

快糙猛GPU编程入门： NVIDIA-CUDA篇

主线

- GPU硬件结构概览
- CUDA语言扩展概览
- GPU编程优化加速核心思想
- GPU编程实例
- GPU编程实用工具概览

什么是GPU

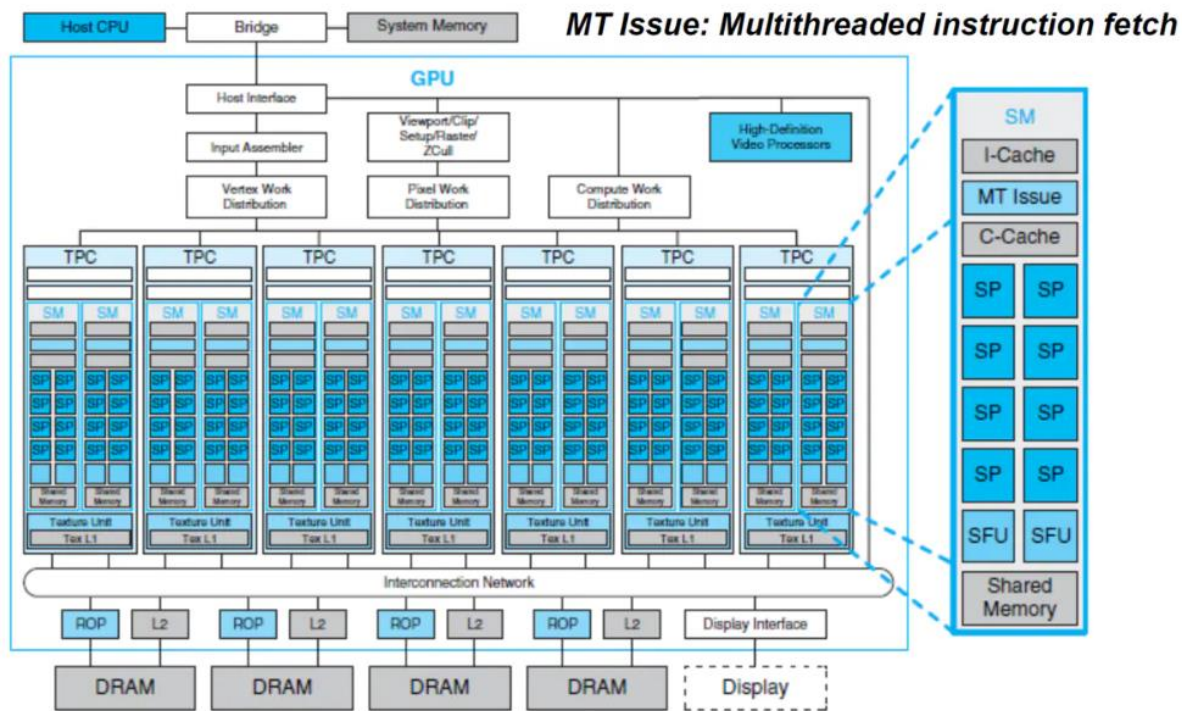
- 相比CPU，核心数目更**多**，单核性能更**弱**（价格更贵）的计算设备
- 核数多：并行吞吐量大
- 单核弱：单个核心效率并不高
- 从PCIe实现数据传输和通信

如何使用GPU实现通用计算

- CUDA-Compute Unified Device Architecture
- 什么是CUDA?
 - 入门级理解：加入了语言扩展以操作GPU硬件的C语言

CUDA硬件基础-GPU简化结构

NVIDIA Tesla Architecture



TPCs: Texture/Processor Clusters
SMs: Stream Multiprocessors

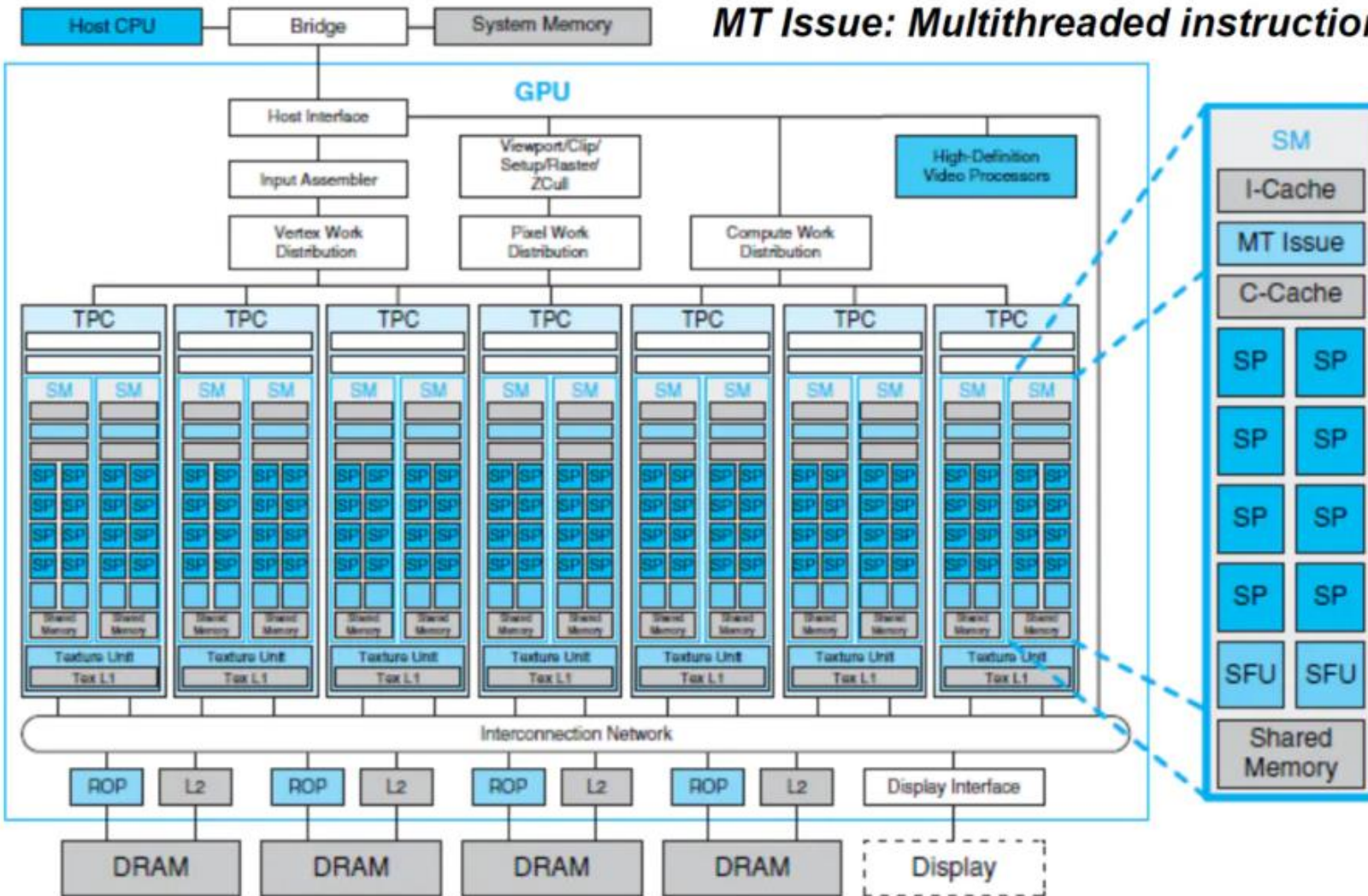
SPs: Streaming Processors
SFU: Special Function Unit
(4 floating-point multipliers)



