《计算机系统结构》课程直播
2020. 5.14

听不到声音请及时调试声音设备，可以下课后补签到

请将ZOOM名称改为“姓名”；

本节内容

- 数据级并行性
  - GPU硬件执行模型
  - GPU存储模型

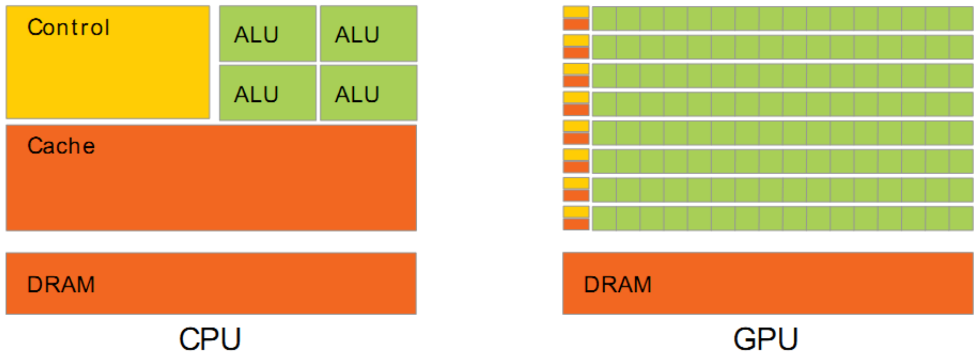
From : H&P Computer Architecture: A Quantitative Approach, Fifth Edition, （5th edition)

Graphics Processing Units

GPU

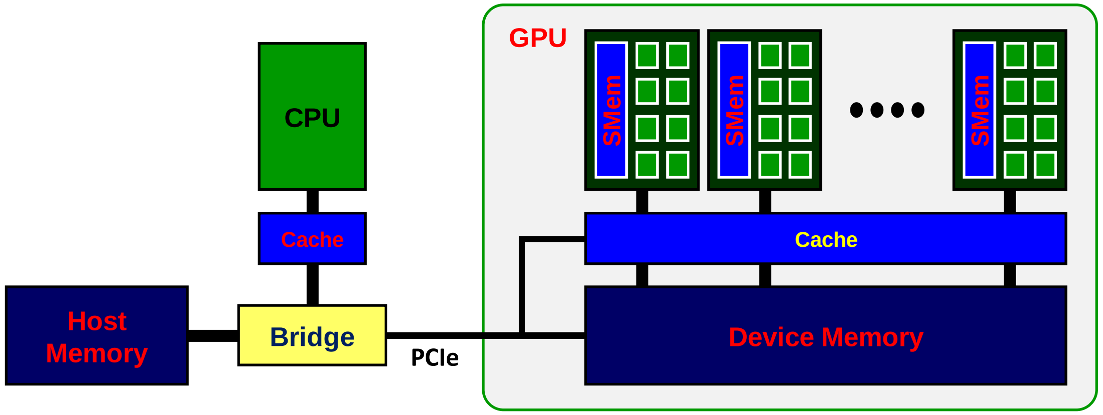
Using CPU+GPU Architecture

- CPU+GPU异构多核系统
  - 针对每个任务选择合适的处理器和存储器
- 通用CPU 适合执行一些串行的线程
  - 串行执行快
  - 带有cache，访问存储器延时低
- GPU 适合执行大量并行线程
  - 可扩放的并行执行
  - 高带宽的并行存取

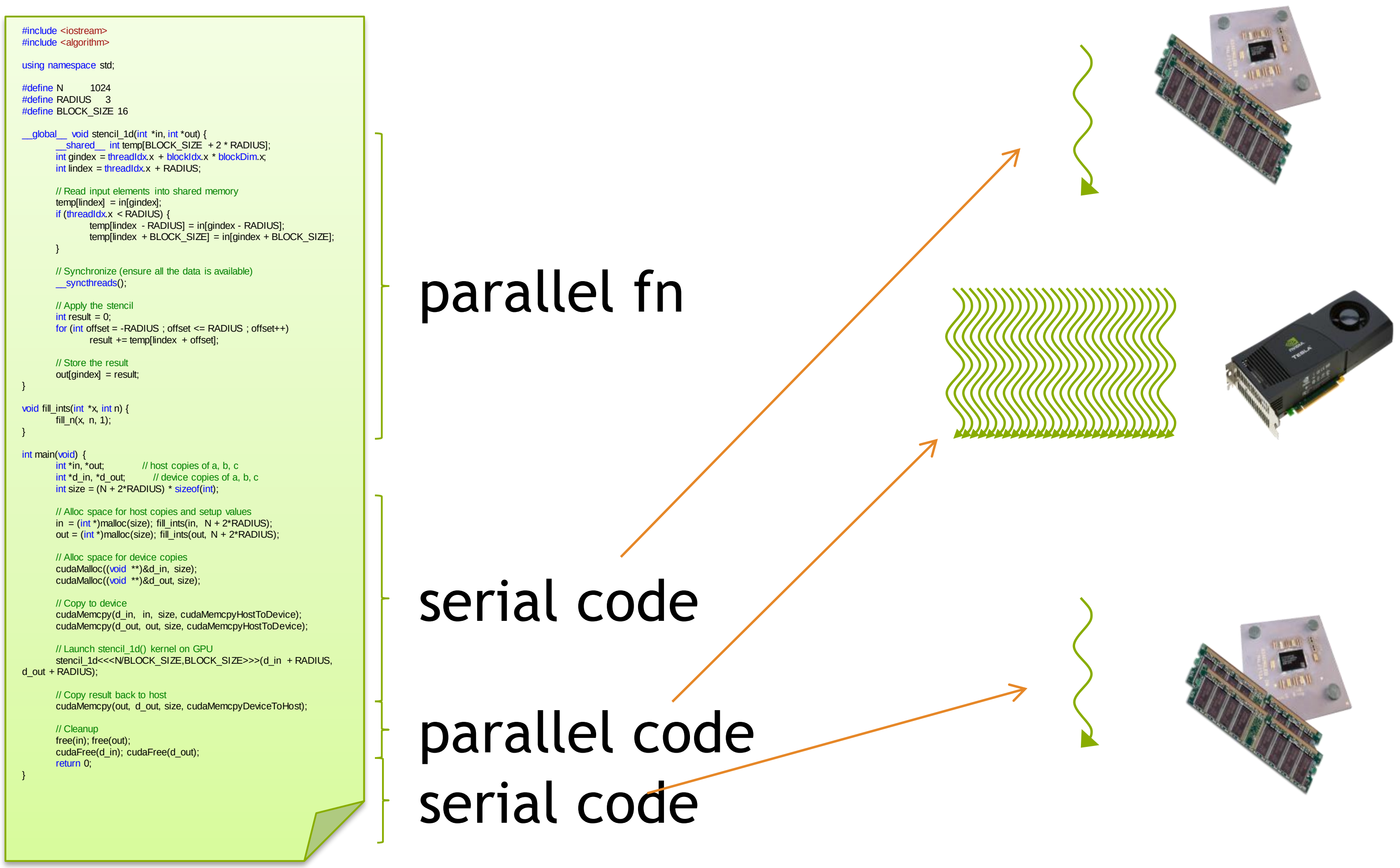


强控制、弱计算

弱控制、强计算



Heterogeneous Computing



```
#include <iostream>
#include <algorithm>

using namespace std;

#define N         1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__  void stencil_1d(int *in, int *out) {
          __shared__  int temp[BLOCK_SIZE  + 2 * RADIUS];
          int gindex = threadIdx.x  + blockIdx.x * blockDim.x;
          int lindex = threadIdx.x  + RADIUS;

          // Read input elements  into shared memory
          temp[lindex]  = in[gindex];
          if (threadIdx.x  < RADIUS) {
                    temp[lindex  - RADIUS] = in[gindex - RADIUS];
                    temp[lindex  + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
          }

          // Synchronize (ensure all the data is available)
          __syncthreads();

          // Apply the stencil
          int result = 0;
          for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                    result += temp[lindex  + offset];

          // Store the result
          out[gindex]  = result;
}

void fill_ints(int *x, int n) {
          fill_n(x,  n, 1);
}

int main(void)  {
          int *in, *out;              // host copies of a, b, c
          int *d_in, *d_out;          // device copies of a, b, c
          int size = (N + 2*RADIUS) * sizeof(int);

          // Alloc space for host copies and setup values
          in  = (int *)malloc(size); fill_ints(in,   N + 2*RADIUS);
          out = (int *)malloc(size); fill_ints(out,  N + 2*RADIUS);

          // Alloc space for device copies
          cudaMalloc((void  **)&d_in,  size);
          cudaMalloc((void  **)&d_out, size);

          // Copy to device
          cudaMemcpy(d_in,   in,  size, cudaMemcpyHostToDevice);
          cudaMemcpy(d_out,  out, size, cudaMemcpyHostToDevice);

          // Launch stencil_1d()  kernel on GPU
          stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in  + RADIUS,
d_out + RADIUS);

          // Copy result back to host
          cudaMemcpy(out,  d_out, size, cudaMemcpyDeviceToHost);

          // Cleanup
          free(in); free(out);
          cudaFree(d_in); cudaFree(d_out);
          return 0;
}
```

