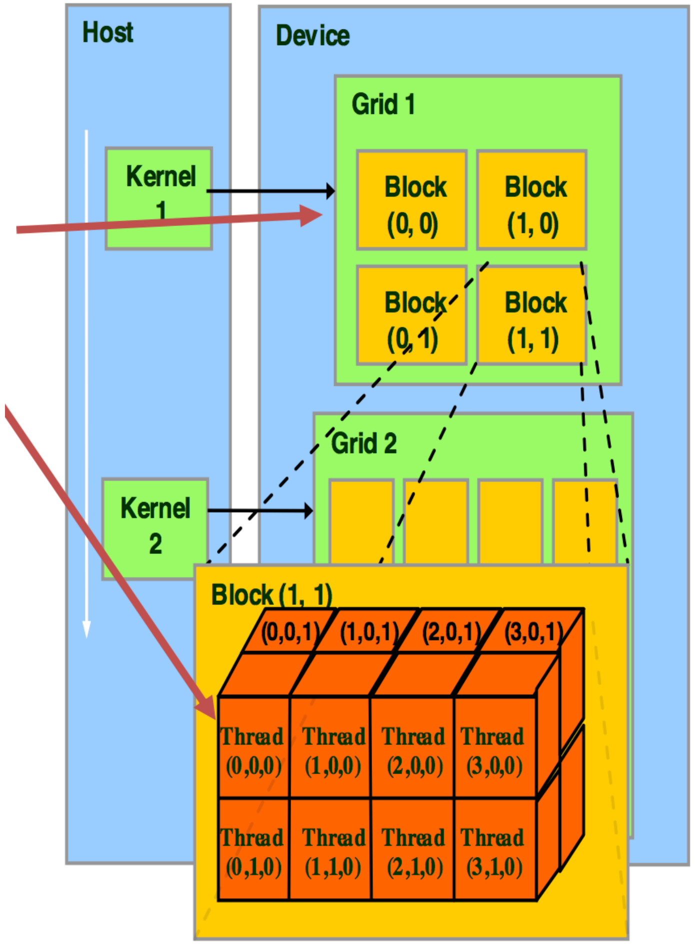
- Block ID: 1D or 2D (or 3D)
- Thread ID: 1D, 2D, or 3D



GPU并行计算与CUDA编程 第3课

本周介绍内容

- 1. CUDA代码的高效策略
  - 1.1 高效公式
  - 1.2 合并全局内存
  - 1.3 避免线程发散
- 2. Kernel加载方式
  - 2.1 查询本机参数
  - 2.2 Kernel加载的1D,2D,3D模式
  - 2.3 Kernel函数的关键字
- 3. CUDA中的各种内存的代码使用
  - 3.1 全局内存
  - 3.2 共享内存
  - 3.3 本地内存
- 4. CUDA同步操作
  - 4.1 原子操作
  - 4.2 同步函数
  - 4.3 CPU/GPU同步
- 5. 并行化高效策略（一）
  - 5.1 归约（实例）
  - 5.2 扫描（实例）