NVIDIA Tesla Architecture



MT Issue: Multithreaded instruction fetch

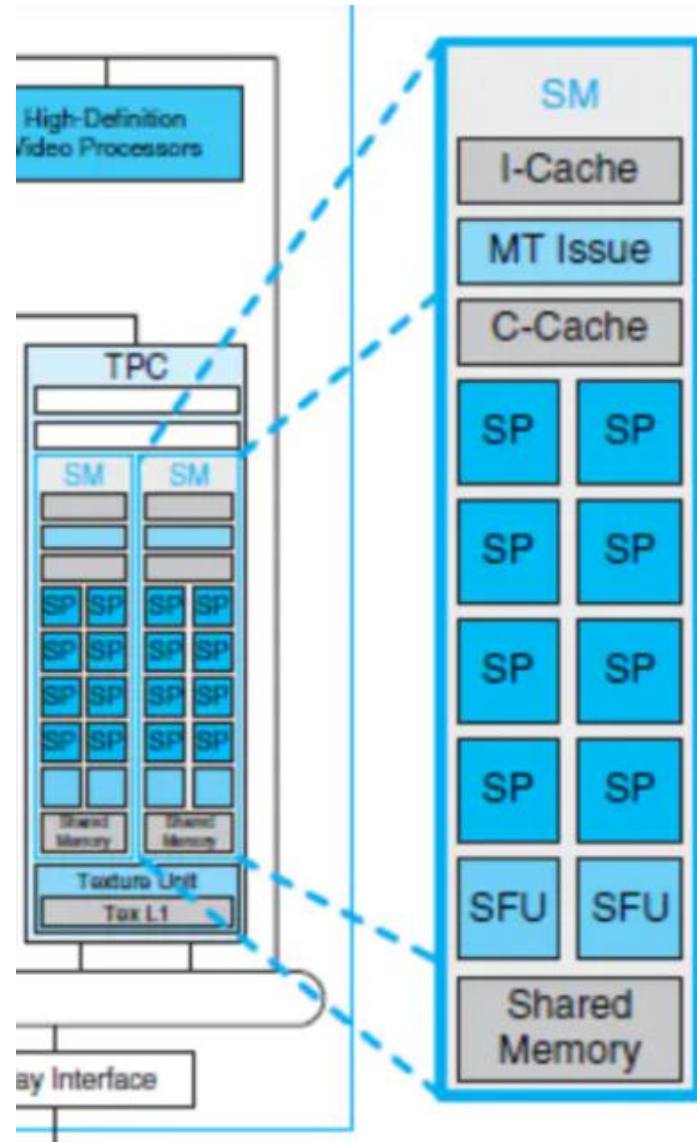


TPCs: Texture/Processor Clusters
SMs: Stream Multiprocessors

SPs: Streaming Processors
SFU: Special Function Unit
(4 floating-point multipliers)