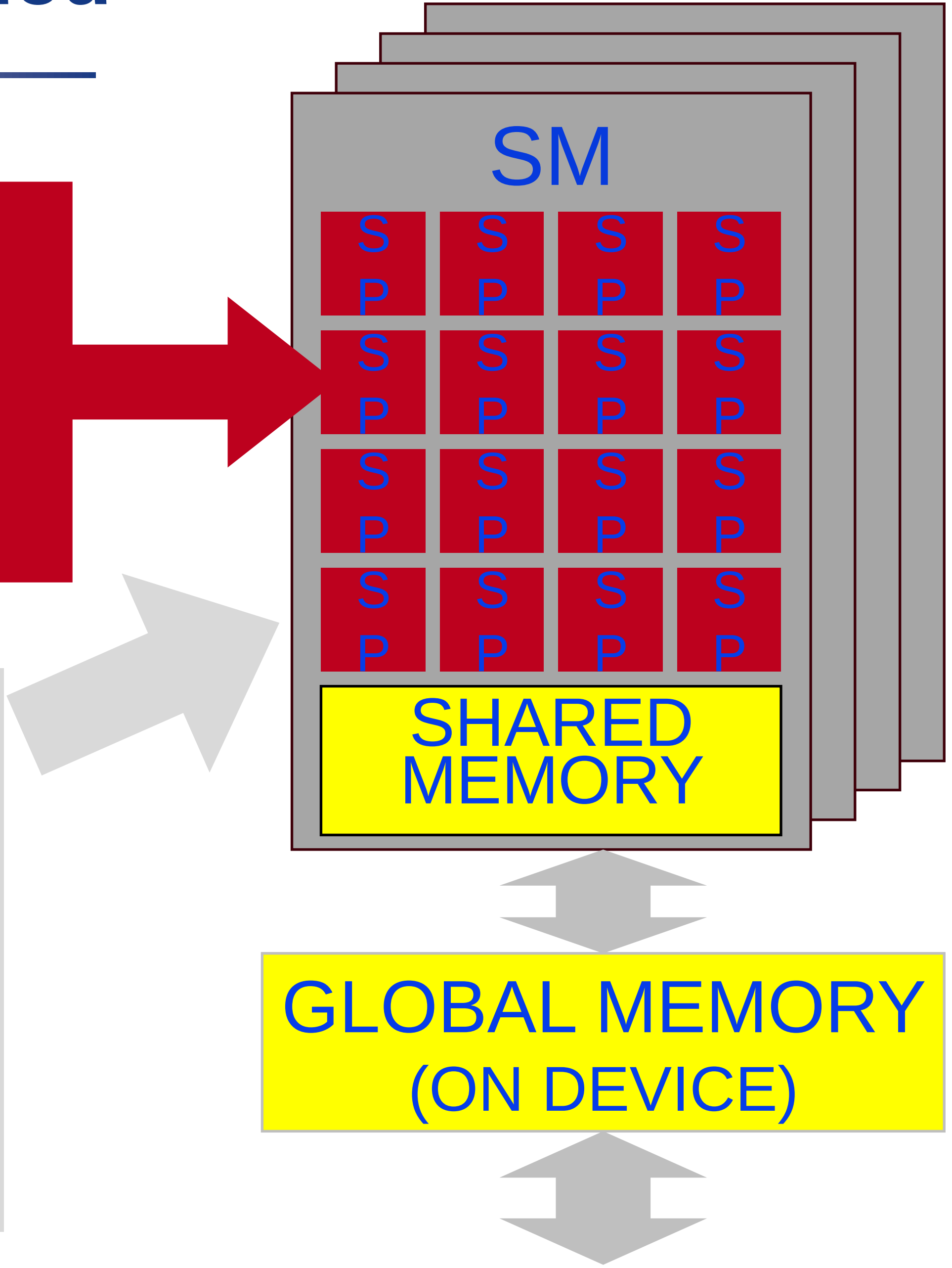
GPU: a multithreaded coprocessor

SP: scalar processor
'CUDA core'

Executes one thread

SM
streaming multiprocessor
32xSP (or 16, 48 or more)

Fast local '**shared memory**'
(shared between SPs)
16 KiB (or 64 KiB)



Simplified CUDA Programming Model

- 计算由大量的相互独立的线程(*CUDA threads* or *microthreads*) 完成，这些线程组合成线程块（*thread blocks*）

```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{ for (int i=0; i<n; i++)
      y[i] = a*x[i] + y[i]; }

// CUDA version.
__host__   // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__   // Piece run on GP-GPU.
void daxpy(int n, double a, double*x, double*y)
{ int i = blockIdx.x*blockDim.x + threadId.x;
  if (i<n) y[i]=a*x[i]+y[i]; }
```

编程模型：SPMD

- Single procedure/program, multiple data
  - 它是一种编程模型而不是计算机组织
- 每个处理单元执行同样的过程，处理不同的数据
  - 这些过程可以在程序中的某个点上同步，例如 barriers

- 多条指令流执行相同的程序
  - 每个程序/过程
    - 操作不同的数据
    - 运行时可以执行不同的控制流路径
  - 许多科学计算应用以这种方式编程，运行在MIMD硬件结构上 (multiprocessors)
  - 现代 GPUs 以这种类似的方式编程，运行在SIMD硬件上

硬件执行模型： GPU is a SIMD (SIMT) Machine

- GPU不是用SIMD指令编程

- 使用线程 (SPMD 编程模型)
  - 每个线程执行同样的代码，但操作不同的数据元素
  - 每个线程有自己的上下文(即可以独立地启动/执行等）

- 一组执行相同指令的线程由硬件动态组织成warp
  - 一个warp是由硬件形成的SIMD操作

Threads and Blocks

- 一个线程对应一个数据元素
- 大量的线程组织成很多线程块
- 许多线程块组成一个网格

- GPU 由硬件对线程进行管理
  - Thread Block Scheduler
  - SIMD Thread Scheduler

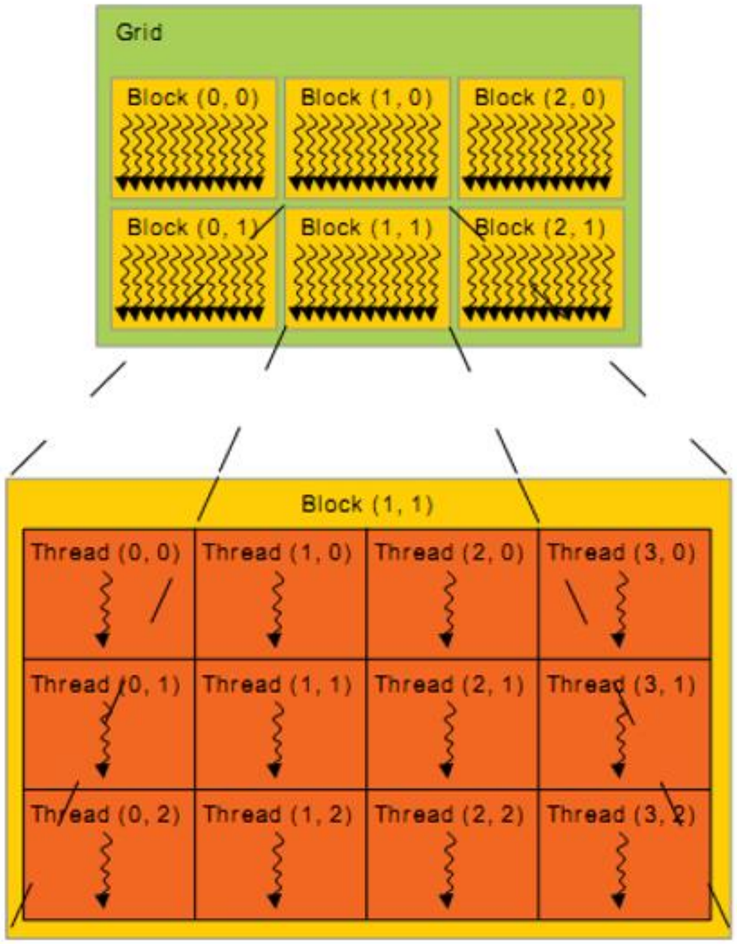


Figure 6 Grid of Thread Blocks

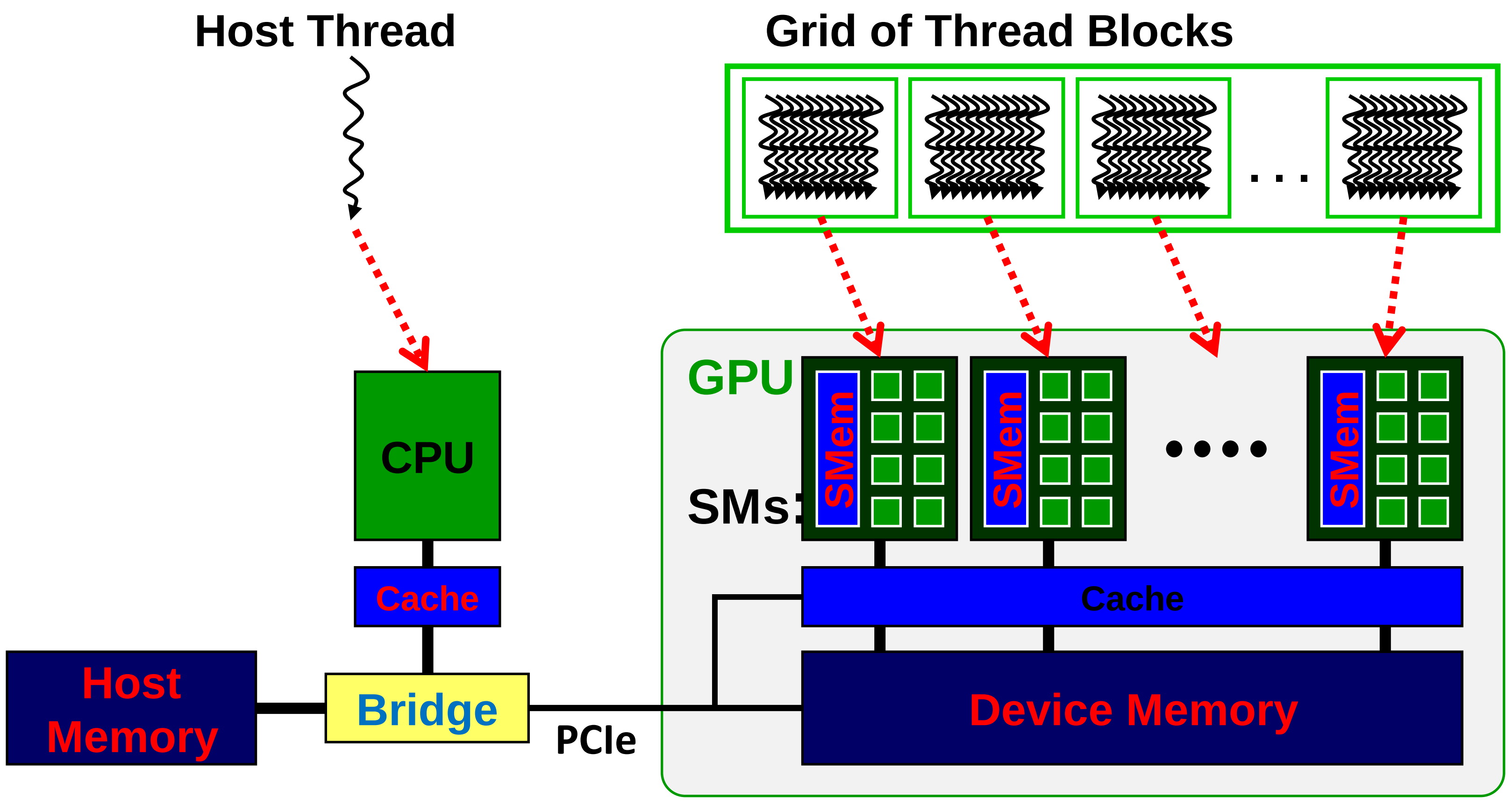
Simplified CUDA Programming Model

- 计算由大量的相互独立的线程(*CUDA threads* or *microthreads*) 完成，这些线程组合成线程块（*thread blocks*）

```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{ for (int i=0; i<n; i++)
      y[i] = a*x[i] + y[i]; }

// CUDA version.
__host__   // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__   // Piece run on GP-GPU.
void daxpy(int n, double a, double*x, double*y)
{ int i = blockIdx.x*blockDim.x + threadId.x;
  if (i<n) y[i]=a*x[i]+y[i]; }
```

