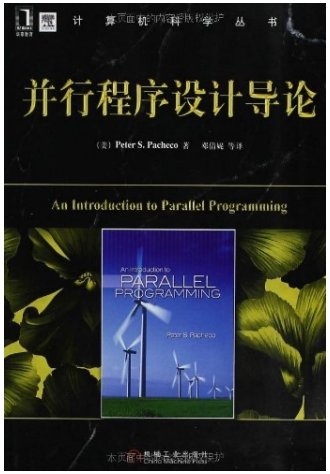


第六章

用CUDA进行GPU编程

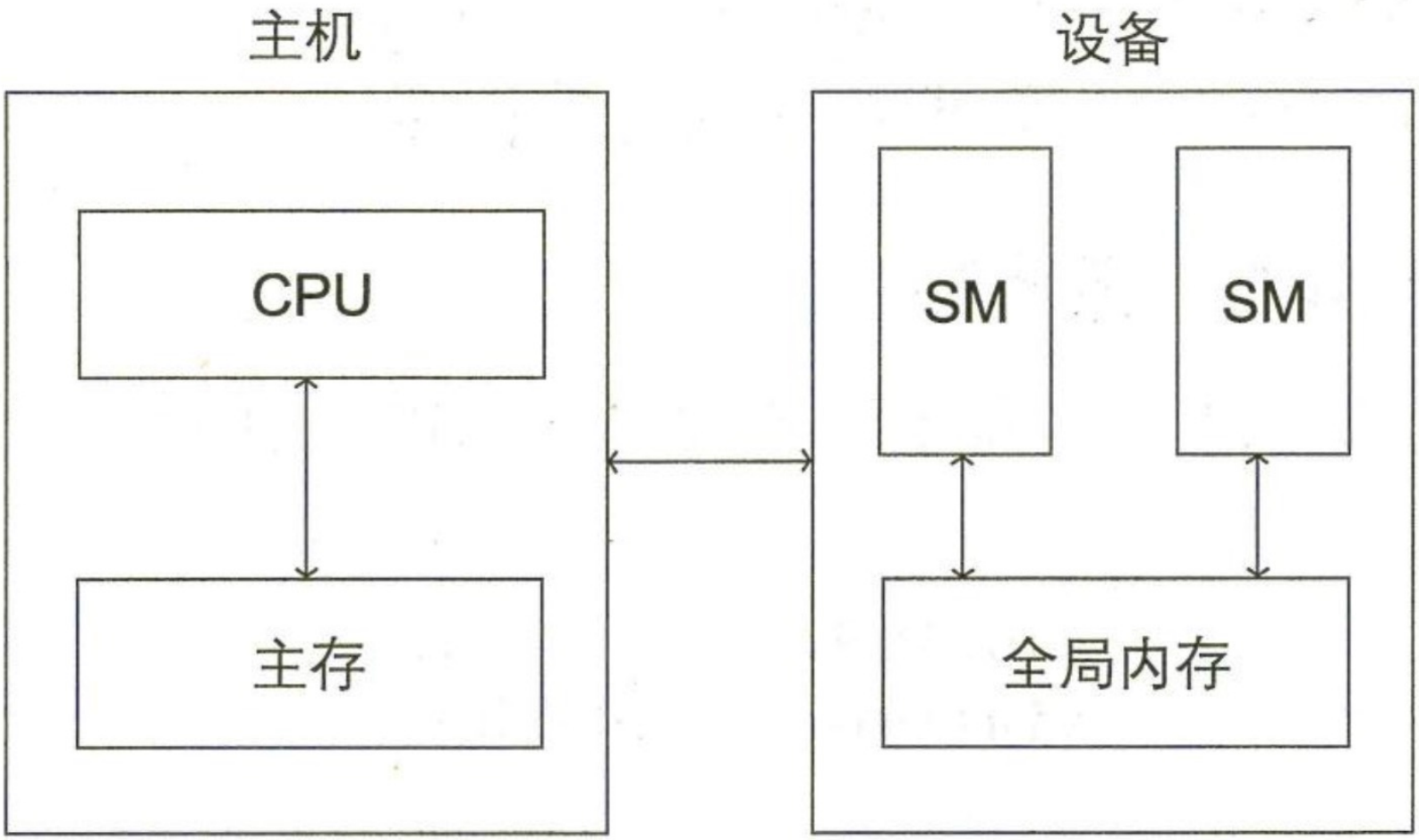
目录

- 异构计算.
- CUDA编程模型.
- 编写你的第一个CUDA程序.
- 用CUDA来实现梯形积分法.
- CUDA性能提升.
- 双调排序.



异构计算

异构计算



常见的异构计算有哪些?

GPU架构

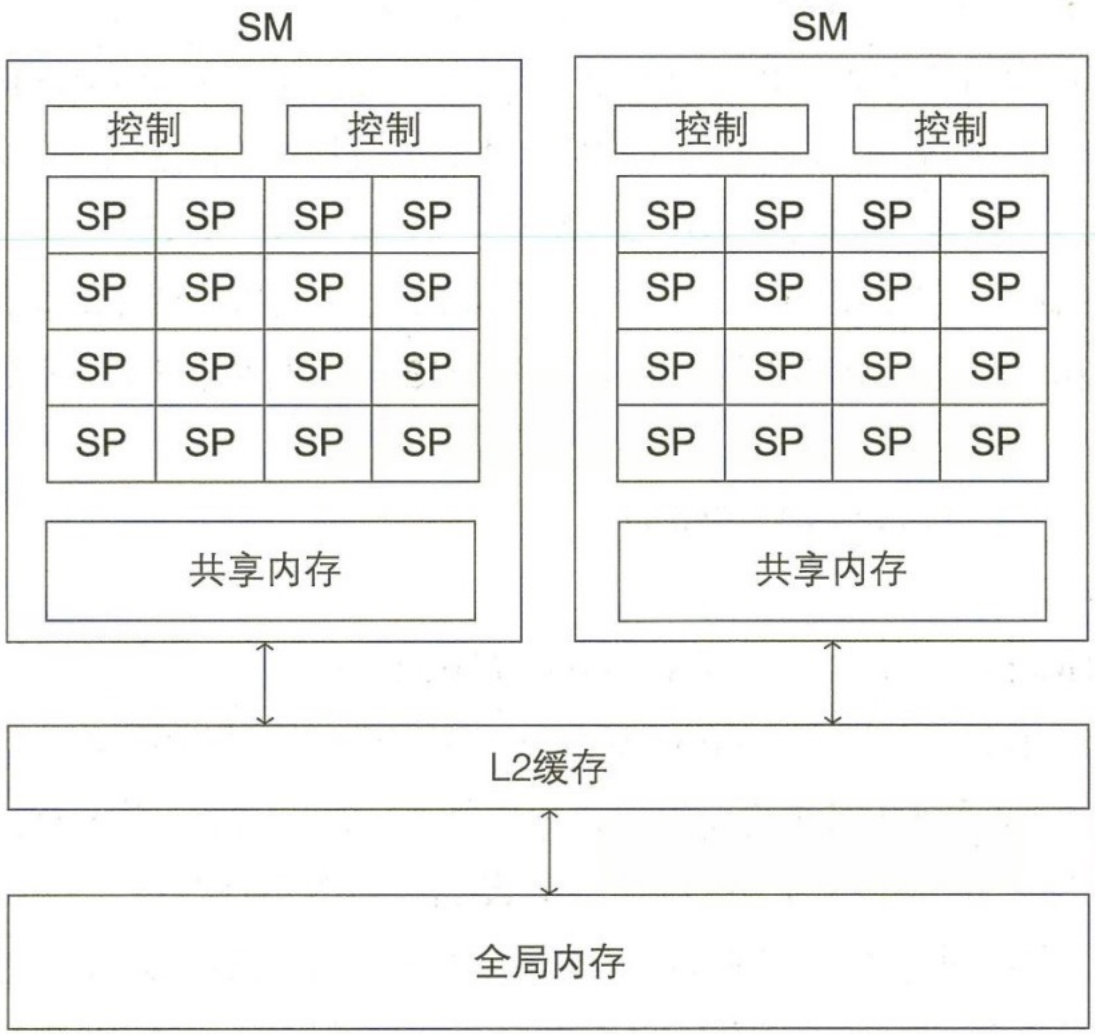
- 图形处理器 Graphics Processing Unit，缩写：GPU.
- 为解决早期计算机图形处理增加CPU负担的问题.



GPU架构

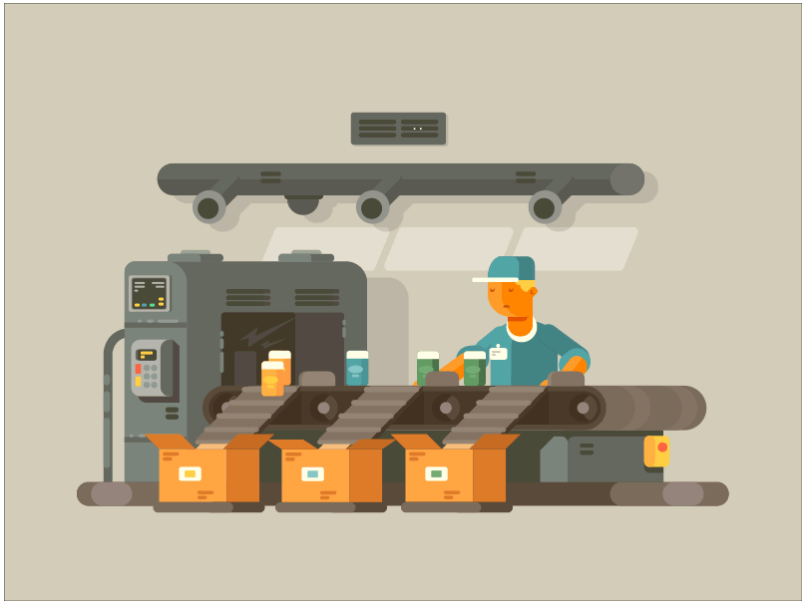
- 单指令多数据流（SIMD）设备.
- NVIDIA GPU由流式多处理器（SM）组成.
- 一个SM可以有多个控制单元和更多的数据通路（SP）.

GPU架构



GPU架构

- 多流水线工作.
- 小缓存提高内存吞吐量.
- 简单控制，没有分支预测，没有数据转发.



SIMD处理器

有分支的执行代码。

```
if (x[i] >= 0)
    x[i] += 1;
else
    x[i] -= 2;
```

在SIMD系统的分支执行

时间	x[i]>=0 的数据通路	x[i]<0 的数据通路
1	测试 x[i]>=0	测试 x[i]>=0
2	x[i]+=1	空闲
3	空闲	x[i]-=2

在多个SM系统上的分支执行

时间	x[i]>=0 的数据通路（在 SM A 上）	x[i]<0 的数据通路（在 SM B 上）
1	测试 x[i]>=0	测试 x[i]>=0
2	x[i]+=1	x[i]-=2

GPU的编程API

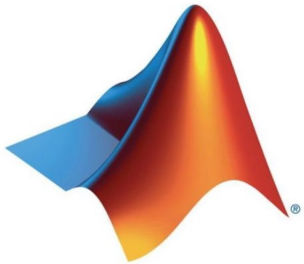
- OpenCL.
- CUDA.

