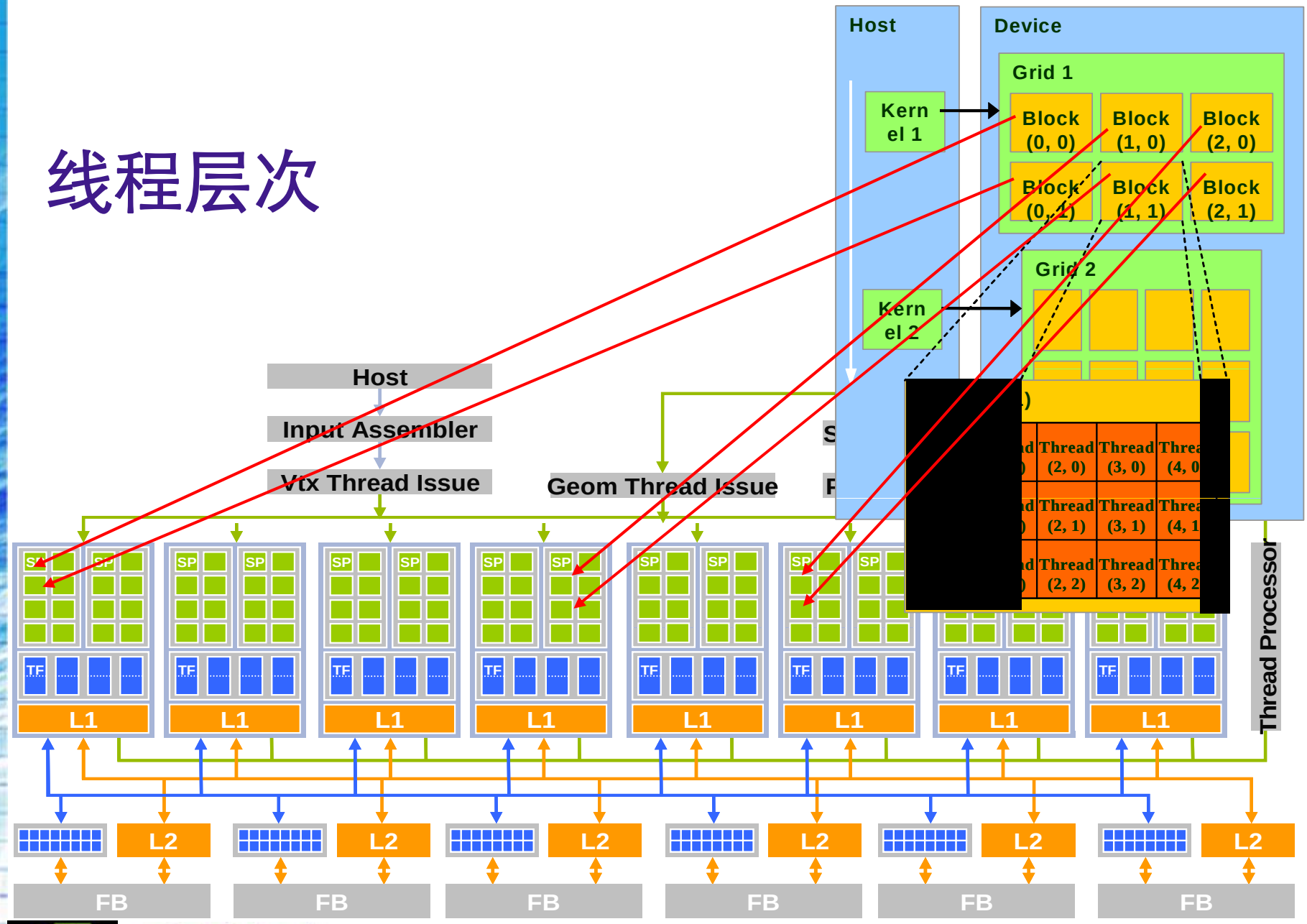


并行线程组织结构

- 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、2维或3维组织
  - 共享全局内存
- Kernel: 在GPU上执行的核心程序
  - One kernel ↔ one grid

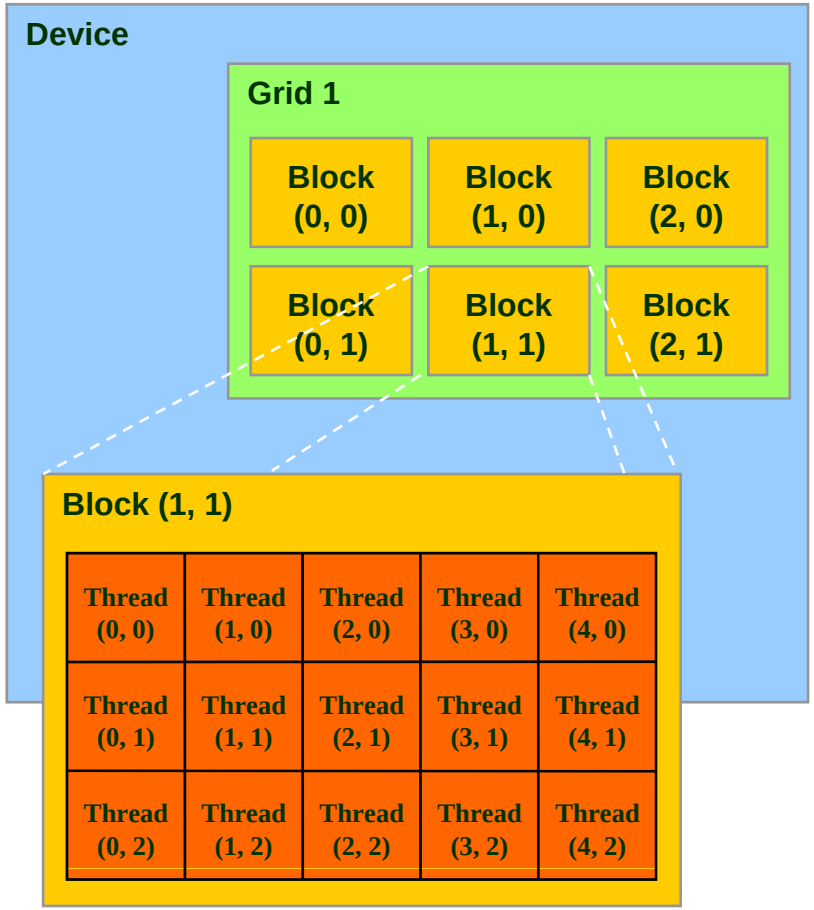


线程层次


Host
Device
Grid 1
Kern el 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Kern el 2
Thread (2, 0)
Thread (3, 0)
Thread (2, 1)
Thread (3, 1)
Thread (2, 2)
Thread (3, 2)
Host
Input Assembler
Vtx Thread Issue
Geom Thread Issue
SP
TF
L1
L2
FB
Thread Processor