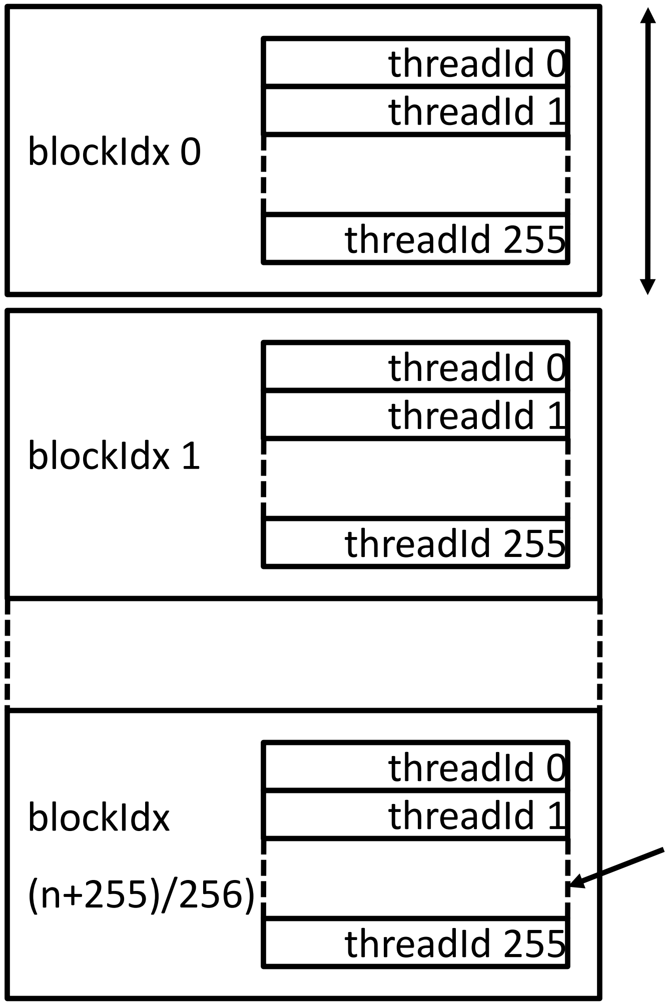
CUDA kernel maps to Grid of Blocks

```
kernel_func<<<nblk, nthread>>>(param, … );
```



Programmer’s View of Execution

创建足够的线程块以适应输入向量 (Nvidia 中将由多个线程块构成的、在GPU上运行的代码称为*Grid*，*Grid可以是2维的*)



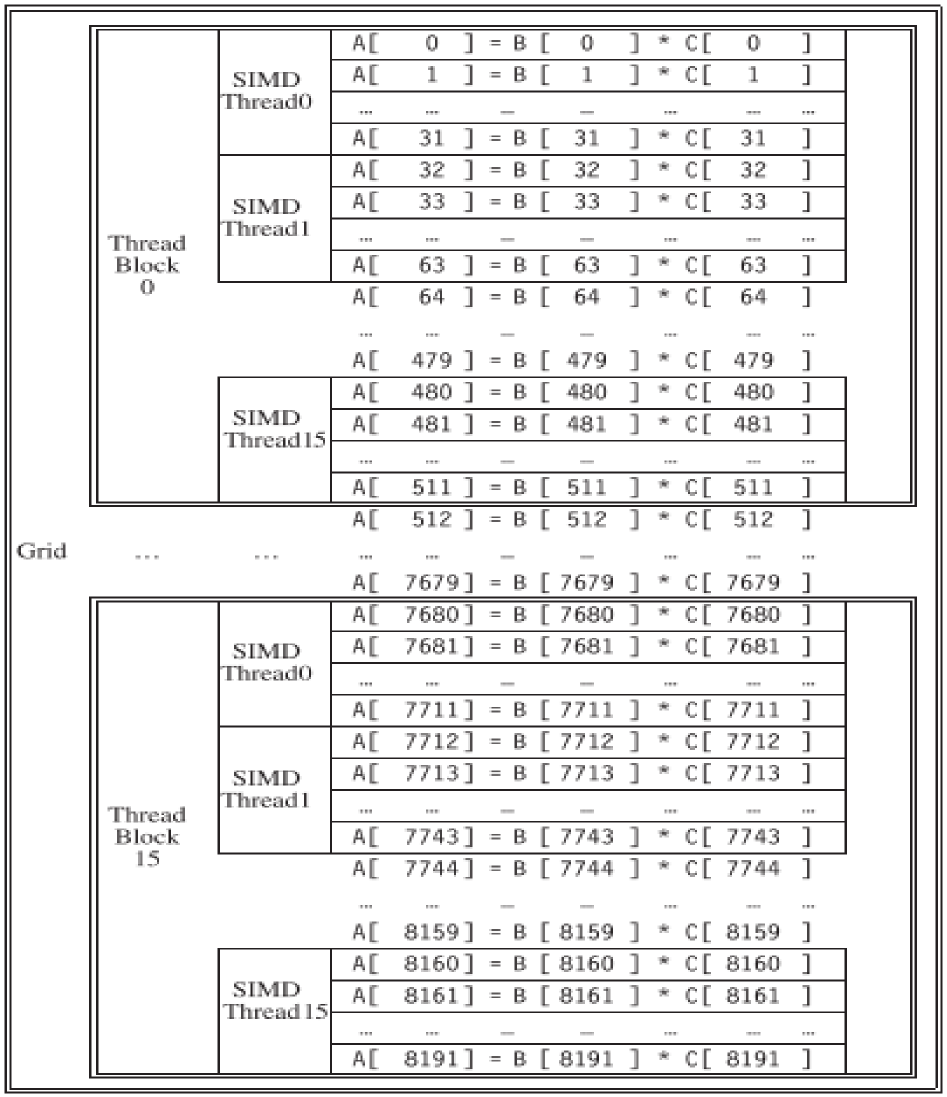
blockDim = 256 (programmer can choose)

Conditional **(i<n)** turns off unused threads in last block

处理向量长度为8192的程序组织：

Grid 包含16个线程块（512个元素/Block）

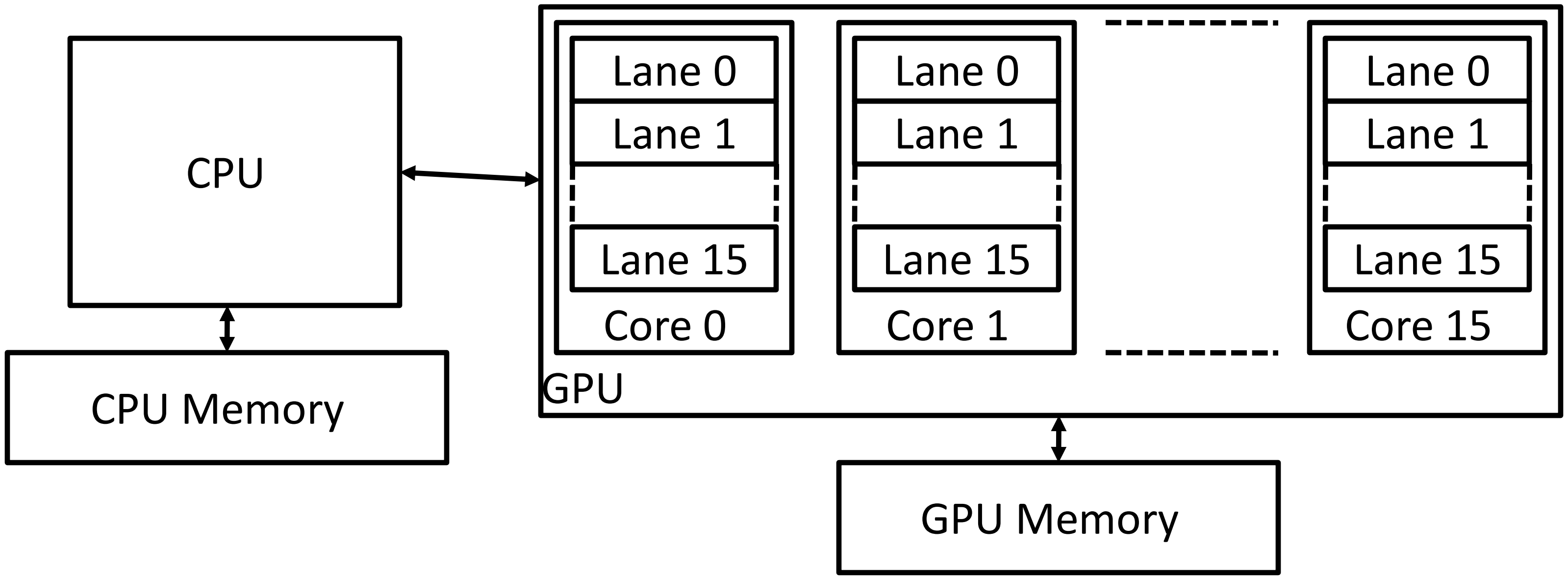
Block包含16个SIMD 线程
32个CUDA 线程/SIMD线程
处理一个元素/CUDA线程



Grid
Thread Block 0
SIMD Thread0
SIMD Thread1
SIMD Thread15
Thread Block 15
A[0] = B [0] * C[0]
A[1] = B [1] * C[1]
A[31] = B [31] * C[31]
A[32] = B [32] * C[32]
A[33] = B [33] * C[33]
A[63] = B [63] * C[63]
A[64] = B [64] * C[64]
A[479] = B [479] * C[479]
A[480] = B [480] * C[480]
A[481] = B [481] * C[481]
A[511] = B [511] * C[511]
A[512] = B [512] * C[512]
A[7679] = B [7679] * C[7679]
A[7680] = B [7680] * C[7680]
A[7681] = B [7681] * C[7681]
A[7711] = B [7711] * C[7711]
A[7712] = B [7712] * C[7712]
A[7713] = B [7713] * C[7713]
A[7743] = B [7743] * C[7743]
A[7744] = B [7744] * C[7744]
A[8159] = B [8159] * C[8159]
A[8160] = B [8160] * C[8160]
A[8161] = B [8161] * C[8161]
A[8191] = B [8191] * C[8191]

Figure 4.13 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector-vector multiply, with each vector being 8192 elements long. Each thread of SIMD instructions calculates 32 elements per instruction, and in this example, each Thread Block contains 16 threads of SIMD instructions and the Grid contains 16 Thread Blocks. The hardware Thread Block Scheduler assigns Thread Blocks to multithreaded SIMD Processors, and the hardware Thread Scheduler picks which thread of SIMD instructions to run each clock cycle within a SIMD Processor. Only SIMD Threads in the same Thread Block can communicate via local memory. (The maximum number of SIMD Threads that can execute simultaneously per Thread Block is 32 for Pascal GPUs.)