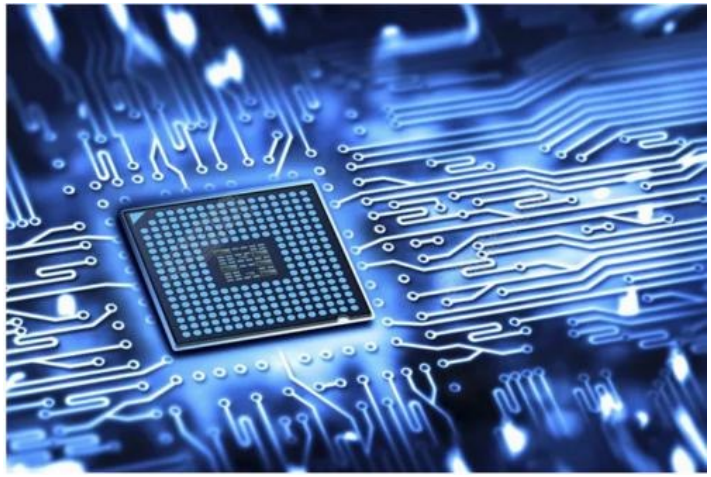








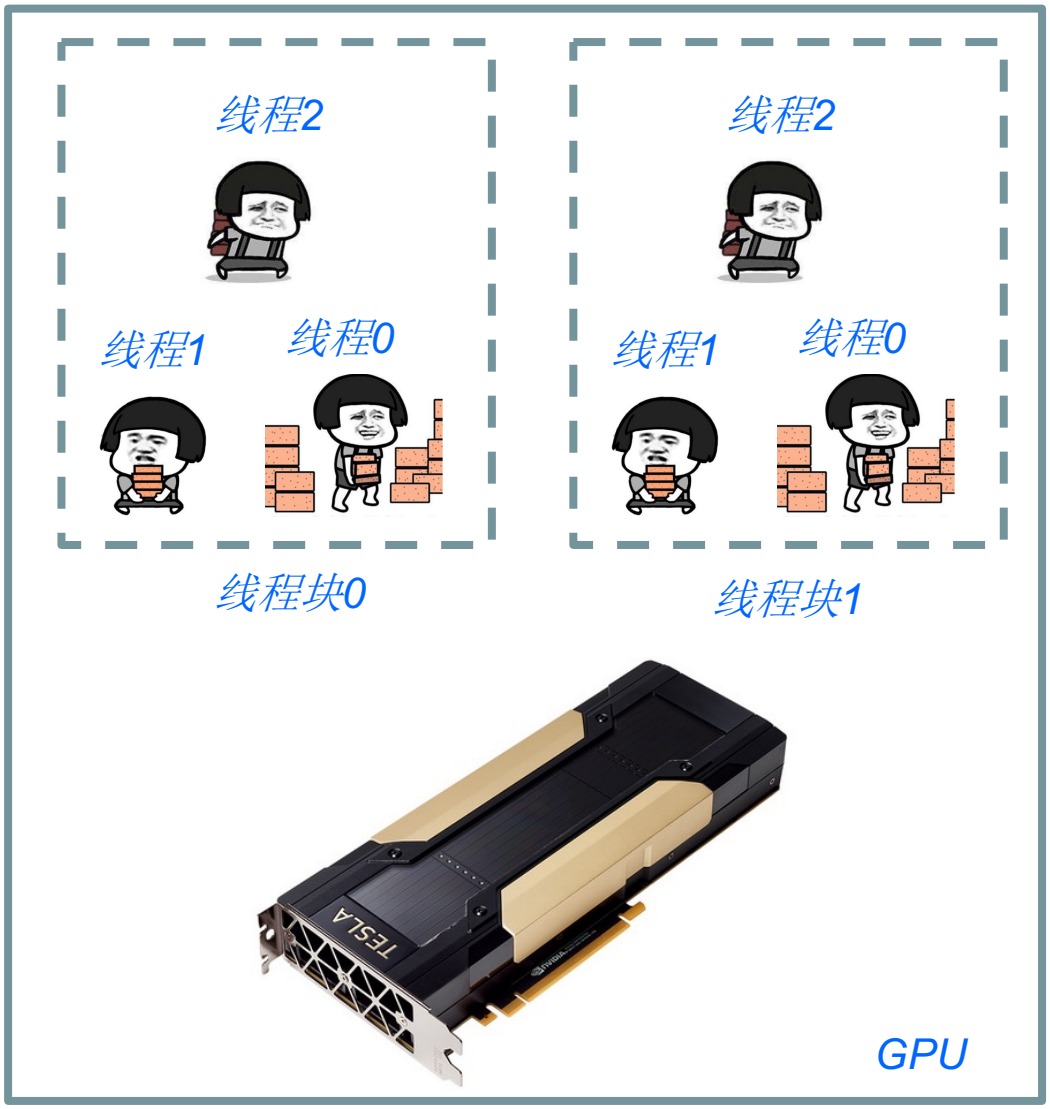




CUDA编程模型

使用CUDA编程





核函数

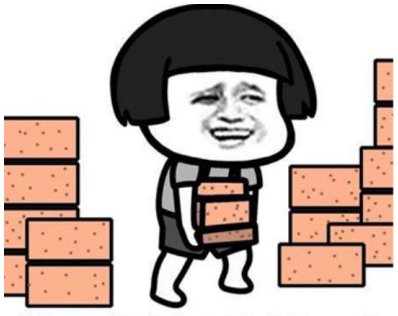
- 核函数，该函数由GPU上的每个线程运行.
- 一个由主机启动，但在设备上运行的函数.
- CUDA的核函数由关键字__global__标识，并且它们始终具有返回类型void.



核函数

__global__ void KernelFunction(…);

__global__ void 搬砖(



);

核函数规定了每个线程将执行的代码

线程、线程块、线程网络

```
int main(){
        KernelFunction();
}
```

```
int main(){
        KernelFunction <<<1,1>>>();
}
```



加入线程调度标记

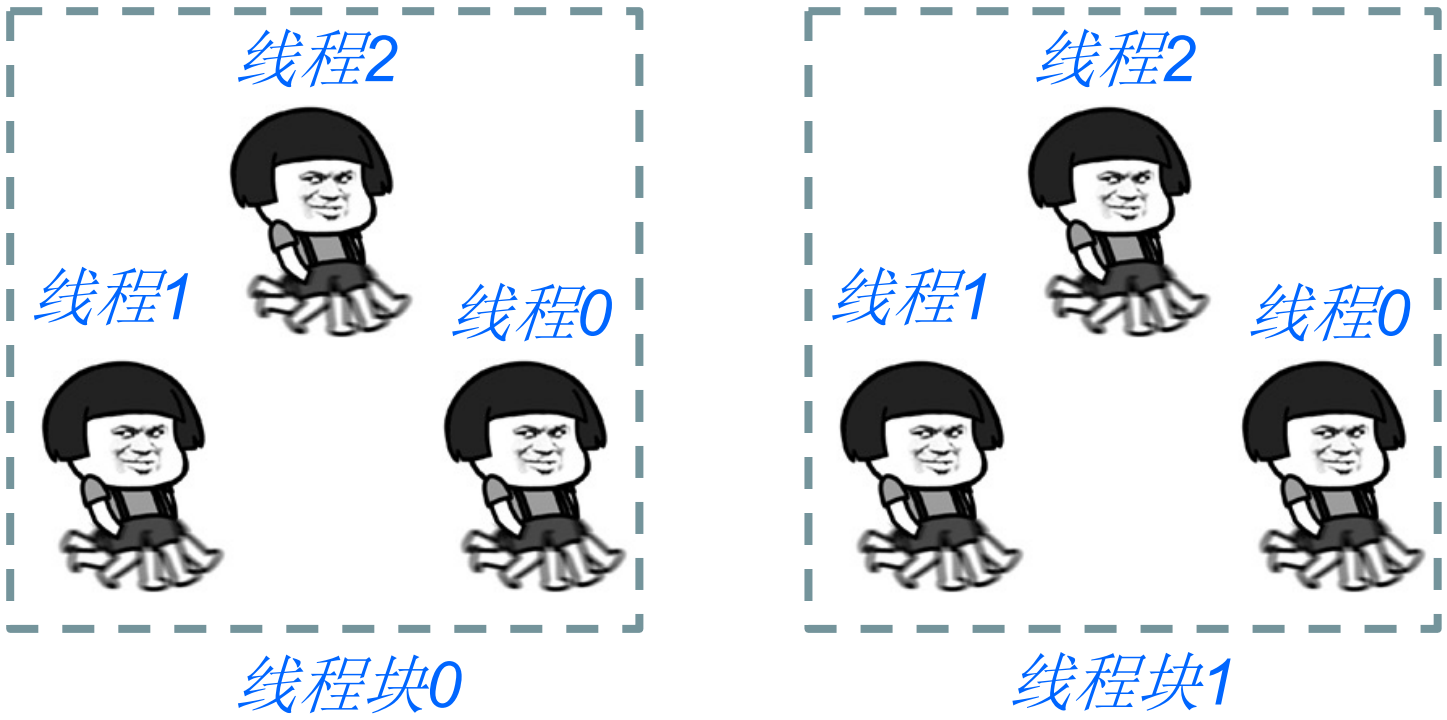
线程、线程块、线程网络

- CUDA将线程组织成块和网络.
- 线程块（简称块）是在单个SM上运行的线程集合.
- 核函数调用时，尖括号中的第一个值指定线程块的数量，第二个值是每个线程块中的线程数量.

线程、线程块、线程网络

```
int main(){
        KernelFunction <<<2,3>>>();
}
```

线程块　线程



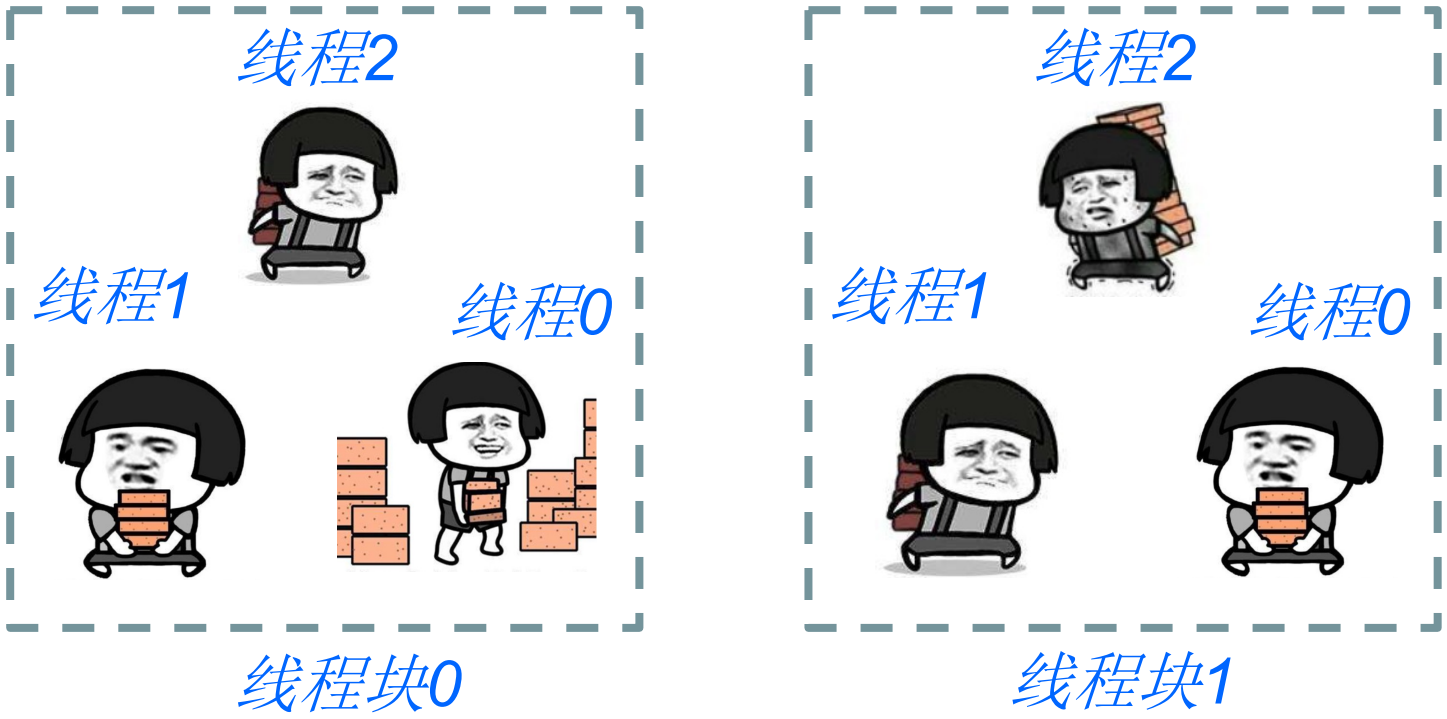
线程、线程块、线程网络

- 当核函数启动时，每个块都会分配一个SM，块中的线程在该SM上运行.
- 系统变量threadIdx.x来给出线程在块中的序列号或索引.
- 系统变量blockIdx.x来给出线程块在线程网格的序列号或索引.

- 网格，只是由核函数启动的线程块的集合.

线程、线程块、线程网络

```
int main(){
        搬砖 <<<2,3>>>(blockIdx.x,threadIdx.x);
}
```



线程、线程块、线程网络

- threadIdx:线程在其线程块中的序号或索引.
- blockDim:线程块的维度、形状或大小.
- blockIdx:块在网格中的序列号或索引.
- gridDim:网格的维度、形状或大小.

线程、线程块、线程网络

```
dim3 grid_dims, block_dims;
grid_dims.x = 2;
grid_dims.y = 3;
grid_dims.z = 1;
block_dims.x = 4;
block_dims.y = 4;
block_dims.z = 4;
...
Kernel <<<grid_dims, block_dims>>> (...);
```

让GPU输出“Hello world!”

```
#include <stdio.h>
#include <cuda.h>    /* CUDA 的头文件 */

/* 设备代码: 运行在GPU上 */
__global__ void Hello(void) {

    printf("Hello from thread %d!\n", threadIdx.x);
}  /* Hello */


/* 主机代码: 运行在CPU上 */
int main(int argc, char* argv[]) {
    int thread_count;       /* GPU上运行的线程数 */

    thread_count = strtol(argv[1], NULL, 10);
                        /* 从命令行获取thread_count */

    Hello <<<1, thread_count>>>();
                        /* 在GPU上启动thread_count线程 */

    cudaDeviceSynchronize();          /* 等待GPU完成 */

    return 0;
}  /* main */
```

```
nvcc -o cuda_hello cuda_hello.cu

./ cuda_hello 10
```

compiling

running with 10 threads

possible outcomes →

```
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
Hello from thread 3!
Hello from thread 4!
Hello from thread 5!
Hello from thread 6!
Hello from thread 7!
Hello from thread 8!
Hello from thread 9!
```

CUDA执行

- 执行从main函数开始，它在主机上运行.
- 线程的数量来自命令行.
- 对Hello函数的调用启动了核函数.
- 调用中的 <<<1, thread_count >>>参数，指定了在设备上启动thread_count个核函数副本。

CUDA执行

- 当核函数启动时，系统会初始threadIdx.x结构体，在我们的例子中，字段包含线程的索引或序列号.
- 每个线程打印其信息并终止.
- main函数中对cudaDeviceSynchonize的调用迫使主机等待，直到所有的线程都完成了核函数的执行，然后主机再继续后续指令及结束执行.