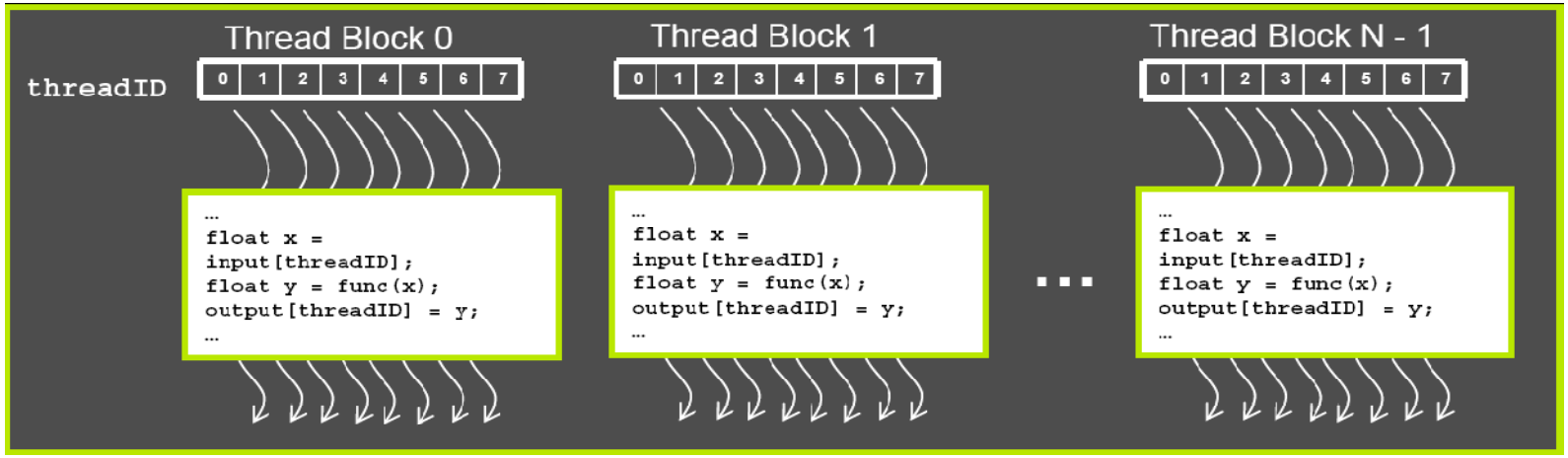
Block and Thread IDs

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



线程组织

- Threads are grouped into thread blocks
- Kernel = a grid of thread blocks
  - Thread blocks的执行没有顺序要求
  - 自适应于具有不同硬件资源的GPU



线程同步

- void __syncthreads();
  - Barrier synchronization
  - 同步thread block之内的所有线程
  - 避免访问共享内存时发生RAW/WAR/WAW 冒险 (hazard)

```
__shared__ float scratch[256];
scratch[threadID] = begin[threadID];
__syncthreads();
int left = scratch[threadID -1];
```

在此等待，直至所有线程到达才开始执行下面的代码

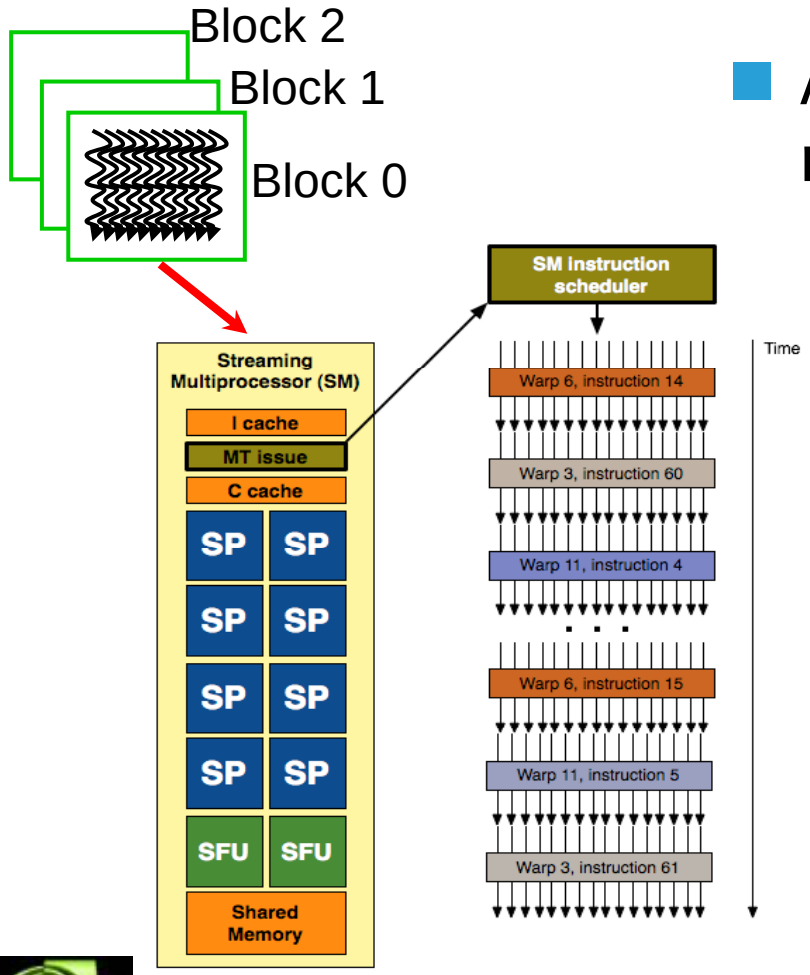
Dead-Lock with __syncthreads

- Dead-lock if
  - Some threads have val larger than threshold
  - And others not

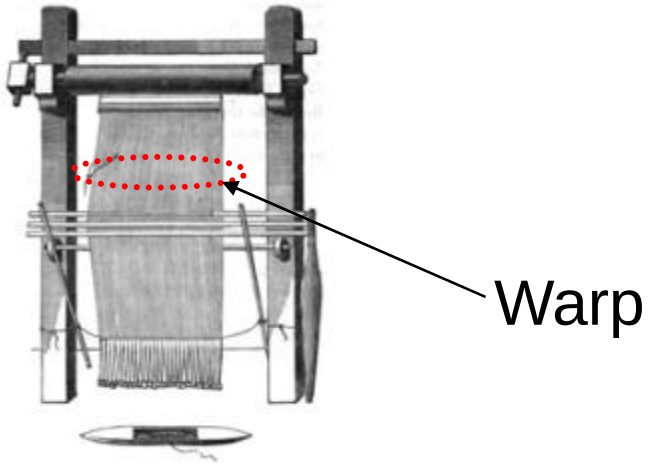
```
__global__ void compute(...)
{
    // do some computation for val
    if( val > threshold )
        return;

    __syncthreads();
    // work with val & store it
    return;
}
```

Thread Execution



- A warp of 32 threads physically running on SM
  - Sharing instructions
  - 4 cycles for 1 warp instruction
    - Why?
  - Dynamically scheduled by SM
    - Executed when operands ready



实例0: Increment Array Elements

```
//CPU program
void inc_cpu(float *a, float *b, int N)
{
    for (intidx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


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

```
//CUDA program
__global__ void inc_gpu(float *a, float *b, int N)
{
    intidx =blockIdx.x* blockDim.x+ threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}

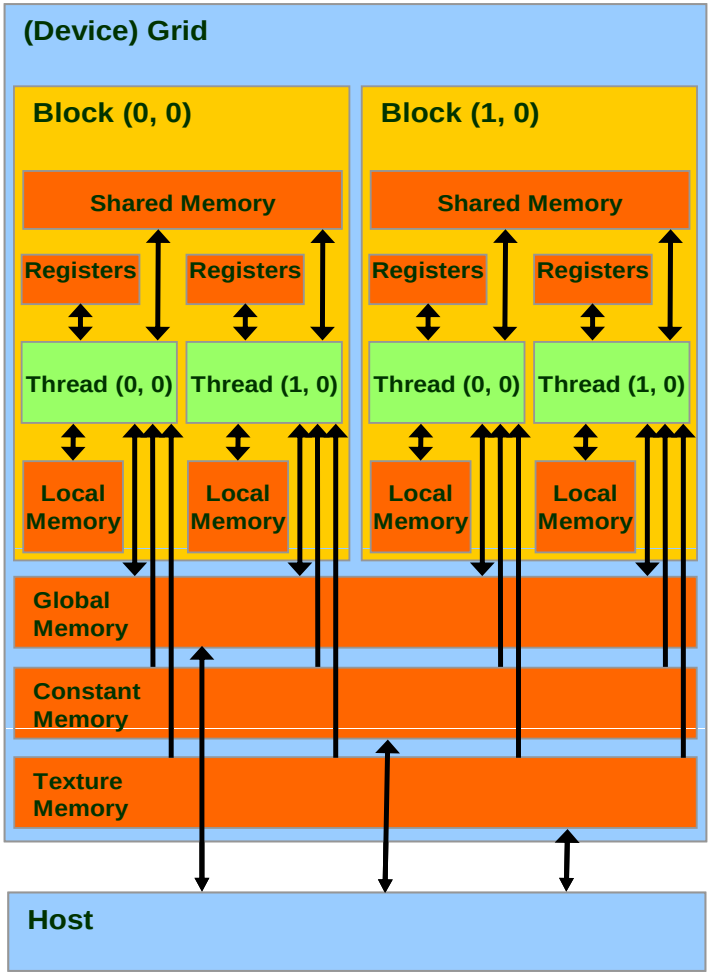
void main()
{
    .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    inc_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