Hardware Execution Model



- GPU 由多个并行核构成，每个核是一个多线程SIMD处理器（包含多个车道（Lanes））
- CPU 发送整个 “grid”到GPU，由GPU将这些线程块分发到多个核上（每个线程块在一个核上运行）
    - GPU上核的数量对程序员而言是透明的

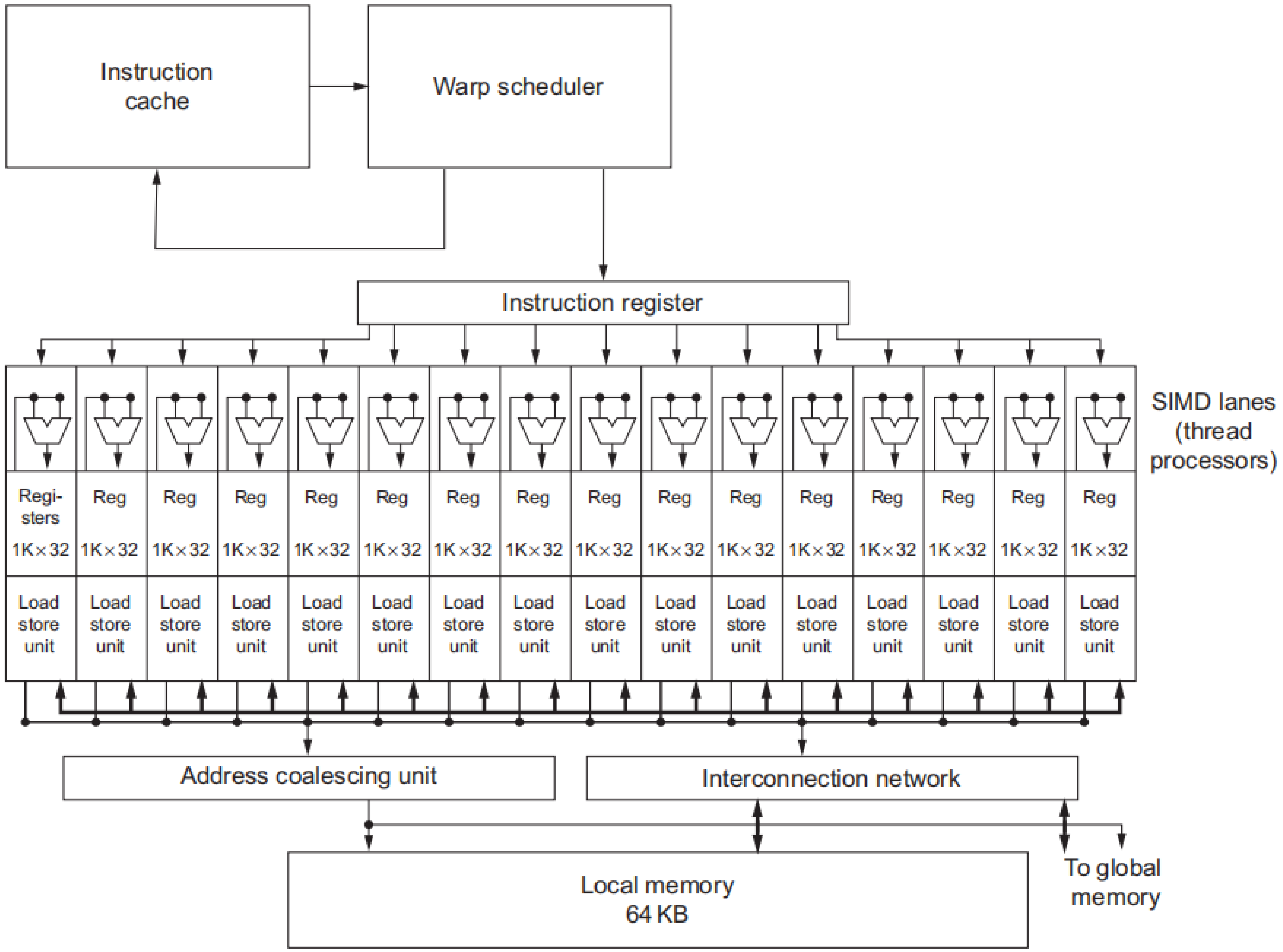
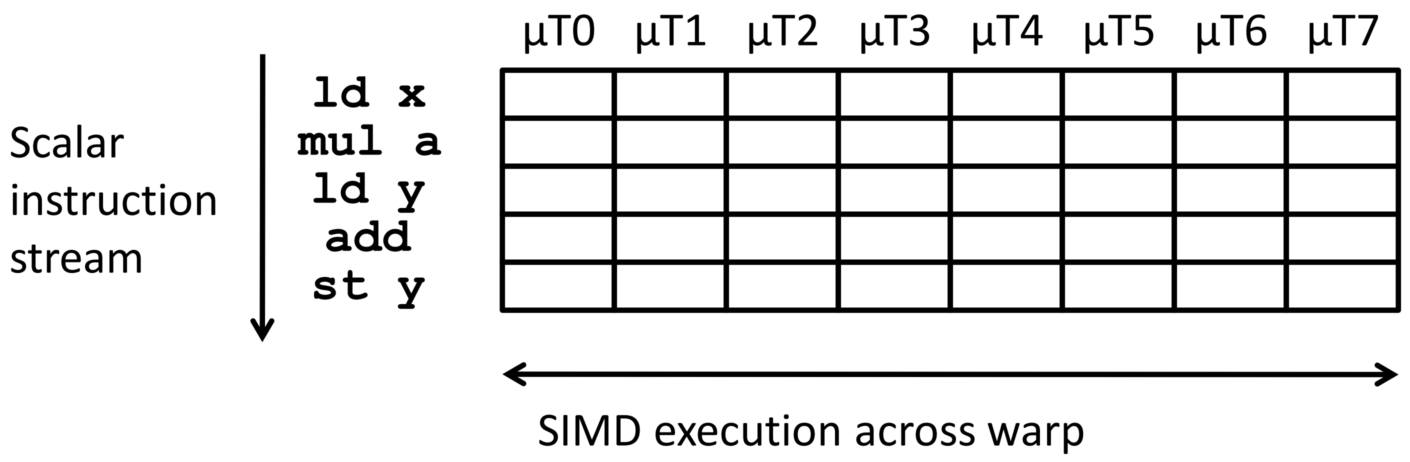


Figure 4.14 Simplified block diagram of a multithreaded SIMD Processor. It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

CUDA: "Single Instruction, Multiple Thread"

- GPUs 使用 SIMT模型, 每个CUDA线程的标量指令流汇聚在一起在硬件上以SIMD方式执行 (Nvidia groups 32 CUDA threads into a warp)



NVIDIA Instruction Set Arch.

- ISA 是硬件指令集的抽象
  - “Parallel Thread Execution (PTX)”
  - 使用虚拟寄存器
  - 用软件将其翻译成机器码

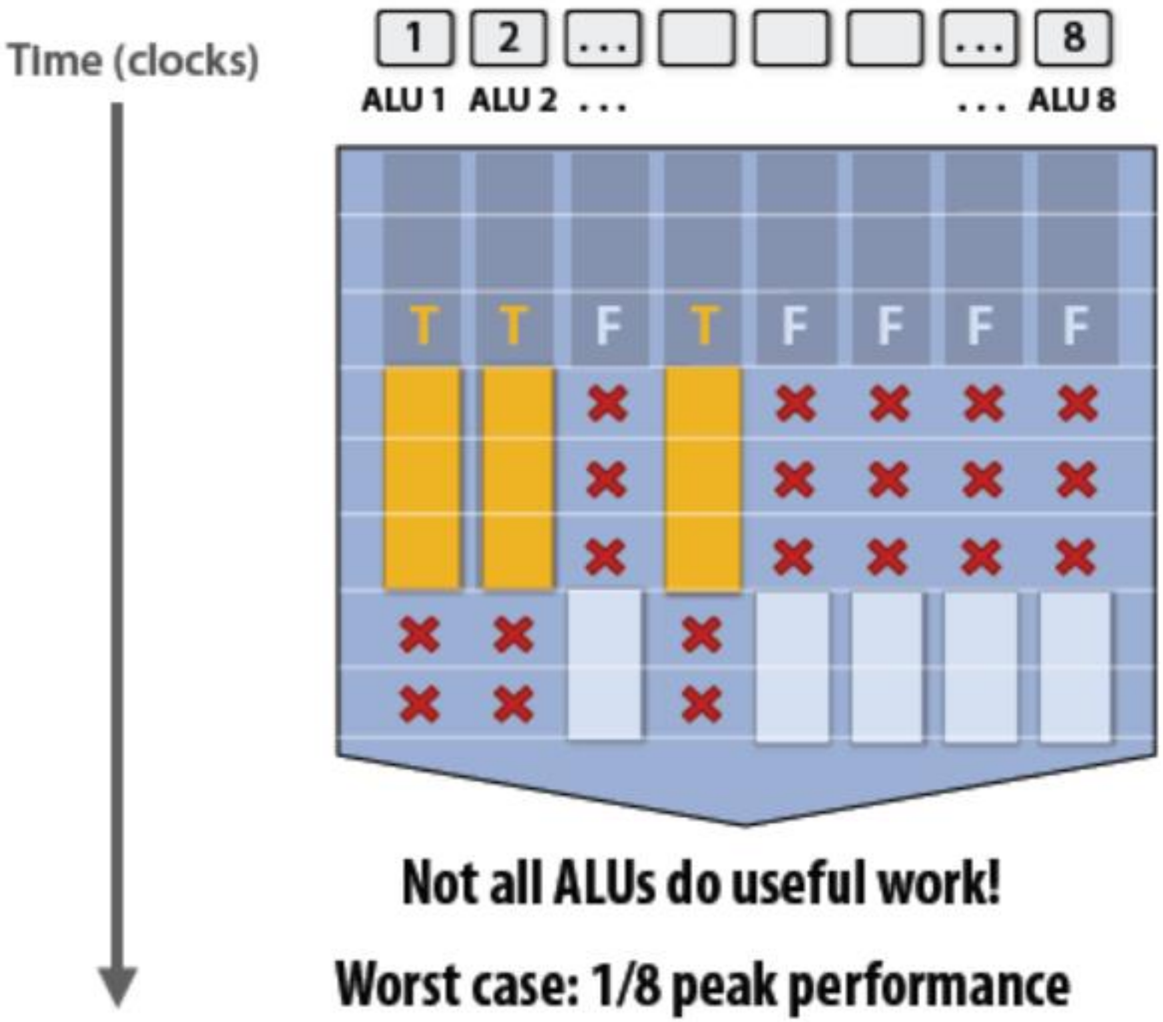
  - Example:

```
shl.s32  R8, blockIdx, 9        ; Thread Block ID * Block size (512 or 2^9)
add.s32  R8, R8, threadIdx  ; R8 = i = my CUDA thread ID
ld.global.f64     RD0, [X+R8]    ; RD0 = X[i]
ld.global.f64     RD2, [Y+R8]    ; RD2 = Y[i]
mul.f64   RD0, RD0, RD4        ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64   RD0, RD0, RD2        ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64    [Y+R8], RD0  ; Y[i] = sum (X[i]*a + Y[i])
```

Conditional Branching

- GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

设置掩码（Mask）寄存器，屏蔽部分ALU的输出



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

 float x = A[i];

 if (x > 0) {
     float tmp = exp(x,5.f);
     tmp *= kMyConst1;
     x = tmp + kMyConst2;
 } else {
     float tmp = kMyConst1;
     x = 2.f * tmp;
 }