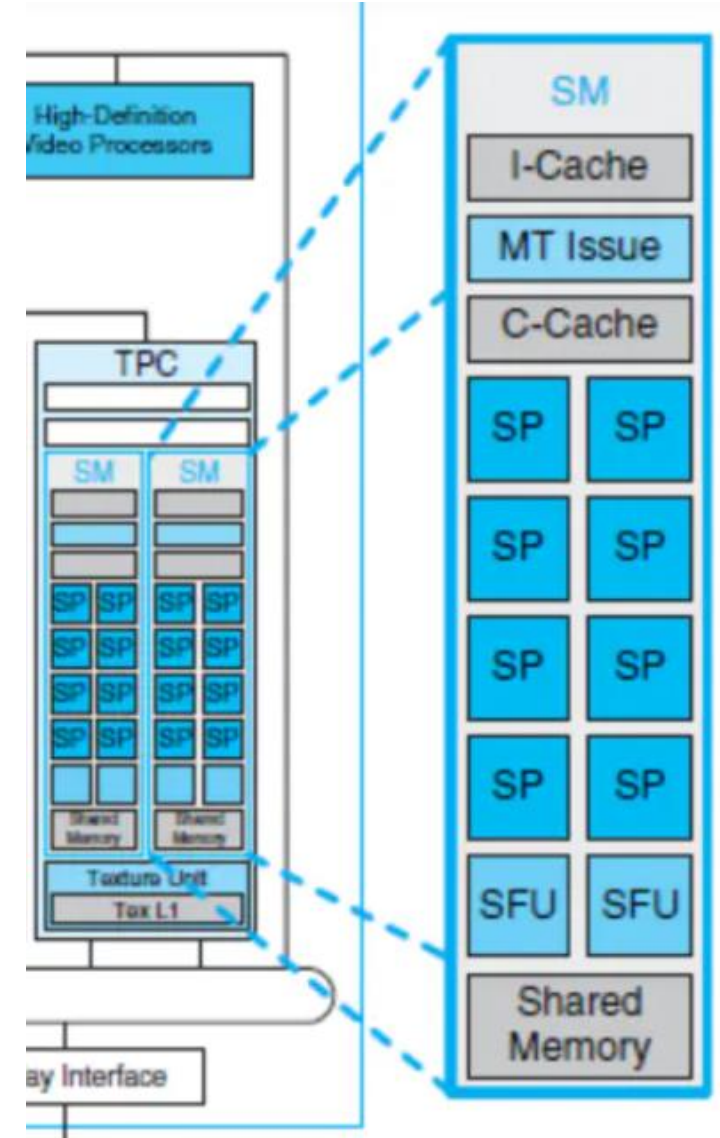
CUDA硬件基础-GPU简化结构

- SM-流多处理器
- 核心-SP流处理器（可以看成CPU中的thread）
- L1缓存，常量缓存，共享内存



CUDA硬件基础-更深入SM

- SM-流多处理器
- **SP-流处理器** (可以看成CPU中的thread)
- L1缓存, 常量缓存, 共享内存

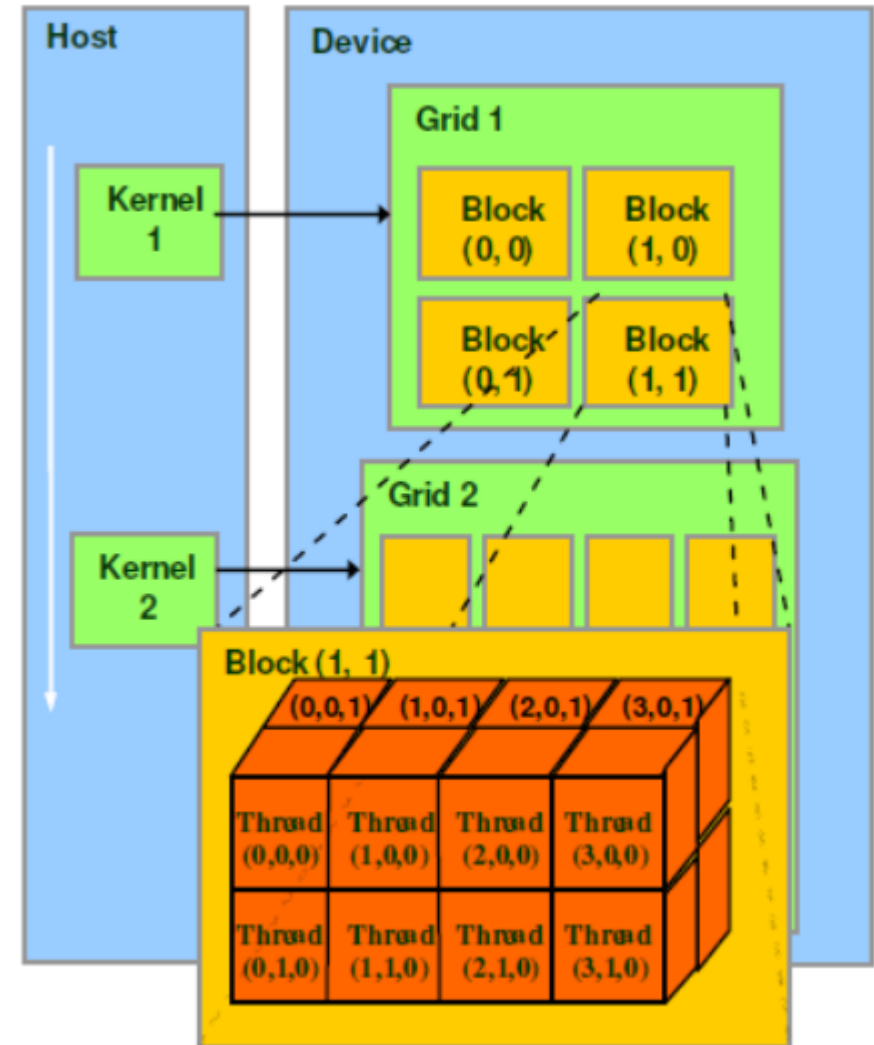


CUDA并行本质-SIMT

- SIMT (Single Instruction Multiple Thread)
- 和SIMD (Single Instruction Multiple Data)的区分
- SIMT: 多个SP执行一个函数 (核函数Kernel)

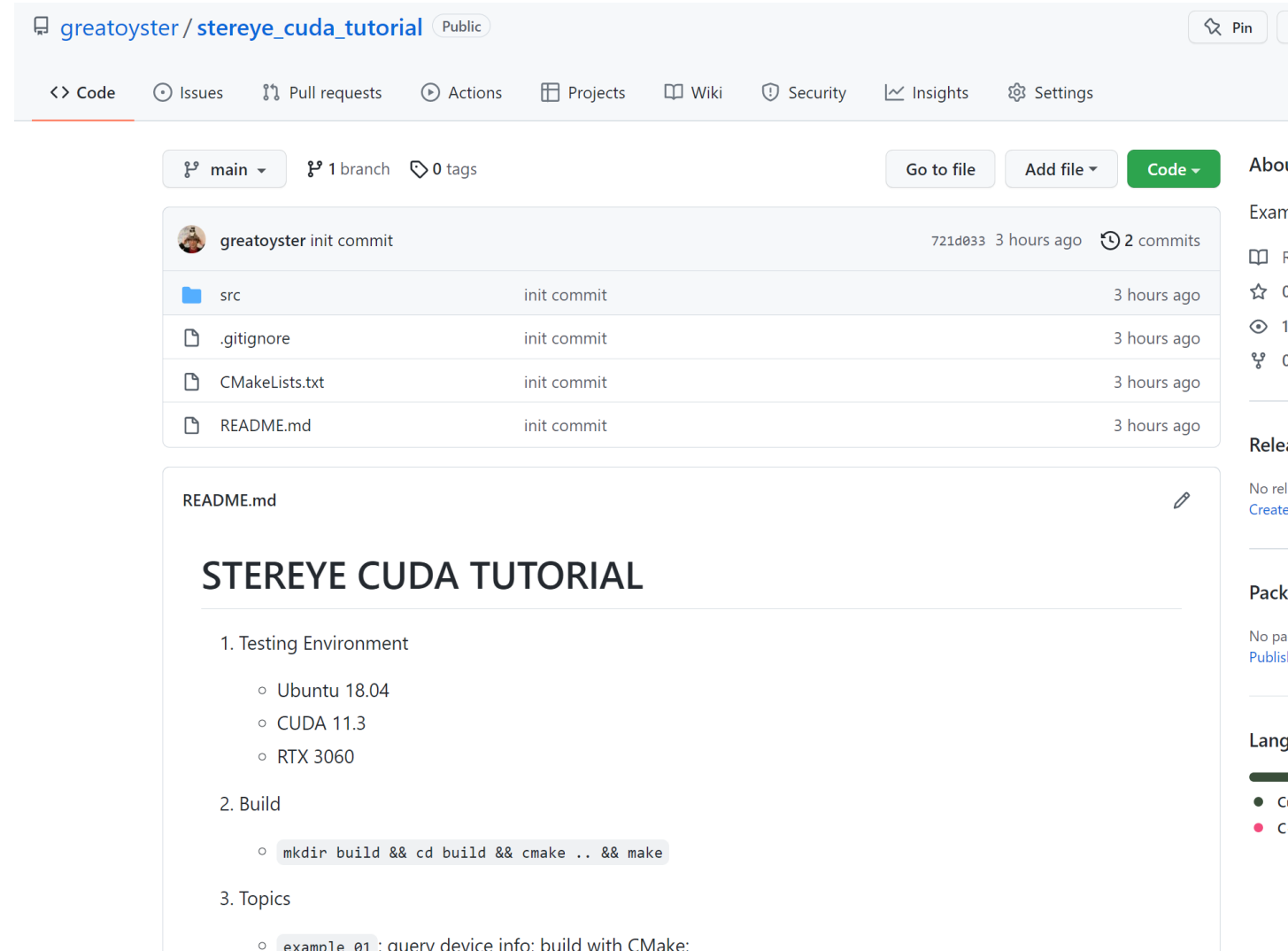
CUDA编程模型-硬件资源抽象

- 划分层级从小到大Thread-Block-Grid
- Block和Grid以及Warp的大小因设备而异
- 计算资源调度的最细粒度-Warp（线程束）
- 即便如此还是可以进行Warp层面的优化



Example_01

- 查询设备信息（非常重要）
- 见src/example_01.cu
- 注：全部代码可见GitHub



The screenshot shows the GitHub interface for the repository 'greatoyster / stereye_cuda_tutorial'. The repository is public and has 1 branch (main) and 0 tags. The commit history shows an initial commit by 'greatoyster' 3 hours ago, with 2 commits in total. The file list includes 'src', '.gitignore', 'CMakeLists.txt', and 'README.md', all from the initial commit. The 'README.md' file is expanded, showing the title 'STEREYE CUDA TUTORIAL' and three sections: '1. Testing Environment' (listing Ubuntu 18.04, CUDA 11.3, and RTX 3060), '2. Build' (with a command to create a build directory and compile), and '3. Topics' (listing 'example 01' as a query device info and build with CMake).

greatoyster / stereye_cuda_tutorial Public

<> Code Issues Pull requests Actions Projects Wiki Security Insights Settings

main 1 branch 0 tags Go to file Add file Code

greatoyster init commit 721d033 3 hours ago 2 commits

src	init commit	3 hours ago
.gitignore	init commit	3 hours ago
CMakeLists.txt	init commit	3 hours ago
README.md	init commit	3 hours ago

README.md

STEREYE CUDA TUTORIAL

1. Testing Environment
 - Ubuntu 18.04
 - CUDA 11.3
 - RTX 3060
2. Build
 - `mkdir build && cd build && cmake .. && make`
3. Topics
 - `example 01` : query device info; build with CMake;