1. CUDA代码的高效策略

1. 高效公式
2. 合并归约
3. 避免线程发散

1.1 高效公式

- 最大化计算强度：

$$\frac{\text{math}}{\text{memory}}$$

数学计算量

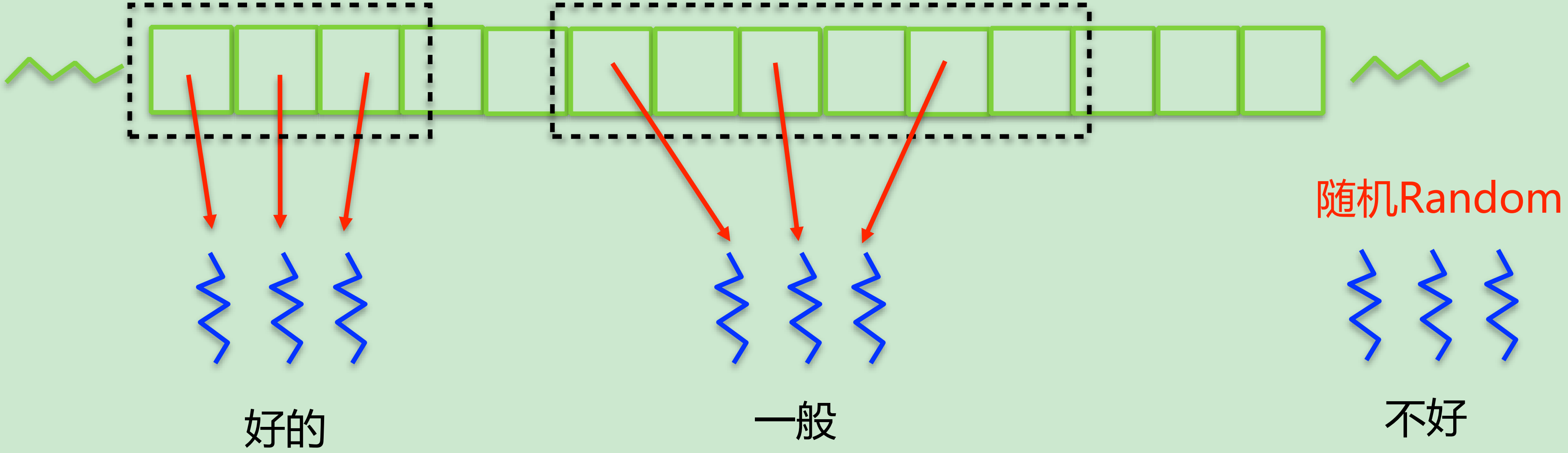
每个线程的内存

- 1. 最大化每个线程的计算量
- 2. 最小化每个线程的内存读取速度
  - \- 每个线程读取的数据量少
  - \- 每个线程读取的速度快

本地内存 ＞ 共享内存 ＞＞ 全局内存

合并全局内存

1.2 合并全局内存




```
__global__ void kernel(float *g){
  float a = 0.00;
  int i = threadIdx.x;
  g[i] = a; // write data
  a = g[i]; // read data
}
```

```
__global__ void kernel(float *g){
  float a = 0.00;
  int i = threadIdx.x;
  g[2*i] = a; // write data
  a = g[2*i]; //?read data
}
```