<resume unconditional code>

 result[i] = x;
```

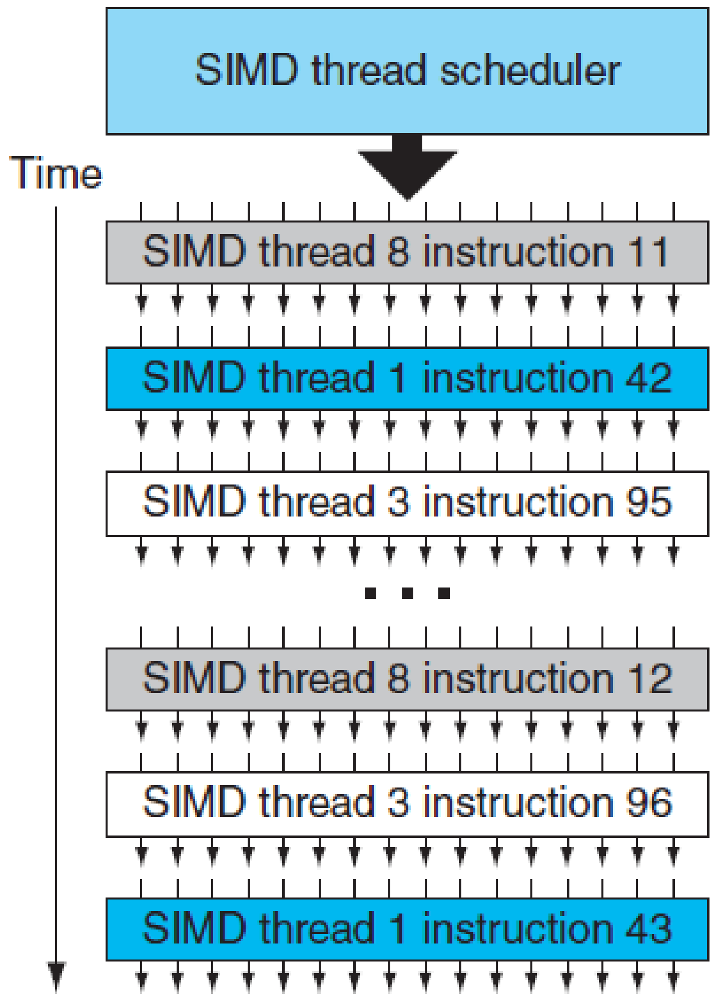
每一条指令，都在所有的ALU上同步执行，如果有不一致的指令流，即分支指令，会导致同步性能下降！最坏会慢8倍！

举例：if (X[i] != 0)
X[i] = X[i] – Y[i];
else X[i] = Z[i];

分支指令，编译后得到的Nvidia GPU“Parallel Thread Execution (PTX)” 指令

```
        ld.global.f64  RD0, [X+R8]      ; RD0 = X[i]
        setp.neq.s32  P1, RD0, #0       ; P1 is predicate register 1

        @!P1, bra     ELSE1,            ; if P1 false, got ELSE1

        ld.global.f64   RD2, [Y+R8]     ; RD2 = Y[i]
        sub.f64 RD0, RD0, RD2           ; Difference in RD0
        st.global.f64   [X+R8], RD0     ; X[i] = RD0

        @P1, bra      ENDIF1            ; if P1 true, go to ENDIF1

ELSE1:  ld.global.f64 RD0, [Z+R8]       ; RD0 = Z[i]
        st.global.f64 [X+R8], RD0       ; X[i] = RD0