CUDA语言扩展

- 函数限定符
 - `__host__`,
 - `__device__`,
 - `__global__`
- 变量限定符 `__constant__`, `__shared__`
 - 一般用于控制数据的存储布局

```
#include <cuda_runtime.h>
#include <cuda.h>
#include <device_launch_parameters.h>
#include <cassert>
#include "example_02.h"

__global__ void kernel_vec_add(float *a, float *b, float *c, int n);

__host__ void vec_add(float *a, float *b, float *c, int n)
{
    float *dev_a, *dev_b, *dev_c;
    int size = sizeof(float) * n;

    cudaMalloc(&dev_a, size);
    cudaMalloc(&dev_b, size);
    cudaMalloc(&dev_c, size);

    cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);

    cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_c, c, size, cudaMemcpyHostToDevice);
    kernel_vec_add<<<n / 32 + 1, 32>>>>(dev_a, dev_b, dev_c, n);

    cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}

__global__ void kernel_vec_add(float *a, float *b, float *c, int n)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n)
    {
        c[idx] = a[idx] + b[idx];
    }
}
```

CUDA语言扩展-函数限定符

- `__device__`
 - 这个限定符声明的函数： a、在设备上运行； b、只能被设备调用；
- `__host__`
 - 这个限定符声明的函数： a、在主机上运行； b、只能被主机调用；
- `__global__`
 - 这个限定符声明的函数： a、在设备上运行； b、可以从主机上被调用； c、可以被计算能力为3.x的设备调用
 - 这个限定符声明的函数也就是核函数， 需要传入特殊的参数

CUDA语言扩展-核函数

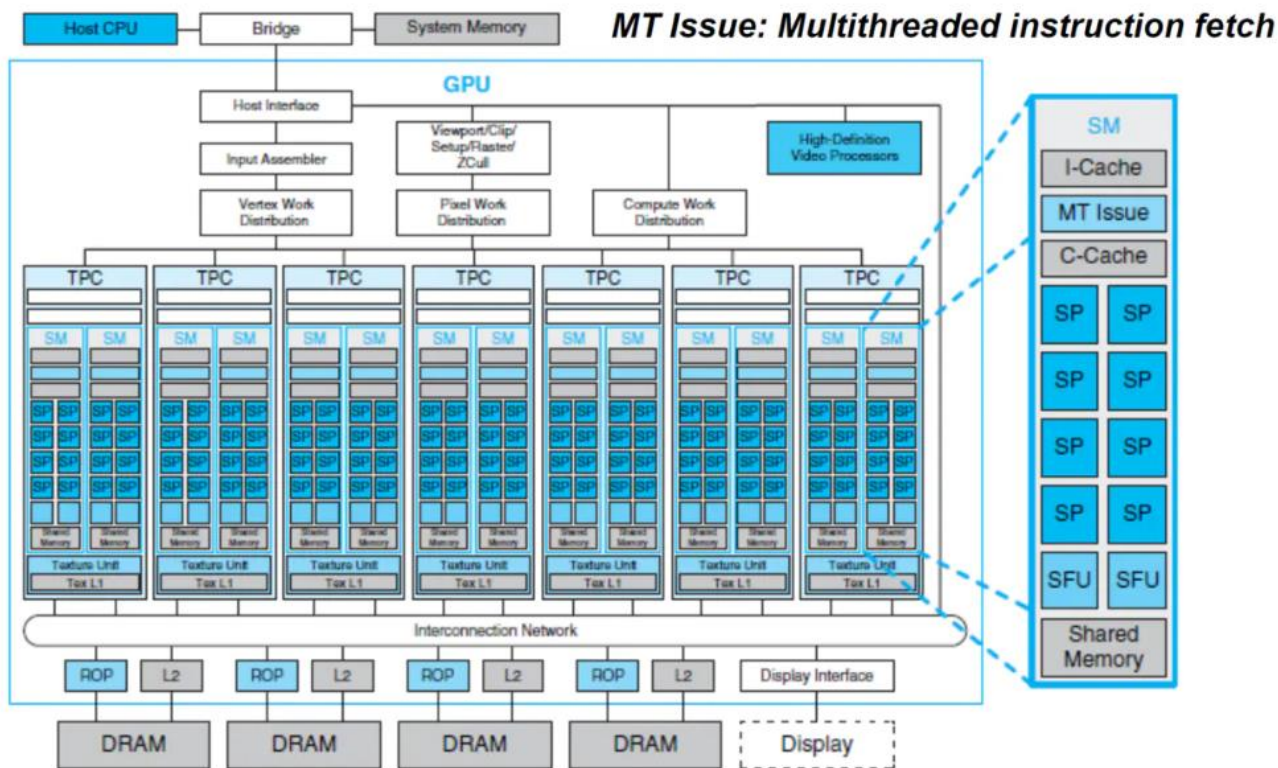
- 被__global__修饰的函数（外表）
- 在GPU SP上执行的指令序列（本质）
- 形如kernel_func<<<numBlocks,threadsPerBlock>>>(args...)
 - 尖括号内是GPU的硬件资源调度参数