另一个“Hello world!”

```
#include <stdio.h>
#include <cuda.h>    /* CUDA 的头文件 */

/* 设备代码: 运行在GPU上 */
__global__ void Hello(void) {

    printf("Hello from thread %d in block %d\n",
           threadIdx.x, blockIdx.x);
}  /* Hello */


/* 主机代码: 运行在CPU上 */
int main(int argc, char* argv[]) {
    int blk_ct;                    /* 线程块的数量 */
    int th_per_blk;       /* 每个块中的线程数 */

    blk_ct = strtol(argv[1], NULL, 10);
                    /* 从命令行获取线程块的数量 */
    th_per_blk = strtol(argv[2], NULL, 10);
        /* 从命令行获取每个块中的线程数 */

    Hello <<<blk_ct, th_per_blk>>>();
              /* 在GPU上启动blk_ct*th_per_blk线程, */

    cudaDeviceSynchronize();          /* 等待GPU完成 */

    return 0;
}  /* main */
```

注意！！！

- CUDA核函数的启动是异步的，加入路障（cudaDeviceSynchonize）来完成同步.
- GPU中的线程数是有限的，如早期GPU每个块的最大线程数为1024（32*32）。



编写你的第一个CUDA程序

向量加法

- 假设有两个长度为n的一维数组：

$$\mathbf{a} = [a_1, a_2, \ldots, a_n], \quad \mathbf{b} = [b_1, b_2, \ldots, b_n]$$

- 它们的元素对应相加得到的新数组c为：

$$\mathbf{c} = \mathbf{a} + \mathbf{b} = [a_1 + b_1,\ a_2 + b_2,\ \ldots,\ a_n + b_n]$$

- 用索引表示为：

$$c_i = a_i + b_i, \quad \text{其中 } i = 1, 2, \ldots, n$$

向量加法的CUDA程序设计

- 核函数设计.
- 线程分配设计.
- 数据拷贝设计.

向量加法的C语言实现

```
void Serial_vec_add(
        const float   x[]    /* in   */,
        const float   y[]    /* in   */,
        float         cz[]   /* out  */,
        const int     n      /* in   */) {

    for (int i = 0; i < n; i++)
        cz[i] = x[i] + y[i];
}  /* Serial_vec_add */
```

向量加法的核函数设计

```
__global__ void Vec_add(
        const float x[]   /* in   */,
        const float y[]   /* in   */,
        float       z[]   /* out  */,
        const int   n     /* in   */) {
    int my_elt = blockDim.x * blockIdx.x + threadIdx.x;
    /* 总线程 = blk _c t* th _per_blk可能>n  */
    if (my_elt < n)
        z[my_elt] = x[my_elt] + y[my_elt];
}  /* Vec_add */
```

- 可以使用公式为每个线程分配一个唯一的“全局”序列号或索引.

```
rank = blockDim.x * blockIdx.x + threadIdx.x
```

- 一个有4个块，每个块有5个线程的网格全局线程序列号.

blockIdx.x	threadIdx.x				
	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

向量加法的线程分配设计

- 参数任意定义，注意不要超过GPU的计算性能.
- 尽量选择1,2,4,8,16,32,64,128,256,512,1024这样的线程数量。

```
Vec_add <<<blk_ct, th_per_blk>>>(x, y, z, n);
```

向量加法的线程分配设计

```
Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);

void Get_args(
      const int argc          /* in  */,
      char*     argv[]        /* in  */,
      int*      n_p           /* out */,
      int*      blk_ct_p      /* out */,
      int*      th_per_blk_p  /* out */,
      char*     i_g           /* out */) {
   if (argc != 5) {
      /* 打印错误消息并退出 */
      ...
   }

   *n_p = strtol(argv[1], NULL, 10);
   *blk_ct_p = strtol(argv[2], NULL, 10);
   *th_per_blk_p = strtol(argv[3], NULL, 10);
   *i_g = argv[4][0];

   /* 是否n > 线程的总数= blk_ct*th_per_blk */
   if (*n_p > (*blk_ct_p)*(*th_per_blk_p)) {
      /* 打印错误消息并退出 */
      ...
   }
} /* Get_args */
```

向量加法的数据拷贝设计

- 设备端无法直接访问主机端的数据.

向量加法的数据拷贝设计

```
void Allocate_vectors(
        float** x_p     /* out */,
        float** y_p     /* out */,
        float** z_p     /* out */,
        float** cz_p    /* out */,
        int     n       /* in  */) {

    /* x, y和z在主机和设备上使用 */
    cudaMallocManaged(x_p, n*sizeof(float));
    cudaMallocManaged(y_p, n*sizeof(float));
    cudaMallocManaged(z_p, n*sizeof(float));

    /* cz只在主机上使用 */
    *cz_p = (float*) malloc(n*sizeof(float));
}  /* Allocate_vectors */
```

统一内存系统

- CUDA在6.0版本之后可以用“统一内存系统”自动管理内存分配.
- 该系统允许程序员编写CUDA程序指向使用cudaMallocManaged分配的内存的指针可以在设备和主机上使用

显式内存传输

```
void Allocate_vectors(
        float** hx_p  /* out */,
        float** hy_p  /* out */,
        float** hz_p  /* out */,
        float** cz_p  /* out */,
        float** dx_p  /* out */,
        float** dy_p  /* out */,
        float** dz_p  /* out */,
        int     n     /* in  */)  {

    /* 设备上使用dx、dy、dz */
    cudaMalloc(dx_p, n*sizeof(float));
    cudaMalloc(dy_p, n*sizeof(float));
    cudaMalloc(dz_p, n*sizeof(float));

    /* 主机上使用hx、hy、hz、cz */
    *hx_p = (float*) malloc(n*sizeof(float));
    *hy_p = (float*) malloc(n*sizeof(float));
    *hz_p = (float*) malloc(n*sizeof(float));
    *cz_p = (float*) malloc(n*sizeof(float));
}  /* Allocate_vectors */
```

显式内存传输

- 另一种方案，程序员可以显示地完成内存传输.
- 这种方案需要程序员使用cudaMalloc函数在设备端分配内存,malloc函数在主机端分配内存.

显式内存传输

```
/* 将向量x和y从主机复制到设备 */
cudaMemcpy(dx, hx, n*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(dy, hy, n*sizeof(float), cudaMemcpyHostToDevice);


Vec_add <<<blk_ct, th_per_blk>>>(dx, dy, dz, n);

/* 等待内核完成并将结果复制到主机 */
cudaMemcpy(hz, dz, n*sizeof(float), cudaMemcpyDeviceToHost);
```

显式内存传输

- 在核函数执行前后需要对数据完成拷贝.
- 使用cudaMemcpy函数实现显式的内存传输.

内存释放

```
void Free_vectors(
        float*  x     /* in/out */,
        float*  y     /* in/out */,
        float*  z     /* in/out */,
        float*  cz    /* in/out */) {

    /* 由cudaMallocManaged 分配 */
    cudaFree(x);
    cudaFree(y);
    cudaFree(z);

    /* 由malloc分配 */
    free(cz);
}   /* Free_vectors */
```

向量加法的CUDA版本

```
__global__ void Vec_add(
        const float x[]   /* in  */,
        const float y[]   /* in  */,
        float       z[]   /* out */,
        const int   n     /* in  */) {
    int my_elt = blockDim.x * blockIdx.x + threadIdx.x;
    /* 总线程 = blk _c t* th _per_blk可能>n */
    if (my_elt < n)
        z[my_elt] = x[my_elt] + y[my_elt];
}  /* Vec_add */

int main(int argc, char* argv[]) {
    int n, th_per_blk, blk_ct;
    char i_g;  /* x和y是用户输入的还是随机的? */
    float *x, *y, *z, *cz;
    double diff_norm;

    /* 获取命令行参数, 并设置向量 */
    Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
    Allocate_vectors(&x, &y, &z, &cz, n);
    Init_vectors(x, y, n, i_g);
```

向量加法的CUDA版本

```
    /* 调用核函数并等待它完成 */
    Vec_add <<<blk_ct, th_per_blk>>>(x, y, z, n);
    cudaDeviceSynchronize();

    /* 检查正确性 */
    Serial_vec_add(x, y, cz, n);
   diff_norm = Two_norm_diff(z, cz, n);
    printf("Two-norm of difference between host and ");
    printf("device = %e\n", diff_norm);

    /* 释放存储并退出 */
    Free_vectors(x, y, z, cz);
    return 0;
}   /* main */
```

另一个版本（显式内存操作）

```
__global__ void Vec_add(
        const float  x[]  /* in  */,
        const float  y[]  /* in  */,
        float        z[]  /* out */,
        const int    n    /* in  */) {
    int my_elt = blockDim.x * blockIdx.x + threadIdx.x;

    if (my_elt < n)
        z[my_elt] = x[my_elt] + y[my_elt];
}  /* Vec_add */

int main(int argc, char* argv[]) {
    int n, th_per_blk, blk_ct;
    char i_g;  /* x和y是用户输入的还是随机的? */
    float *hx, *hy, *hz, *cz; /* 主机数组 */
    float *dx, *dy, *dz;      /* 设备数组 */
    double diff_norm;

    Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
    Allocate_vectors(&hx, &hy, &hz, &cz, &dx, &dy, &dz, n);
    Init_vectors(hx, hy, n, i_g);
```

另一个版本（显式内存操作）

```
    /* 将向量x和y从主机复制到设备 */
    cudaMemcpy(dx, hx, n*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dy, hy, n*sizeof(float), cudaMemcpyHostToDevice);


    Vec_add <<<blk_ct, th_per_blk>>>(dx, dy, dz, n);

    /* 等待内核完成并将结果复制到主机 */
    cudaMemcpy(hz, dz, n*sizeof(float), cudaMemcpyDeviceToHost);

    Serial_vec_add(hx, hy, cz, n);
    diff_norm = Two_norm_diff(hz, cz, n);
    printf("Two-norm of difference between host and ");
    printf("device = %e\n", diff_norm);

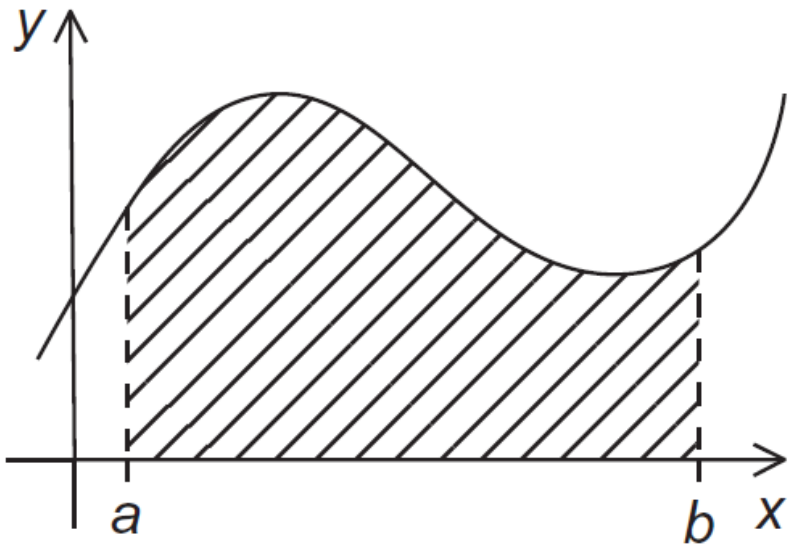
    Free_vectors(hx, hy, hz, cz, dx, dy, dz);

    return 0;
}   /* main */
```

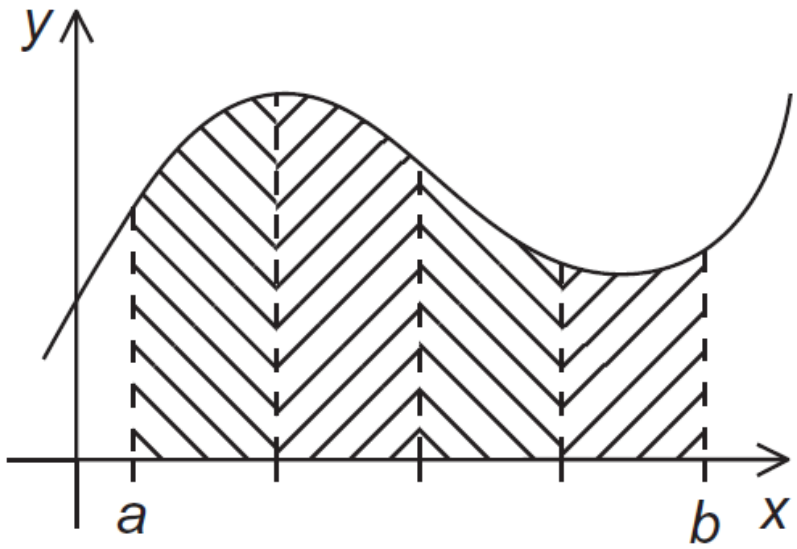


用CUDA来实现梯形积分法

梯形积分法



(a)



(b)

$$h\left[\frac{1}{2}\left(f(a)+f(b)\right)+\left(f(x_1)+f(x_2)+\cdots+f(x_{n-1})\right)\right]$$

梯形积分法

```
float Serial_trap(
      const float  a  /* in */,
      const float  b  /* in */,
      const int    n  /* in */) {
   float x, h = (b-a)/n;
   float trap = 0.5*(f(a) + f(b));

   for (int i = 1; i <= n-1; i++) {
      x = a + i*h;
      trap += f(x);
   }
   trap = trap*h;

   return trap;
}  /* Serial_trap */
```

如何用CUDA加速？

- 每一个循环都可以执行一个串行for循环中的迭代.

