CUDA优化

张峰

信息楼427

中國人民大學
RENMIN UNIVERSITY OF CHINA

目录

- 性能考虑因素
- 优化例子：reduction

性能考虑因素

前提：注意！

- 永远记得测量时间花费在哪里！
  - 即使你认为知道哪个部分花费时间久
  - 由粗粒度到细粒度慢慢测量
- 优化任何代码时时刻谨记Amdahl优化定律
  - 不要一直在优化只占总体执行时间一小部分的程序代码

阿姆达尔曾致力于并行处理系统的研究。对于固定负载情况下描述并行处理效果的加速比s，阿姆达尔经过深入研究给出了如下公式：

$$S=1/(1-a+a/n)$$

其中，a为并行计算部分所占比例，n为并行处理结点个数。这样，当1-a=0时，(即没有串行，只有并行)最大加速比s=n；当a=0时（即只有串行，没有并行），最小加速比s=1；当n→∞时，极限加速比s→ 1/（1-a），这也就是加速比的上限。

性能考虑因素

- Memory Coalescing
- Shared Memory Bank Conflicts
- Occupancy

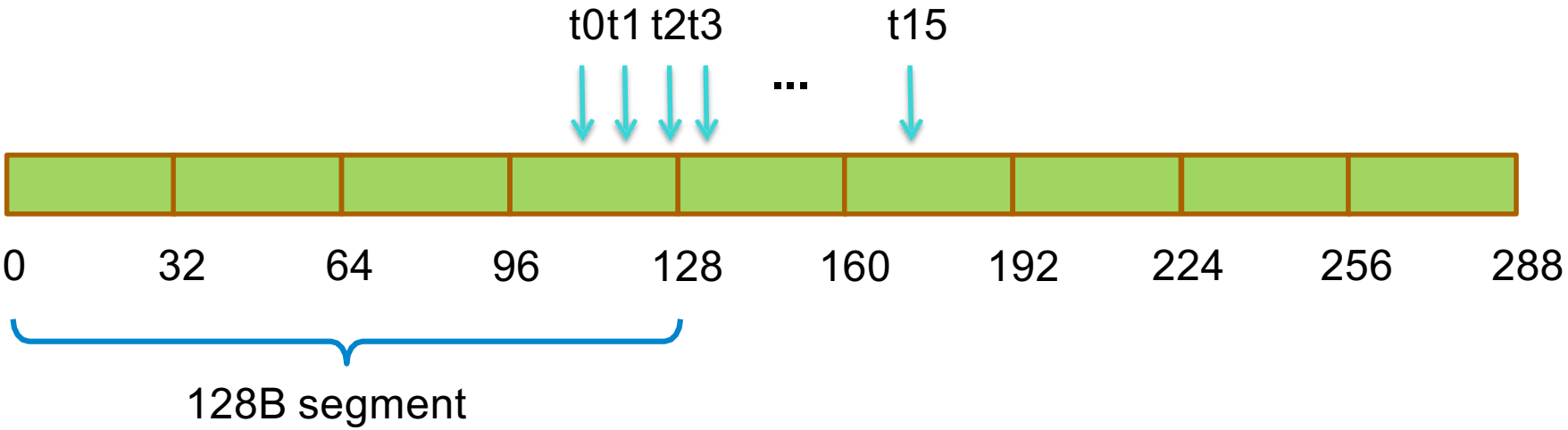
- Kernel launch overheads（kernel开启时间）
- Loop iteration count divergence（loop分支）

Memory Coalescing

- off-chip memory以chunk为单位进行访问
  - 即使只读一个字（word, 2 bytes）
  - 如果不使用整个chunk的内容，内存带宽就被浪费了
- chunk通常以32/64/128字节进行对齐
  - 没有对齐(unaligned)会带来性能开销

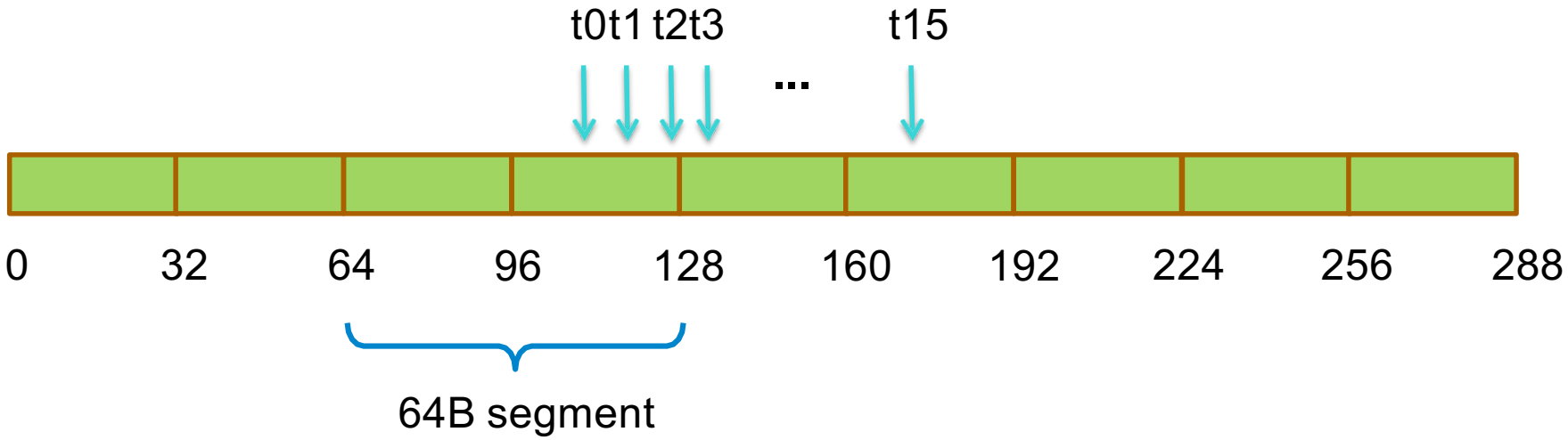
线程0到线程15以4byte为单位依次读取内存地址116—176的数据

- 线程0访问地址116的4 byte数据
- 线程1访问地址120的4 byte数据
- …
- 前128-byte的部分：0-127



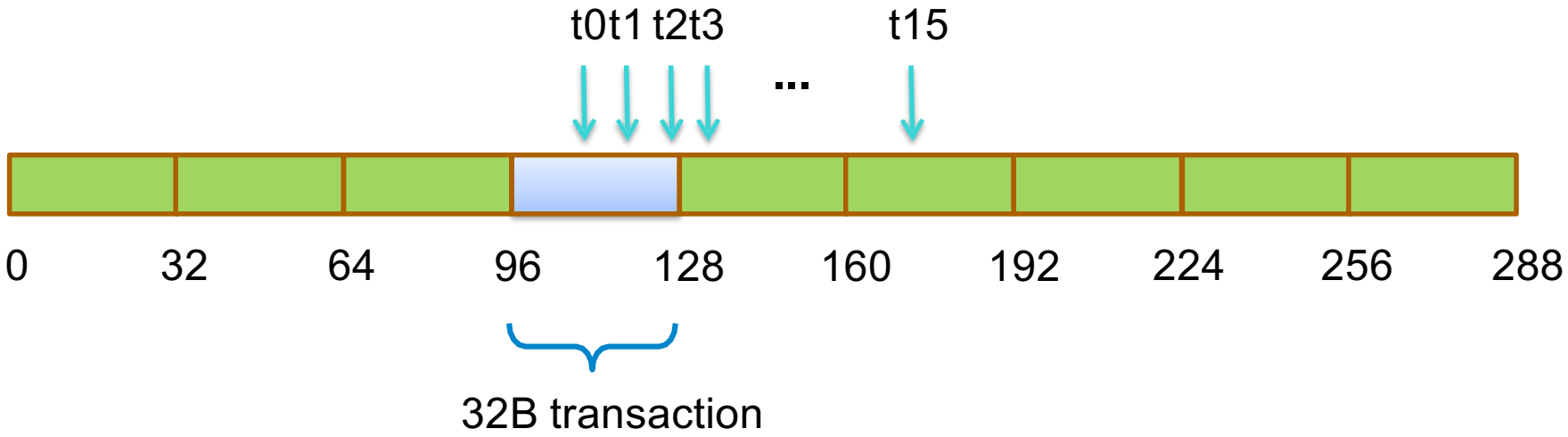
线程0到线程15以4byte为单位依次读取内存地址116—176的数据

- 线程0访问地址116的4 byte数据
- 线程1访问地址120的4 byte数据
- …
- 前128-byte的部分：0-127(reduce to 64)



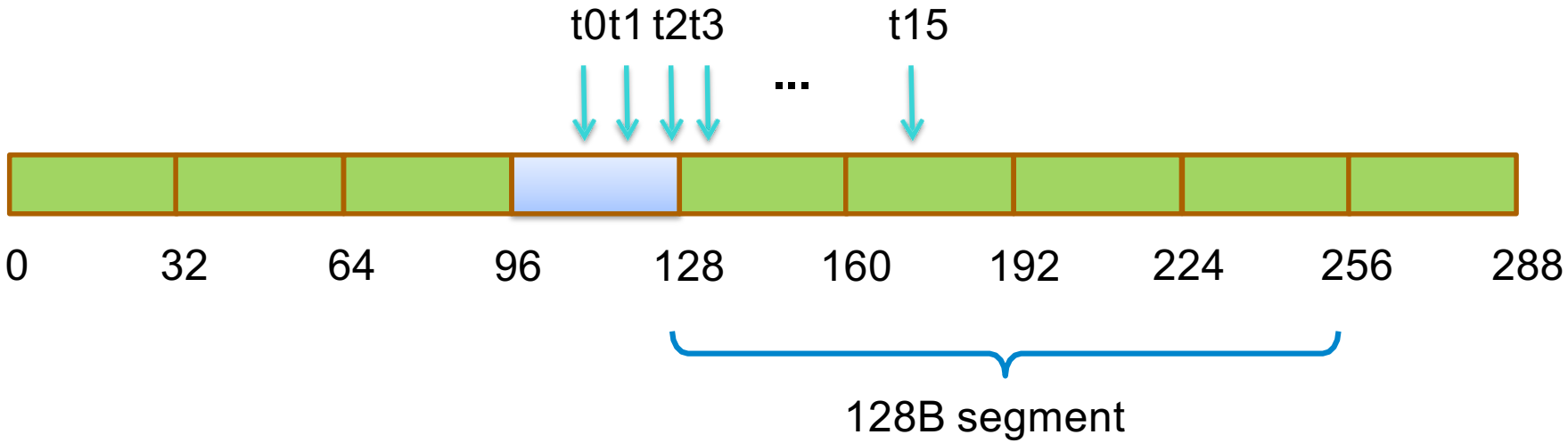
线程0到线程15以4byte为单位依次读取内存地址116—176的数据

- 线程0访问地址116的4 byte数据
- 线程1访问地址120的4 byte数据
- …
- 前128-byte的部分：0-127(**reduce to 32**)