ENDIF1: <next instruction>,             ;
```

Scheduling of threads of SIMD instructions

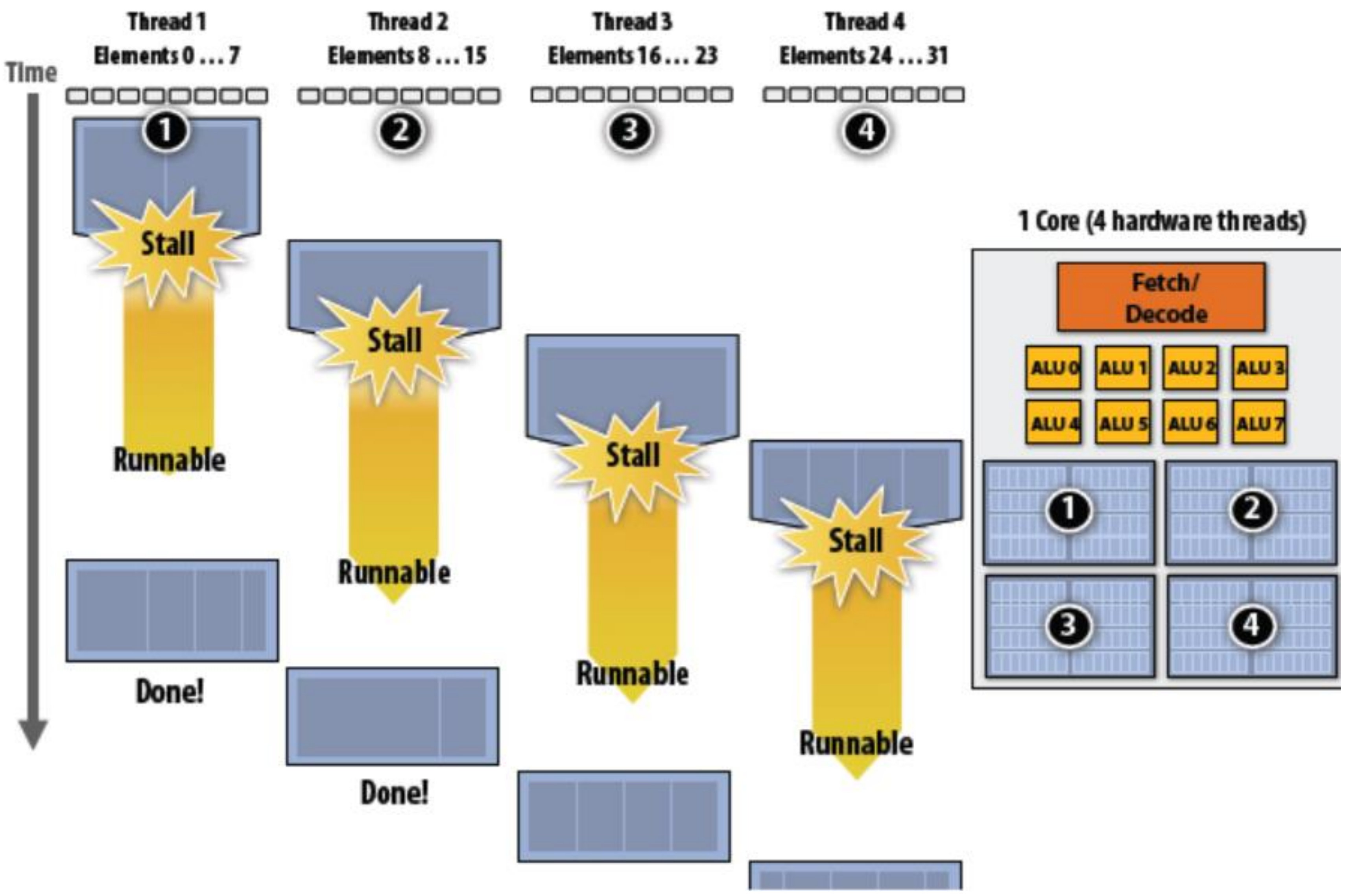

SIMD thread scheduler
Time
SIMD thread 8 instruction 11
SIMD thread 1 instruction 42
SIMD thread 3 instruction 95
SIMD thread 8 instruction 12
SIMD thread 3 instruction 96
SIMD thread 1 instruction 43

调度器每次选择一个不同的 SIMD thread 发射其ready的指令

一条指令会让所有的SIMD lane（16个着色器、SP）同时执行.

A Thread of SIMD Instructions (SIMD thread)
= A warp

利用多线程隐藏访存延迟

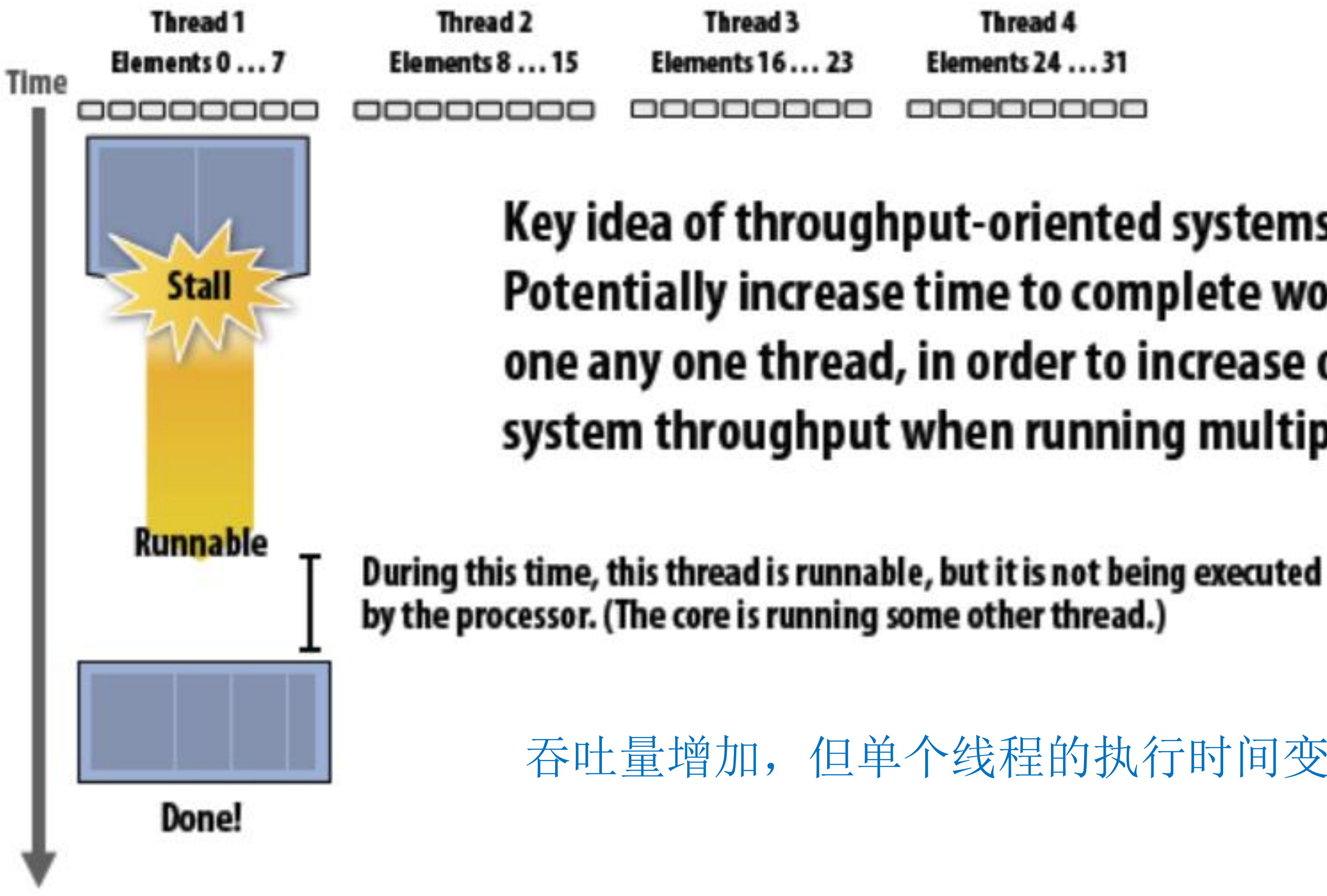


Example of Hiding Memory Latency

- G80:
  - 执行warp全部线程的一条指令需要4个时钟cycle
  - 假定1 global memory access / 8 instructions
  - A 400-cycle global memory latency
  - How many warps are needed to tolerate the latency?

400 cycles × 1 MEM/ 8 cycles = 50 cycle-latency per instruction on average
50 cycles / 4 cycles per warp = 12.5 ⇒ 13 warps to keep an SM busy

吞吐量增加引起的trade off

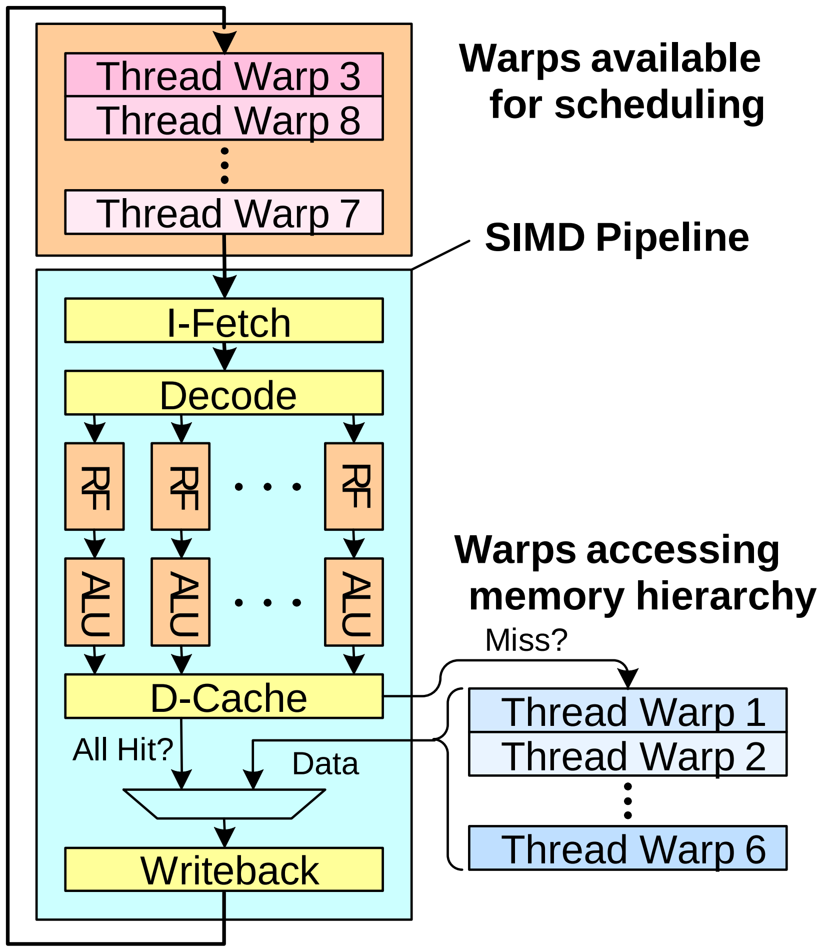

Thread 1
Elements 0 … 7
Thread 2
Elements 8 … 15
Thread 3
Elements 16 … 23
Thread 4
Elements 24 … 31
Time
Stall
Runnable
During this time, this thread is runnable, but it is not being executed by the processor. (The core is running some other thread.)