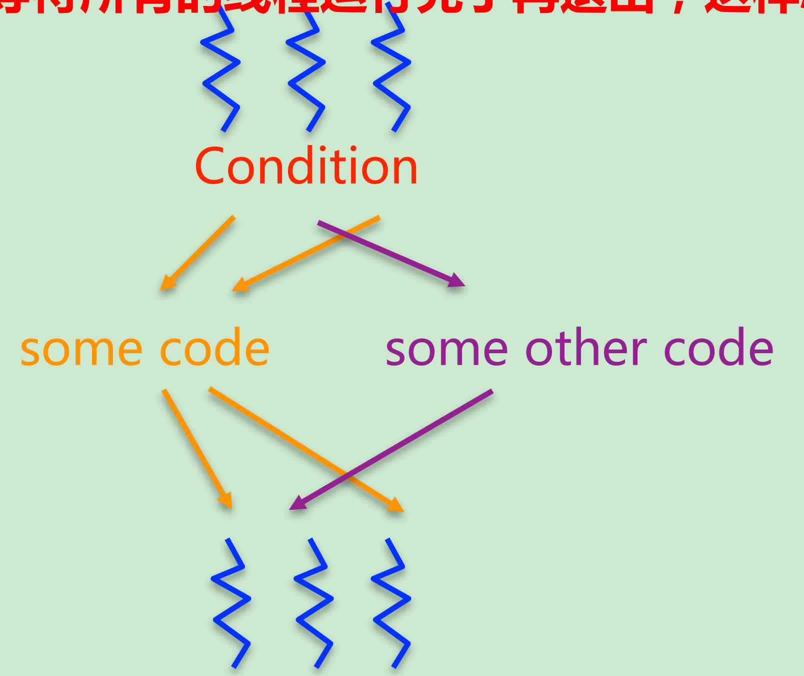
对数组的读取有三种方式:1.按照顺序，按照一定规律，3.随机读取

1.3避免线程发散

- 线程发散：同一个线程块中的线程执行不同内容的代码
- 导致发散的例子：
- 1. kernel中做条件判断

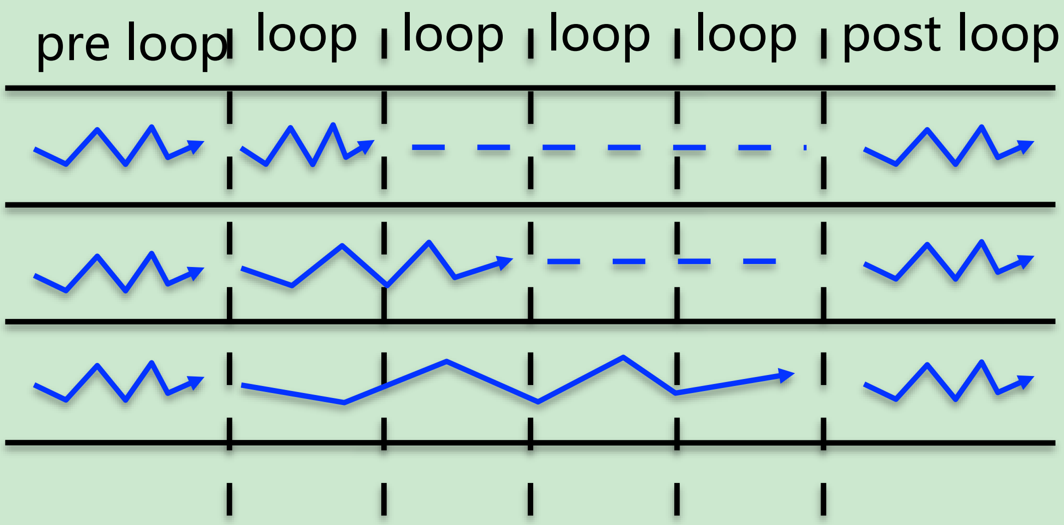
1.是指线程束发散吗？
2.线程块中的线程运行的内容不一样，有可能导致每个线程的运行速度不
，但是线程块会等待所有的线程运行完了再退出，这样就可能导致程序变

```
__global__ void kernel(){
  if(/* condition */){
    // some code
  }else{
    // some other  code
  }
}
```



- 2. 循环长度不一

```
__global__ void kernel(){
  // pre loop code
  for(int i=0;i<threadIdx.x;++i){
    // loop code
  }
  // post loop code
}
```