如何用CUDA加速？

```
/* h和trap是核函数的形式参数 */
int my_i = blockDim.x * blockIdx.x + threadIdx.x;
float my_x = a + my_i*h;
float my_trap = f(my_x);
float trap += my_trap;
```

如何用CUDA加速？

- 为每一个线程分配一个唯一的整数序列号.
- 每一个线程都计算一个函数值.
- 将函数值累加到“运行总和”中.

一些问题

1. 没有初始化h或者trap，以及trap结果的处理（trap总和与h相乘）
2. 串行代码中的trap由函数返回.
3. my_i的值可能太大，也可能太小.
4. 变量trap必须在线程间共享.

问题1

- 没有初始化h或者trap，以及trap结果的处理（trap总和与h相乘）

如何完成变量的初始化和最后更新?

初始化、最后更新

```
int my_i = blockDim.x * blockIdx.x + threadIdx.x;
if (my_i == 0) {
    h = (b-a)/n;
    trap = 0.5*(f(a) + f(b));
}
...
if (my_i == 0)
    trap = trap*h;
```

问题2

- 串行代码中的trap由函数返回.

trap是多个线程更新的，如何多线程共享trap?

预分配内存

```
/* 主机代码 */
float* trap_p;
cudaMallocManaged(&trap_p, sizeof(float));
...

*trap_p = 0.5*(f(a) + f(b));

/* 调用核函数 */
...

/* 从核函数返回之后 */
*trap_p = h*(*trap_p);
```

问题3

- my_i的值可能太大，也可能太小.

如何保证线程的正确？

使用正确的线程

- 用if语句约束线程号

```
if (0 < my_i && my_i < n) {
    /* 计算x, f(x), 并加到* trap_p中 */
    ...
}
```

问题4

* trap_p += my_trap;

是否可行?

原子操作

- 内存位置共享，从而形成了竞争条件，最终trap_p的实际值是不可预测的.

- 解决方案：原子操作.
- 原子操作，是指对全局内存或共享内存中的数据进行不可分割的读-改-写操作.
- 它保证在一个线程执行该操作的过程中，没有其他线程能中断或干扰.

原子操作

- 原子操作，加法

```
__device__ float atomicAdd(
        float*   float_p     /* in/out */,
        float    val         /* in     */);
```

CUDA实现梯形积分法

```
__global__ void Dev_trap(
      const float  a        /* in      */,
      const float  b        /* in      */,
      const float  h        /* in      */,
      const int    n        /* in      */,
      float*       trap_p   /* in/out  */) {
   int my_i = blockDim.x * blockIdx.x + threadIdx.x;

   /* f(x_0)和 f(x_n) 在主机上计算                    */
   /* 所以计算f(x_1),f(x_2), …, f(x_(n-1))          */
   if (0 < my_i && my_i < n) {
      float my_x = a + my_i*h;
      float my_trap = f(my_x);
      atomicAdd(trap_p, my_trap);
   }
}  /* Dev_trap */

/* 主机代码 */
void Trap_wrapper(
```

CUDA实现梯形积分法

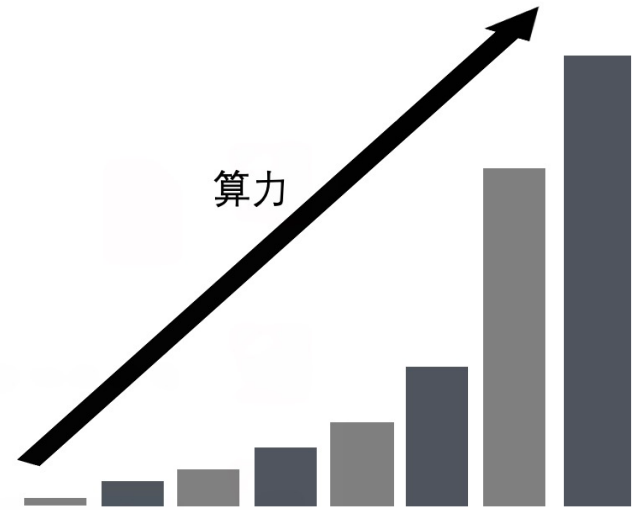
```
    const float  a              /* in  */,
    const float  b              /* in  */,
    const int    n              /* in  */,
    float*       trap_p         /* out */,
    const int    blk_ct         /* in  */,
    const int    th_per_blk     /* in  */) {

  /* 用cudaMallocManaged在main中分配trap_p存储 */

  *trap_p = 0.5*(f(a) + f(b));
  float h = (b-a)/n;

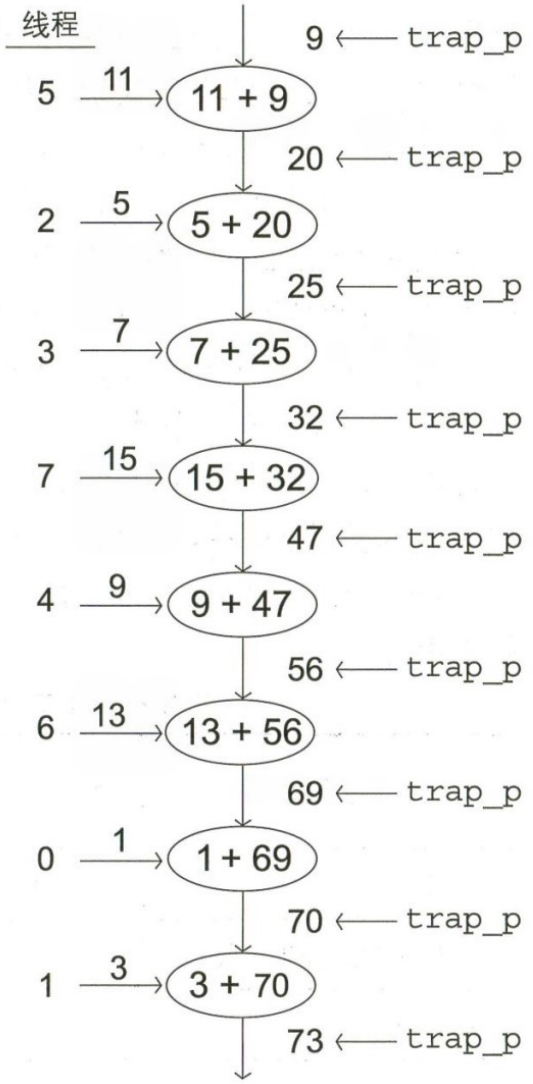
  Dev_trap<<<blk_ct, th_per_blk>>>(a, b, h, n, trap_p);
  cudaDeviceSynchronize();

  *trap_p = h*(*trap_p);
} /* Trap_wrapper */
```



CUDA性能提升

原子操作，高效？



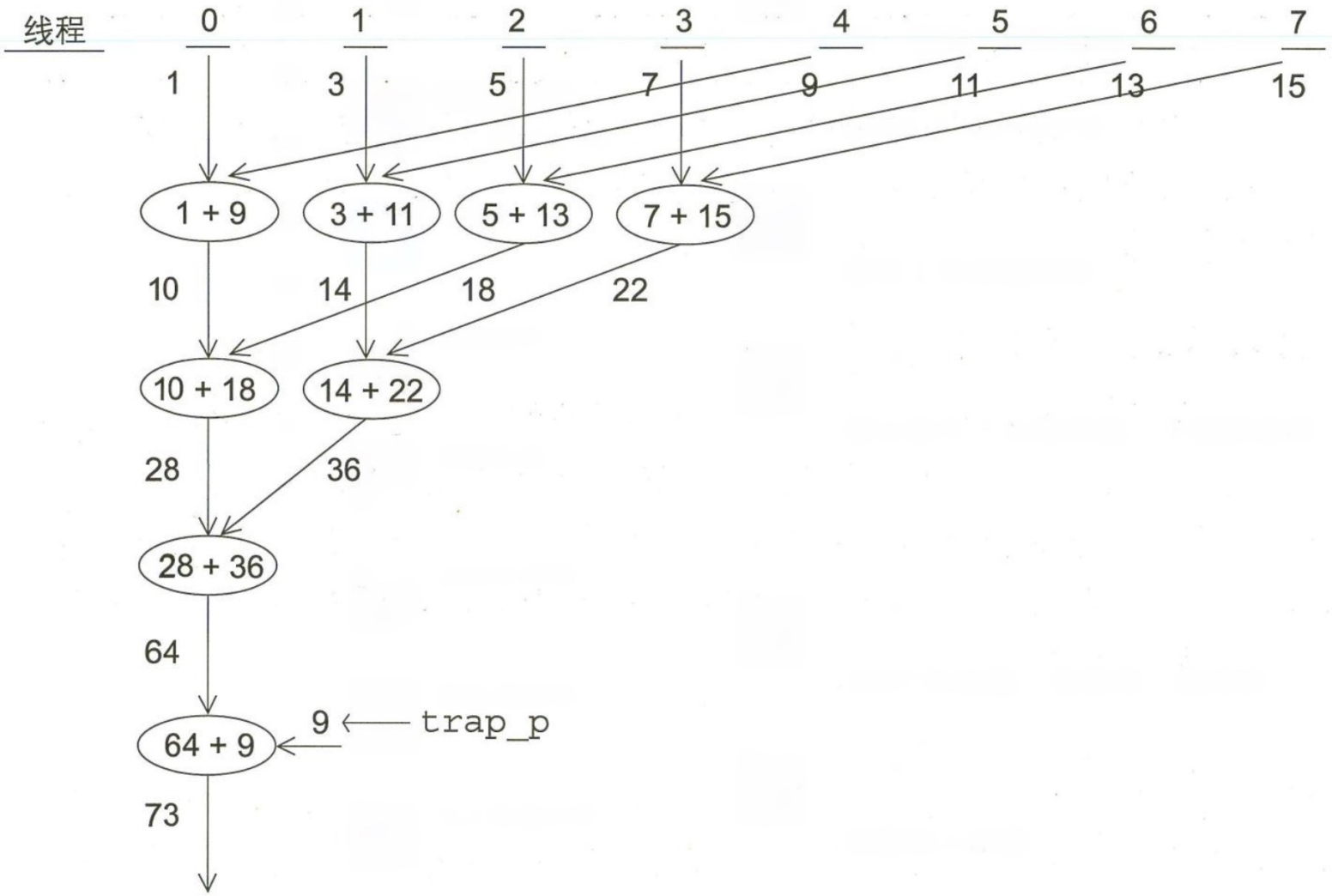
可能出现的原子操作过程

原子操作，高效？

- 原子操作使用串行化线程的手段解决了访问的竞争问题.

如何进一步提升效率?

树形通信（规约）

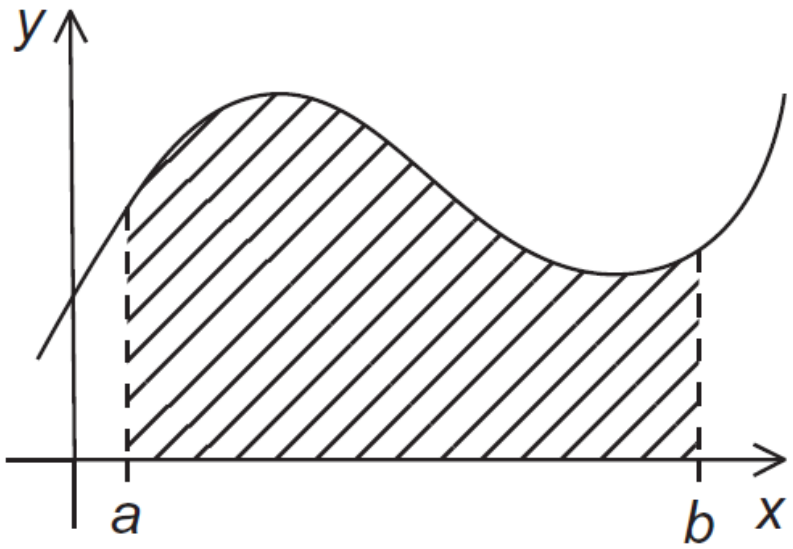

线程
0
1
2
3
4
5
6
7
1
3
5
7
9
11
13
15
1 + 9
3 + 11
5 + 13
7 + 15
10
14
18
22
10 + 18
14 + 22
28
36
28 + 36
64
9
trap_p
64 + 9
73

树形通信（规约）

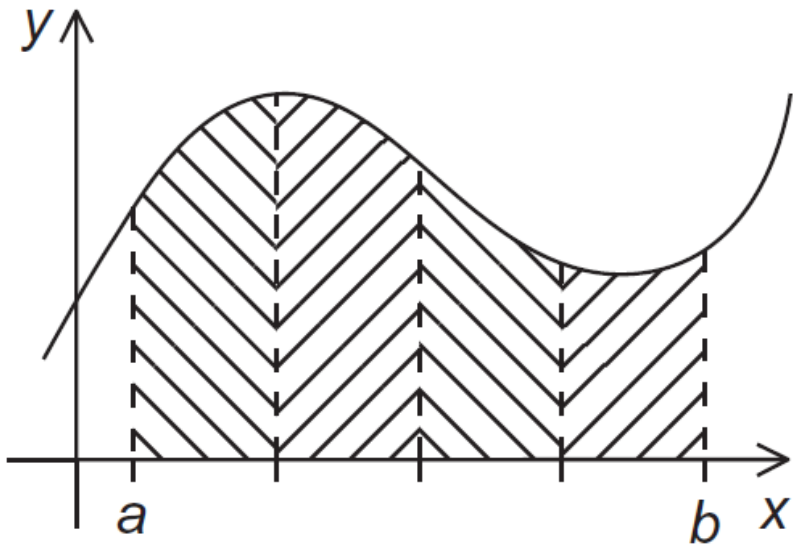
- 树形结构的规约方法.
- 树结构方法需要计算$[\log_2(n)]+1$次加法.