Done!

Key idea of throughput-oriented systems:
Potentially increase time to complete work by any one any one thread, in order to increase overall system throughput when running multiple threads.

吞吐量增加，但单个线程的执行时间变长

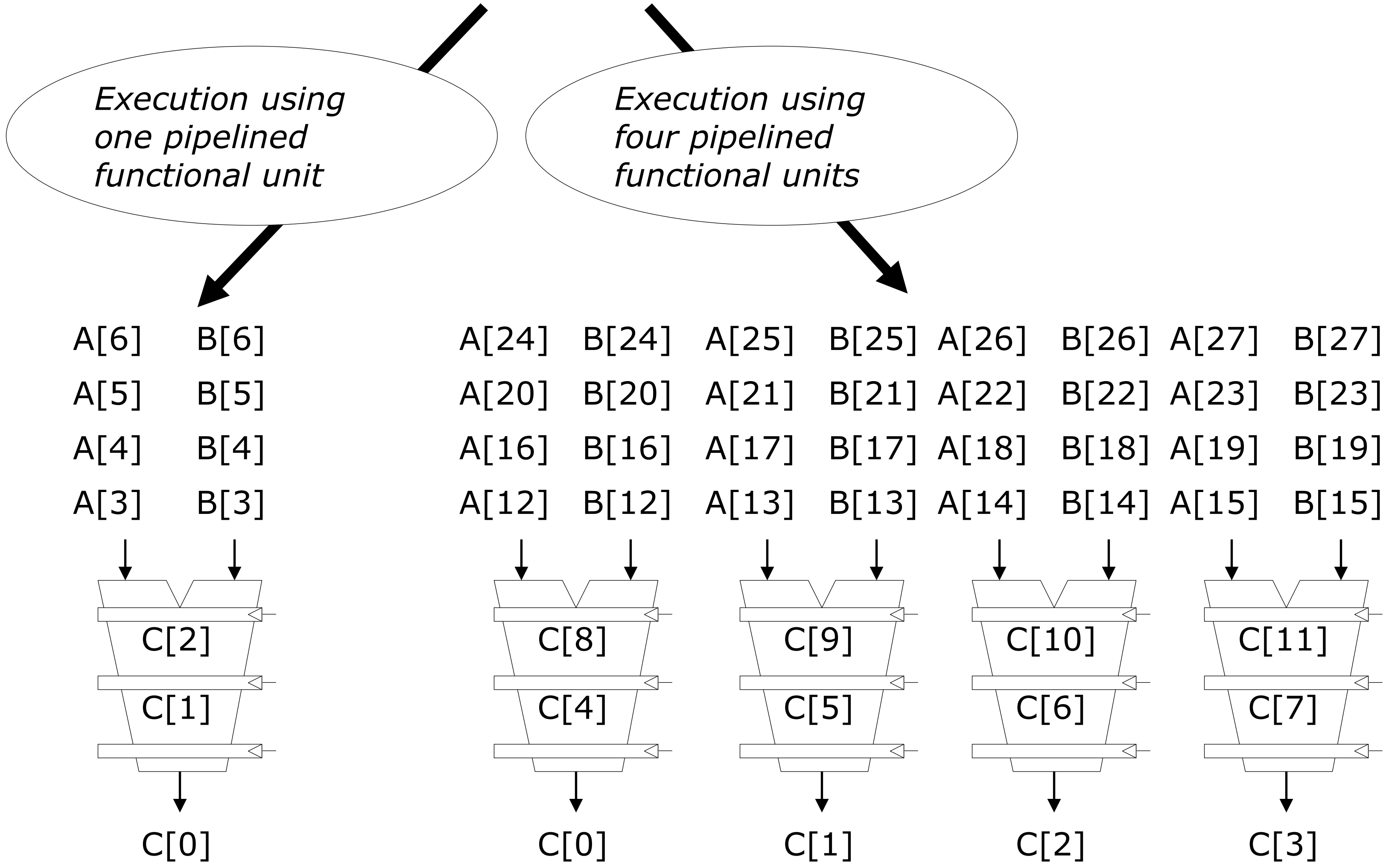
Latency Hiding via Warp-Level FGMT

- Warp: 一组执行相同指令的线程（作用于不同的数据元素）
- Fine-grained multithreading
  - 流水线上每次执行每个线程的一条指令 (No interlocking)
  - 通过warp的交叉执行来隐藏延时
- 所有线程的寄存器值保存在寄存器文件中
- FGMT 可以容忍长延时
  - Millions of pixels



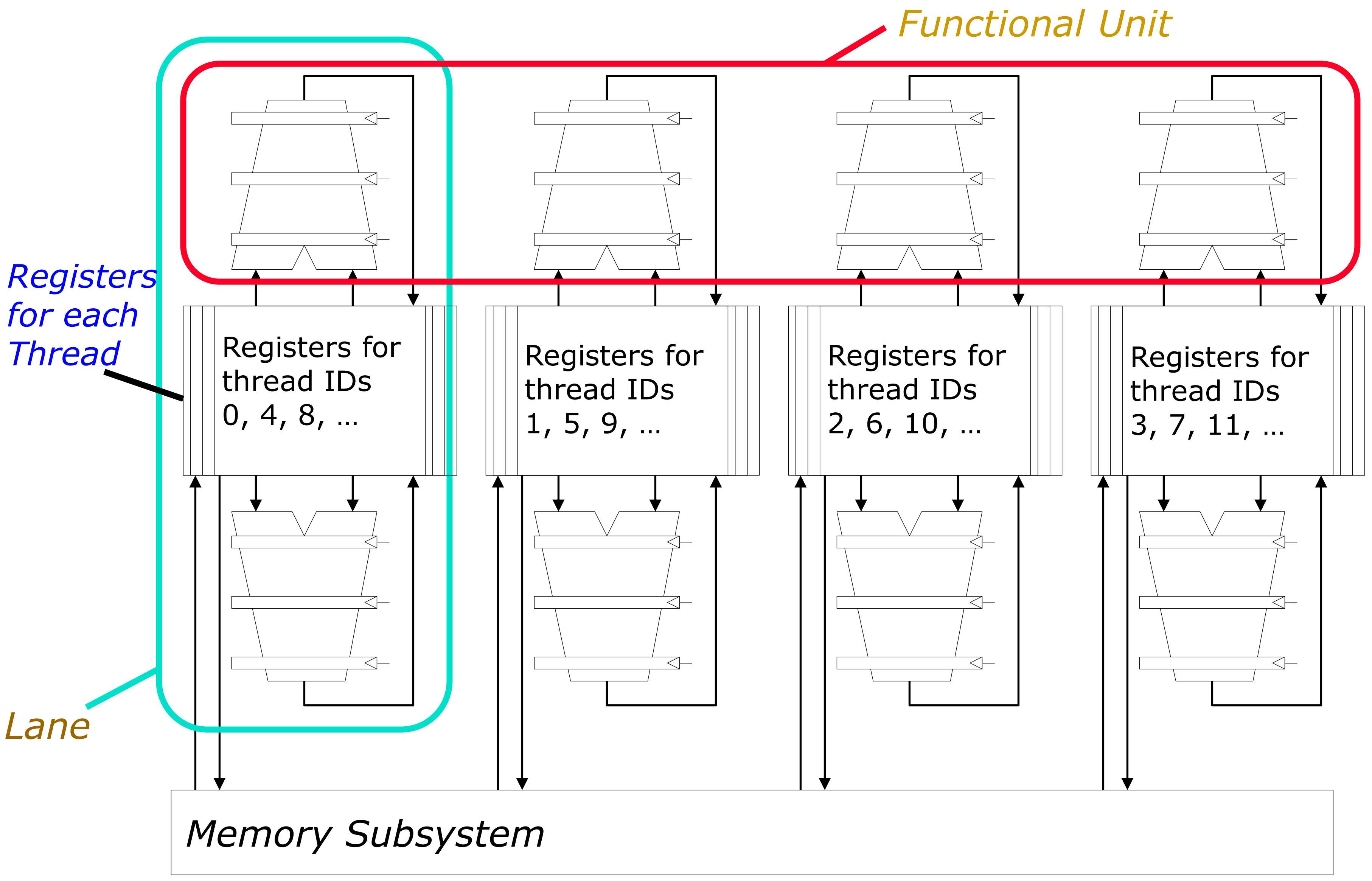
Warp Execution (Recall the Slide)

32-thread warp executing ADD A[tid],B[tid] → C[tid]

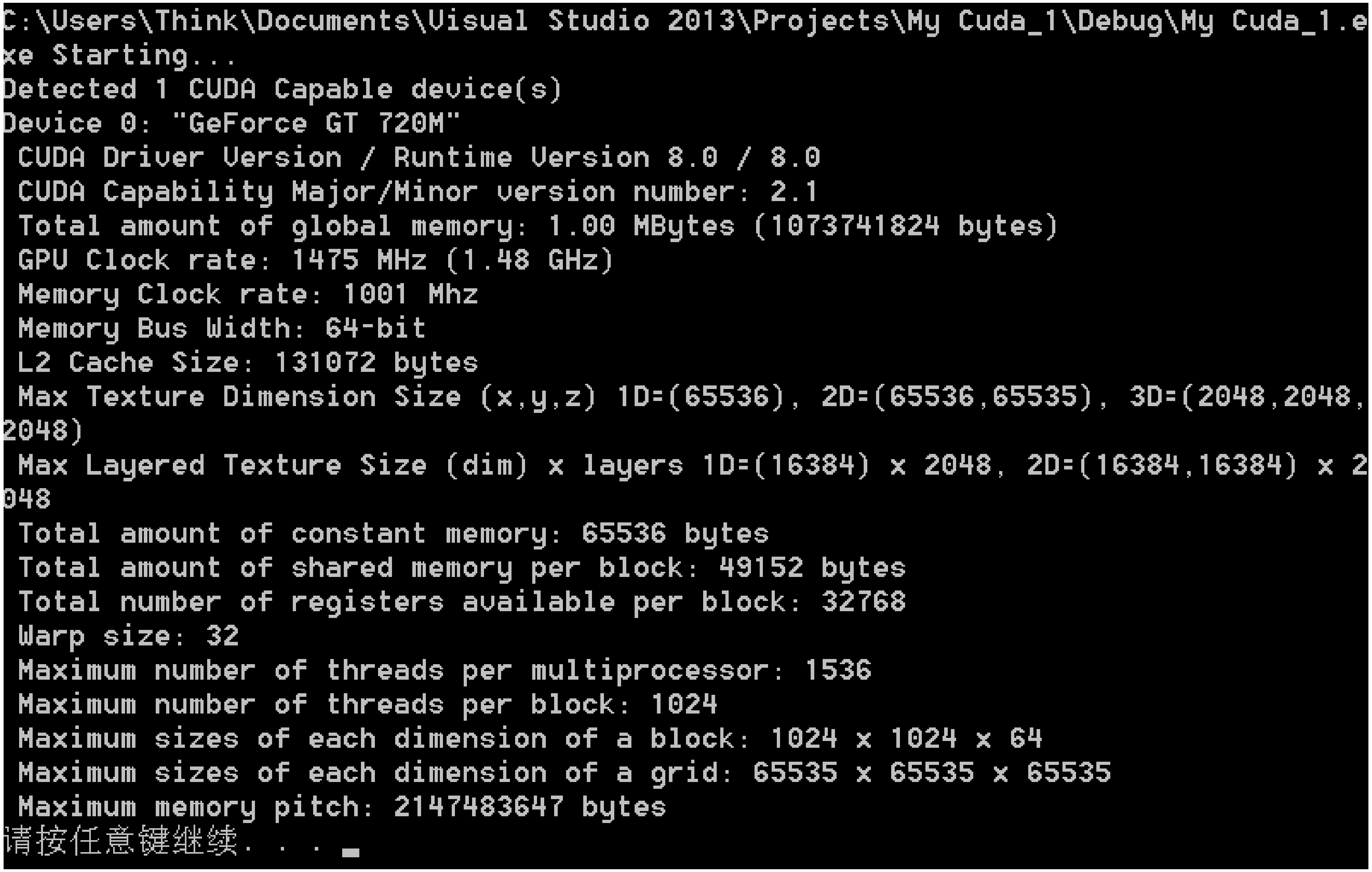


Execution using one pipelined functional unit
Execution using four pipelined functional units
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]
A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]

SIMD Execution Unit Structure


Functional Unit
Registers for each Thread
Registers for thread IDs 0, 4, 8, …
Registers for thread IDs 1, 5, 9, …
Registers for thread IDs 2, 6, 10, …
Registers for thread IDs 3, 7, 11, …
Lane
Memory Subsystem

CUDA configuration data reported by deviceQuery