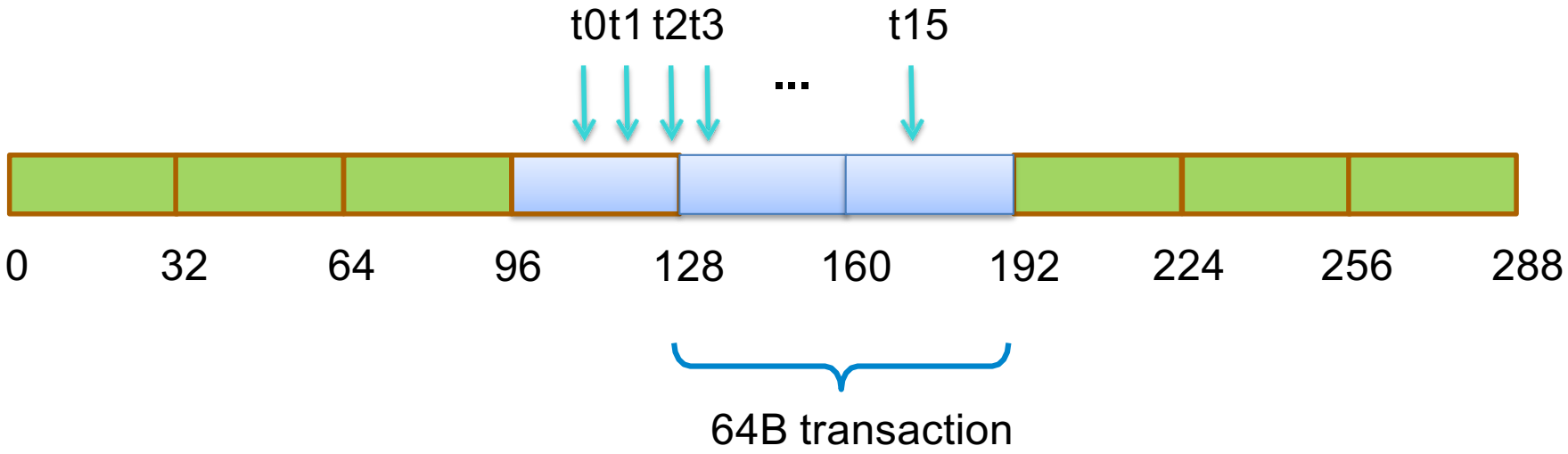
线程0到线程15以4byte为单位依次读取内存地址116—176的数据

- 线程0访问地址116的4 byte数据
- 线程1访问地址120的4 byte数据
- …
- 后128-byte的部分：128-255



线程0到线程15以4byte为单位依次读取内存地址116—176的数据

- 线程0访问地址116的4 byte数据
- 线程1访问地址120的4 byte数据
- …
- 后128-byte的部分：128-255(**reduce to 64**)



数据访问步长


```
__global__ void foo(int* input,
                    float3* input2)
{
  int i = blockDim.x * blockIdx.x
        + threadIdx.x;
  // Stride 1
  int a = input[i];
  // Stride 2, half the bandwidth is wasted
  int b = input[2*i];
  // Stride 3, 2/3 of the bandwidth wasted
  float c = input2[i].x;
}
```

例子：Array of Structures (AoS)


```
struct record
{
   int key;
   int value;
   int flag;
};

record  *d_records;
cudaMalloc((void**)&d_records, ...);
```

例子：Structure of Arrays (SoA)


```
struct SoA
{
  int  * keys;
  int  * values;
  int  * flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);
```

SoA vs AoS


```
__global__ void bar(record *AoS_data,
                    SoA SoA_data)
{
  int i = blockDim.x * blockIdx.x
        + threadIdx.x;
  // AoS wastes bandwidth
  int key = AoS_data[i].key;
  // SoA efficient use of bandwidth
  int key_better = SoA_data.keys[i];
}
```

Memory Coalescing

- 通常structure of array要优于array of structure
  - 对于步长为1的访问模式更好
  - 对于不规则访问需要具体问题具体分析

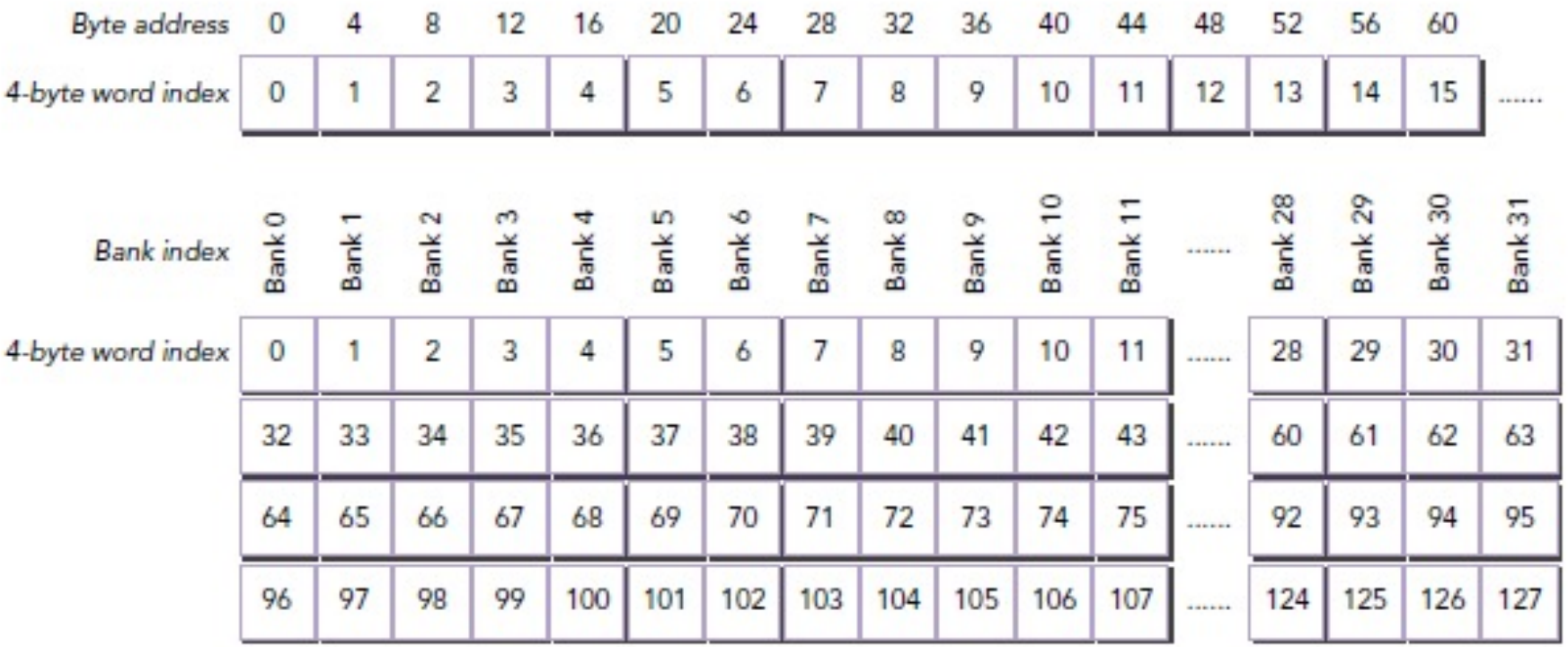
思考1

- 什么是Memory Coalescing？

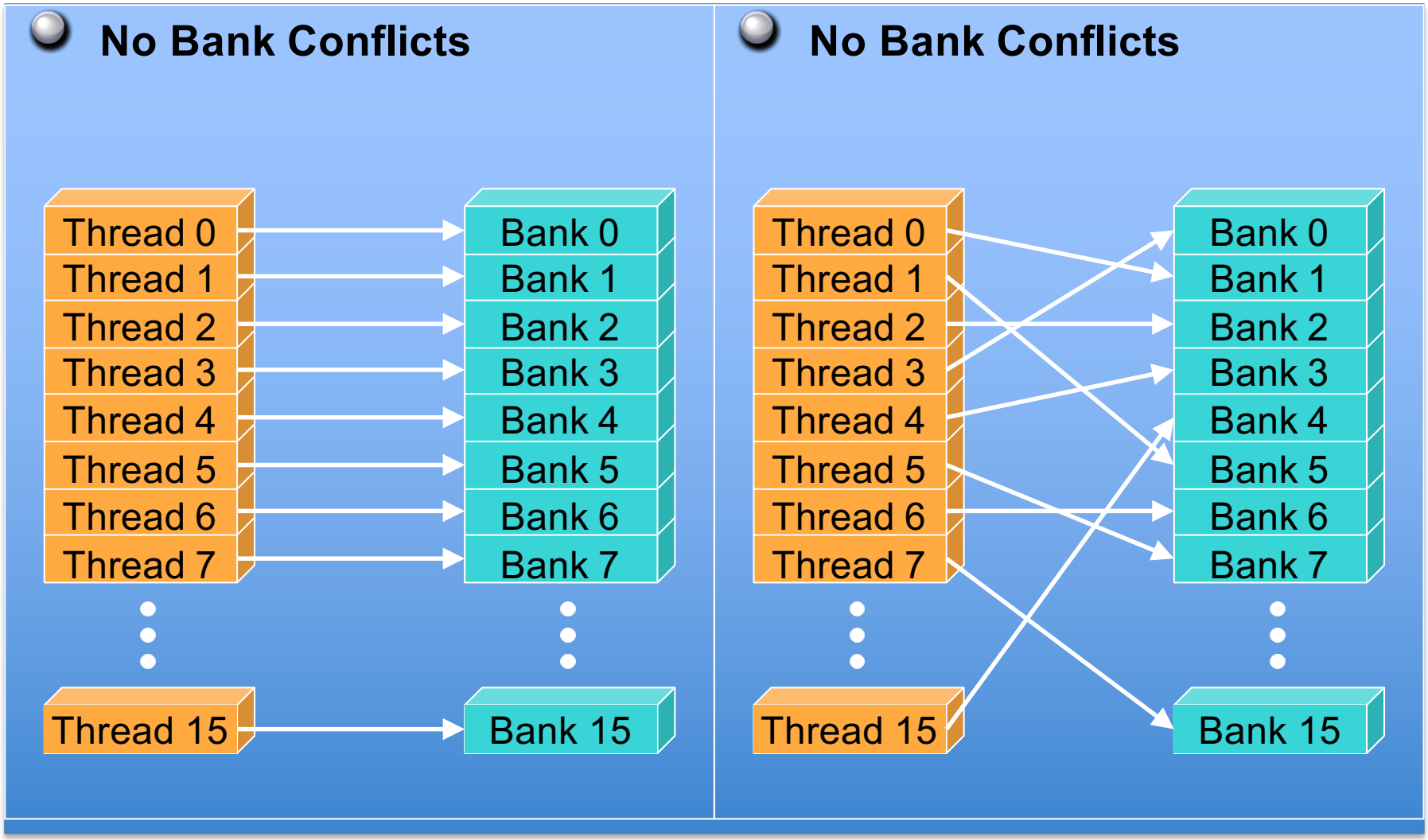
shared memory bank conflicts

- shared memory以bank方式进行存储
  - 只对warp内的线程有影响
  - 达到最大性能需要满足一些约束条件
  - 多个线程可以访问不同的bank
  - 或warp内线程访问同一个地址
- 连续的内容存储在不同的bank中
- 如果2个或多个线程访问同一个bank的不同内容，将产生bank conflict