每个线程运行的速度最好一样

2. Kernel加载方式

1. 查询本机参数

2. Kernel加载的1D,2D,3D模式

2.1 查看本机参数

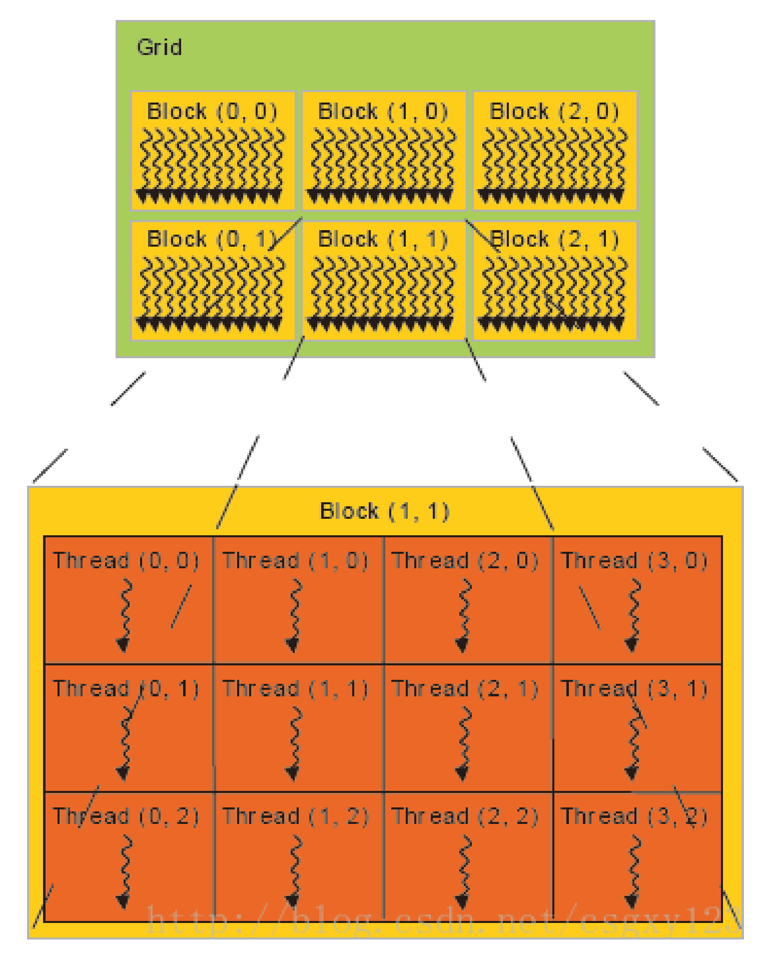
```
luoyun@luoyun-CW65S:~/NVIDIA_CUDA-8.0_Samples/1_Utilities/deviceQuery$ ls
Makefile  NsightEclipse.xml  deviceQuery  deviceQuery.cpp  deviceQuery.o  readme.txt
luoyun@luoyun-CW65S:~/NVIDIA_CUDA-8.0_Samples/1_Utilities/deviceQuery$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 950M"
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 4044 MBytes (4240965632 bytes)
  ( 5) Multiprocessors, (128) CUDA Cores/MP:     640 CUDA Cores
  GPU Max Clock rate:                            928 MHz (0.93 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 8.0, CUDA Runtime Version = 8.0, NumDevs = 1, Device0 = GeForce GTX 950M
Result = PASS
```

了解自己的电脑才能合理地根据情况来写程序。
通过deviceQuery文件来查询。

- 注意事项：
- Kernel的加载中，自定义的线程数，线程块的数量等都不要超过系统本身的设定，否则，会影响机器的效率。

2.2Kernel的加载

- 回顾1：Grid，Block，Thread的关系



- 回顾2：

```
#include <stdio.h>

__global__ void square(float* d_out,float* d_in){
  int idx = threadIdx.x;
  float f = d_in[idx];
  d_out[idx] = f * f;
}

int main(int argc,char** argv){
  const int ARRAY_SIZE = 64;
  const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

  // generate the input array on the host
  float h_in[ARRAY_SIZE];
  for(int i=0;i<ARRAY_SIZE;i++){
    h_in[i] = float(i);
  }
  float h_out[ARRAY_SIZE];

  // declare GPU memory pointers
  float* d_in;
  float* d_out;

  // allocate GPU memory
  cudaMalloc((void**) &d_in,ARRAY_BYTES);
  cudaMalloc((void**) &d_out,ARRAY_BYTES);

  // transfer the array to GPU
  cudaMemcpy(d_in,h_in,ARRAY_BYTES,cudaMemcpyHostToDevice);
```

```
// launch the kernel
square<<<1,ARRAY_SIZE>>>(d_out,d_in);

// copy back the result array to the GPU
cudaMemcpy(h_out,d_out,ARRAY_BYTES,cudaMemcpyDeviceToHost);

// print out the resulting array
for(int i=0;i<ARRAY_SIZE;i++){
  printf("%f",h_out[i]);
  printf(((i%4) != 3) ? "\t" : "\n");
}

// free GPU memory allocation
cudaFree(d_in);
cudaFree(d_out);

return 0;

}
```