如何更高效率的实现这一方法?
全局内存? 共享内存? 寄存器?

梯形积分法



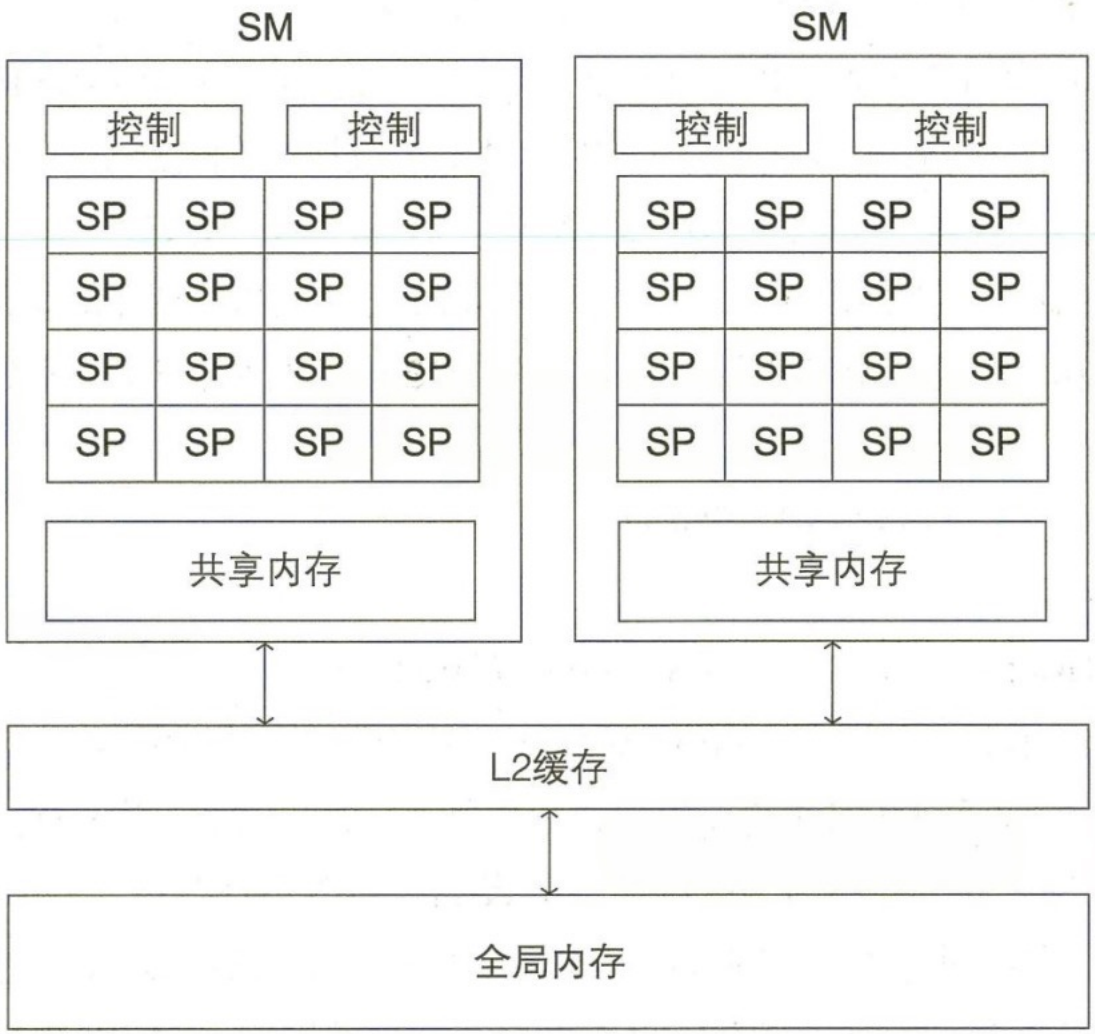
(a)



(b)

$$h\left[\frac{1}{2}\left(f(a)+f(b)\right)+\left(f(x_1)+f(x_2)+\cdots+f(x_{n-1})\right)\right]$$

CUDA内存模型


SM
SM
控制
控制
控制
控制
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
SP
共享内存
共享内存
L2缓存
全局内存

CUDA内存模型

- 三个“级别”的层次结构。
- 全局内存，最慢，最大.
- 共享内存，更快，更小.
- 寄存器，最快，最小.

CUDA内存模型

- 4个字节的整数复制时间：
- 寄存器，1个周期.
- 共享内存，多花一个数量级.
- 全局内存，多花两到三个数量级.

表 6.7　一些 NVIDIA GPU 的内存统计

GPU	计算能力	寄存器：字节每线程	共享内存：字节每块	全局内存：字节每 GPU
Quadro 600	2.1	504	48K	1G
GK20A (Jetson TK1)	3.2	504	48K	2G
GeForce GTX Titan X	5.2	504	48K	12G

CUDA内存模型

- 全局内存，所有SP和所有线程都可以访问全局内存.
- 共享内存，每个SM都可以访问自己的“共享内存”该内存只能由该块中的线程访问.
- 寄存器，每个线程自己访问，储存局部变量.如没有足够空间，局部变量将“溢出”到全局内存中.

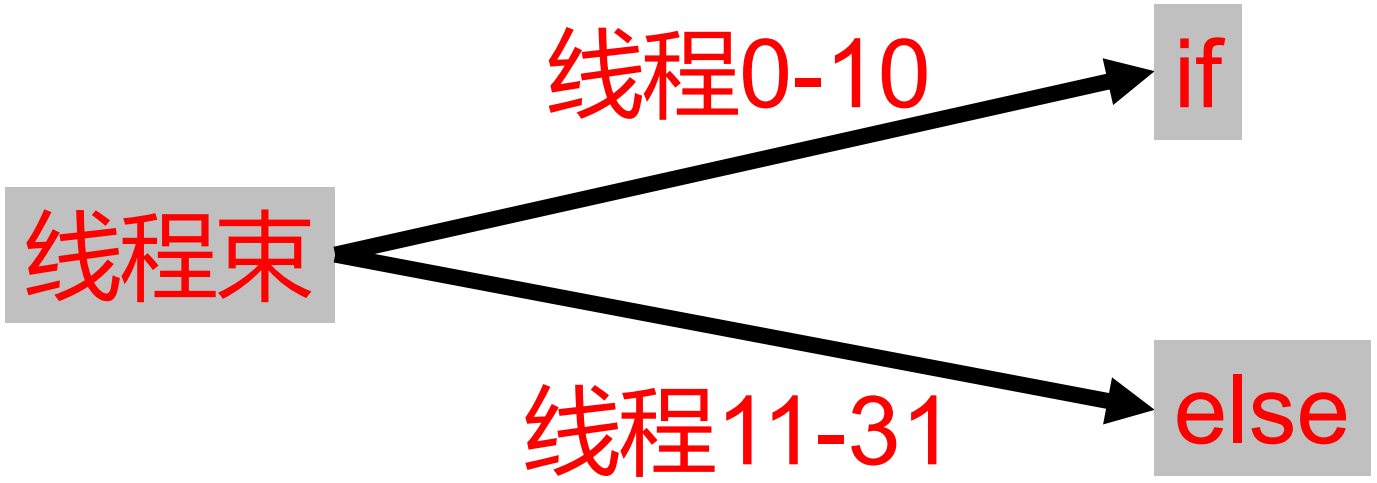
线程束

- 线程束，一组线程属于一个线程块.
- 目前一个线程束中的线程数是32，NVIDIA在声明中表示这可能会改变.
- 系统初始化了一个变量，用于存储线程束的大小“int warpSize”.

线程分歧与收敛

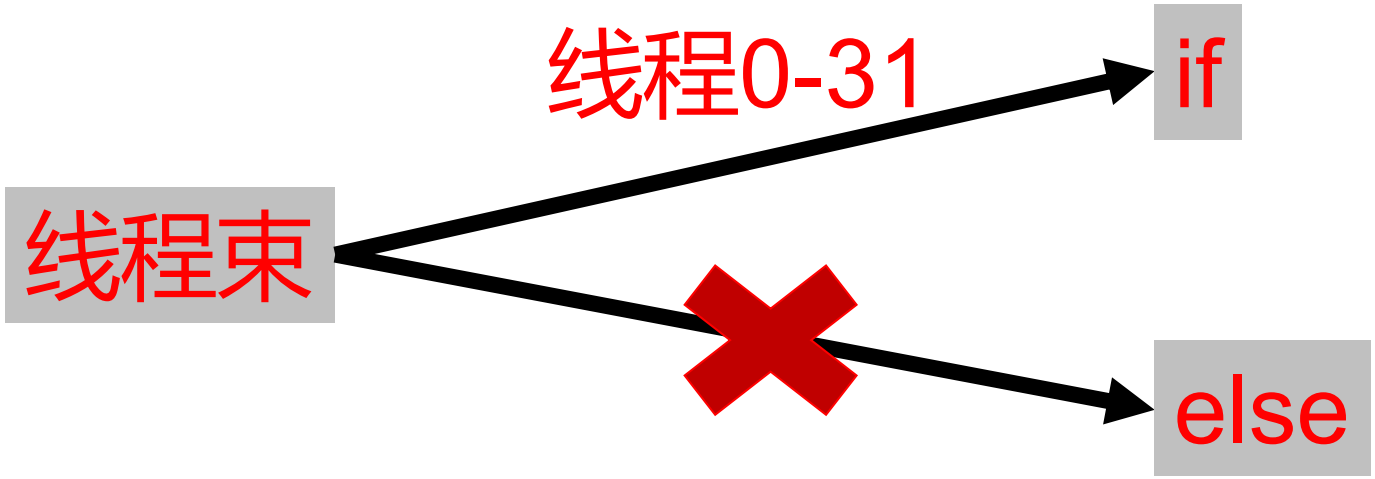
- 不同线程束中的线程可以执行不同的语句而不会产生不利后果.
- 同一个线程束中的线程必须执行相同的语句.
- 线程束中线程的序列号称为线程的通道(lane),lane=threadIdx.x%warpSize;

线程分歧与收敛



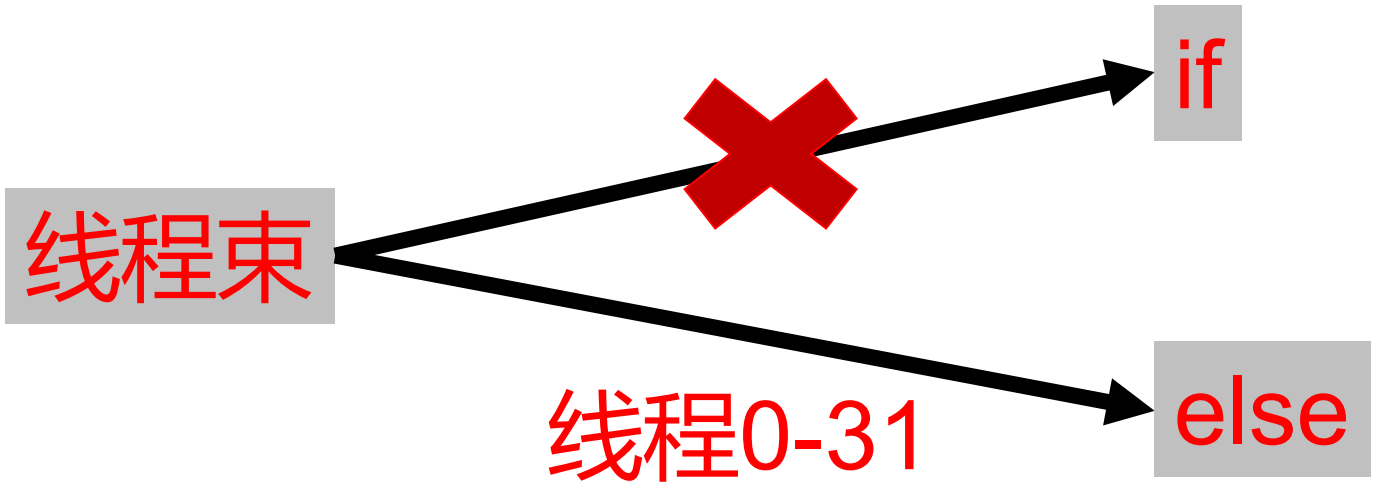
线程分歧

线程分歧与收敛



线程收敛

线程分歧与收敛



线程收敛

线程束洗牌

- 线程束洗牌函数允许线程束中的一个线程从同一个线程束中的另一个线程使用的寄存器中读取数据.