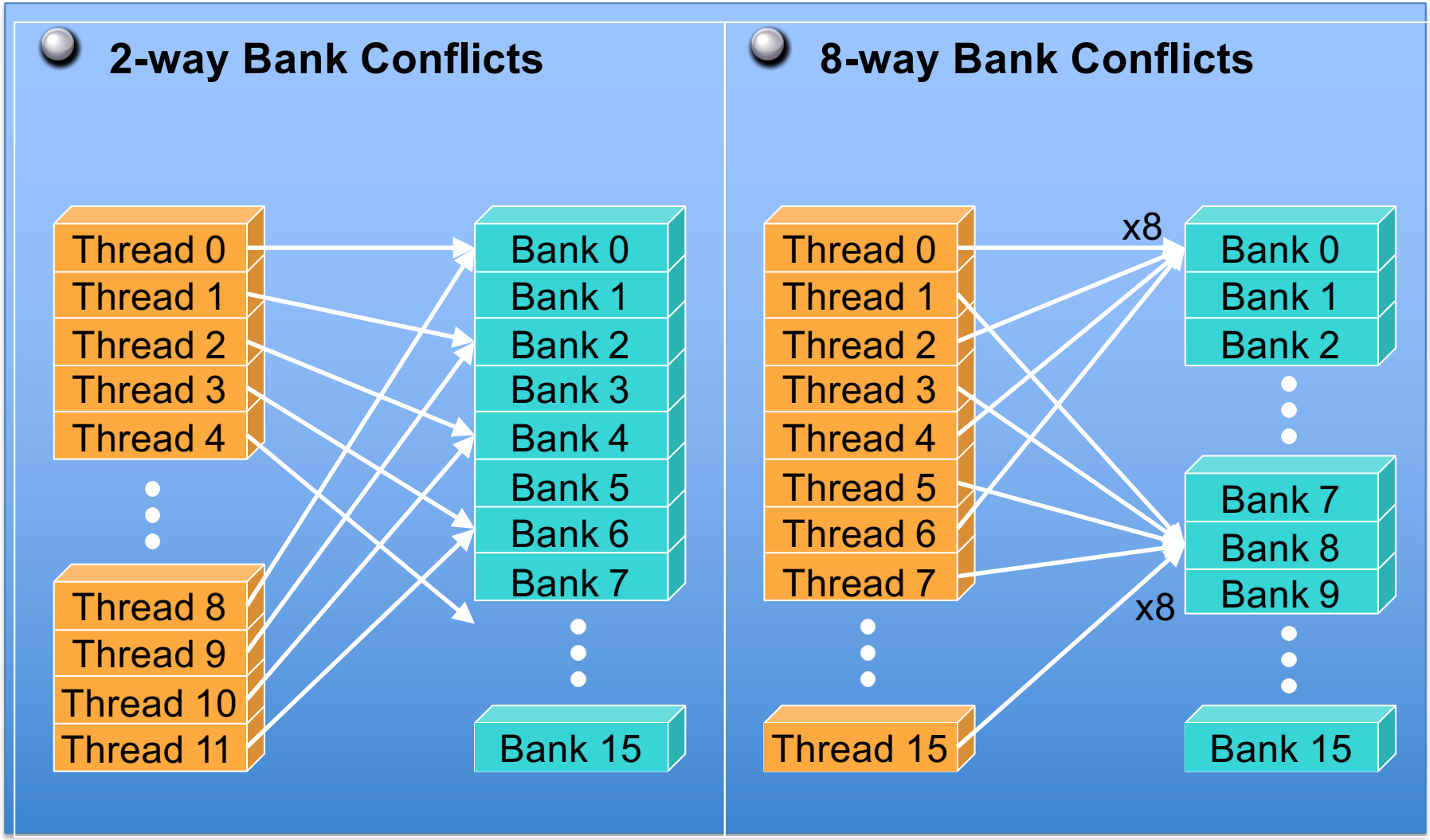
memory bank


Byte address
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
4-byte word index
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 .....
Bank index
Bank 0 Bank 1 Bank 2 Bank 3 Bank 4 Bank 5 Bank 6 Bank 7 Bank 8 Bank 9 Bank 10 Bank 11 ..... Bank 28 Bank 29 Bank 30 Bank 31
4-byte word index
0 1 2 3 4 5 6 7 8 9 10 11 ..... 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 ..... 60 61 62 63
64 65 66 67 68 69 70 71 72 73 74 75 ..... 92 93 94 95
96 97 98 99 100 101 102 103 104 105 106 107 ..... 124 125 126 127

bank例子


No Bank Conflicts
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 15
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 15
No Bank Conflicts
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 15
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 15

bank例子

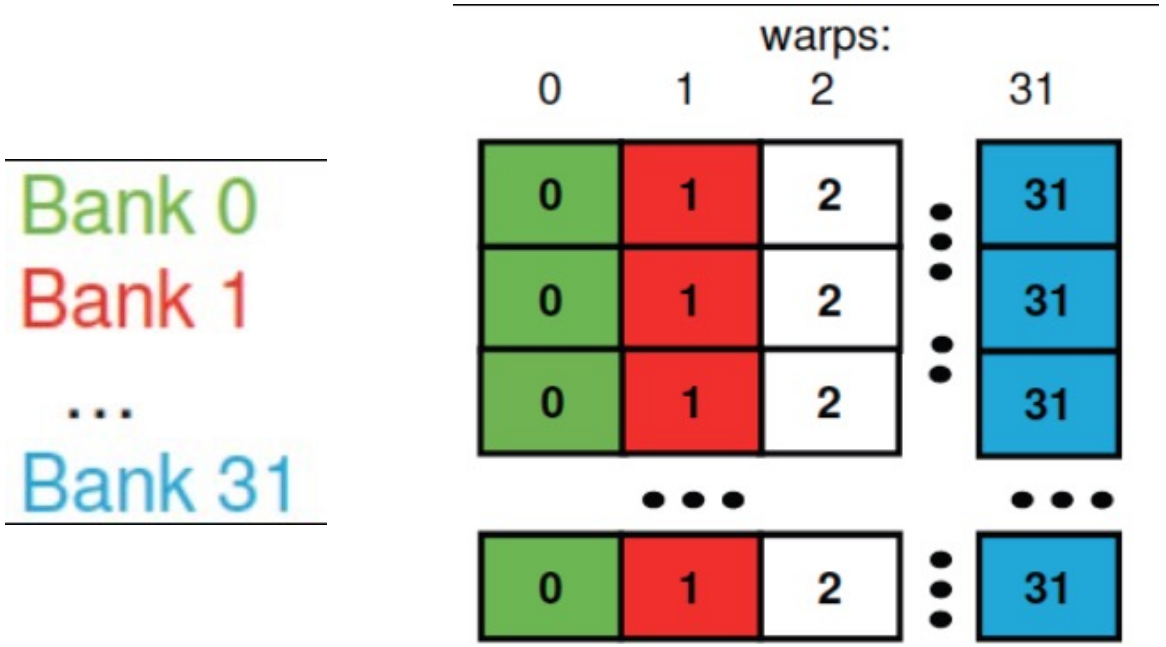

2-way Bank Conflicts
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 8
Thread 9
Thread 10
Thread 11
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 15
8-way Bank Conflicts
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 15
x8
x8
Bank 0
Bank 1
Bank 2
Bank 7
Bank 8
Bank 9
Bank 15

避免bank conflict

- 改变shared memory访问模式
  - 连续访问
  - 访问同一个地址（读）broadcast
- memory padding
  - 增加元素数目避免bank conflict

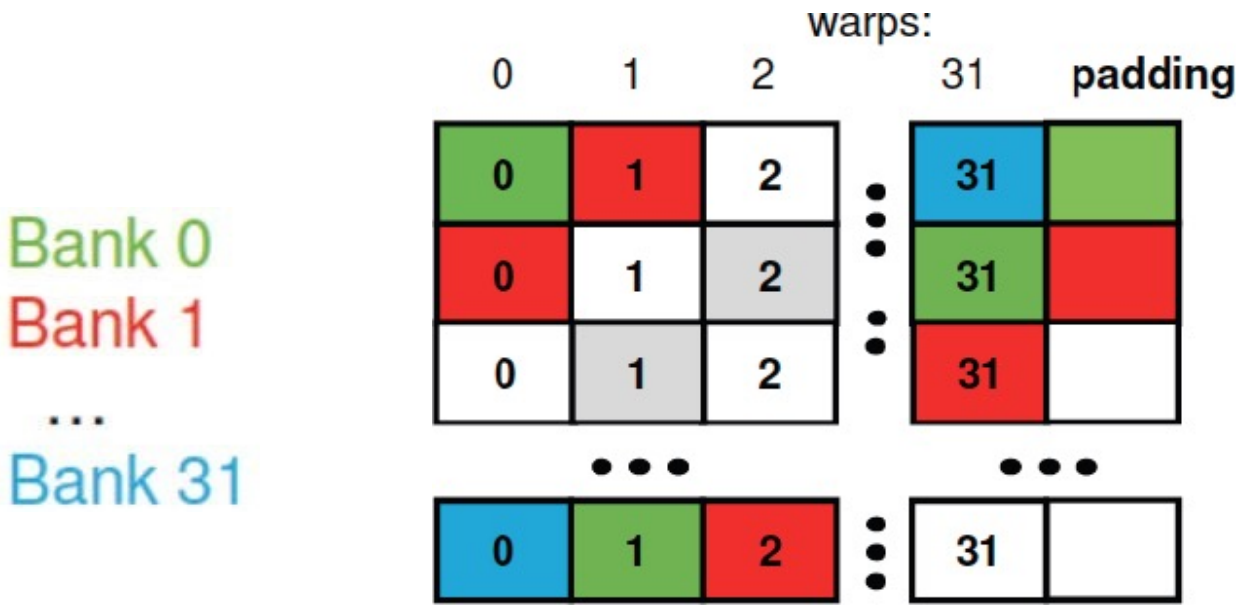
例子：2D矩阵

- 32*32 SMEM array
- warp访问一列，有bank conflict



memory padding

- 增加一列：
- 32*33 SMEM ARRAY
- warp访问一列
- 32个不同的bank，没有bank conflicts



shared memory

- 目的：
  - block内的线程间通信
  - 缓存数据降低全局内存访问
  - 使用shared memory避免non-coalesced access
- 组织结构：
  - 16 banks, 32-bit bank宽度（Tesla）
  - 32 banks, 32-bit bank宽度（Fermi）
  - 连续32-bit属于不同bank
- 性能：
  - 每个bank约2个clock/SM

思考2

- 什么是bank conflict？如何避免？

control flow divergence

- 指令发射给某个warp（32个线程）
- divergent branches：
  - 同一个warp中的线程执行不同的路径：
    - if-else
  - 同一个warp内的不同执行路径需要串行执行
- 不同的warp可以执行不同部分的代码，没有性能损失

例子：divergent iteration


```
__global__ void per_thread_sum(int *indices,
                               float *data,
                               float *sums)
{
  ...
  // number of loop iterations is data
  // dependent
  for(int j=indices[i];j<indices[i+1]; j++)
  {
    sum += data[j];
  }
  sums[i] = sum;
}
```

iteration divergence

- 单个线程可能拖慢整个warp的运行时间
- 了解数据访问模式
- 如果数据不可预测，尝试让每个线程处理多个数据项

```
__global__ void per_thread_sum(...)
{
  while(!done)
  {
    for(int j=indices[i];
   j<min(indices[i+1],indices[i]+MAX_ITER);
         j++)
    {...}
  }
}
```

思考3

- 什么是control flow divergence？如何避免？

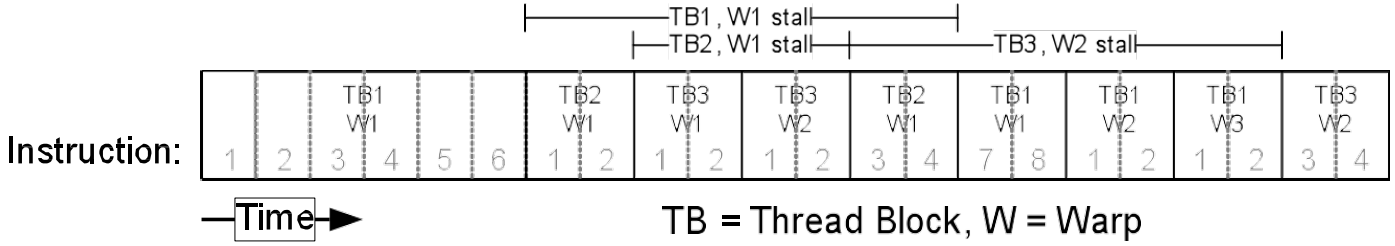
提高访问性能

- 所有的shared memory read都指向同一个地址
  - GPU内部采用broadcast操作，没有冲突

- 对于shared memory write
  - 访问shared memory时尽量根据threadIdx.x访问
    - 这一方法也会对read操作有帮助

占用时间（Occupancy）

- 线程调度
  - SM进行warp切换几乎没有开销
  - 任何时间，每个warp scheduler只能处理一个warp
  - 能够继续处理指令的warp会等待warp scheduler处理
  - GPU内有调度策略决定接下来执行哪个warp
  - warp内所有的线程都要执行同样的指令