Example_02

- 高维向量的并行二元操作（向量相加）
- 见src/example_02.h, src/example_02.cu, src/example_02.cc

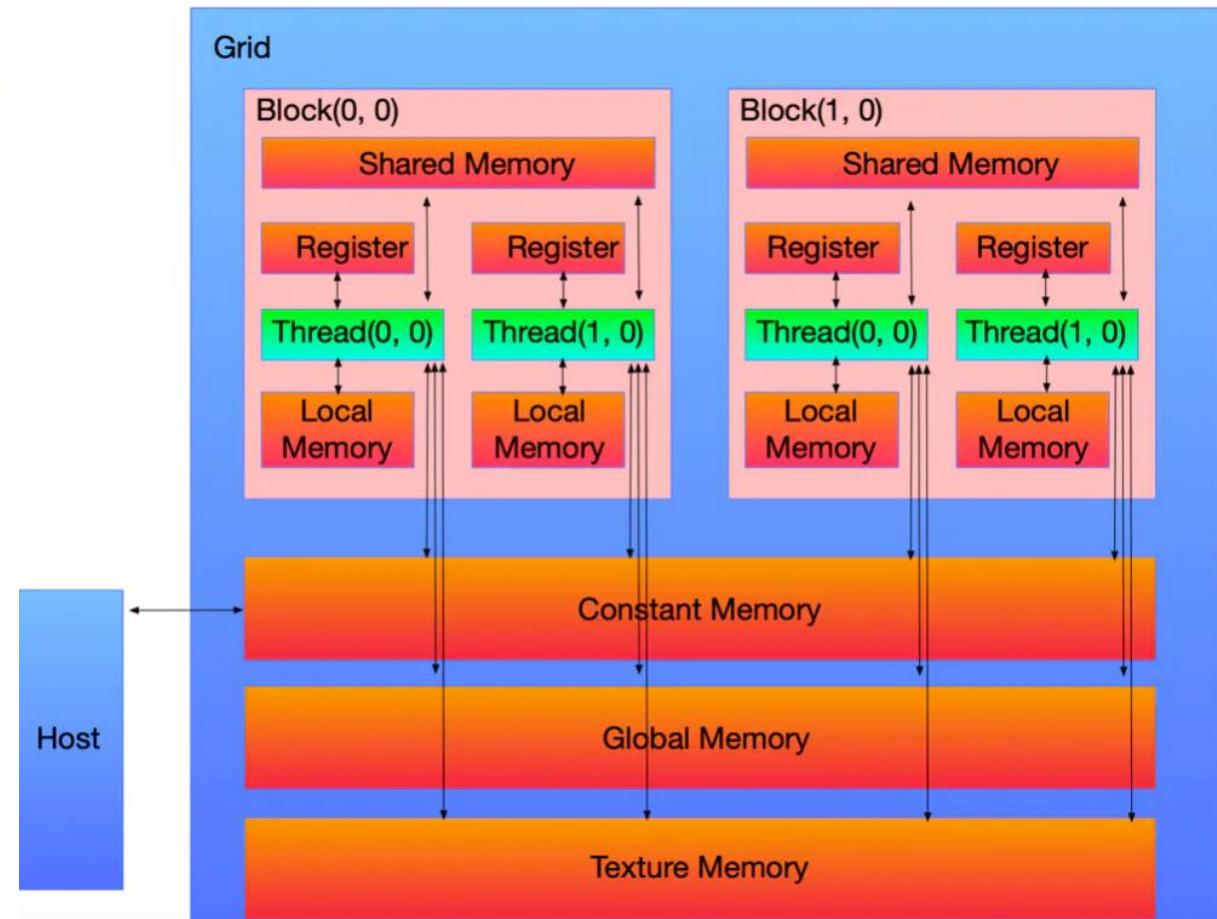
CUDA硬件-逻辑对应关系

NVIDIA Tesla Architecture



TPCs: Texture/Processor Clusters
SMS: Stream Multiprocessors

SPs: Streaming Processors
SFU: Special Function Unit
 (4 floating-point multipliers)



CUDA默认流的基本操作

- 向GPU传入数据(cudaMemcpy等)
- 在GPU完成计算(kernel function)
- 向HOST传回数据(cudaMemcpy等)



CUDA默认流的基本操作

```
cudaMalloc(&dev_a, size);
cudaMalloc(&dev_b, size);
cudaMalloc(&dev_c, size);

cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);

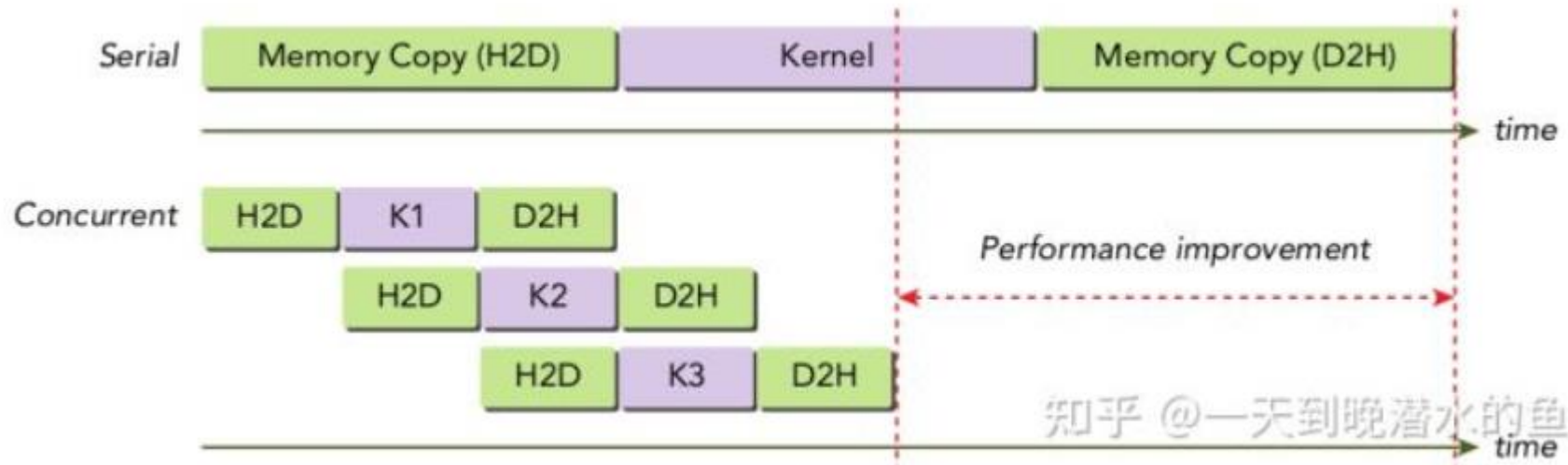
cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_c, c, size, cudaMemcpyHostToDevice);
kernel_vec_add<<<n / 32 + 1, 32>>>(dev_a, dev_b, dev_c, n);

cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
```


CUDA事件-基本概念

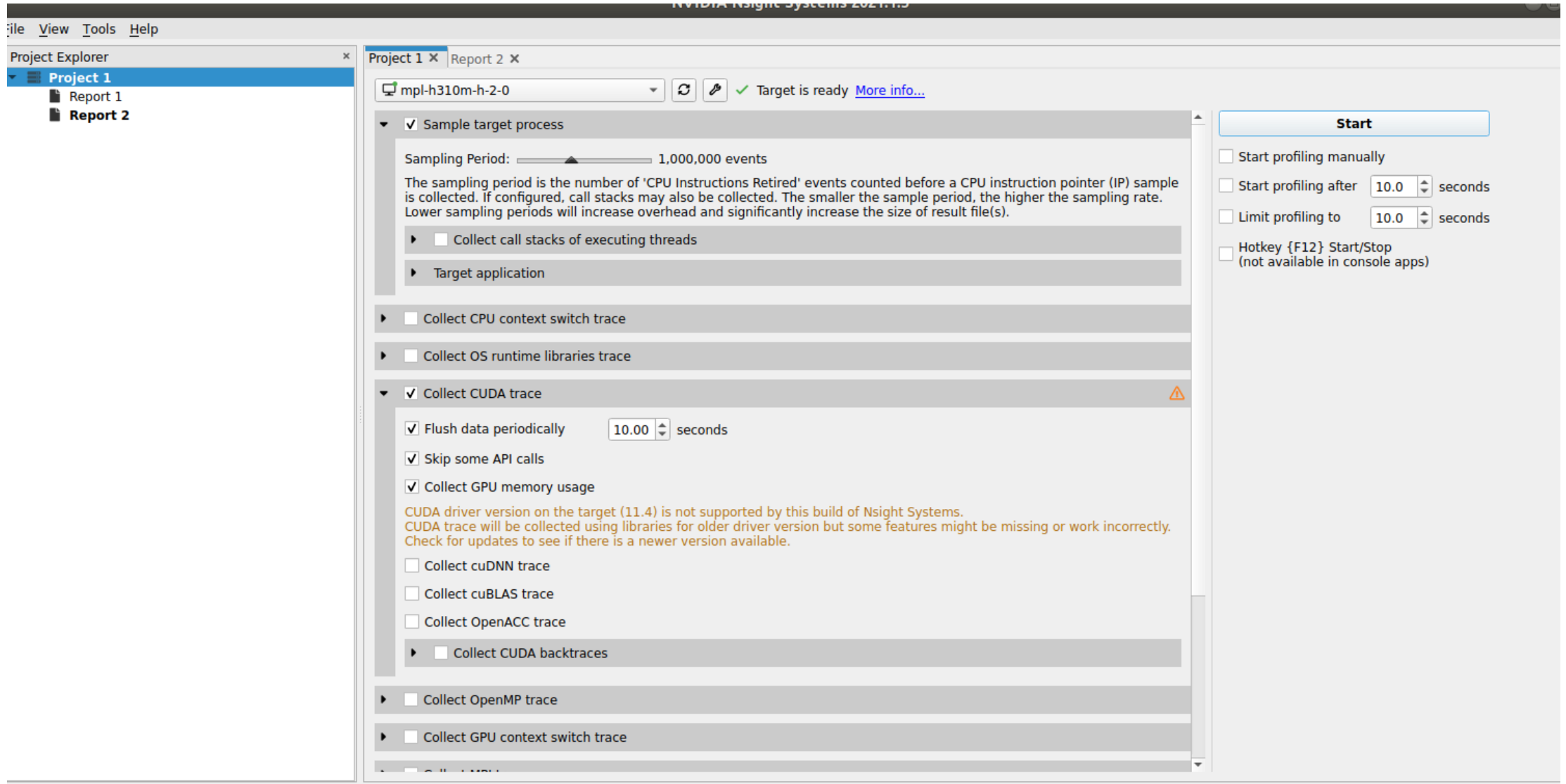
- CUDA中Event用于在流的执行中添加标记点，用于检查正在执行的流是否到达给定点。
 - 可以用于统计时间，在需要测量的函数前后插入Event。调用 `cudaEventElapsedTime()` 查看时间间隔
 - 还有更重要的作用（流同步），见CUDA多流并发