线程束洗牌

```
__device__ float __shfl_down_sync(
        unsigned    mask                    /* in */,
        float       var                     /* in */,
        unsigned    diff                    /* in */,
        int         width = warpSize        /* in */);
```

线程束洗牌

- mask参数表示哪些线程正在参与调用.
- 必须为每个参与线程设置一个线程通道的位，确保调用线程都已经收敛.

mask = 0xffffffff;

1111 1111 1111 1111
1111 1111 1111 1111

线程束洗牌

- 如果具有通道l的线程调用__shfl_down_sync，那么$l+diff$的线程上存储在var中的值被返回给线程l.
  - lane = $l+diff$;

线程束洗牌

- 假设一个warp有32个线程,每个线程手里有一个值var.
- 线程 *l* 拿到线程 *l*+1 的var.

```
__shfl_down_sync(0xffffffff, var, 1);
```

线程束洗牌

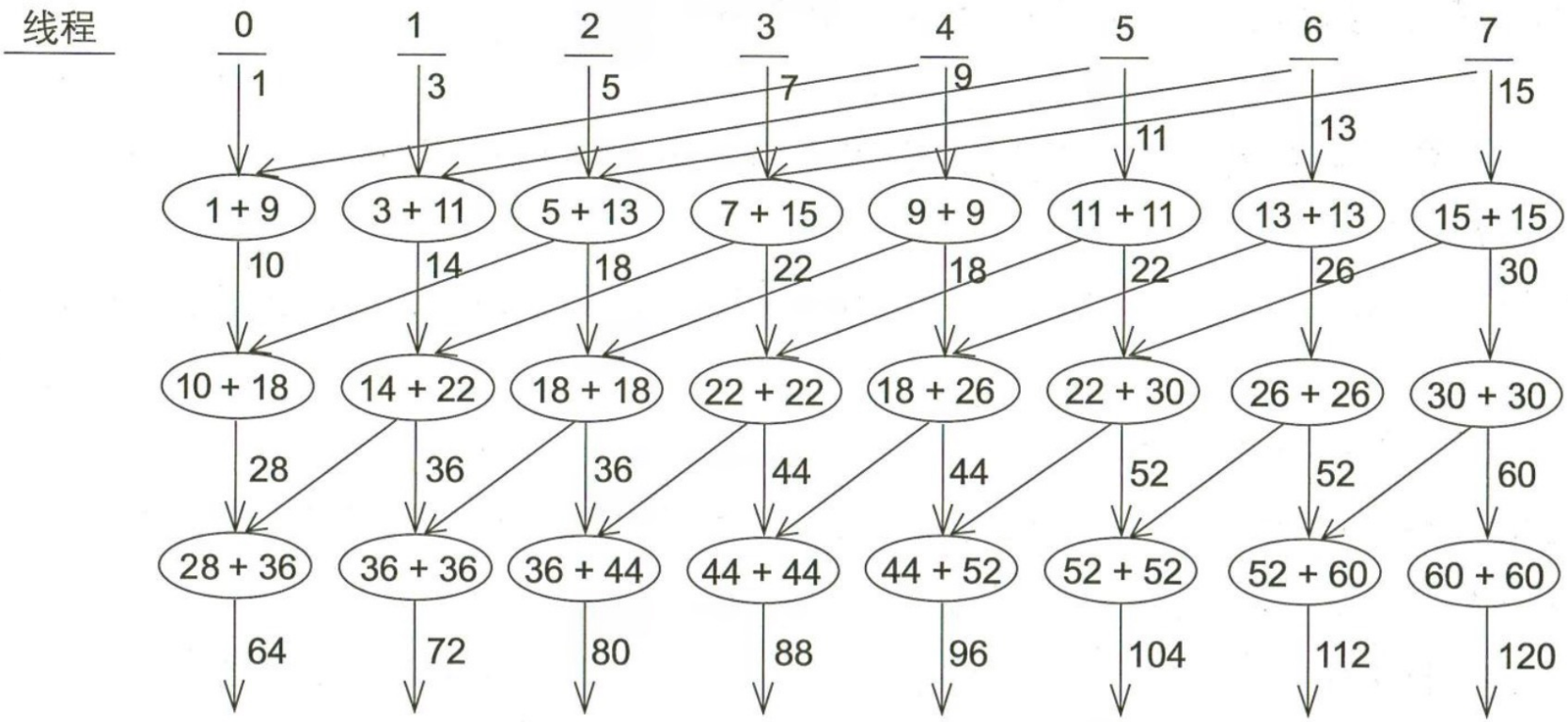
- __shfl_down_sync, lane *l* 拿到 lane *l+diff* 的var.
- __shfl_up_sync, lane *l* 拿到 lane *l-diff*的var.
- __shfl_xor_sync, lane *l* 与 lane *l* XOR *diff* 交换var.
- __shfl_sync, 广播 所有线程都拿到 lane *diff* 的var.

使用线程束洗牌实现规约求和

```
__device__ float Warp_sum(float var) {
    unsigned mask = 0xffffffff;

    for (int diff = warpSize/2; diff > 0; diff = diff/2)
        var += __shfl_down_sync(mask, var, diff);
    return var;
}   /* Warp_sum */
```

使用线程束洗牌实现规约求和



warpSize为8

线程束洗牌

- 如果线程 *l* 调用洗牌函数，但线程*l+diff* 没有调用会发生什么？
- 如果线程 *l* 调用洗牌函数，但线程*l+diff* =>*warpSize*会发生什么？
- 如果线程 *l* 调用洗牌函数，但线程*l+diff* <*warpSize*会发生什么，但*l+diff*>线程束中的最大通道数会发生什么？

线程束洗牌

- 线程束中的所有线程调用洗牌函数.
- 线程块大小需要是warpSize的倍数.

另一种方法，共享内存

- 可以在核函数中定义一个共享内存空间，仅限于每个线程块中内部使用.
- 共享内存需要用“__shared__”来定义.
- 必须直接定义！！！

另一种方法，共享内存

- 两种定义方式

```
__global__ void kernel() {
    __shared__ float shared_vals[32]; //大小固定
}
```

另一种方法，共享内存

- 两种定义方式

```
__global__ void kernel() {
    extern __shared__ float shared_vals[]; // 大小未知，稍后指定
}

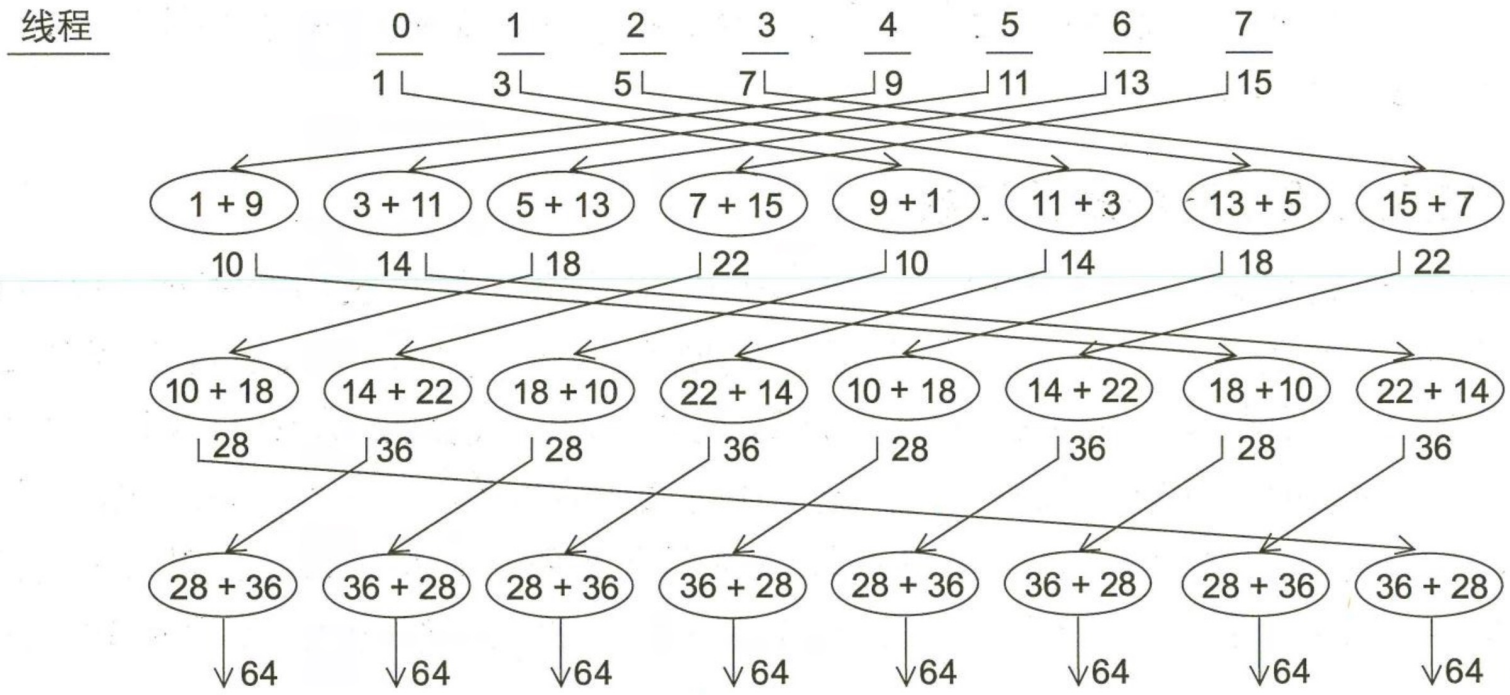
// 启动时指定：每个 block 分配 32*sizeof(float) 字节
kernel<<<grid, block, 32 * sizeof(float)>>>();
```

另一种方法，共享内存

```
__device__ float Shared_mem_sum(float shared_vals[]) {
    int my_lane = threadIdx.x % warpSize;

    for (int diff = warpSize/2; diff > 0;  diff = diff/2) {
        /* 确保0<= source <线程束大小 */
        int source = (my_lane + diff) % warpSize;
        shared_vals[my_lane] += shared_vals[source];
    }
    return shared_vals[my_lane];
}
```

另一种方法，共享内存


线程
0 1 2 3 4 5 6 7
1 3 5 7 9 11 13 15
1 + 9
3 + 11
5 + 13
7 + 15
9 + 1
11 + 3
13 + 5
15 + 7
10 14 18 22 10 14 18 22
10 + 18
14 + 22
18 + 10
22 + 14
10 + 18
14 + 22
18 + 10
22 + 14
28 36 28 36 28 36 28 36
28 + 36
36 + 28
28 + 36
36 + 28
28 + 36
36 + 28
28 + 36
36 + 28
64 64 64 64 64 64 64 64

多个线程块的实现（线程束洗牌）

```
__global__ void Dev_trap(
      const float  a          /*    in    */,
      const float  b          /*    in    */,
      const float  h          /*    in    */,
      const int    n          /*    in    */,
      float*       trap_p     /* in/out   */) {
   int my_i = blockDim.x * blockIdx.x + threadIdx.x;

   float my_trap = 0.0f;
   if (0 < my_i && my_i < n) {
      float my_x = a + my_i*h;
      my_trap = f(my_x);
   }

   float result = Warp_sum(my_trap);

   /* 只有在线程0上的结果是正确的 */
   if (threadIdx.x == 0) atomicAdd(trap_p, result);
}  /* Dev_trap */
```

多个线程块的实现（共享内存）

```
__global__ void Dev_trap(
      const float  a          /* in  */,
      const float  b          /* in  */,
      const float  h          /* in  */,
      const int    n          /* in  */,
      float*       trap_p     /* out */) {
   __shared__ float shared_vals[WARPSZ];
   int my_i = blockDim.x * blockIdx.x + threadIdx.x;
   int my_lane = threadIdx.x % warpSize;

   shared_vals[my_lane] = 0.0f;
   if (0 < my_i && my_i < n) {
      float my_x = a + my_i*h;
      shared_vals[my_lane] = f(my_x);
   }

   float result = Shared_mem_sum(shared_vals);

   /* 在一个块中的所有线程上，结果是相同的 */
   if (threadIdx.x == 0) atomicAdd(trap_p, result);
}  /* Dev_trap */
```

性能比较

- [-3,3]上积分$f(x)=x^2+1$,$2^{20}=1048576$个梯形
- 线程块大小32，32768个线程块（1048576个线程）<<<32768, 32>>>

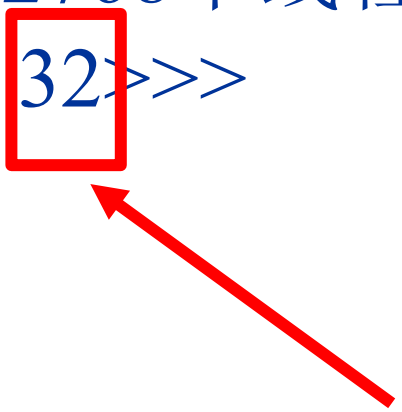
性能比较

表 6.8　使用 32 个线程块的梯形法则的平均运行时间

系统	ARM Cortex-A15	NVIDIA GK20A	Intel Core i7	NVIDIA GeForce GTX Titan X
时钟	2.3　GHz	852 MHz	3.5　GHz	1.08　GHz
SM 的数量，SP 的数量		1, 192		24, 3 072
原始的	33.6	20.7	4.48	3.08
线程束洗牌		14.4		0.210
共享内存		15.0		0.206

使用具有多个线程束的线程块

- [-3,3]上积分$f(x)=x^2+1$,2^{20}=1048576个梯形
- 线程块大小32，32768个线程块（1048576个线程）<<<32768, 32>>>



如何扩展成多个线程束的块

线程束的同步

- __device__ void __syncthreads(void);
- 线程束之间是独立的，在必要情况下，可以使用该函数来使一个线程块中的所有线程同步.