提纲

- 综述
- 并行线程组织
- 存储器模型和内存分配
- CUDA变量和函数
- CUDA程序设计实例
- CUDA libraries和程序设计工具

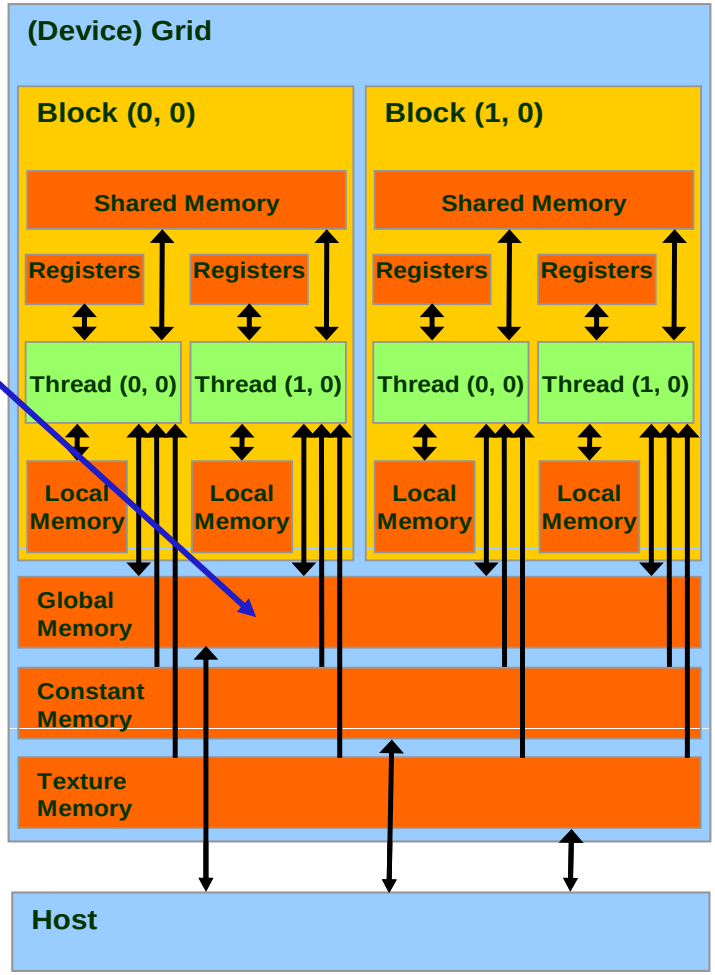
内存空间

- 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
  - But bank conflicts may drag down
- R/W per-grid global memory
  - ~500-cycle latency
  - 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
    - 数组尺寸
- cudaFree()
  - 释放显存中的global memory
    - 对象数组指针



GPU Global Memory分配

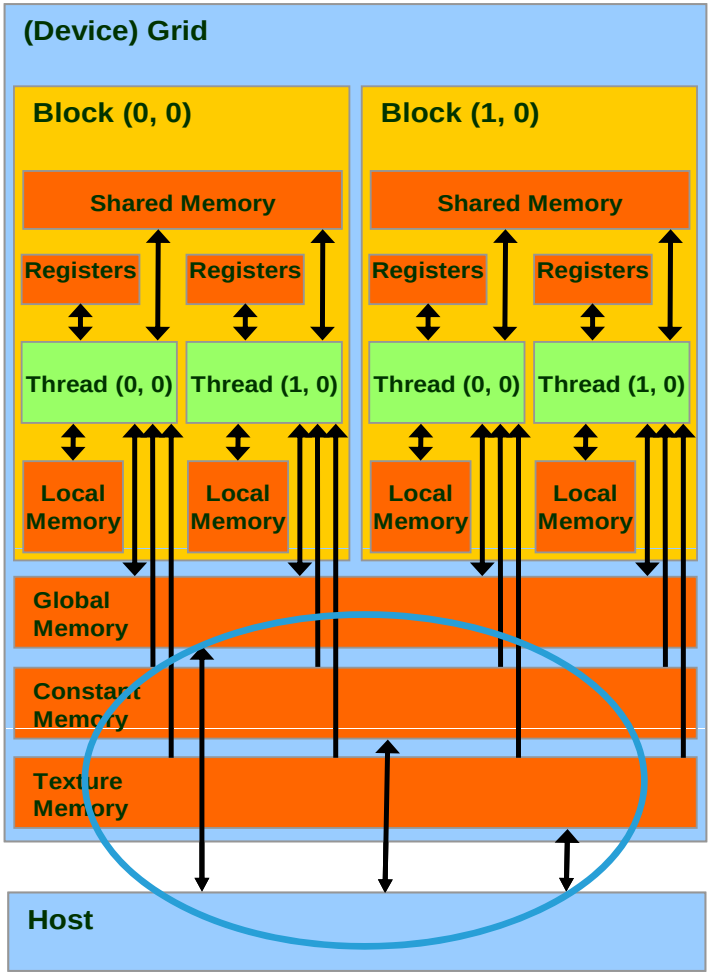
- 代码实例
  - 分配64 × 64单精度浮点数组
  - 数组指针Md
  - 建议用 “d” 表示GPU显存数据结构

```
int BLOCK_SIZE = 64;
float* Md;
int size = BLOCK_SIZE * BLOCK_SIZE * 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



Host – Device数据交换

- 代码实例
  - M.elements: CPU主存
  - Md: GPU显存
  - 符号常数: cudaMemcpyHostToDevice和cudaMemcpyDeviceToHost

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

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

提纲

- 综述
- 并行线程组织
- 存储器模型和内存分配
- CUDA变量和函数
- CUDA程序设计实例
- CUDA libraries和程序设计工具

CUDA引入的新变量

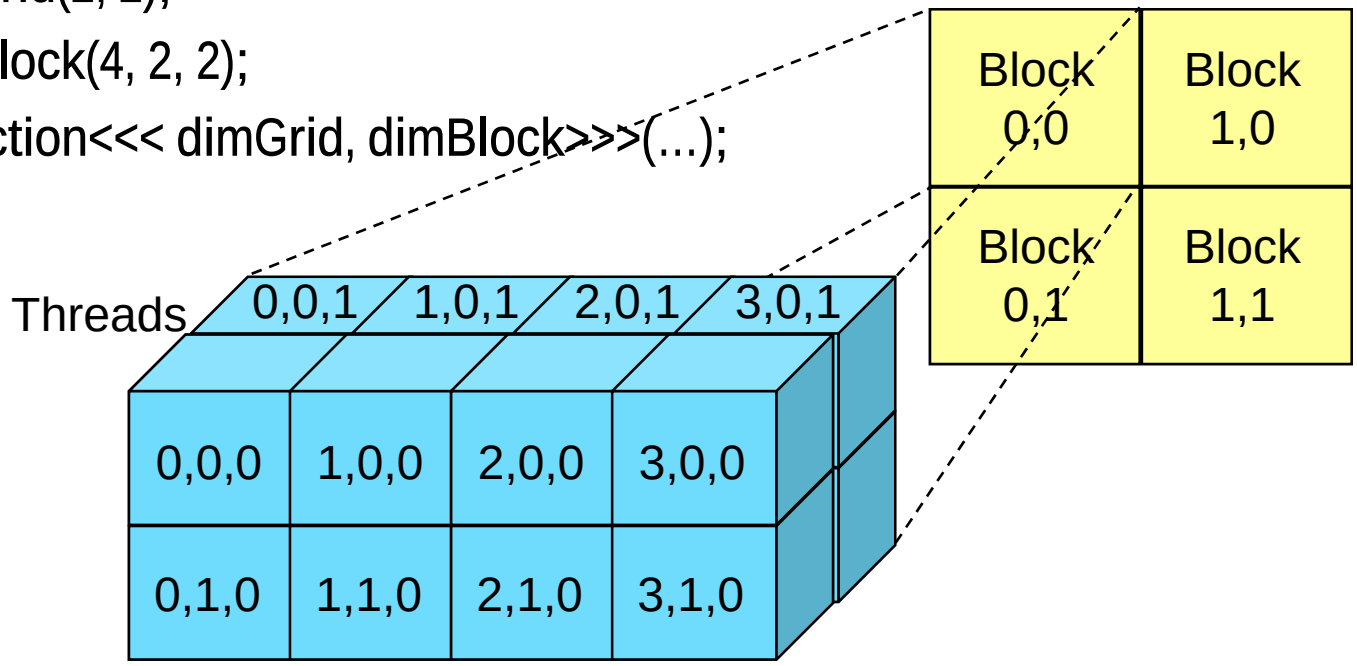
- __device__
  - 储存于GPU上的global memory空间
  - 和应用程序具有相同的生命期(lifetime)
  - 可被grid中所有线程存取, CPU代码通过runtime函数存取
- __constant__
  - 储存于GPU上的constant memory空间
  - 和应用程序具有相同的生命期(lifetime)
  - 可被grid中所有线程存取, CPU代码通过runtime函数存取
- __shared__
  - 储存于GPU上thread block内的共享存储器
  - 和thread block具有相同的生命期(lifetime)
  - 只能被thread block内的线程存取
- Local变量
  - 储存于SM内的寄存器和local memory
  - 和thread具有相同的生命期(lifetime)
  - Thread私有

CUDA矢量数据类型

- Built-in矢量数据类型
  - int1, int2, int3, int4, float1, float2, float3, float4, …
    - 例如: **int4 iv(1, 2, 3, 4)**
      - > **iv.w = 1, iv.x = 2, iv.y = 3, iv.z = 4**
- Texture type
  - texture<Type, Dim, ReadMode> texRef;
    - Type: integer or float
    - Dim: 1, 2, or 3
    - ReadMode:
      - > cudaReadModeNormalizedFloat
      - > cudaReadModeElementType

Built-in dim3 Type

- 定义grid和thread block的组织
  - dim3 dimGrid(2, 2);
  - dim3 dimBlock(4, 2, 2);
  - kernelFunction<<< dimGrid, dimBlock>>>(...);



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__和__host__ 可以组合使用
  - 则被定义的函数在CPU和GPU上都被编译

CUDA函数定义

- __device__ 函数不能用&运算符取地址
- 限制:
  - 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
    - type va_list(stdarg.h)
    - double average(int count, ...)

Kernel函数调用

- A kernel function must be called with an execution configuration:

```
__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 >>>(...);
```

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
- 每个多处理器包含两个超越函数计算单元

提纲

- 综述
- 并行线程组织
- 存储器模型和内存分配
- CUDA变量和函数
- CUDA程序设计实例
- CUDA libraries和程序设计工具

实例1: SAXPY (Sum of Alpha X Plus Y)

```
void main(){          //compute vector sum S = alpha*X + Y
    float *S = …, *X = …, *Y;              //allocate host (CPU) memory
    int N = …;        //length of vector

    float *Xd, *Yd, *Sd;          //allocate device (GPU) memory
    cudaMalloc( (void**) &Xd, N * sizeof(float));
    cudaMalloc( (void**) &Yd, N * sizeof(float));
    cudaMalloc( (void**) &Sd, N * sizeof(float));

    //copy host memory to device
    cudaMemCopy(Xd, X, N * sizeof(float), cudaMemcpyHosttoDevice);
    cudaMemCopy(Yd, Y, N * sizeof(float), cudaMemcpyHosttoDevice);

    //execute the kernel on N/256 blocks of threads
    saxpy<<<N/256, 256>>>(Xd, Yd, Sd, alpha, N);

    cudaMemCopy(S, Sd, N * sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(Xd); cudaFree(Yd); cudaFree(Sd);              //free memory
}
```

实例1: SAXPY kernel

```
//compute vector sum S = alpha*X + Y
//each thread performs a pair-wise addition
__global__ void saxpy(float *Xd, float *Yd, float *Sd, float alpha, int
    N)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i < N)
            Sd[i] = alpha * Xd[i] + Yd[i];
}
```

PTX汇编代码

```
cvt.u32.u16     $blockid, %ctaid.x;     // Calculate i from thread/block IDs
cvt.u32.u16     $blocksize, %ntid.x;
cvt.u32.u16     $tid, %tid.x;
mad24.lo.u32    $i, $blockid, $blocksize, $tid;
ld.param.u32    $n, [N];                // Nothing to do if n ≤ i
setp.le.u32     $p1, $n, $i;
@$p1 bra        $L_finish;

mul.lo.u32      $offset, $i, 4;         // Load y[i]
ld.param.u32    $yaddr, [Y];
add.u32         $yaddr, $yaddr, $offset;
ld.global.f32   $y_i, [$yaddr+0];
ld.param.u32    $xaddr, [X];            // Load x[i]
add.u32         $xaddr, $xaddr, $offset;
ld.global.f32   $x_i, [$xaddr+0];

ld.param.f32    $alpha, [ALPHA];        // Compute and store alpha*x[i] + y[i]
mad.f32         $y_i, $alpha, $x_i, $y_i;
st.global.f32   [$yaddr+0], $y_i;