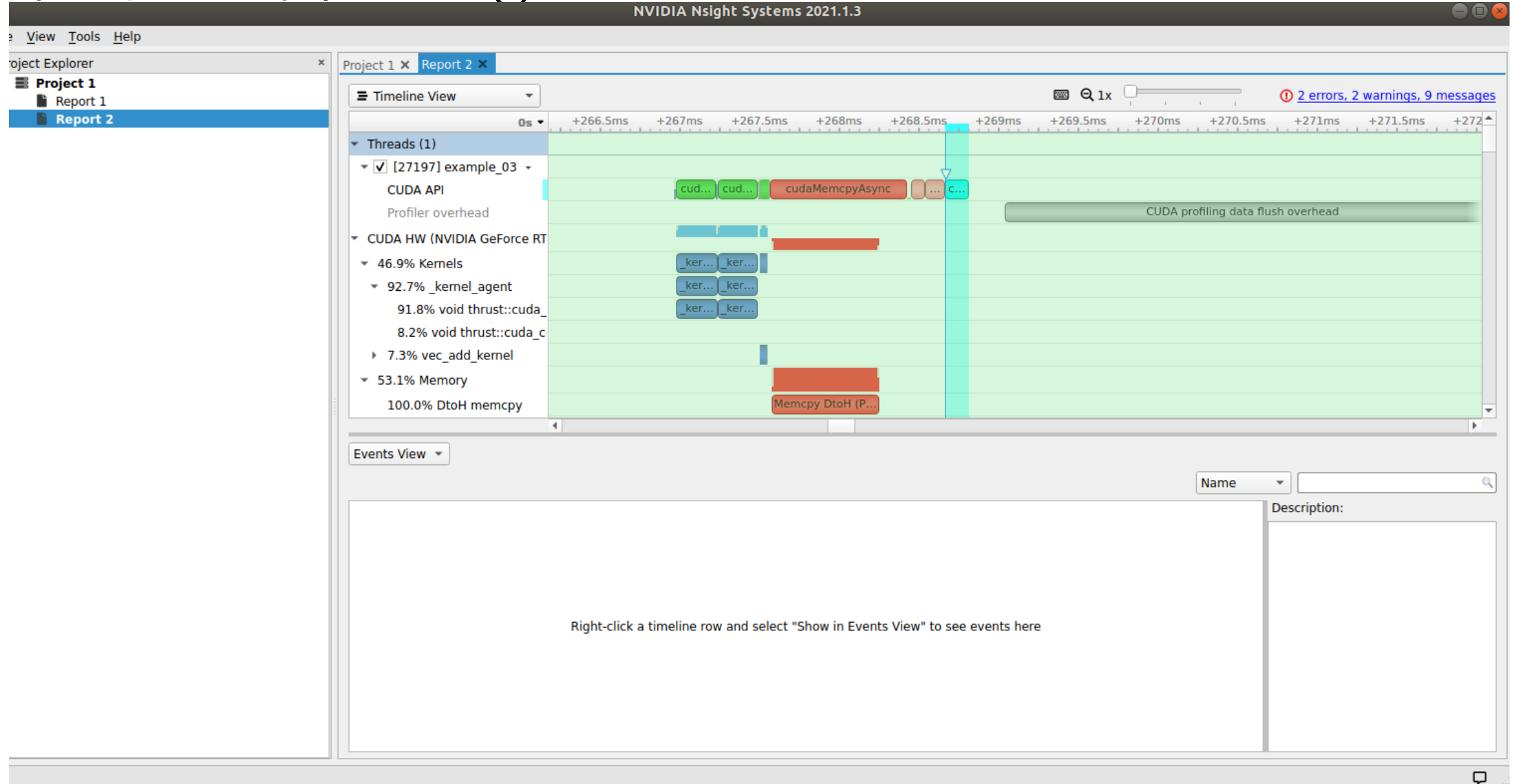
Example_03

- 在CPU和GPU上向量相加，并且记录时间
- 见src/example_03.cu

性能分析-Nsight

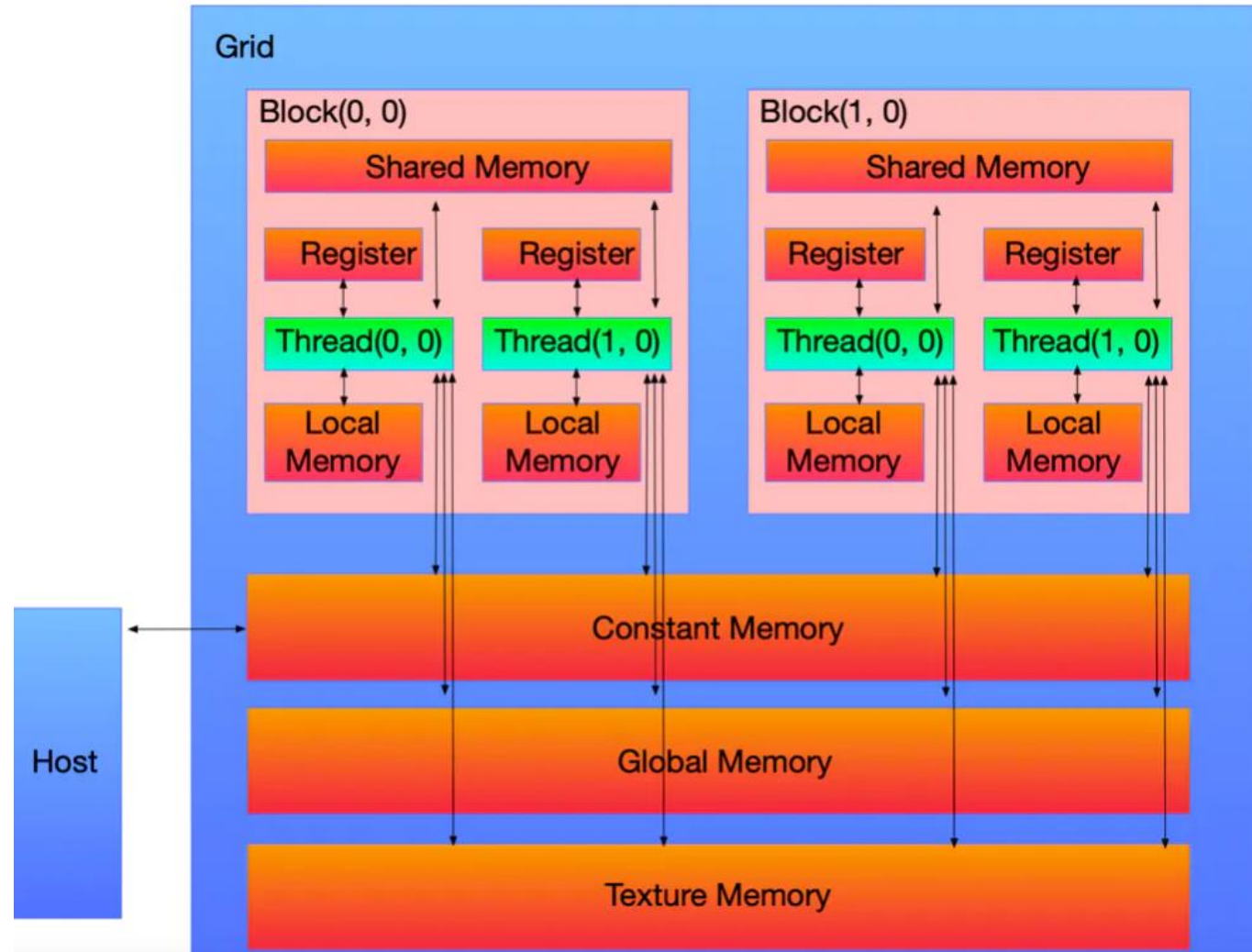


性能分析-Nsight



CUDA-Shared Memory初探

- 十分重要的概念



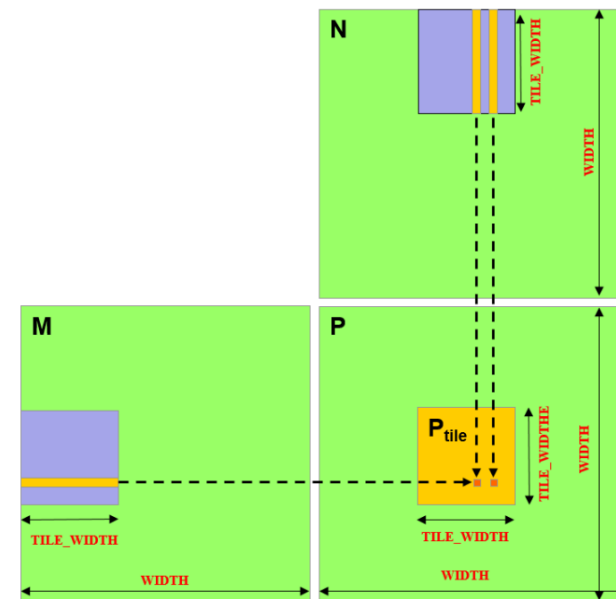
Example_04

Exploiting data reuse

- 矩阵乘法
- 见src/example_04.cu

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        for (int k = 0; k < Width; ++k)  
            Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];  
        d_P[Row*Width+Col] = Pvalue; } }
```

- P_{tile} contains a block of TILE_WIDTH^2 threads.
- Every thread loads all data it needs by itself.
 - TILE_WIDTH^2 threads in P_{tile} each loads $2 * \text{TILE_WIDTH}$ data $\Rightarrow 2 * \text{TILE_WIDTH}^3$ global memory reads.
- But notice all threads in P_{tile} need data from purple tiles.
 - Purple data can be reused!
- Threads in P_{tile} cooperate to load purple tiles, eliminating redundant global memory reads.
 - Only $2 * \text{TILE_WIDTH}^2$ global memory reads in total. A factor of TILE_WIDTH less!



CUDA原子操作

- [Programming Guide :: CUDA Toolkit Documentation \(nvidia.com\)](https://docs.nvidia.com/cuda/cuda-toolkit/documentation/index.html)

Can perform atomics on global or shared memory variables.

```
int atomicInc(int *addr)
```

- Reads value at addr, increments it, returns old value.
- Hardware ensures all 3 instructions happen without interruption from any other thread.

```
int atomicAdd(int *addr, int val)
```

- Reads value at addr, adds val to it, returns old value.

```
int atomicMax(int *addr, int val)
```

- Reads value at addr, sets it to max of current value and val, returns old value.

```
int atomicExch(int *addr1, int val)
```

- Sets val at addr to val, returns old value at val.