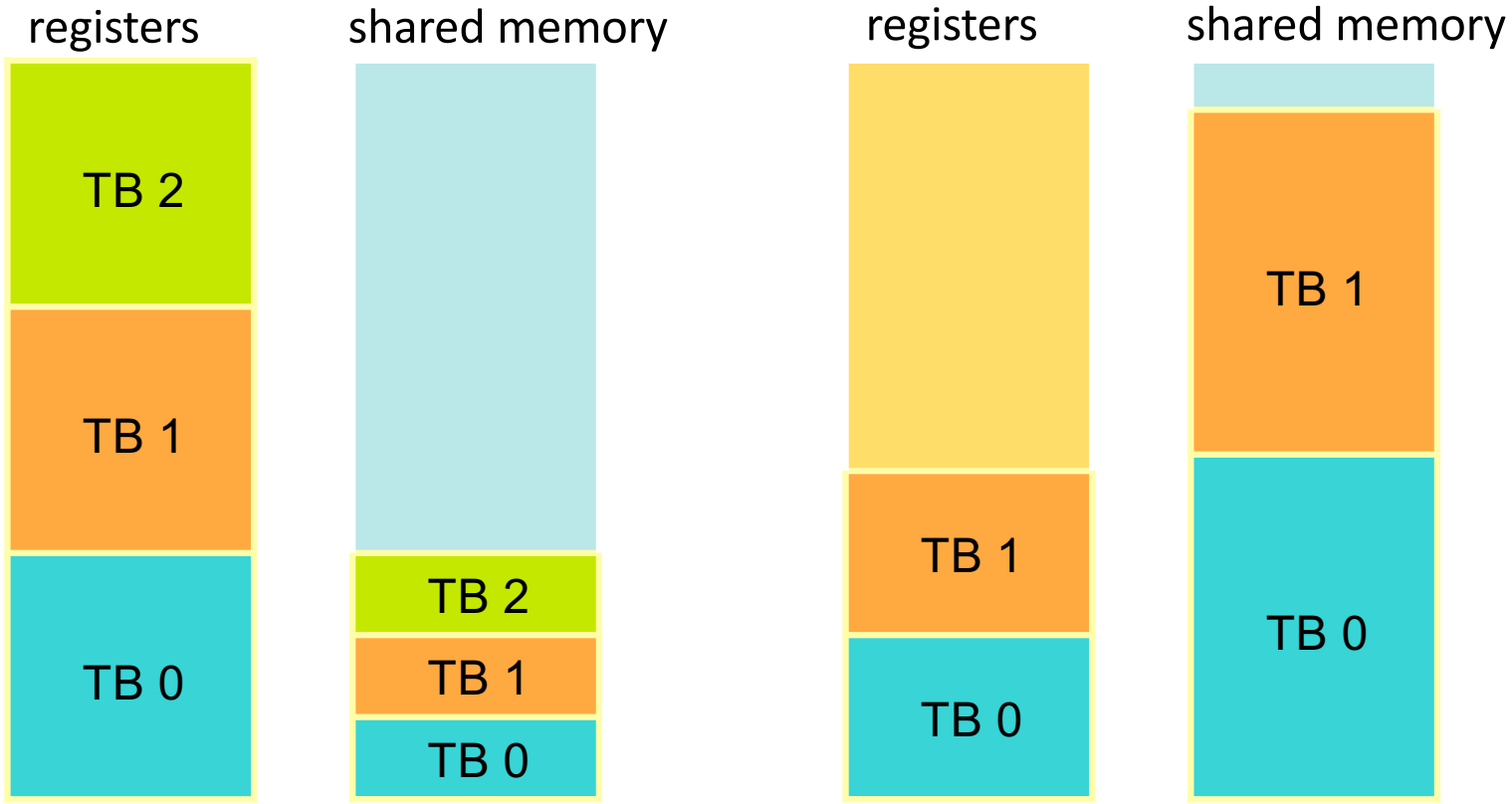
线程调度

- 如果所有的warp都被延迟了，怎么办？
  - 不会有指令执行 —> 性能损失
- 造成延迟的最常见原因？
  - 等待global memory取数据
- 如果程序每隔几个指令就需要访问global memory
  - 尝试最大化occupancy

occupancy由哪些因素决定？

- 每个线程的register usage
- 每个线程使用的shared memory

资源限制（1）



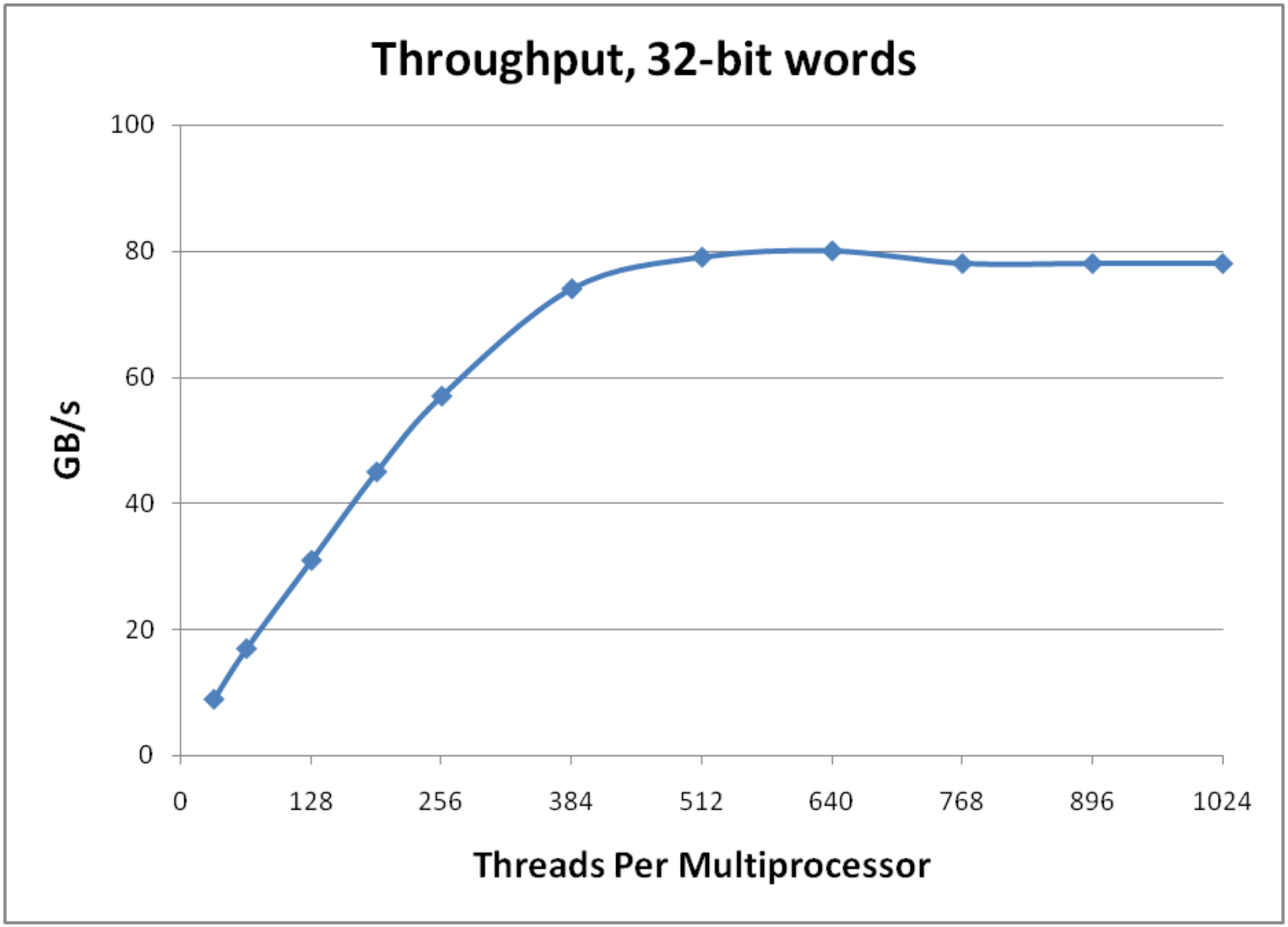
- 每个SM的寄存器和shared memory资源有限
  - 每个thread block使用一定量的register和shared memory
  - 如果其中一项被使用满了 —> 不会有更多的thread block

资源限制（2）

- 不同系列GPU的SM最大block数目不一样
- 如果thread不够多，不能够占满SM

- 高占用率（occupancy）有助于隐藏访问延迟

大量线程隐藏访问延迟


Throughput, 32-bit words
GB/s
100
80
60
40
20
0
0
128
256
384
512
640
768
896
1024
Threads Per Multiprocessor

如何知道使用了多少资源？

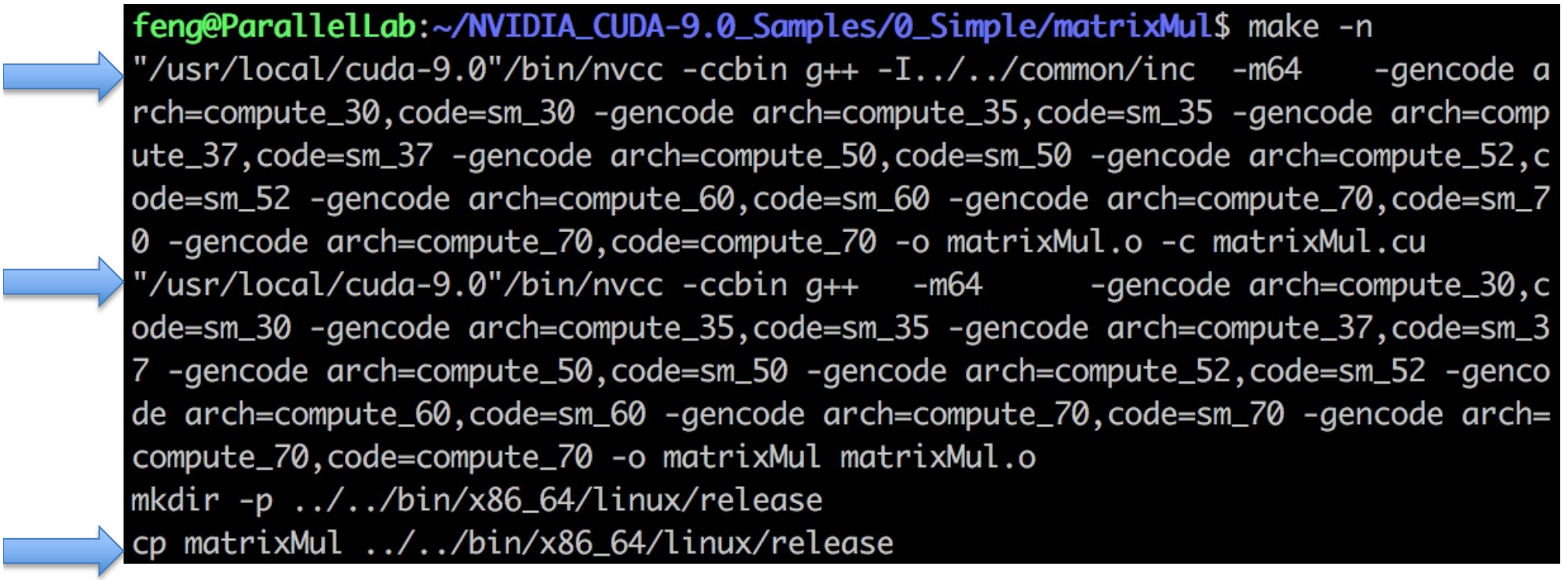
- 使用 -Xptxas –v 获得相应信息
- 利用这些信息可以计算occupancy

```
feng@ParallelLab:~/hw4$ nvcc -Xptxas -v -O3 main.cu -o main
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z4spmvPiS_PfS0_S0_ii' for 'sm_30'
ptxas info    : Function properties for _Z4spmvPiS_PfS0_S0_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 32 registers, 368 bytes cmem[0]
```