$L_finish:      exit;
```

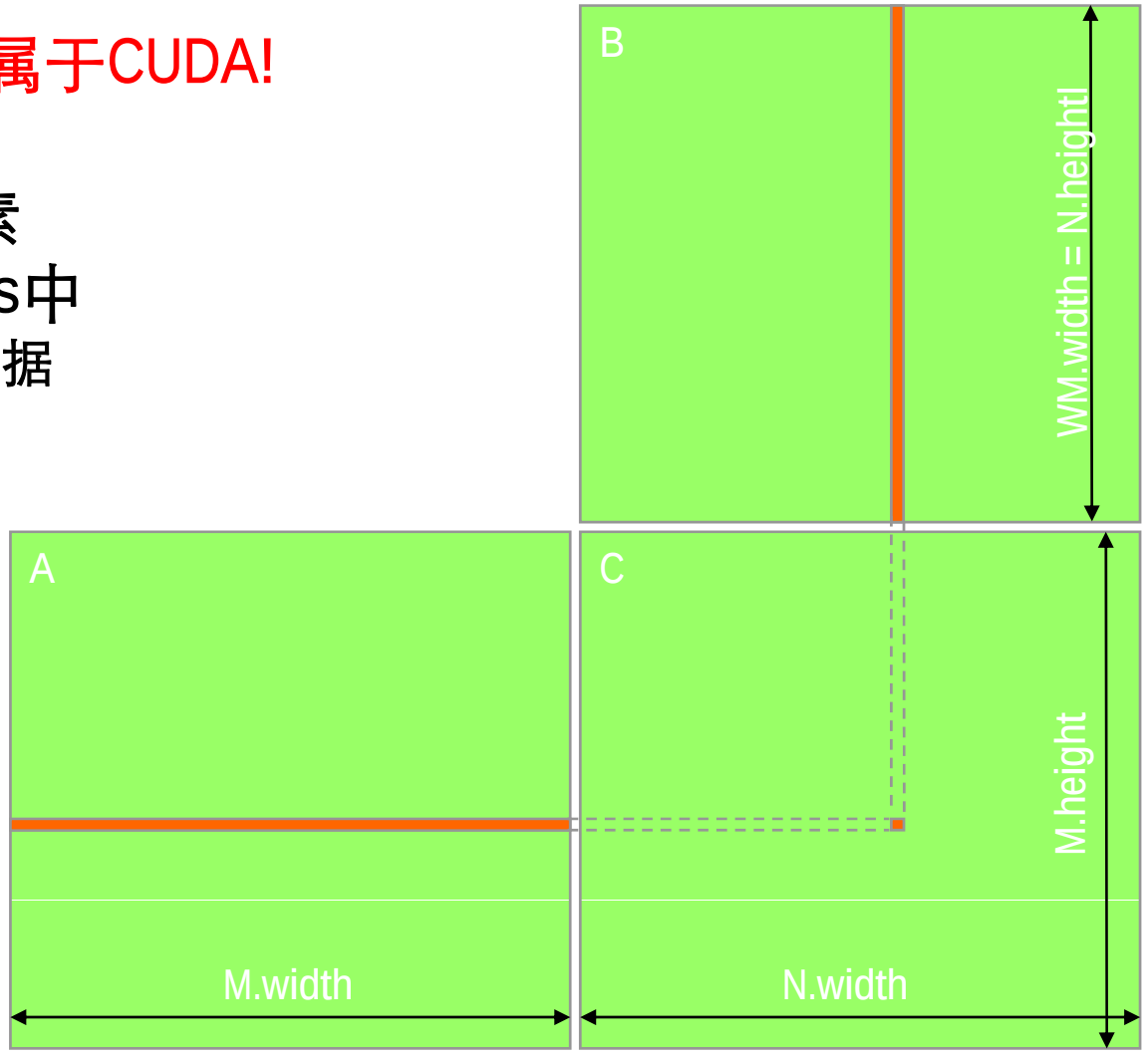
threadIdx.x + blockDim.x * blockIdx.x

if(i < N)
  Sd[i] = alpha * Xd[i] + Yd[i];

实例2: 矩阵相乘

- 矩阵数据类型 – 不属于CUDA!
- 单精度浮点数
- 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

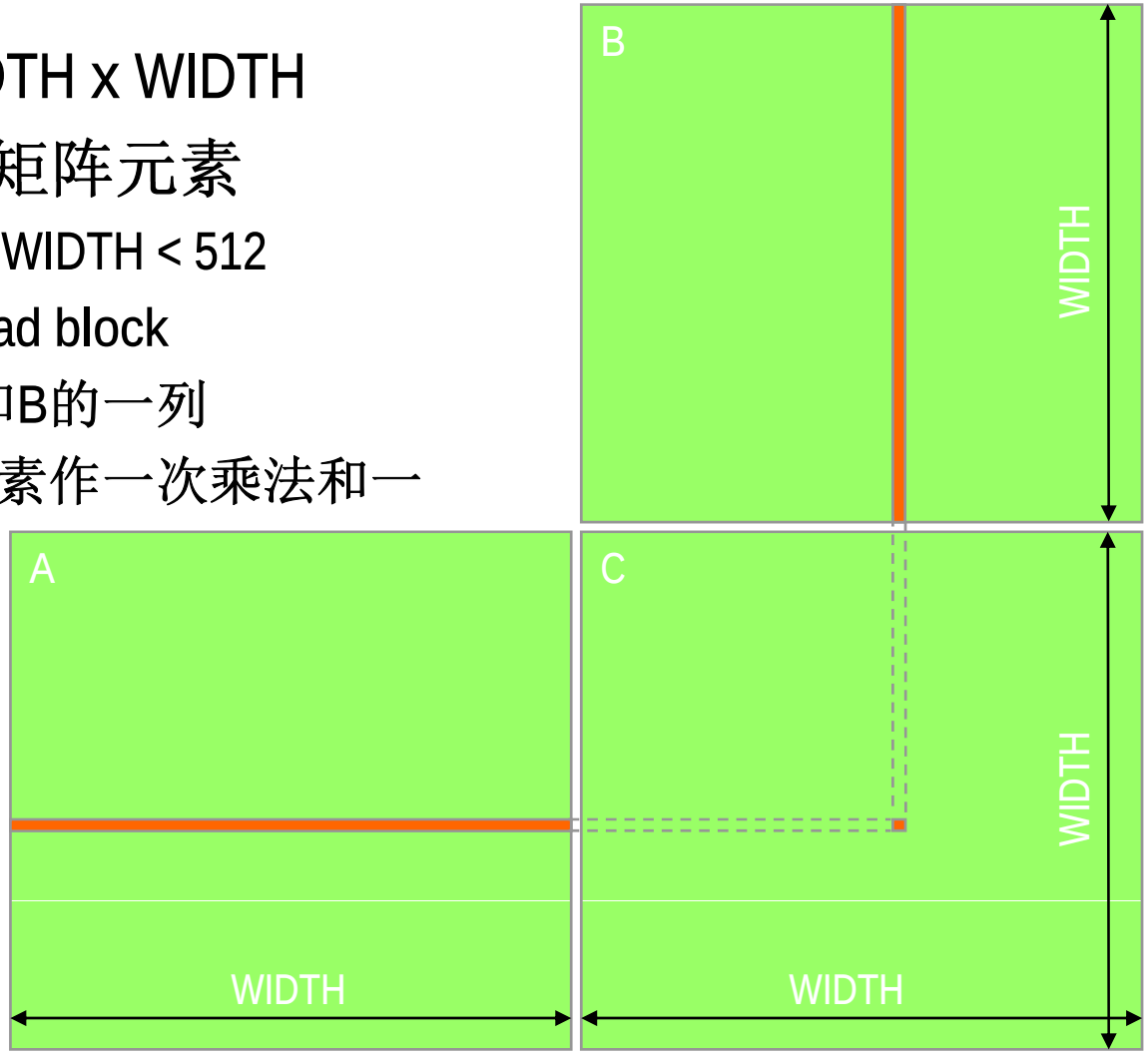
```
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```



NVIDIA
清華大學 Tsinghua University

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



NVIDIA
清華大學 Tsinghua University

CUDA Implementation – Main()

```
int main(void) {        //host side
    // Allocate and initialize the matrices
    Matrix  A  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  B  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  C  = AllocateMatrix(WIDTH, WIDTH, 0);

    // C = A × B on the device
    Mul(A, B, C);

    // Free matrices
    FreeMatrix(A);
    FreeMatrix(B);
    FreeMatrix(C);
    return 0;
}
```