```
int atomicCAS(int *addr, old, new)
```

- “Compare and swap”, a conditional atomic.
- Reads value at addr. If value equals old, sets value to new. Else does nothing.
- Indicates whether state changed, i.e. if your view is up to date.
- Universal operation, i.e. can be used to perform any other kind of synchronization.

Example_05

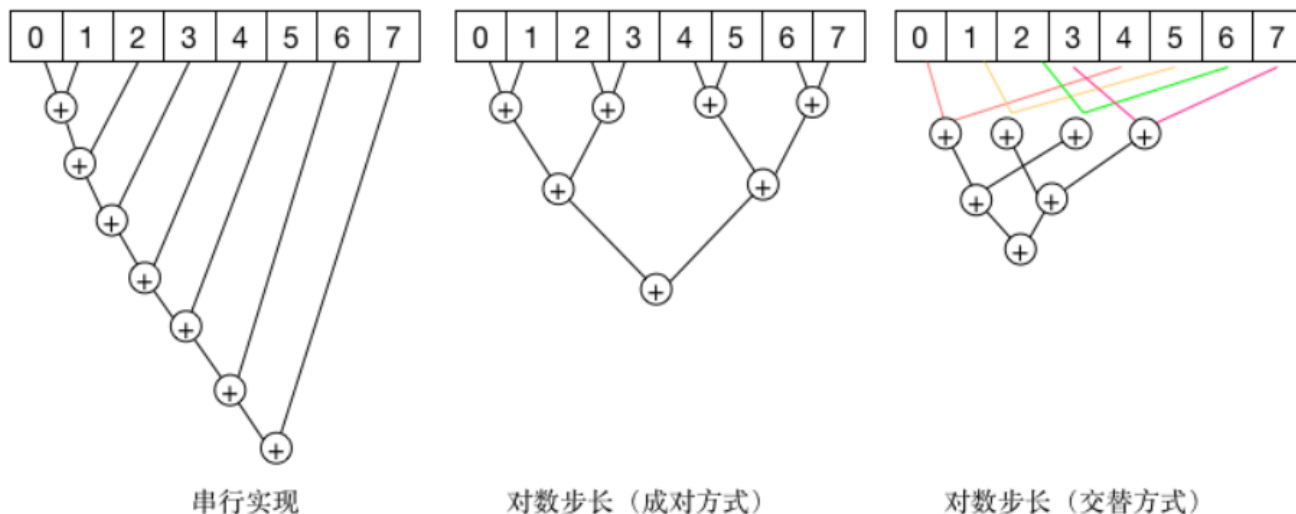
- 数组求和
- 见src/example_05.cu

CUDA并行规约

对于N个输入数据和操作符+, 规约可表示为:

$$\sum_i a_i = a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7$$

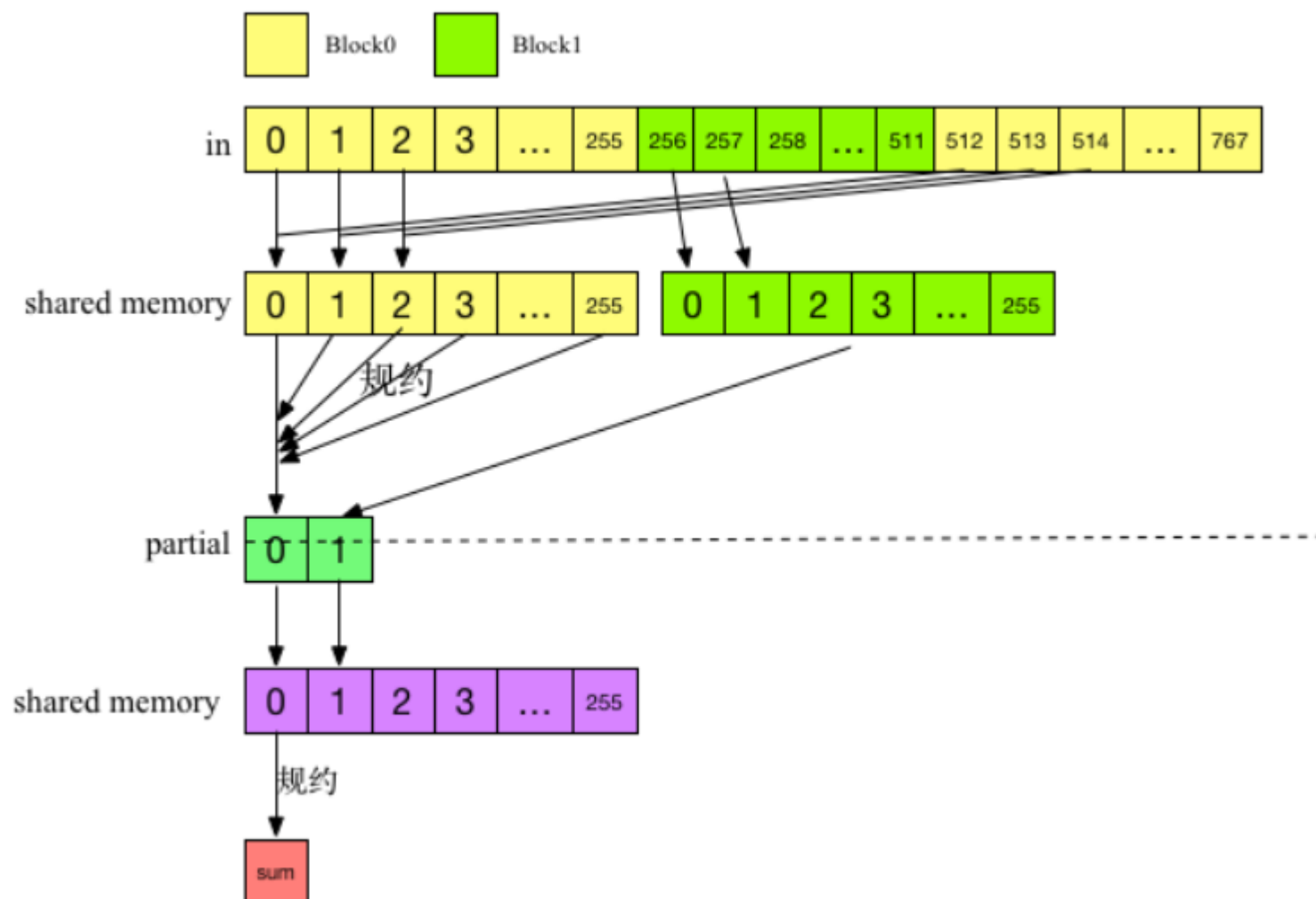
下图展示了一些处理8个元素规约操作的实现:



从图中可以看到, 不同的实现其时间复杂度也是不一样的, 其中串行实现完成计算需要7步, 性能比较差。成对的方式是典型的分治思想, 只需要 $\lg N$ 步来计算结果, 由于不能合并内存事务, 这种实现在CUDA中性能较差。

在CUDA中, 无论是对全局内存还是共享内存, 基于交替策略效果更好。对于全局内存, 使用 $\text{blockDim.x} \times \text{gridDim.x}$ 的倍数作为交替因子有良好的性能, 因为所有的内存事务将被合并。对于共享内存, 最好的性能是按照所确定的交错因子来累计部分结果, 以避免存储片冲突, 并保持线程块的相邻线程处于活跃状态。

CUDA并行规约



Example_06

- 高维向量的并行规约（基于thrust）
- 见src/example_06.cu
- [CUDA Webinar 2 \(nvidia.cn\)](http://nvidia.cn/CUDA-Webinar-2)

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}

template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```



Final Optimized Kernel

Performance for 4M element reduction



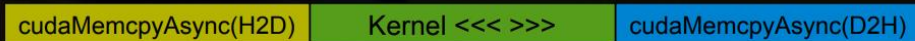
	Time (2 ²² ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 32M elements: 73 GB/s!

Example_07

- CUDA异步流

Serial (1x)



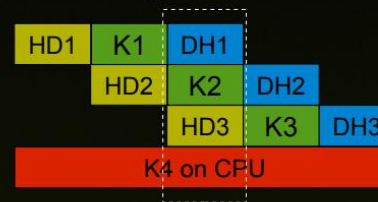
2-way concurrency (up to 2x)



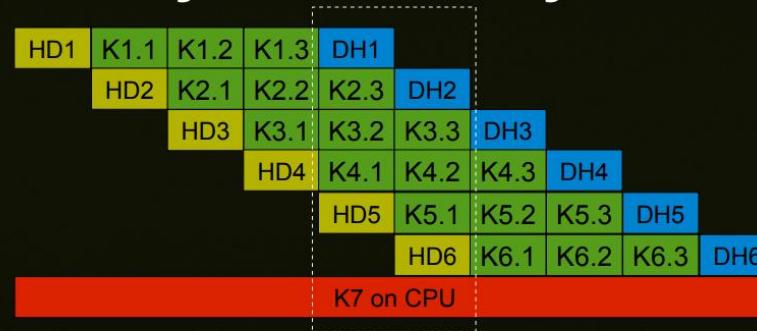
3-way concurrency (up to 3x)



4-way concurrency (3x+)



4+ way concurrency



Example_07

- CUDA异步流[Slide 1 \(nvidia.cn\)](#)
- 见src/example_07.cu

```
cudaStream_t stream1, stream2, stream3, stream4 ;
cudaStreamCreate ( &stream1 );
...
cudaMalloc ( &dev1, size ) ;
cudaMallocHost ( &host1, size ) ;
...
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 ) ;
kernel2 <<< grid, block, 0, stream2 >>> ( ..., dev2, ... ) ;
kernel3 <<< grid, block, 0, stream3 >>> ( ..., dev3, ... ) ;
cudaMemcpyAsync ( host4, dev4, size, D2H, stream4 ) ;
some_CPU_method ();
...
```

// pinned memory required on host

**potentially
overlapped**

Fully asynchronous / concurrent

Data used by concurrent operations should be independent

总结-加速原则

- 设计并且应用合适的并行算法
- 同时利用尽可能多的计算单元
- 减小访存操作
- 针对GPU结构优化

更多展望

- 更多基础并行算法 (sort, prefix_sum...)
- 分布式并行算法 (非PRAM模型)
- Warp级别的优化-Intrinsics
- 其他官方库
 - cuBlas(基础线性代数)
 - tensorRT(神经网络加速)
 - cuSparse(稀疏矩阵计算)
 - nccl(多卡通信)