into CUDA Occupancy Calculator

如何分析CUDA sample中的程序

make -n



```
feng@ParallelLab:~/NVIDIA_CUDA-9.0_Samples/0_Simple/matrixMul$ make -n
"/usr/local/cuda-9.0"/bin/nvcc -ccbin g++ -I../../common/inc  -m64    -gencode a
rch=compute_30,code=sm_30 -gencode arch=compute_35,code=sm_35 -gencode arch=comp
ute_37,code=sm_37 -gencode arch=compute_50,code=sm_50 -gencode arch=compute_52,c
ode=sm_52 -gencode arch=compute_60,code=sm_60 -gencode arch=compute_70,code=sm_7
0 -gencode arch=compute_70,code=compute_70 -o matrixMul.o -c matrixMul.cu
"/usr/local/cuda-9.0"/bin/nvcc -ccbin g++   -m64      -gencode arch=compute_30,c
ode=sm_30 -gencode arch=compute_35,code=sm_35 -gencode arch=compute_37,code=sm_3
7 -gencode arch=compute_50,code=sm_50 -gencode arch=compute_52,code=sm_52 -genco
de arch=compute_60,code=sm_60 -gencode arch=compute_70,code=sm_70 -gencode arch=
compute_70,code=compute_70 -o matrixMul matrixMul.o
mkdir -p ../../bin/x86_64/linux/release
cp matrixMul ../../bin/x86_64/linux/release
```


```
feng@ParallelLab:~/NVIDIA_CUDA-9.0_Samples/0_Simple/matrixMul$ "/usr/local/cuda-
9.0"/bin/nvcc -Xptxas -v -ccbin g++ -I../../common/inc  -m64    -gencode arch=co
mpute_30,code=sm_30 -gencode arch=compute_35,code=sm_35 -gencode arch=compute_37
,code=sm_37 -gencode arch=compute_50,code=sm_50 -gencode arch=compute_52,code=sm
_52 -gencode arch=compute_60,code=sm_60 -gencode arch=compute_70,code=sm_70 -gen
code arch=compute_70,code=compute_70 -o matrixMul.o -c matrixMul.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z13matrixMulCUDAILi32EEvPfS0_S0_ii' f
or 'sm_30'
ptxas info    : Function properties for _Z13matrixMulCUDAILi32EEvPfS0_S0_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 29 registers, 8192 bytes smem, 352 bytes cmem[0]
```

CUDA_Occupancy_calculator.xlsm - Microsoft Excel

Security Warning Macros have been disabled. Options...

MyRegCount 25

1.) Select Compute Capability (click):	1.3 (Help)
2.) Enter your resource usage:	
Threads Per Block	128 (Help)
Registers Per Thread	25
Shared Memory Per Block (bytes)	640
(Don't edit anything below this line)	
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	512 (Help)
Active Warps per Multiprocessor	16
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	50%

Physical Limits for GPU Compute Capability:	1.3
Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384
Register allocation unit size	512
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	4
Registers	3584
Shared Memory	1024

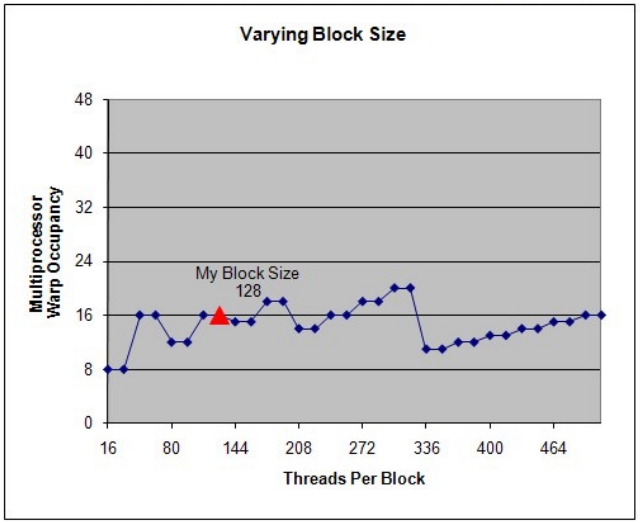
These data are used in computing the occupancy data in blue

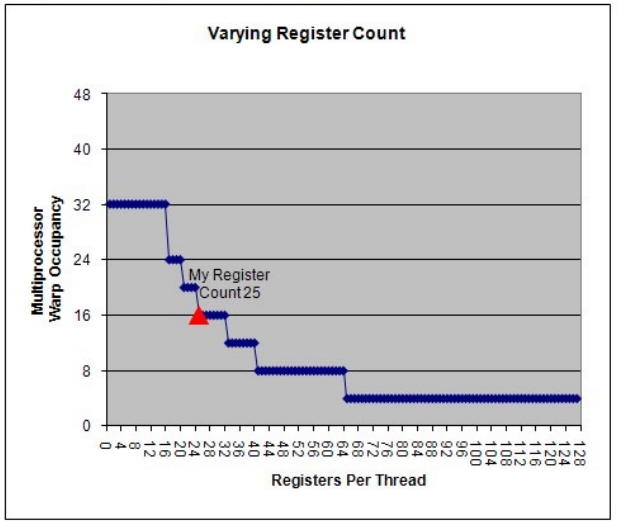
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	8
Limited by Registers per Multiprocessor	4
Limited by Shared Memory per Multiprocessor	16

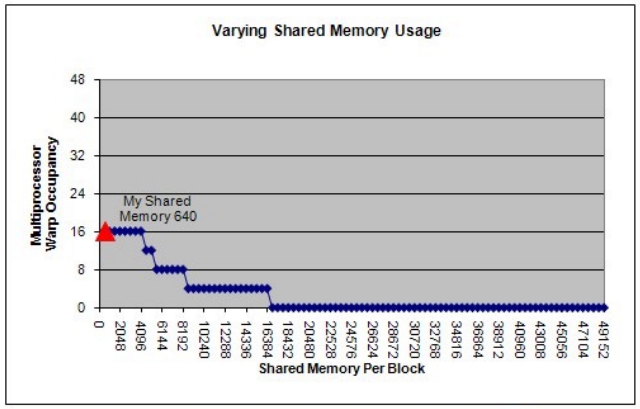
Thread Block Limit Per Multiprocessor highlighted RED