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);
  MuId<<<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);
}
```

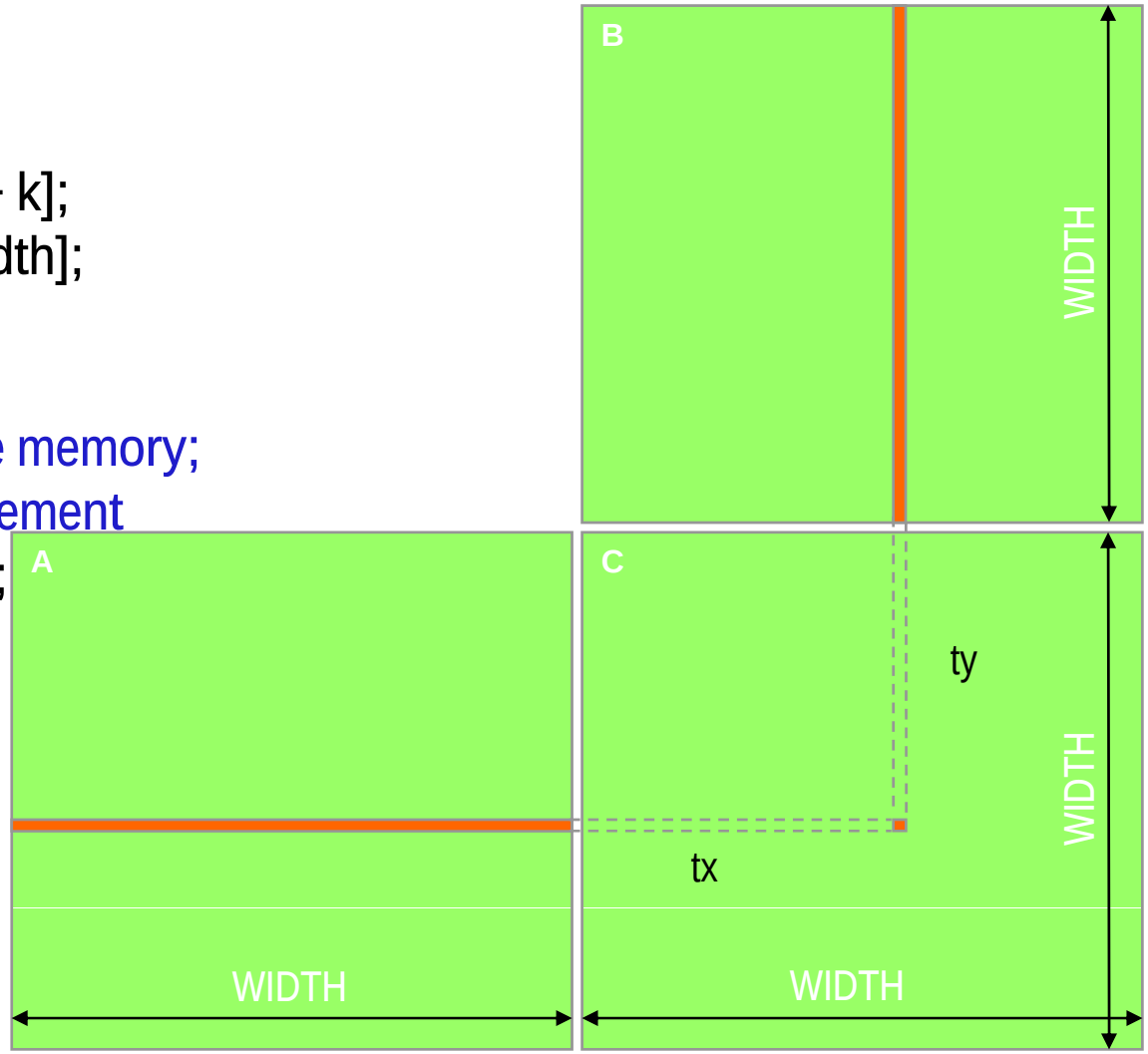
CUDA Implementation – Kernel

```
// 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;
```

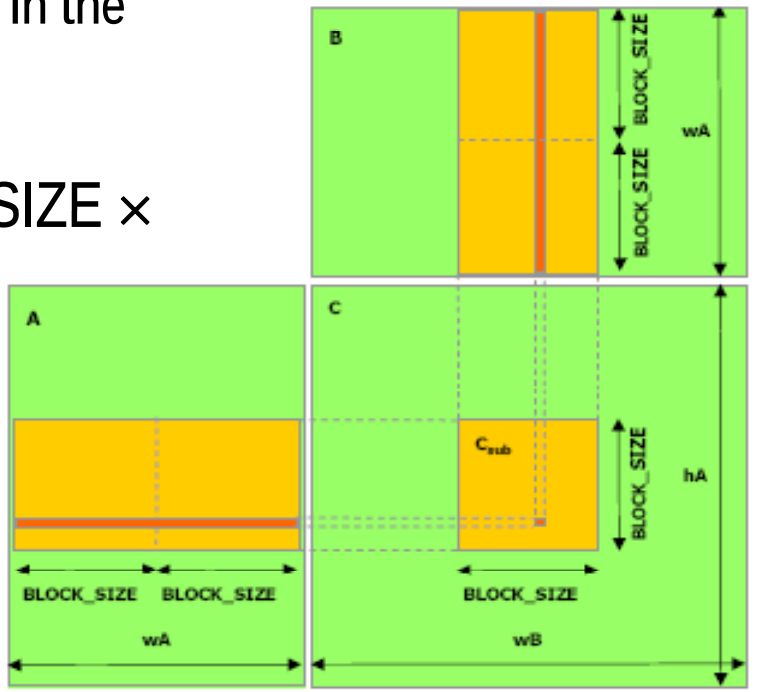
CUDA Implementation – Kernel

```
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;
}
```



实例3: 任意尺寸矩阵相乘

- A 2D thread block
  - Computes a BLOCK_SIZE × BLOCK_SIZE sub-matrix of the result matrix
    - Each has $(BLOCK_SIZE)^2$ threads
  - Use shared memory to store elements in the corresponding blocks of A and B
    - Shared by all threads in thread block
- Generate a 2D grid of WA/BLOCK_SIZE × WB/BLOCK_SIZE blocks



CUDA Implementation - Host

```
// Thread block size
#define BLOCK_SIZE 16
// Forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);

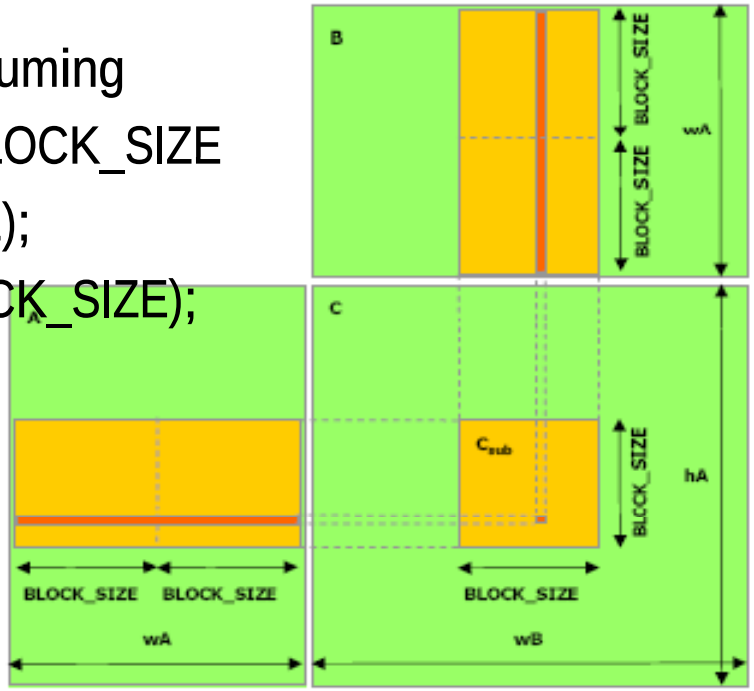
// Host multiplication function
void Mul(const Matrix A, const Matrix B, Matrix C) {
    // Load A and B to the device
    float *Ad, *Bd;
    size = A.height * A.width * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);

    size = B.height * B.width * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
```

CUDA Implementation - Host

```
// Allocate P on the device
float* Cd;
size = A.height * B.width * sizeof(float);
cudaMalloc((void**)&Cd, size);

// Compute the execution configuration assuming
// the matrix dimensions are multiples of BLOCK_SIZE
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(wB / BLOCK_SIZE, hA / BLOCK_SIZE);
```



NVIDIA
清華大學 Tsinghua University

CUDA Implementation - Host

```
    // Launch the device computation
    MuId<<<dimGrid, dimBlock>>>(Ad, Bd, A.width, B.width, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

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





CUDA Implementation - Kernel

```
// Device multiplication function called by Mul()
// Compute C = A × B
// wA is the width of A and wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x; int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x; int ty = threadIdx.y;
```