注意事项

- 必须保证块中所有线程都在调用__syncthreads函数.
- __syncthreads函数只保证一个块的线程同步.

```
int my_x = threadIdx.x;
if (my_x < blockDim.x/2)
    __syncthreads();
my_x++;
```

跨线程束的数据共享

共享内存

```
__shared__ float warp_sum_arr[WARPSZ];
int my_warp = threadIdx.x / warpSize;
int my_lane = threadIdx.x % warpSize;
// 线程计算它们的贡献;
...
float my_result = Warp_sum(my_trap);
if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
__syncthreads();
// 线程束0在warp_sum_arr中添加求和
...
```

共享内存库

- NVIDIA将SM上的共享内存划为32个库.
- 线程束中的线程可以同时访问共享内存.
- 当两个或多个线程访问单个库中的不同元素时，必须对这些访问进行串行化.

共享内存库

```
// 使最大线程块大小在编译时可用
#define MAX_BLKSZ 1024
...
__shared__ float thread_calcs[MAX_BLKSZ];
```

表 6.9 共享内存库：列是内存库，表格中的项展示了 thread_calcs 中元素的下标

	内存库					
	0	1	2	...	30	31
下标	0	1	2	...	30	31
	32	33	34	...	62	63
	64	65	66	...	94	95
	96	97	98	...	126	127
	⋮	⋮	⋮	⋮	⋮	⋮
	992	993	994	...	1022	1023

共享内存库

- 如果线程束中的每个线程访问不同的库，则访问同时发生.
- 如果多个线程访问单个库的不同内存位置，则必须对访问进行串行化.
- 如果多个线程读取单个库的相同内存位置，则读取的值将被广播到各个读取线程，并且读取是同时进行的.

共享内存库

用于存储每个线程计算的结果

```
__shared__ float thread_calcs[MAX_BLKSZ];
__shared__ float warp_sum_arr[WARPSZ];
float* shared_vals = thread_calcs + my_warp*warpSize;
...
float my_result = Shared_mem_sum(shared_vals);
if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
__syncthreads();
...
```

用于存储线程求和

更大线程的共享内存积分代码

```
__global__ void Dev_trap(
      const float  a          /* in  */,
      const float  b          /* in  */,
      const float  h          /* in  */,
      const int    n          /* in  */,
      float*       trap_p     /* out */) {
   __shared__ float thread_calcs[MAX_BLKSZ];
   __shared__ float warp_sum_arr[WARPSZ];
   int my_i = blockDim.x * blockIdx.x + threadIdx.x;
   int my_warp = threadIdx.x / warpSize;
   int my_lane = threadIdx.x % warpSize;
   float* shared_vals = thread_calcs + my_warp*warpSize;
   float blk_result = 0.0;

   shared_vals[my_lane] = 0.0f;
   if (0 < my_i && my_i < n) {
      float my_x = a + my_i*h;
      shared_vals[my_lane] = f(my_x);
   }
```

更大线程的共享内存积分代码

```
    float my_result = Shared_mem_sum(shared_vals);
    if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
    __syncthreads();

    if (my_warp == 0) {
        if (threadIdx.x >= blockDim.x/warpSize)
            warp_sum_arr[threadIdx.x] = 0.0;
        blk_result = Shared_mem_sum(warp_sum_arr);
    }

    if (threadIdx.x == 0) atomicAdd(trap_p, blk_result);
} /* Dev_trap */
```

一个块的和

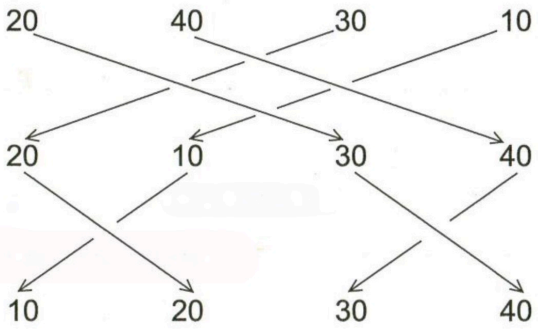
多个块的和

性能比较

- 单个块的线程改为1024.

表 6.10　使用随机块大小的梯形法则的运行时间

系统	ARM Cortex-A15	NVIDIA GK20A	Intel Core i7	NVIDIA GeForce GTX Titan X
时钟 SM 的数量，SP 的数量	2.3　GHz	852 MHz 1, 192	3.5　GHz	1.08　GHz 24, 3 072
原始的	33.6	20.7	4.48	3.08
线程束洗牌，32 ths/blk 共享内存，32 ths/blk		14.4 15.0		0.210 0.206
线程束洗牌 共享内存		12.8 14.3		0.141 0.150



双调排序

算法原理

- 假设n是一个2的幂的正整数.
- 有一个包含n个整数键的列表.
- 任何一对连续元素都可以通过交换转换为递增或递减序列，即2元素蝶式交换.

算法原理

- 如果有一个4元素列表.
- 可以通过2元素蝶式交换获得双调序列.
- 然后用4元素蝶式交换获得排序列表.

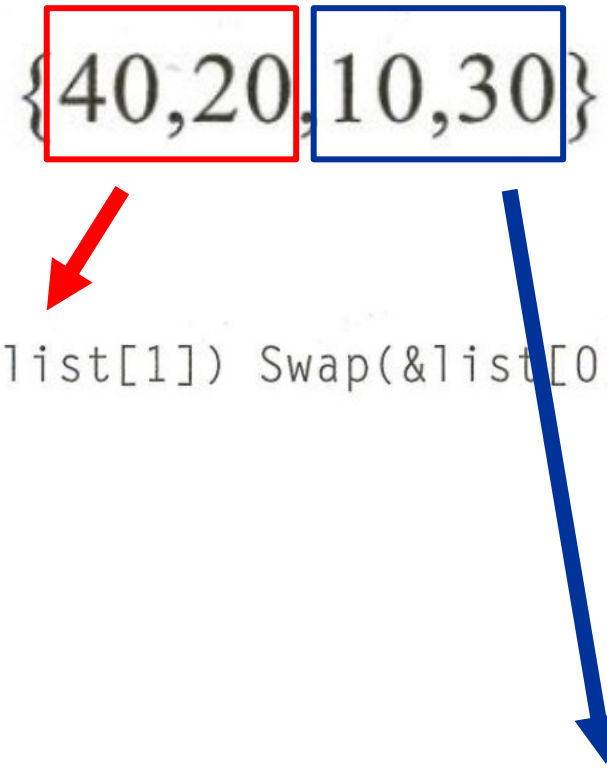
{40,20,10,30}

算法原理

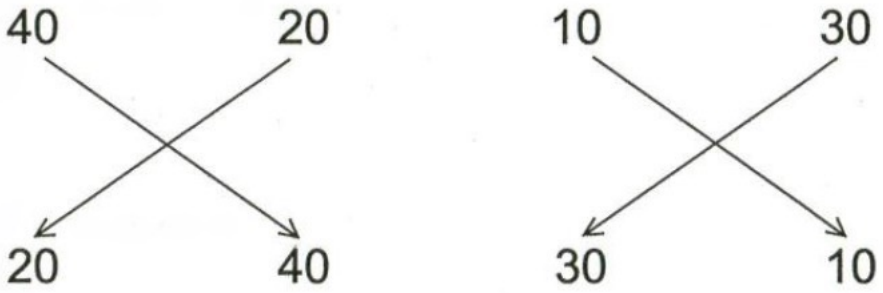
- 双调序列指的是先增后减，再增.
- 或先减后增，再减.

- 即“两个单调”！
- “3，5，4，2，1”，“5，4，2，1，3”

算法原理

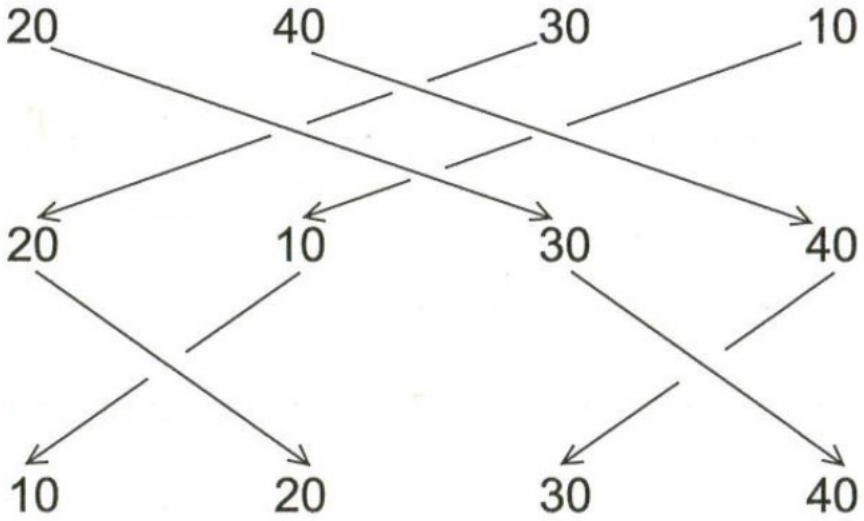

{40,20,10,30}
if (list[0] > list[1]) Swap(&list[0], &list[1]);
if (list[2] < list[3]) Swap(&list[2], &list[3]);

算法原理



双调

图 6.7　两个 2 元素蝶式交换



有序

图 6.8　一个 4 元素蝶式交换

算法原理

表 6.11　4 元素序列的双调排序：首先是两个 2 元素蝶式交换，然后是一个 4 元素蝶式交换

	下标			
	0	1	2	3
开始时的列表	40	20	10	30
期望的两个元素顺序	递增		递减	
2 元素蝶式配对 比较交换后的列表	1 20	0 40	3 30	2 10
期望的四个元素顺序	递增			
4 元素蝶式：阶段 A 配对 阶段 A：比较交换后的列表	2 20	3 10	0 30	1 40
阶段 B 配对 阶段 B：比较交换后的列表	1 10	0 20	3 30	2 40

算法原理

- 如果有一个8元素列表.
- 四个2元素蝶式交换.前四个元素双调，后四个元素双调.
- 两个4元素蝶式交换.前四个递增，后四个递减.
- 一个8元素蝶式变换.有序序列.