CUDA Occupancy Calculator	
Version:	2.0
Copyright and License	

The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

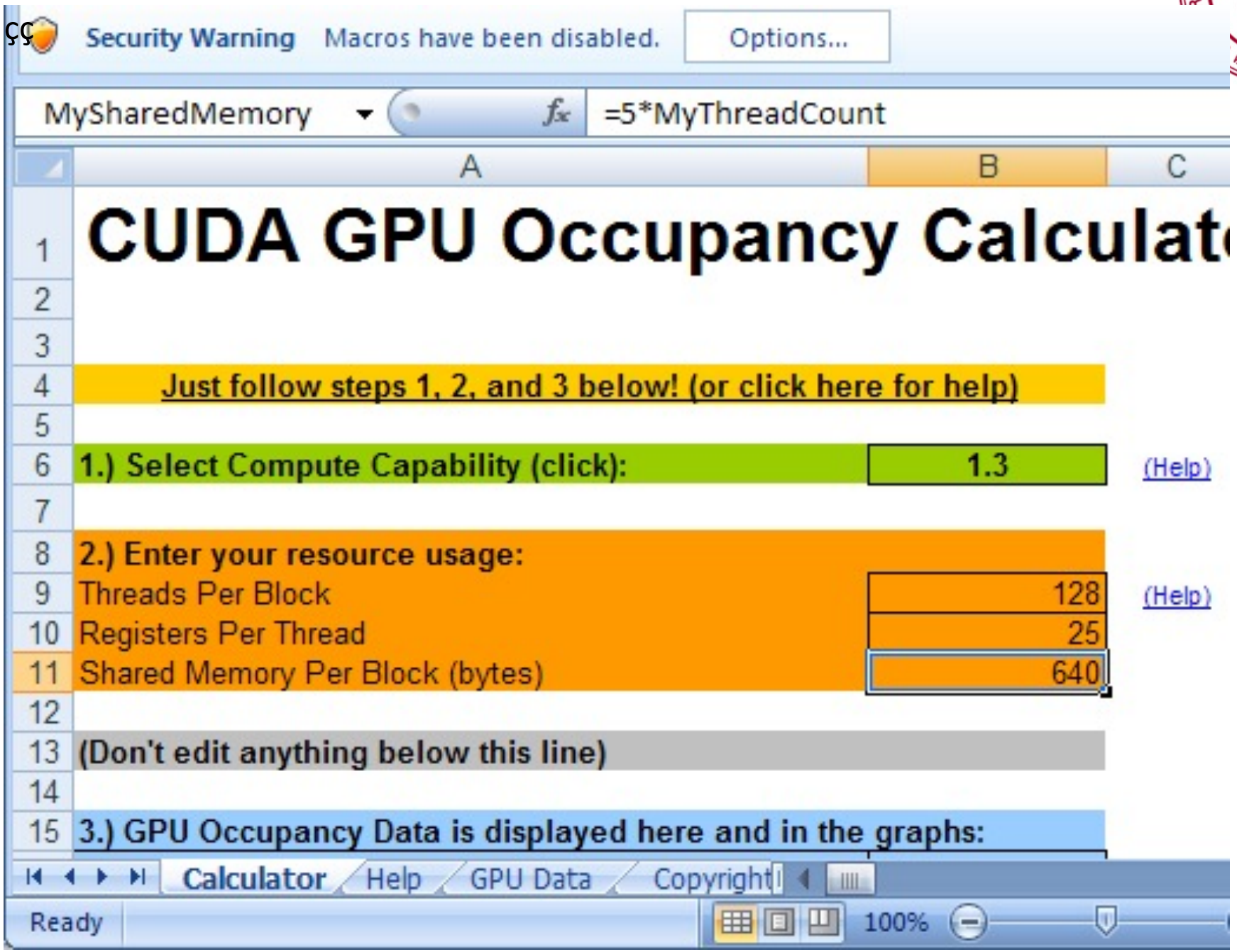


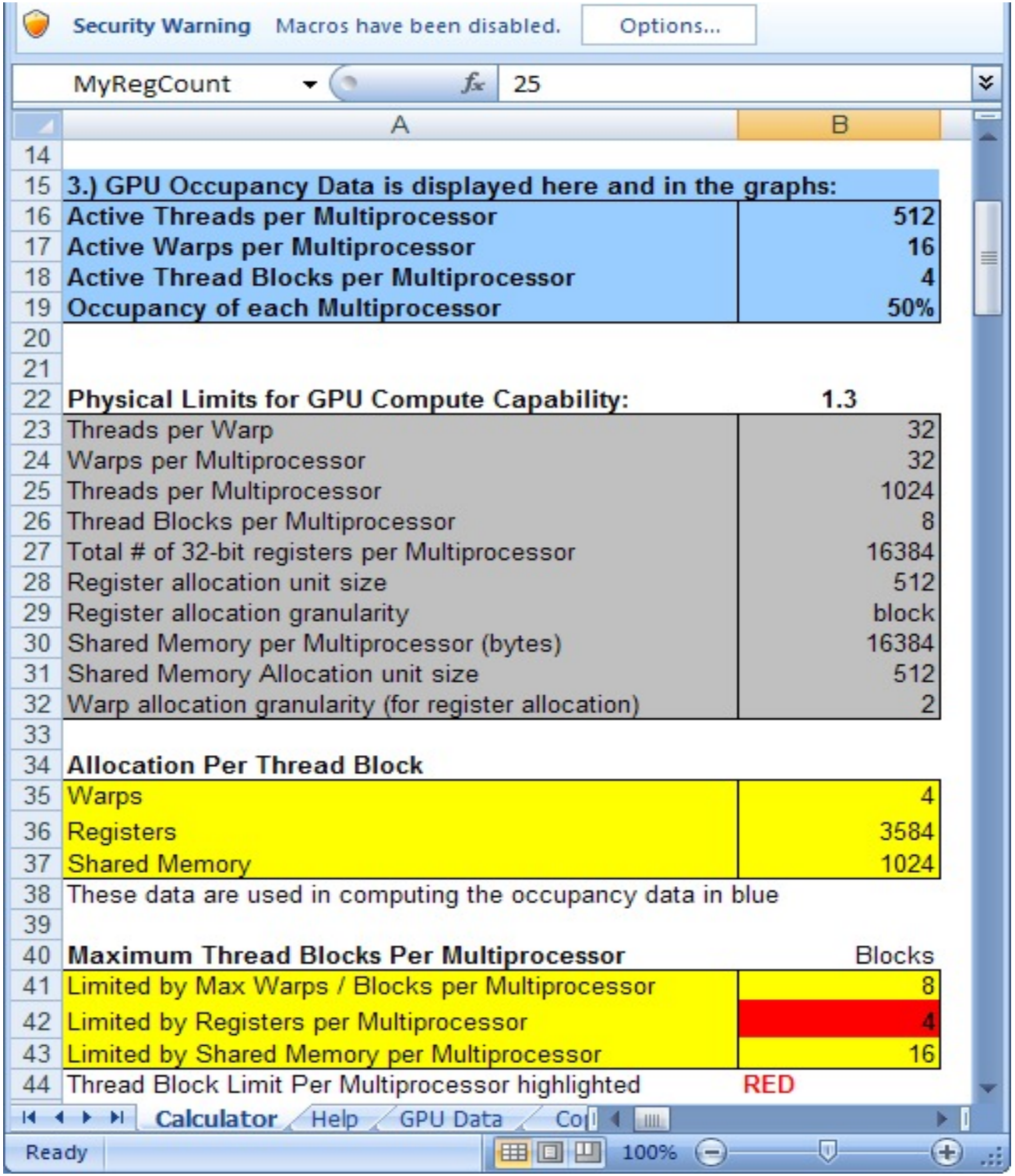




Calculator | Help | GPU Data | Copyright & License

Ready


Security Warning
Macros have been disabled.
Options...
MySharedMemory
=5*MyThreadCount
A
B
C
CUDA GPU Occupancy Calculat
Just follow steps 1, 2, and 3 below! (or click here for help)
1.) Select Compute Capability (click):
1.3
(Help)
2.) Enter your resource usage:
Threads Per Block
128
(Help)
Registers Per Thread
25
Shared Memory Per Block (bytes)
640
(Don't edit anything below this line)
3.) GPU Occupancy Data is displayed here and in the graphs:
Calculator
Help
GPU Data
Copyright
Ready
100%



Security Warning Macros have been disabled. Options...

MyRegCount | fx | 25

	A	B
14		
15	**3.) GPU Occupancy Data is displayed here and in the graphs:**	
16	**Active Threads per Multiprocessor**	**512**
17	**Active Warps per Multiprocessor**	**16**
18	**Active Thread Blocks per Multiprocessor**	**4**
19	**Occupancy of each Multiprocessor**	**50%**
20		
21		
22	**Physical Limits for GPU Compute Capability:**	**1.3**
23	Threads per Warp	32
24	Warps per Multiprocessor	32
25	Threads per Multiprocessor	1024
26	Thread Blocks per Multiprocessor	8
27	Total # of 32-bit registers per Multiprocessor	16384
28	Register allocation unit size	512
29	Register allocation granularity	block
30	Shared Memory per Multiprocessor (bytes)	16384
31	Shared Memory Allocation unit size	512
32	Warp allocation granularity (for register allocation)	2
33		
34	**Allocation Per Thread Block**	
35	Warps	4
36	Registers	3584
37	Shared Memory	1024
38	These data are used in computing the occupancy data in blue	
39		
40	**Maximum Thread Blocks Per Multiprocessor**	Blocks
41	Limited by Max Warps / Blocks per Multiprocessor	8
42	Limited by Registers per Multiprocessor	4
43	Limited by Shared Memory per Multiprocessor	16
44	Thread Block Limit Per Multiprocessor highlighted	RED

Calculator | Help | GPU Data | Co

Ready 100%

如何设置register的使用情况？

- nvcc使用 `-maxrregcount=X` 进行设置
- 谨慎使用！

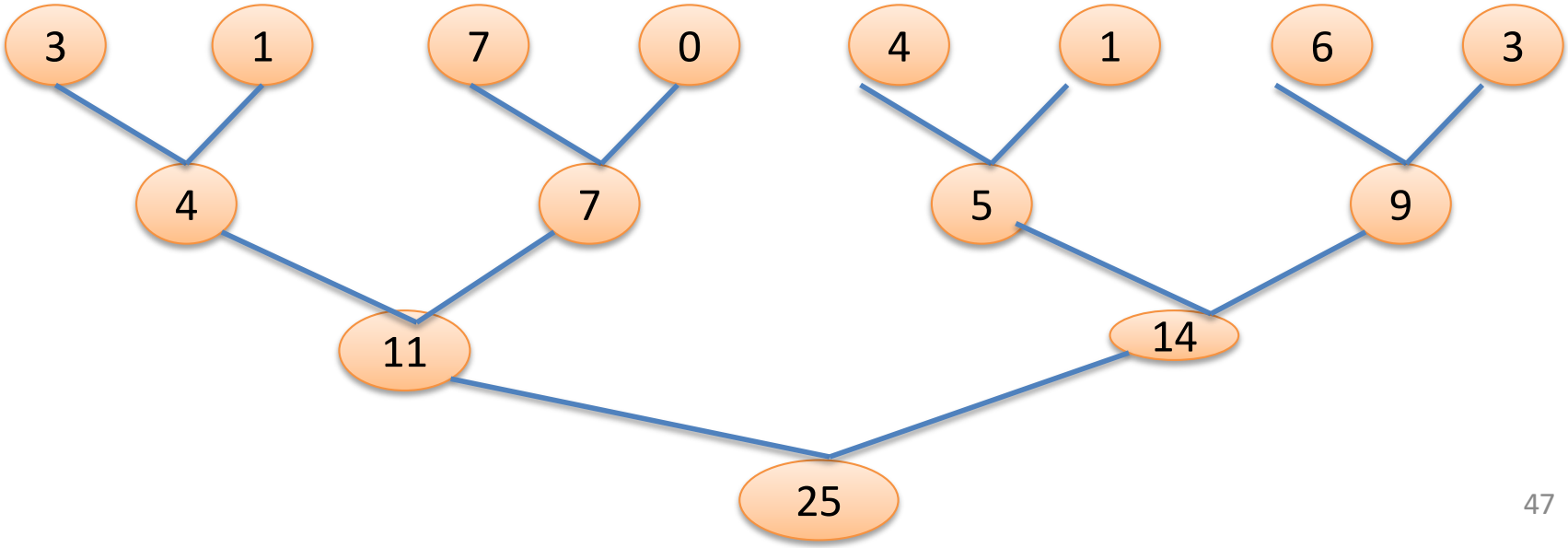
性能考虑因素

- 测试各部分的时间
- 一旦找到了性能瓶颈，进行合适的调优
  - 取决于程序代码
  - 通常会有一系列的性能瓶颈，每个优化点可能只有一小部分的性能提升

优化程序的例子

数组求和（reduction）

- 将向量数组中的各值求和成为一个标量
  - 通过相关操作符，如+/*/min/max/AND/OR…
  - CPU: 串行实现
    - for(int i = 0, i < n, ++i) …
  - GPU: 树状方式并行实现



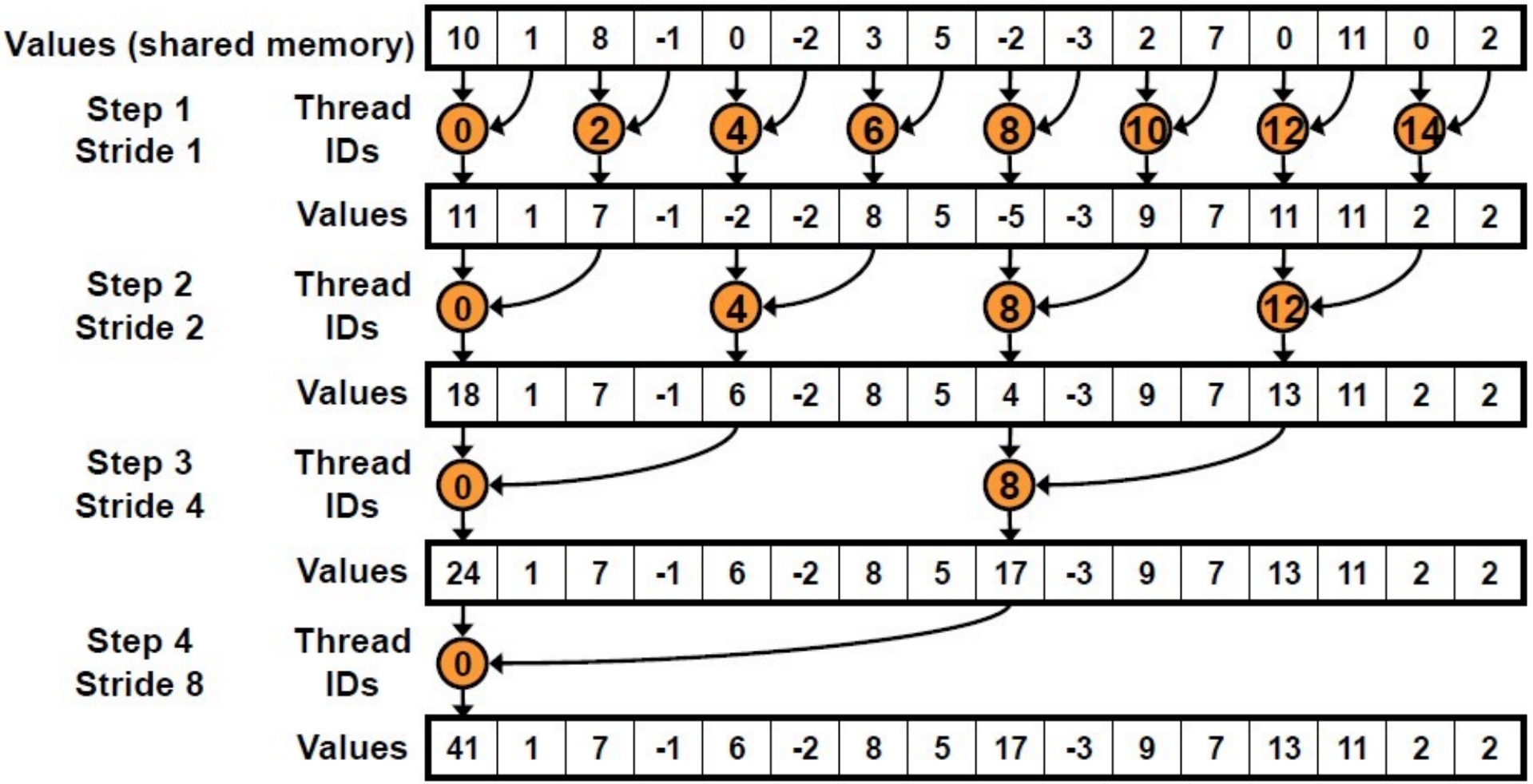
串行reduction

```
// reduction via serial iteration
float sum(float *data, int n)
{
  float result = 0;
  for(int i = 0; i < n; ++i)
  {
    result += data[i];
  }

  return result;
}
```

我们的优化目标

- 尽量接近GPU的峰值性能
- 选择正确的指标：
  - GFLOPs/s: Compute Bound Kernel
  - bandwidth: Memory Bound Kernel
- reduction有较低的计算密度
  - 1 flop/element (bandwidth-optimal)
- 优化内存带宽

版本1: Parallel Reduction(交错访问)



版本1: Parallel Reduction(交错访问)


```
__global__void reduce0(int *g_idata, int *g_odata) {
    extern__shared__int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

版本1: Parallel Reduction(交错访问)

```
__global__void reduce1(int *g_idata, int *g_odata) {
   extern__shared__int sdata[];

   // each thread loads one element from global to shared mem
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
   sdata[tid] = g_idata[i];
   __syncthreads();

   // do reduction in shared mem
   for (unsigned int s=1; s < blockDim.x; s *= 2) {
      if (tid % (2*s) == 0) {
         sdata[tid] += sdata[tid + s];
      }
      __syncthreads();
   }

   // write result for this block to global mem
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem: highly divergent branching results in very poor performance!