CUDA Implementation - Kernel

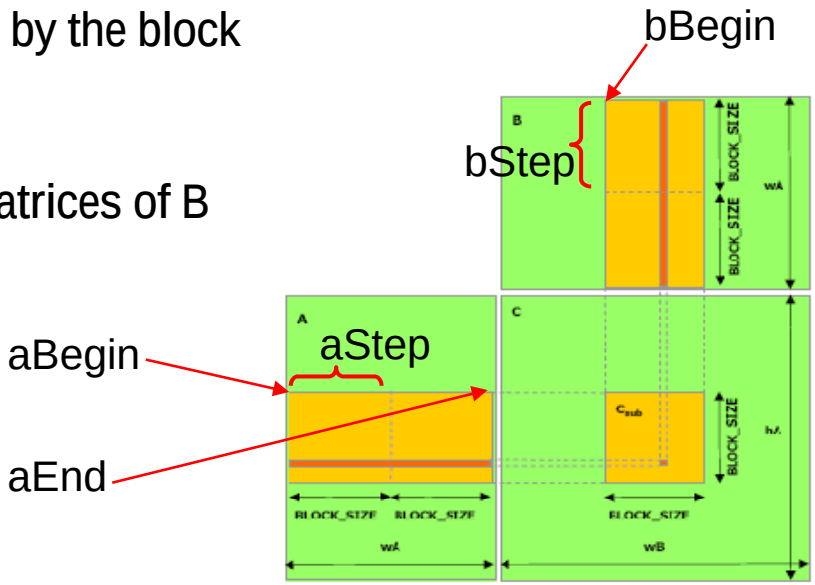
```
// Index of the first sub-matrix of A processed by the block
int aBegin = wA * BLOCK_SIZE * by;

// Index of the last sub-matrix of A processed by the block
int aEnd = aBegin + wA - 1;

// Step size used to iterate through the sub-matrices of A
int aStep = BLOCK_SIZE;

// Index of the first sub-matrix of B processed by the block
int bBegin = BLOCK_SIZE * bx;

// Step size used to iterate through the sub-matrices of B
int bStep = BLOCK_SIZE * wB;
```

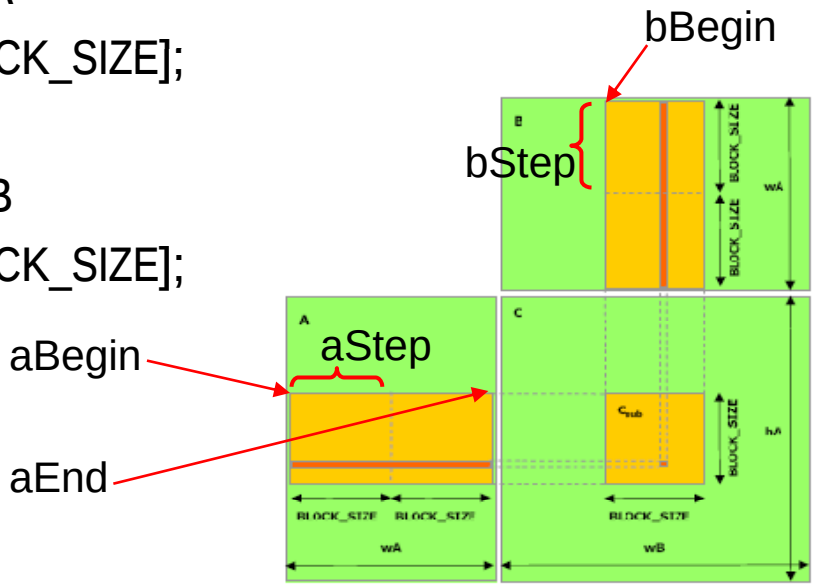


CUDA Implementation - Kernel

```
// The element of the block sub-matrix that is computed by the thread
float Csub = 0;

// Loop over all the sub-matrices of A and B required to
// compute the block sub-matrix
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
```



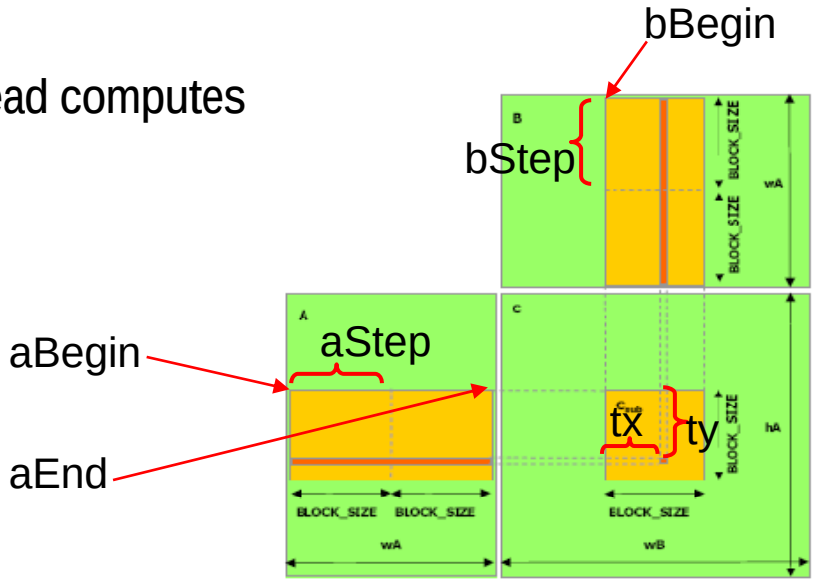
NVIDIA
清華大學 Tsinghua University

CUDA Implementation - Kernel

```
// Load the matrices from global memory to shared memory;
// each thread loads one element of each matrix
As[ty][tx] = A[a + wA * ty + tx];
Bs[ty][tx] = B[b + wB * ty + tx];

// Synchronize to make sure the matrices are loaded
__syncthreads();

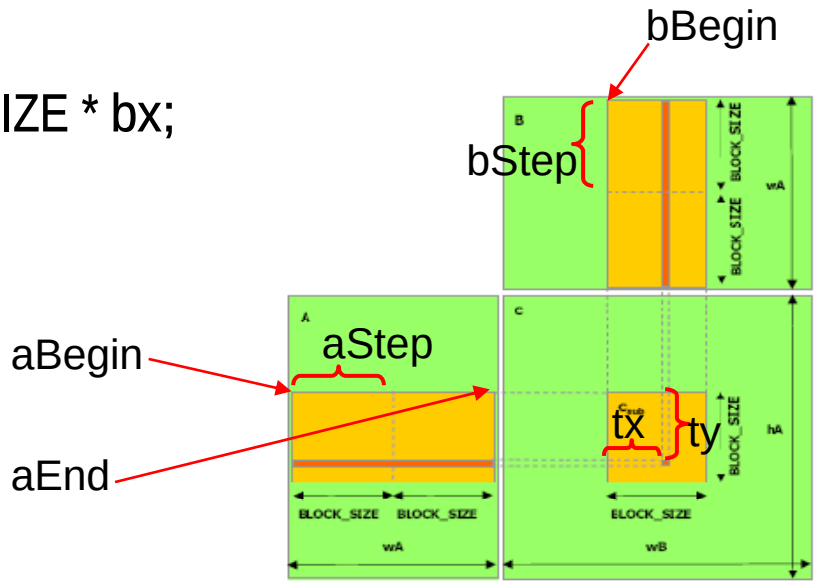
// Multiply the two matrices together; each thread computes
// one element of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += As[ty][k] * Bs[k][tx];
```