```
C:\Users\Think\Documents\Visual Studio 2013\Projects\My Cuda_1\Debug\My Cuda_1.e
xe Starting...
Detected 1 CUDA Capable device(s)
Device 0: "GeForce GT 720M"
 CUDA Driver Version / Runtime Version 8.0 / 8.0
 CUDA Capability Major/Minor version number: 2.1
 Total amount of global memory: 1.00 MBytes (1073741824 bytes)
 GPU Clock rate: 1475 MHz (1.48 GHz)
 Memory Clock rate: 1001 Mhz
 Memory Bus Width: 64-bit
 L2 Cache Size: 131072 bytes
 Max Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536,65535), 3D=(2048,2048,
2048)
 Max Layered Texture Size (dim) x layers 1D=(16384) x 2048, 2D=(16384,16384) x 2
048
 Total amount of constant memory: 65536 bytes
 Total amount of shared memory per block: 49152 bytes
 Total number of registers available per block: 32768
 Warp size: 32
 Maximum number of threads per multiprocessor: 1536
 Maximum number of threads per block: 1024
 Maximum sizes of each dimension of a block: 1024 x 1024 x 64
 Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535
 Maximum memory pitch: 2147483647 bytes
请按任意键继续. . .
```

Warp-based SIMD vs. Traditional SIMD

- 传统的SIMD仅包含一个线程
  - Lock step: 一条向量指令执行完，然后启动下一条向量指令
  - 编程模型为SIMD (no extra threads) → SW 需要知道向量长度
  - ISA 包含vector/SIMD指令

- Warp-based SIMD 由多个标量线程构成，以SIMD方式执行 (即所有的线程执行相同的指令)
  - 不需要lock step
  - 每个线程可以单独对待(即可以映射到不同的warp中），编程模型不是SIMD
    - SW 不必知道向量长度
    - 多个线程可以动态构成warp
  - ISA是标量ISA → 可以动态形成逻辑上的向量指令
  - 是一种在SIMD硬件上实现的SPMD编程模型

总结：GPU的三个key idea

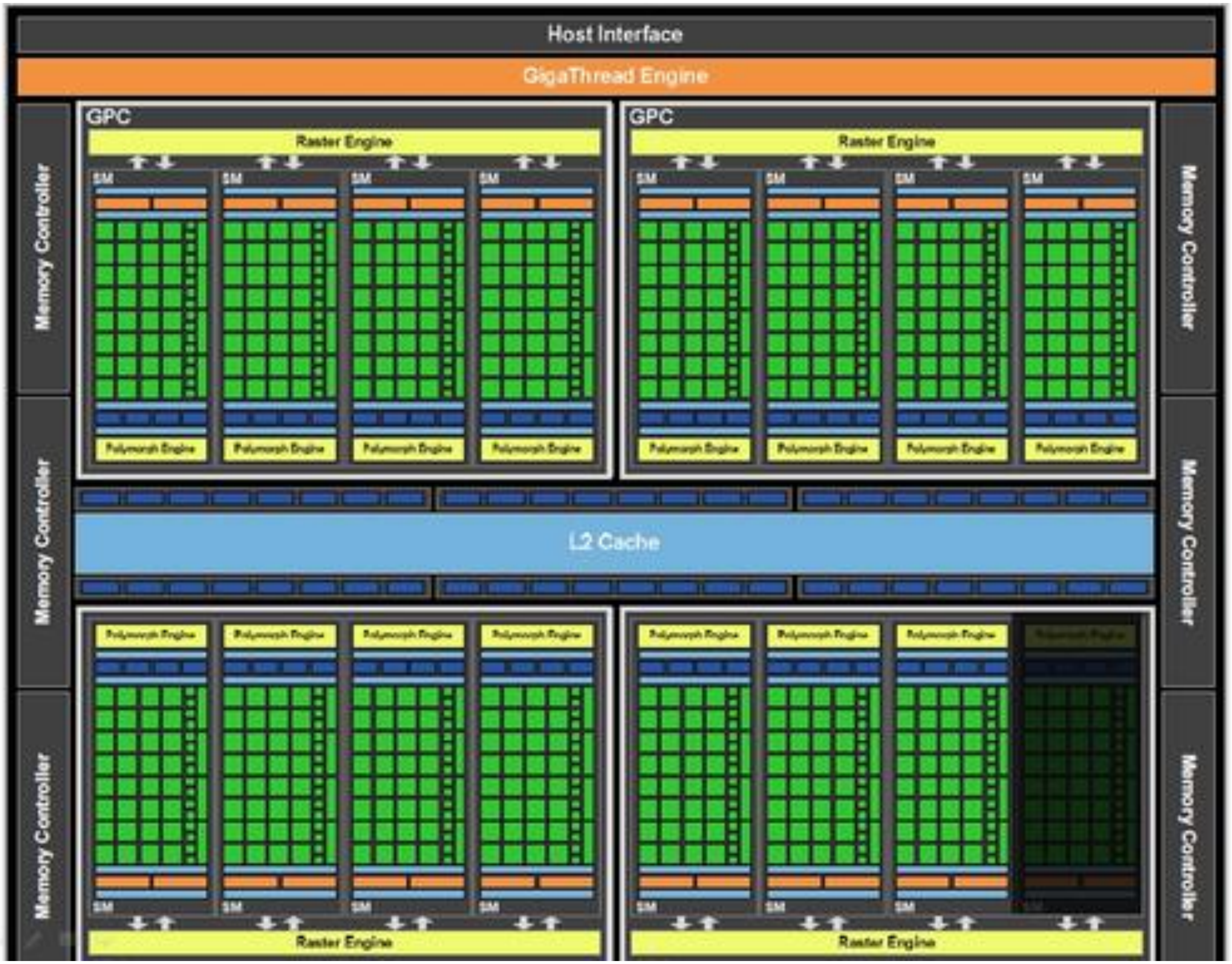
- Employ multiple processing cores
  - 简单的核 (比起ILP， 更注重TLP： thread-level parallelism)
- Pack cores full of ALU
  - 开发数据级的并行性 (SIMD)
  - 一条指令执行时，多个ALU同步处理不同的数据
  - 芯片面积额外增加的开销不大，但大大增强了计算能力
- 利用多线程提高系统的处理能力（吞吐量）
  - 轮流交替执行不同线程的代码段以隐藏延迟

Nvidia 通用图形加速单元体系结构

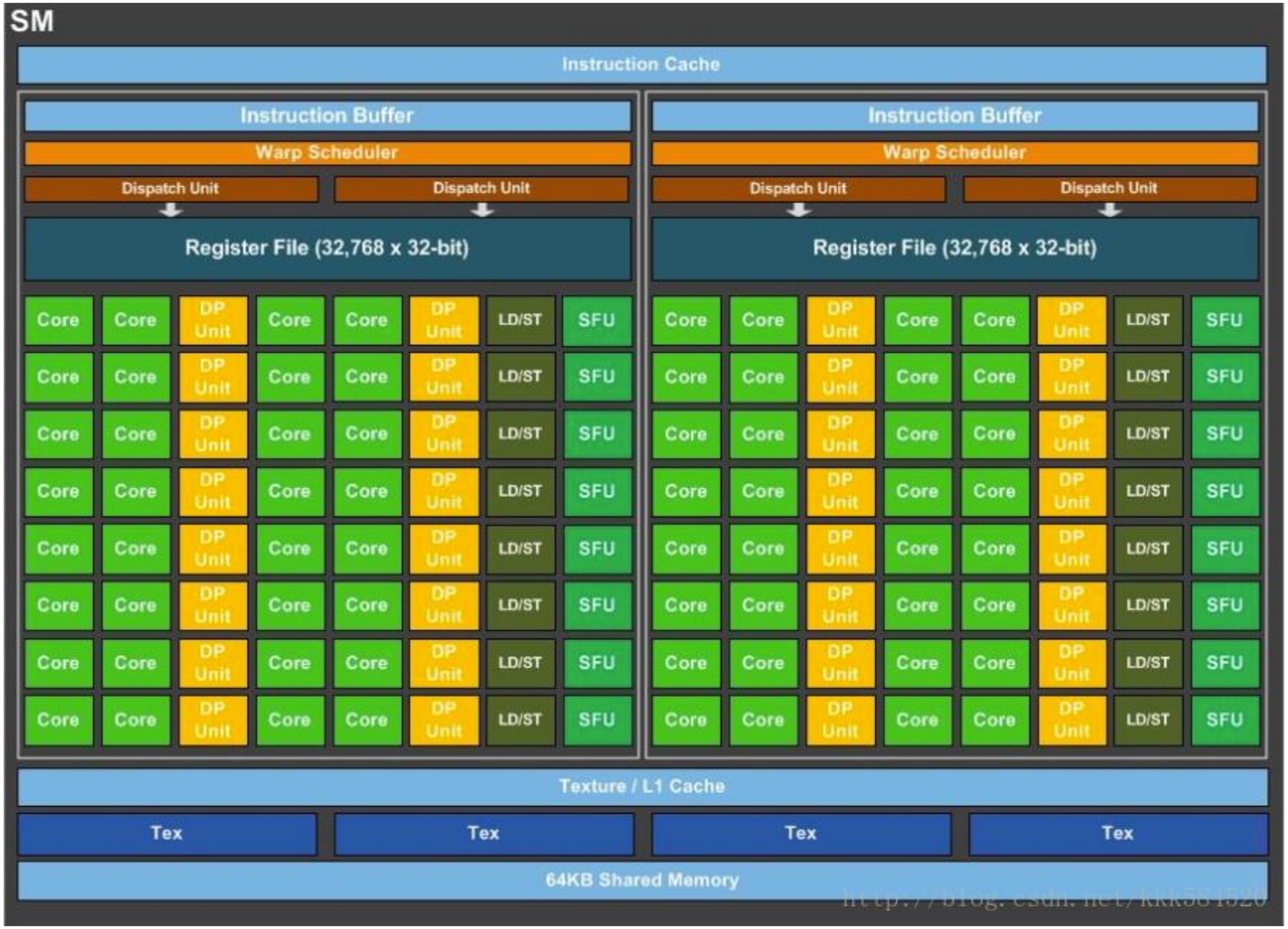
- 2008 Tesla
- 2010 Fermi
- 2012 Kepler
- 2014 Maxwell
- 2016 Pascal

- 每个SM包含的SP（GPU core）数量依据GPU架构而不同，Fermi架构GF100是32个，GF10X是48个，Kepler架构都是192个，Maxwell都是128个，Pascal64个。

Floorplan of Fermi GTX480



Pascal 架构(Tesla P100 2016)



Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
Peak FP32 GFLOPs[1]	5040	6840	10600
Peak FP64 GFLOPs[1]	1680	210	5300
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion
GPU Die Size	551 mm²	601 mm²	610 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET

MEMORY MODEL OF CUDA AND COOPERATING THREADS

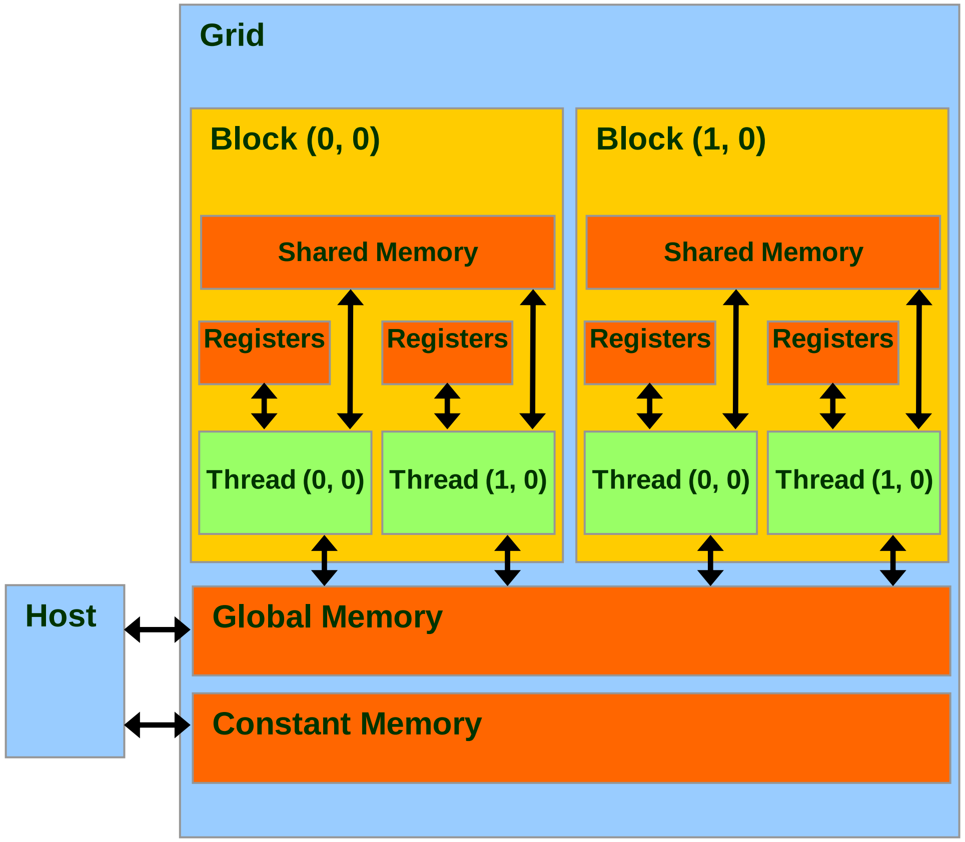
CUDA Variable Type Qualifiers

Variable declaration		Memory	Scope	Lifetime
`__device__ __local__`	`int LocalVar;`	local	thread	thread
`__device__ __shared__`	`int SharedVar;`	shared	block	block
`__device__`	`int GlobalVar;`	global	grid	application
`__device__ __constant__`	`int ConstantVar;`	constant	grid	application

- **`__device__`** is optional when used with **`__local__`**, **`__shared__`**, or **`__constant__`**
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

G80 Implementation of CUDA Memories

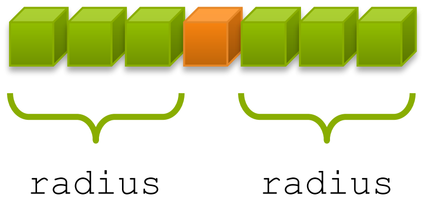
- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread local memory
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read/only per-grid **constant memory**



1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

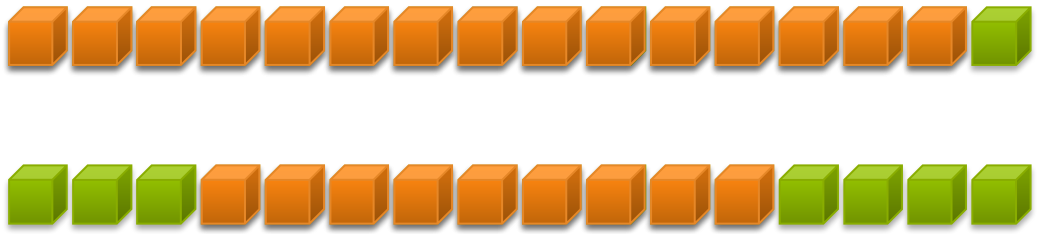
- If radius is 3, then each output element is the sum of 7 input elements:



Implementing Within a Block

- Each thread processes one output element
  - blockDim.x elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times



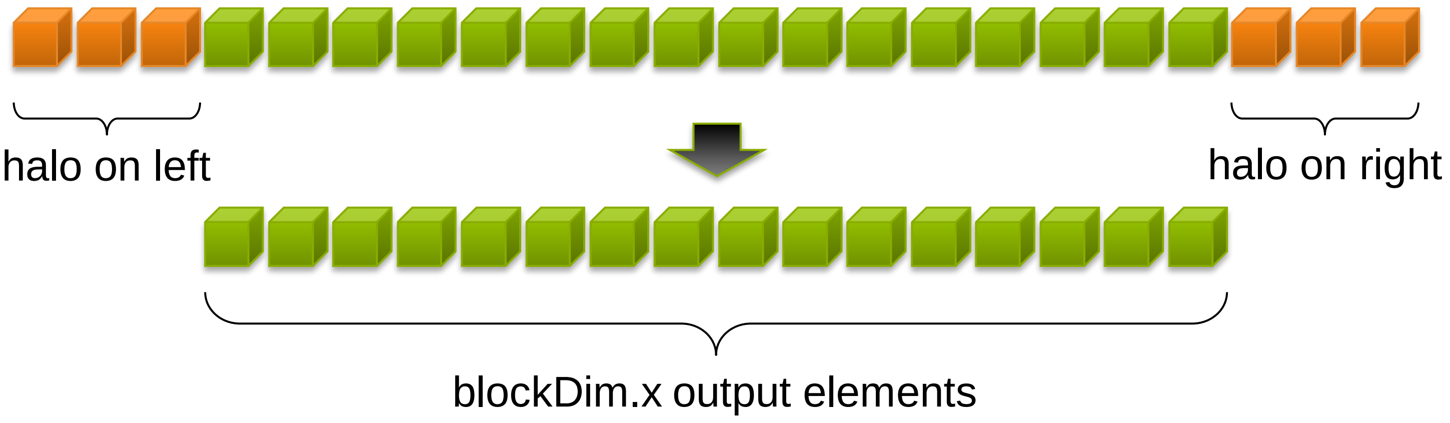
Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

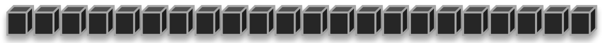
- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory

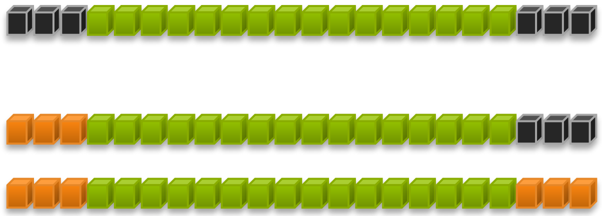
  - Each block needs a halo of radius elements at each boundary



Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```





Stencil Kernel

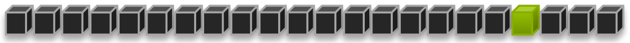
```
  // Apply the stencil
      result = 0;
  for (    offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}
```

Data Race!

- The stencil example will not work…
- Suppose thread 15 reads the halo before thread 0 has fetched it…

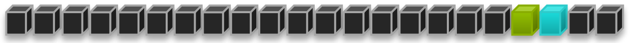
```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex – RADIUS = in[gindex – RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]



Skipped, threadIdx > RADIUS

Load from temp[19]



__syncthreads()

- `__syncthreads();`

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
               int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
                 ();
```

Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

Review (1 of 2)

- Launching parallel threads
  - Launch `N` blocks with `M` threads per block with `kernel<<<N,M>>>(…);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x
```

Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks

- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

下一节

- 线程级并行性（TLP）
  - 多核处理器中的关键问题
  - 有预习视频

From : H&P Computer Architecture: A Quantitative Approach, Fifth Edition, （5th edition)

谢　谢！