G80 GPU 384-bit memory interface, 900 MHz DDR 384 * 1800 / 8 = 86.4 GB/s

Kernel 1: interleaved addressing with divergent branching	**Time (2^{22} ints)**	**Bandwidth**
	8.054 ms	**2.083 GB/s**

Note: Block Size = 128 threads for all tests

版本2: Parallel Reduction(交错访问)

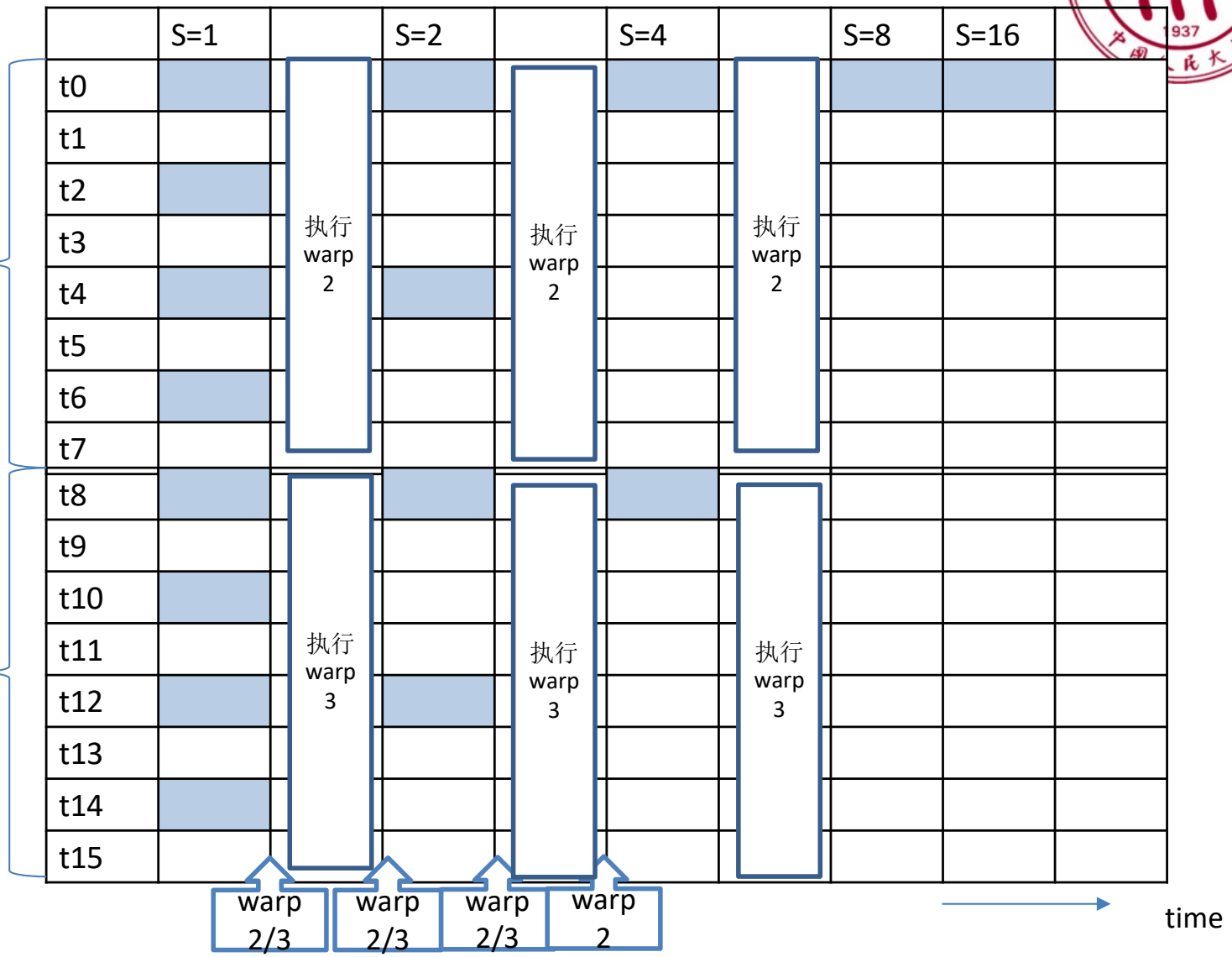
Just replace divergent branch in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

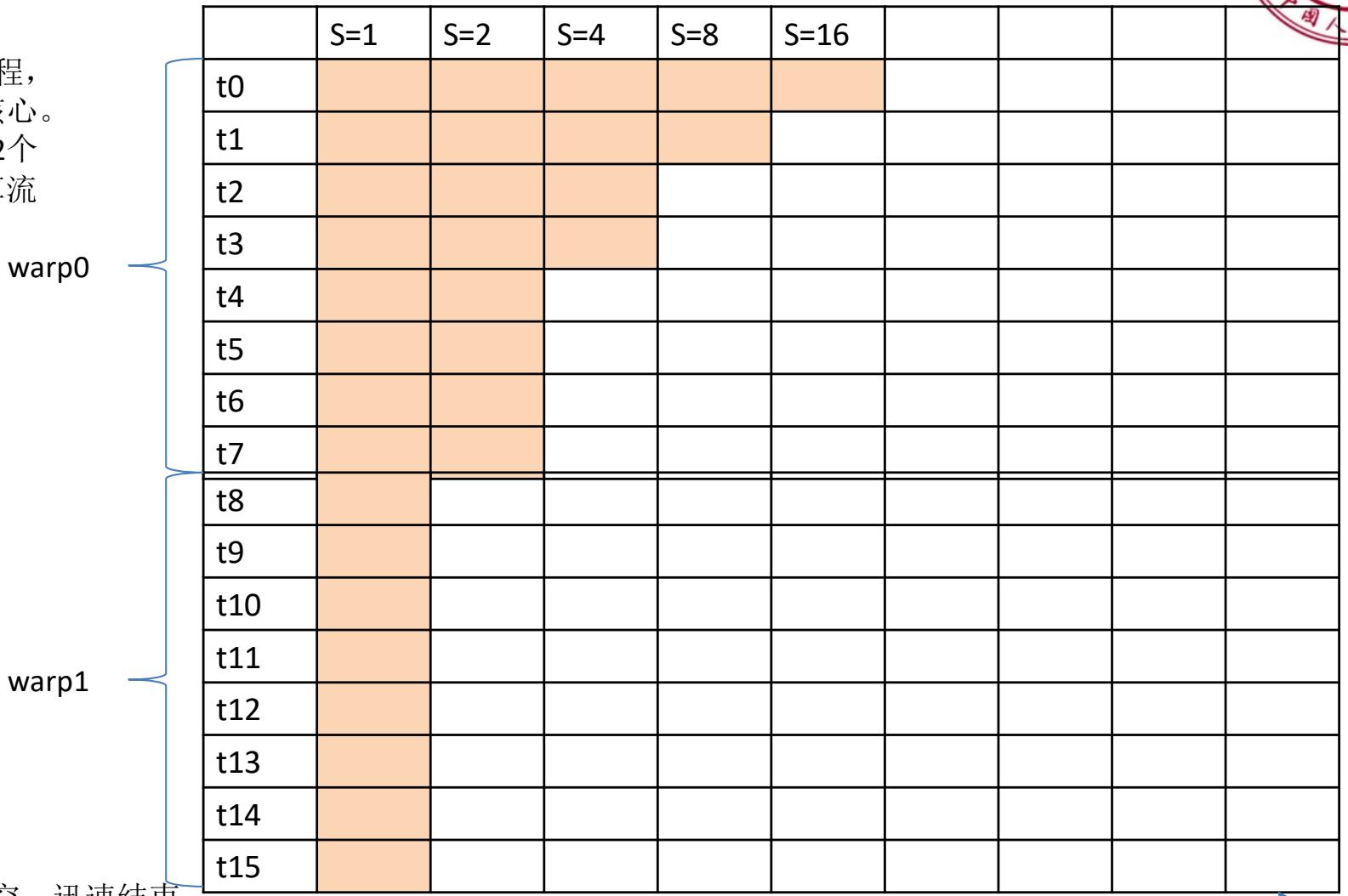
With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

假设GPU每个warp有8个线程，共16个计算核心。用户开启了32个线程，则计算流程如右图。



假设GPU每个warp有8个线程，共16个计算核心。用户开启了32个线程，则计算流程如右图。

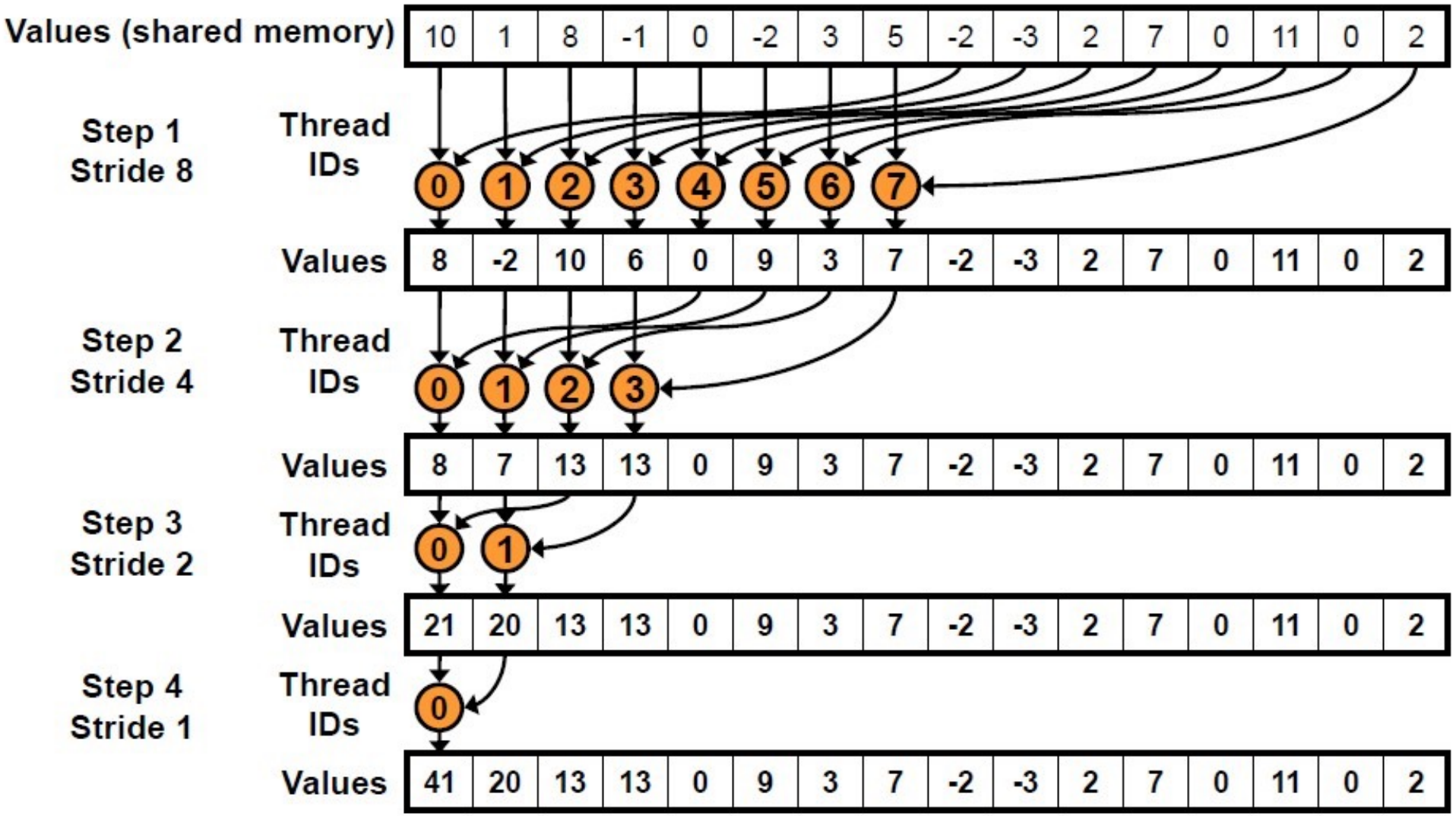


warp2/3均为空，迅速结束

性能结果

	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

版本3: Parallel Reduction(连续访问)



Sequential addressing is conflict free

版本3: Parallel Reduction(连续访问)