算法原理

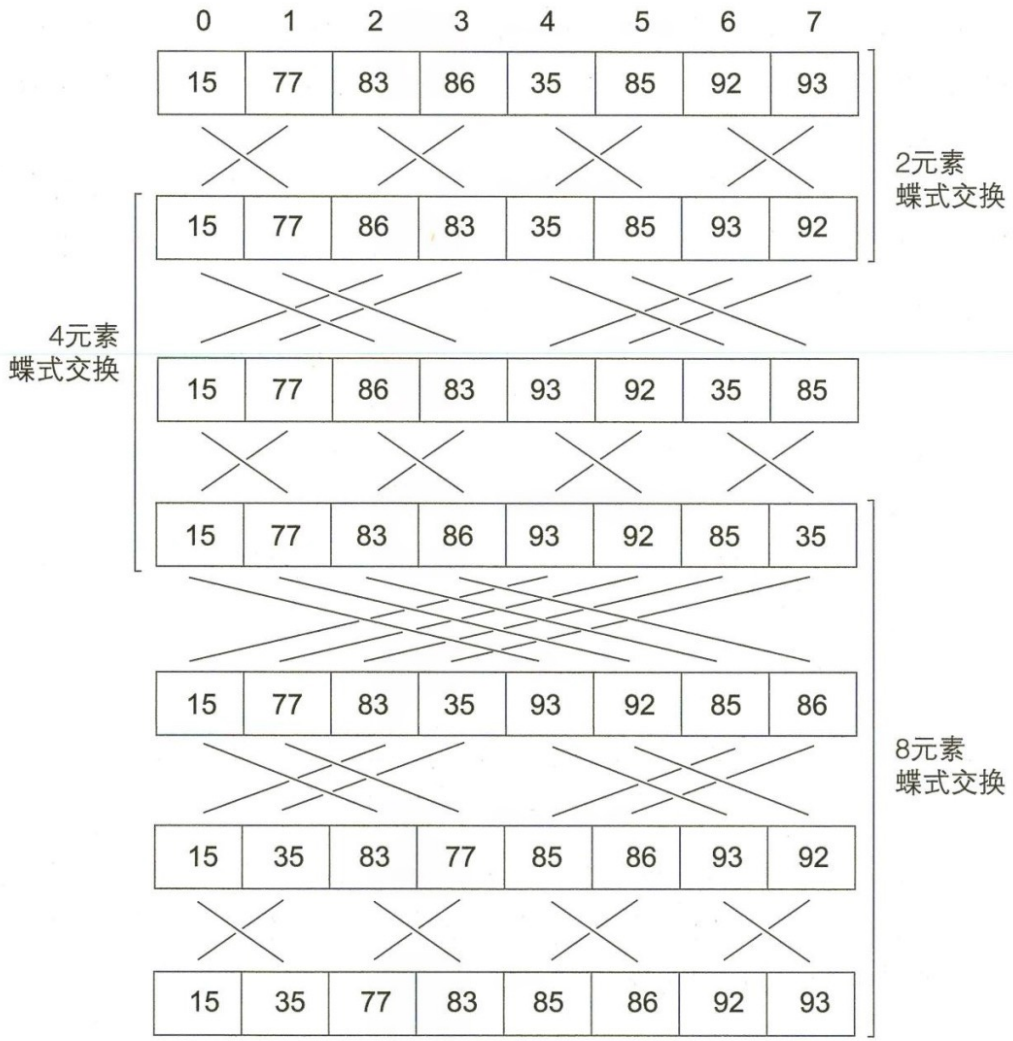


图 6.9　用双调排序为 8 元素列表排序

算法原理

表 6.12　8 元素列表的双调排序

	下标							
	0	1	2	3	4	5	6	7
开始时的列表	15	77	83	86	35	85	92	93
期望的两个元素顺序	递增		递减		递增		递减	
2 元素配对	1	0	3	2	5	4	7	6
比较交换后的列表	15	77	86	83	35	85	93	92
期望的四个元素顺序	递增				递减			
4 元素：阶段 A 配对	2	3	0	1	6	7	4	5
阶段 A：比较交换后的列表	15	77	86	83	93	92	35	85
4 元素：阶段 B 配对	0	1	3	2	5	4	7	6
阶段 B：比较交换后的列表	15	77	83	86	93	92	85	35
期望的八个元素顺序	递增							
8 元素：阶段 A 配对	4	5	6	7	0	1	2	3
阶段 A：比较交换后的列表	15	77	83	35	93	92	85	86
8 元素：阶段 B 配对	2	3	0	1	6	7	4	5
阶段 B：比较交换后的列表	15	35	83	77	85	86	93	92
8 元素：阶段 C 配对	1	0	3	2	5	4	7	6
阶段 C：比较交换后的列表	15	35	77	83	85	86	92	93

双调排序伪代码

迭代蝶式变换的大小

```
for (bf_sz = 2; bf_sz <= n; bf_sz = 2*bf_sz)
    for (stage = bf_sz/2; stage > 0; stage = stage/2)
        for (th = 0; th < n/2; th++) {
            Get_pair(th, stage, &my_elt1, &my_elt2);
            if (Increasing_seq(my_elt1, bf_sz))
                Compare_swap(list, my_elt1, my_elt2, INC);
            else
                Compare_swap(list, my_elt1, my_elt2, DEC);
        }
```

交换

双调排序伪代码

- 如果列表中有n个元素，那么将产生n/2对，这每一对可以由单个线程负责，对应最里面的循环变量th.

算法的二进制表示

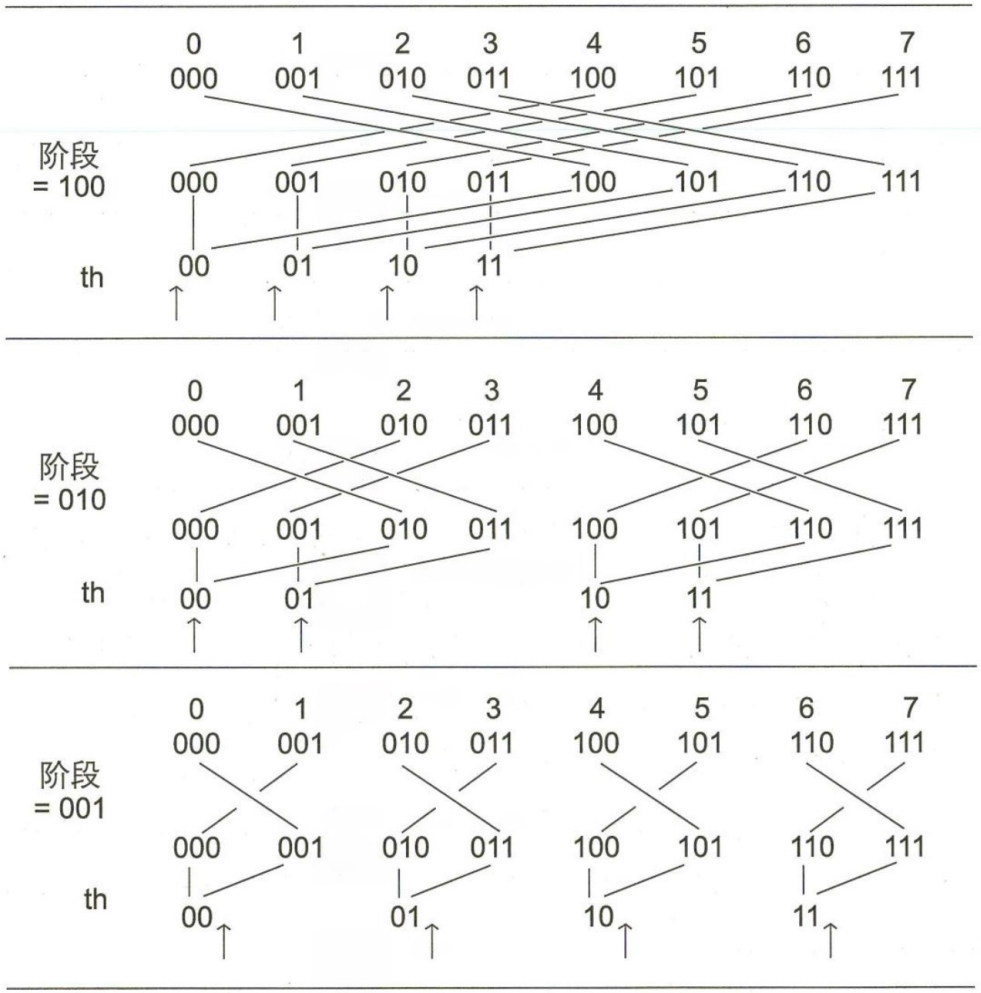


图 6.10　双调排序中用到的不同值的二进制表示

算法的二进制表示

表 6.13　双调排序中线程和元素的映射

阶段	线程	第一个元素	第二个元素
100 = 4	00 = 0	000 = 0	100 = 4
	01 = 1	001 = 1	101 = 5
	10 = 2	010 = 2	110 = 6
	11 = 3	011 = 3	111 = 7
010 = 2	00 = 0	000 = 0	010 = 2
	01 = 1	001 = 1	011 = 3
	10 = 2	100 = 4	110 = 6
	11 = 3	101 = 5	111 = 7
001 = 1	00 = 0	000 = 0	001 = 1
	01 = 1	010 = 2	011 = 3
	10 = 2	100 = 4	101 = 5
	11 = 3	110 = 6	111 = 7

算法的二进制表示

表 6.14　元素是递增（Inc）序列中的一部分还是递减（Dec）序列中的一部分

元素	bf_sz		
	0010 = 2	0100 = 4	1000 = 8
0000 = 0	Inc	Inc	Inc
0001 = 1	Inc	Inc	Inc
0010 = 2	Dec	Inc	Inc
0011 = 3	Dec	Inc	Inc
0100 = 4	Inc	Dec	Inc
0101 = 5	Inc	Dec	Inc
0110 = 6	Dec	Dec	Inc
0111 = 7	Dec	Dec	Inc

按位和为0时递增，否则递减

如2&4=0010&0100增，5&4=0101&0100减

双调排序伪代码（二进制）

```
for (bf_sz = 2; bf_sz <= n; bf_sz = 2*bf_sz)
   for (stage = bf_sz/2; stage > 0; stage = stage/2) {
      which_bit = Which_bit(stage);
      for (th = 0; th < n/2; th++) {
         my_elt1 = Insert_zero(th, which_bit);
         my_elt2 = my_elt1 ^ stage;
         Compare_swap(list, my_elt1, my_elt2, my_elt1 & bf_sz);
      }
   }
```

双调排序的并行实现

```
unsigned th = threadIdx.x;

for (bf_sz = 2; bf_sz <= n; bf_sz = 2*bf_sz) {
   for (stage = bf_sz/2; stage > 0;  stage = stage/2) {

      which_bit = Which_bit(stage);
      my_elt1 = Insert_zero(th, which_bit);
      my_elt2 = my_elt1 ^ stage;
      Compare_swap(list, my_elt1, my_elt2, my_elt1 & bf_sz);
      __syncthreads();
   }
}
```

双调排序的并行实现（更大规模）

```
/* 对包含2* th_per_blk元素的子列表进行排序 */
Pbitonic_start <<<blk_ct, th_per_blk>>> (list, n);

for (bf_sz = 4*th_per_blk; bf_sz <= n; bf_sz = bf_sz/2) {
    for (stage = bf_sz/2; stage >= 2*th_per_blk;
            stage = stage/2) {
        /* 完成蝶式的单个阶段 */
        Pbutterfly_one_stage <<<blk_ct, th_per_blk>>> (list, n,
                bf_sz, stage);
    }
    /* 通过使用“短”子列表来完成当前的蝶式 */

    Pbutterfly_finish <<<blk_ct, th_per_blk>>> (list, n,
            bf_sz);
}
```

如何实现更大规模的双调排序?

性能比较

表 6.18　含有 2 048 个整数的列表的不同排序平均运行时间（ms）

系统	Intel Xeon 4116	Intel Xeon 4116	Nvidia Quadro P5000
时钟 SM 的数量，SP 的数量	2.1GHz	2.1GHz	1.73 GHz 20，1 280
排序种类 块数，线程数 运行时间	qsort 0.116	双调排序 1.19	并行双调排序 1，1 024 0.128

性能比较

表 6.19　含有 2 097 152 个整数的列表上不同排序的平均运行时间

系统	Intel Xeon 4116	Intel Xeon 4116	NVIDIA Quadro P5000
时钟 SM 的数量，SP 的数量	2.1GHz	2.1GHz	1.73GHz 20，1 280
排序种类 块数，线程数 运行时间	qsort 539	双调排序 3 710	并行双调排序 1 024，1 024 13.1

小结(1)

- NVIDIA GPU由流式多处理器（SM）组成. 而一个SM可以有多个控制单元和更多的数据通路（SP）.
- SP和SM在软件上的类比分别是线程和线程块.
- 线程束是线程块中锁步执行的线程子集，现阶段，线程束由32个连续的线程组成

小结(2)

- 每个SM都有一小块快速共享内存，由线程块中的线程共享.
- 全局内存在所有线程之间共享，内存空间大，但速度较慢。

小结(3)

- 核函数是由主机调用，但运行在GPU或设备端的CUDA函数.
- 核函数由特殊标识符“__global__”标识，具有返回类型void.

小结(4)

- 每个SM都有一小块快速共享内存，由线程块中的线程共享.
- 全局内存在所有线程之间共享，内存空间大，但速度较慢.

小结(5)

- 用4个线程块调用核函数，每个线程有256个线程，可以写作.

```
KernelFunction <<<4,256>>>();
```

- 由核函数启动的线程块集合称为网格。
- threadIdx:线程在其线程块中的序号或索引.
- blockDim:线程块的维度、形状或大小.
- blockIdx:块在网格中的序列号或索引.
- gridDim:网格的维度、形状或大小.

小结(6)

- 统一内存访问，提供了一种在主机和设备上自动复制内存的访存方式.

“cudaMallocManaged”

- 内存操作也可以是显式的.

“cudaMalloc”

“cudaMemcpy”

小结(7)

- 当线程束中所有线程在执行相同的语句时，线程将有效地锁步执行.
- 如果一些线程正在执行一条语句，其它线程正在执行另一条语句，那么将发生线程分歧.

小结(8)

- 原子操作保证在一个线程执行该操作的过程中，没有其他线程能中断或干扰.

小结(9)

- 线程束洗牌提供了一种在寄存器层次上的访存方法.

```
unsigned mask = 0xffffffff;
float result = my_val;
for (unsigned diff = warpSize/2; diff > 0; diff = diff/2)
    result += __shfl_down_sync(mask, result, diff);
```

小结(10)

- 共享内存提供了线程束洗牌的替代方法.
- 声明共享内存块时需要静态指定其大小.

```
__global__ void My_kernel(...) {
    ...
    // 元素的数量必须在编译时可用
    __shared__ float sum_arr[32];
    ...
    // 初始化sum_arr [my_lane]
    int my_lane = ...;
    ...
    for (unsigned diff = 16; diff > 0; diff >>= 1) {
        int source = (my_lane + diff) % warpSize;
        sum_arr[my_lane] += sum_arr[source];
    }
    ...
}
```

小结(11)

- 线程束之间是独立的，在必要情况下，可以使用__syncthreads()来使一个线程块中的所有线程同步.
- 必须保证块中所有线程都在调用__syncthreads函数.
- __syncthreads函数只保证一个块的线程同步.

小结(12)

- 共享内存被划分为库.
- 库的设计目的是让一个线程束中的所有线程可以同时执行载入和存储.