CUDA Implementation - Kernel

```
        // Synchronize to make sure that the preceding computation is done
        // before loading two new sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

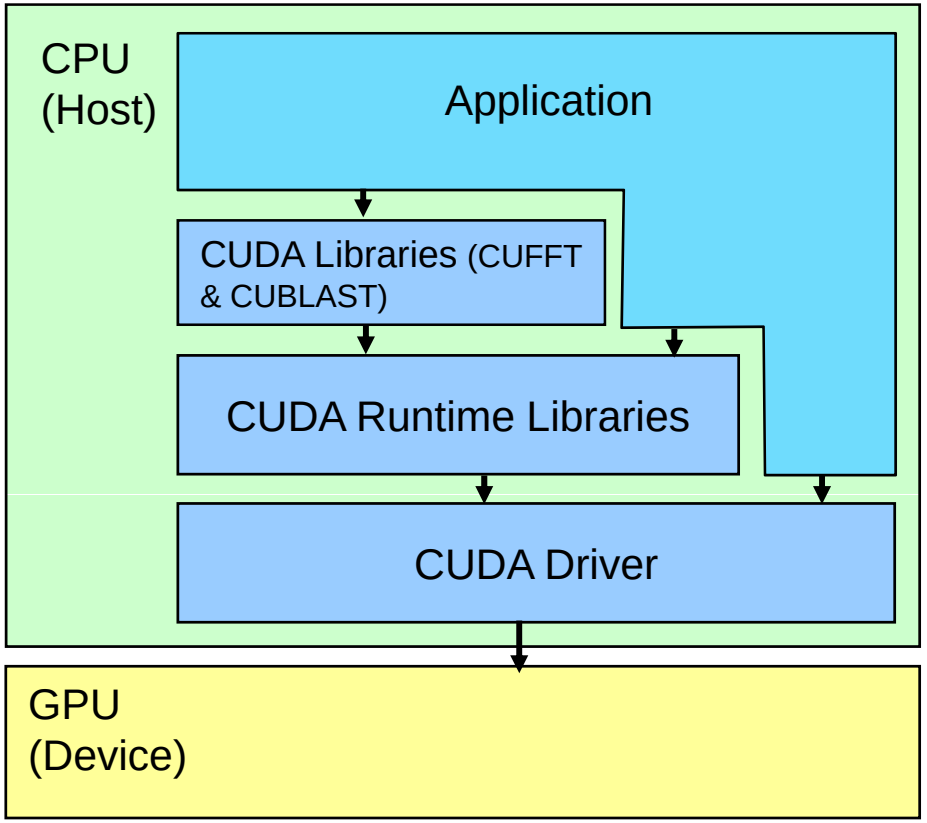


提纲

- 综述
- 并行线程组织
- 存储器模型和内存分配
- CUDA变量和函数
- CUDA程序设计实例
- CUDA libraries和程序设计工具

CUDA Libraries

- CUBLAS
  - BLAS implementation
- CUFFT
  - FFT implementation



CUBLAS

- 在CUDA硬件驱动上实现的BLAS (Basic Linear Algebra Subprograms) API
  - 程序员无需关心CUDA硬件驱动
- 基本使用模式
  - 在GPU存储器上分配矩阵和矢量数据对象
  - 填充数据
  - 调用CUBLAS完成一系列数学运算
  - 把GPU上的结果传回CPU host

Supported Features

- 单精度BLAS函数
  - 实数: level 1, 2 and 3
  - 虚数: level1 and CGEMM

(Level 1=vector vector O(N), Level 2= matrix vector O(N^2), Level 3=matrix matrix O(N^3))

- 数据结构
  - Column-major storage
  - 1-based indexing

CUBLAS Code Example

```
int main( void ){
        ...
        float* host_vector, * device_vector;

        host_vector = (float*) malloc(M*sizeof(float));

        ... // Initialize vector of M floats
        cublasAlloc(M, sizeof(float), (void**) &device_vector);

        cublasSetVector(M, sizeof(float), host_vector, device_vector, 1);
        cublasSscal(M, ALPHA, device_vector, 1);
        cublasGetVector(M, sizeof(float), device_vector,host_vector, 1);

        cublasFree(device_vector);
        ...
}
```

CUFFT

- Very efficient data parallel implementation
- Supported features:
  - 1D, 2D and 3D transforms of complex and real-valued data
  - Batched execution for doing multiple 1D transforms in parallel
  - 1D transform size up to 8M elements
  - 2D and 3D transform sizes in the range [2,16384]
  - In-place and out-of-place transforms for real and complex data.

CUFFT Code Example

```
#define NX 256
#define BATCH 10
cufftHandle plan;
cufftComplex *data;
cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*BATCH);

/* Create a 1D FFT configuration. */
cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);

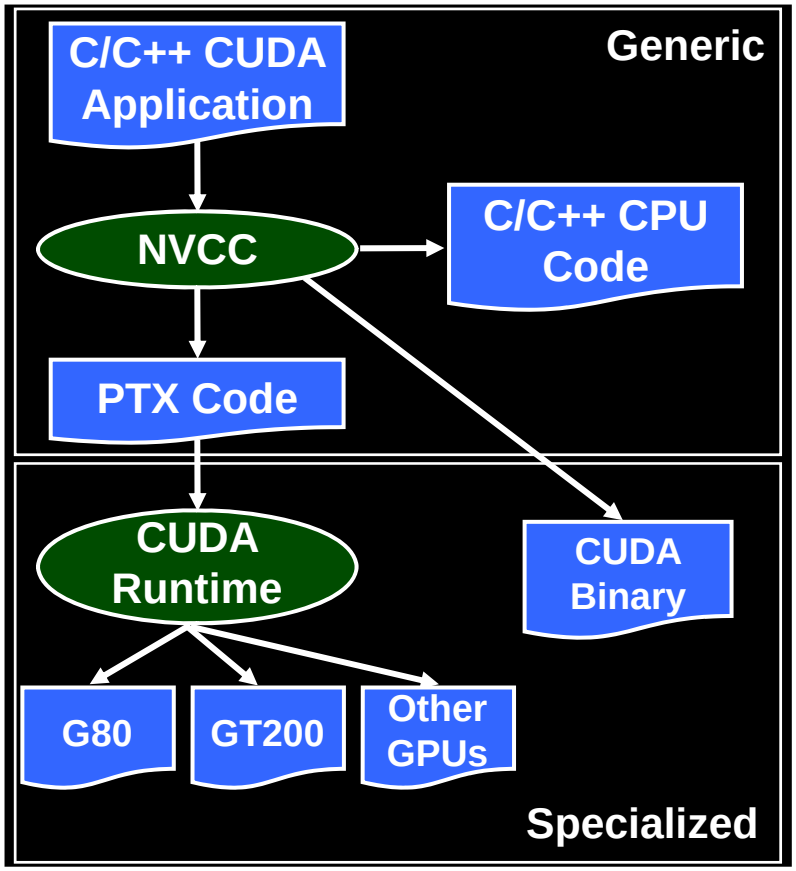
/* Identical pointers to input and output arrays implies in-place transformation. */
cufftExecC2C(plan, data, data, CUFFT_FORWARD); //Forward transform

cufftExecC2C(plan, data, data, CUFFT_INVERSE); //Inverse transform

/* Destroy the CUFFT plan. */
cufftDestroy(plan);
cudaFree(data);
```

CUDA程序的编译(compile)

- CUDA源文件被nvcc编译
  - 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



CUDA程序的链接(Link)

- CUDA代码要求两个动态链接库
  - CUDA runtime library (cudart)
  - CUDA core library (cuda)

Debugging in Device Emulation Mode

- 由device emulation mode (nvcc -deviceemu)方式编译的CUDA代码完全在CPU上运行
  - 仍然需要CUDA runtime library
  - 不需要GPU硬件和CUDA driver
  - 每一device(GPU)线程由一个host(CPU)线程仿真
- 支持:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. printf) and vice-versa
  - Detect deadlock situations caused by improper usage of __syncthreads

Device Emulation Mode Pitfalls

- Device emulation mode代码由CPU执行
  - CPU线程仿真GPU线程
  - 线程竞争不容易发生
    - 由于并行线程数量大幅度减少
- 浮点运算结果细微差别
- 允许对device(GPU)函数取指针
  - 在GPU执行模式下不允许