Just replace strided indexing in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

With reversed loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

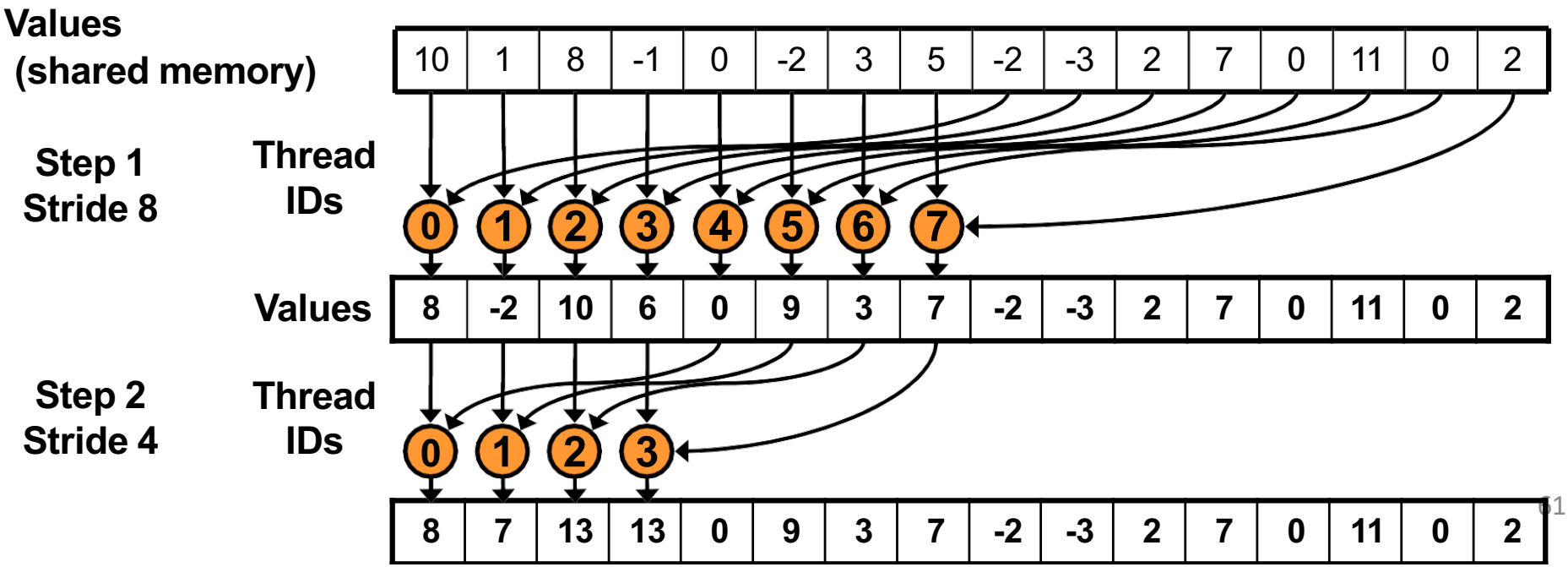
性能结果

	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

版本4: Parallel Reduction (减少空闲线程数目)

- 第一轮迭代时一半的线程空闲

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```



版本4: Parallel Reduction (减少空闲线程数目)

Halve the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

性能结果

	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

Instruction Bottleneck

17 GB/s 和 86 GB/s 距离很远

并且我们知道**reduction**计算密度很低

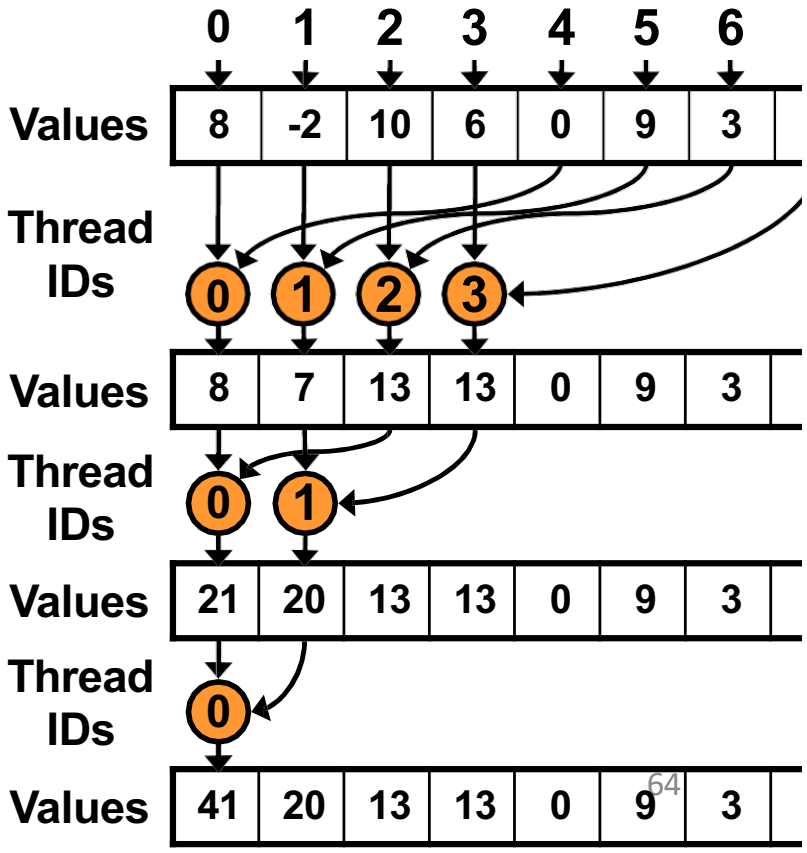
有可能有指令方面的开销

额外的辅助指令： **Ancillary instructions that are not loads, stores, or arithmetic for the core computation**

In other words: address arithmetic and loop overhead

Strategy: unroll loops

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```



unrolling the Last Warp

- 随着reduction的进行，真正用于做计算的线程数目减少
  - 当需要做加法的元素数目小于32时，只有一个warp
- warp内指令以SIMD的方式执行
- 这意味着当元素数目小于等于32时：
  - 不需要使用__syncthreads()
  - 不要写if

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

版本5: Parallel Reduction (unroll the last warp)


```
for (unsigned int s=blockDim.x/2; s>32; s>>=1)
{
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}

if (tid < 32)
{
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}
```

Note: This saves useless work in *all* warps, not just the last one!

Without unrolling, all warps execute every iteration of the for loop and if statement

性能结果

	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

进一步展开

- 如果能在编译阶段确定迭代的数目，我们就可以展开循环
- CUDA支持C++ template

- **Specify block size as a function template parameter:**

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

版本6: Parallel Reduction (进一步展开)


```
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) {
    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];
}
```

Note: all code in RED will be evaluated at compile time.

Results in a very efficient inner loop!

调用CUDA template


```
switch (threads)
        {
        case 512:
            reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        case 256:
            reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        case 128:
            reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        case 64:
            reduce5< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        case 32:
            reduce5< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        case 16:
            reduce5< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        case  8:
            reduce5<    8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        case  4:
            reduce5<    4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        case  2:
            reduce5<    2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        case  1:
            reduce5<    1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
        }
```

性能结果

	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

版本7: Parallel Reduction (每个线程做更多的计算)

Replace load and add of two elements:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
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();
```

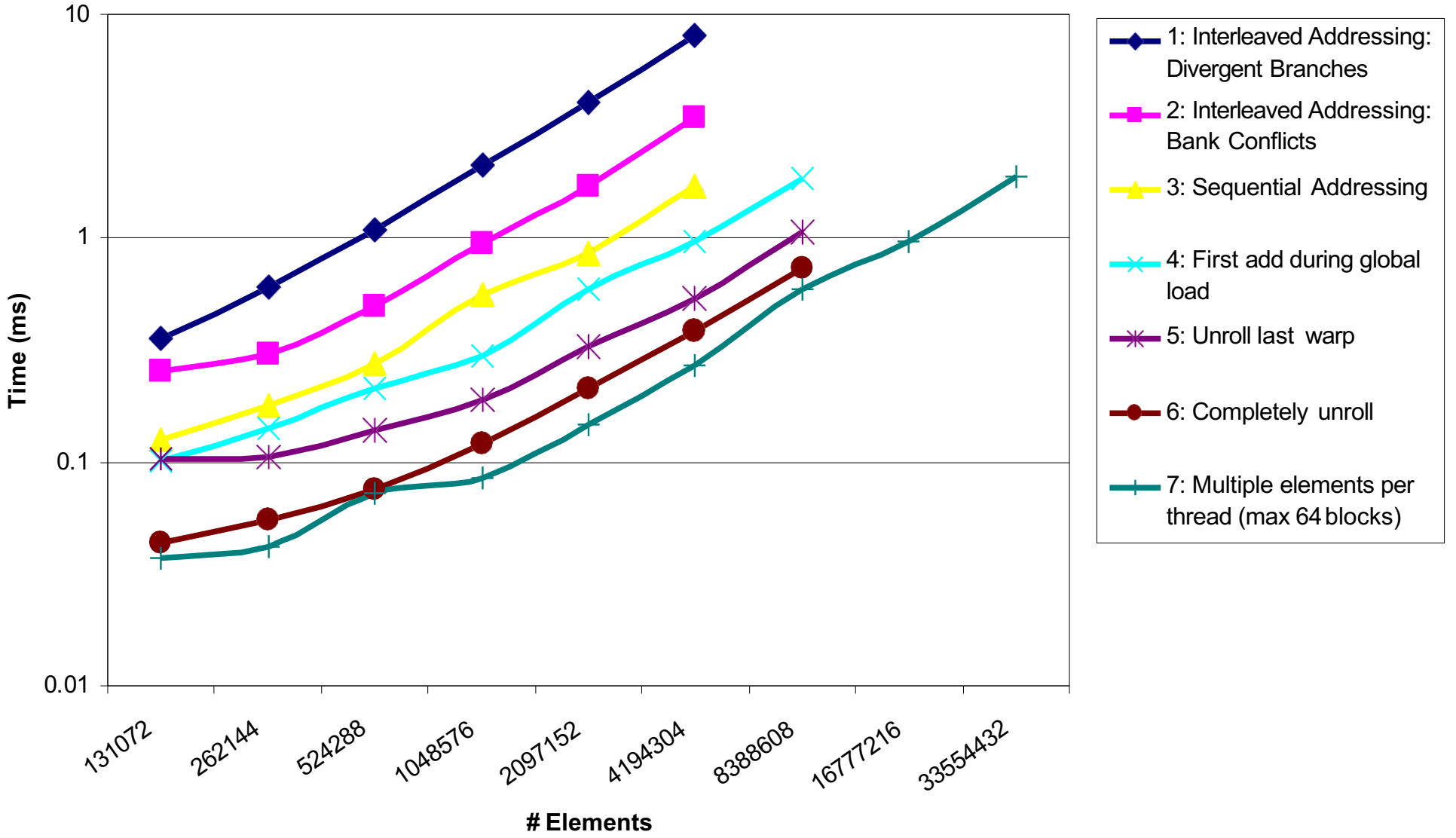
性能结果（理论带宽86GB/s）

	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

```
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) {
        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];
    }

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

Final Optimized Kernel

性能对比


10
1
0.1
0.01
Time (ms)
131072
262144
524288
1048576
2097152
4194304
8388608
16777216
33554432
Elements
1: Interleaved Addressing: Divergent Branches
2: Interleaved Addressing: Bank Conflicts
3: Sequential Addressing
4: First add during global load
5: Unroll last warp
6: Completely unroll
7: Multiple elements per thread (max 64 blocks)

优化类型

- 来自算法层面的优化
  - 11.84X
- 代码优化
  - 2.54X

思考4

- 回忆、思考数组求和例子中用到了哪些优化点？

结论

- 理解CUDA的性能特性
  - Memory coalescing
  - Divergent branching
  - Bank conflicts
- 使用peak performance的指标指导优化
- 理解并行计算复杂性相关的理论
- 知道如何确定程序的性能瓶颈
  - memory/core computation/instruction overhead
- 优化算法，unroll loops
- 使用template

谢谢！