Kernel加载——1D模式

- 网格(grid)是**1D**的- 线程块(block)是**1D**

```
int idx = blockIdx.x *blockDim.x +
threadIdx.x;
```

加载方式：

```
Kernel<<<numBlock,threadsPerBlock>>>(argv)
```

- 网格(grid)是**1D**的- 线程块(block)是**2D**

```
int idx = blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x +
threadIdx.x;
```

加载方式：

```
dim3 dimBlock(x,y)
Kernel<<<numBlock,dimBlock>>>(argv)
```

- 网格(grid)是1D的- 线程块(block)是3D

```
int idx = blockIdx.x * blockDim.x * blockDim.y * blockDim.z + threadIdx.z * blockDim.y *
blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
```

加载方式：

```
dim3 dimBlock(x,y,z)
Kernel<<<numBlock,dimBlock>>>(argv)
```

Kernel加载——2D模式

- 网格(grid)是2D的- 线程块(block)是1D

```
int blockId = blockIdx.y * gridDim.x + blockIdx.x;
int Idx = blockId * blockDim.x + threadIdx.x;
```

加载方式：

```
dim3 dimGrid(x,y)；
Kernel<<<dimGrid,threadsPerBlock>>>(argv)；
```

- 网格(grid)是2D的- 线程块(block)是2D

```
int blockId = blockIdx.y * gridDim.x + blockIdx.x;
int Idx = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) +
threadIdx.x;
```

加载方式：

```
dim3 dimGrid(x1,y1),dimBlock(x2,y2)；
Kernel<<<dimGrid,dimBlock>>>(argv)；
```

- 网格(grid)是**2D**的- 线程块(block)是**3D**

```
int blockId = blockIdx.y * gridDim.x + blockIdx.x;
int Idx = blockId * (blockDim.x * blockDim.y * blockDim.z) + (threadIdx.z * (blockDim.x *
blockDim.y)) + (threadIdx.y * blockDim.x)+ threadIdx.x;
```

加载方式：

```
dim3 dimGrid(x1,y1),dimBlock(x2,y2,z2)；
Kernel<<<dimGrid,dimBlock>>>(argv)；
```

Kernel加载——3D模式

- 网格(grid)是3D的- 线程块(block)是1D

```
int blockId = blockIdx.x+ blockIdx.y * gridDim.x+ gridDim.x * gridDim.y *
blockIdx.z;
int Idx = blockId * blockDim.x + threadIdx.x;
```

加载方式：

```
dim3 dimGrid(x,y,z)；
Kernel<<<dimGrid,threadsPerBlock>>>(argv)；
```

- 网格(grid)是3D的- 线程块(block)是2D

```
int blockId = blockIdx.x+ blockIdx.y * gridDim.x + gridDim.x * gridDim.y *
blockIdx.z;
int Idx = blockId * (blockDim.x * blockDim.y)+ (threadIdx.y * blockDim.x) +
threadIdx.x;
```

加载方式：

```
dim3 dimGrid(x1,y1,z1),dimBlock(x2,y2)；
Kernel<<<dimGrid,dimBlock>>>(argv)；
```

- 网格(grid)是3D的- 线程块(block)是3D

```
int blockId = blockIdx.x+ blockIdx.y * gridDim.x+ gridDim.x * gridDim.y * blockIdx.z;
int Idx = blockId * (blockDim.x * blockDim.y * blockDim.z) + (threadIdx.z * (blockDim.x *
blockDim.y)) + (threadIdx.y * blockDim.x)+ threadIdx.x;
```

加载方式：

```
dim3 dimGrid(x1,y1),dimBlock(x2,y2,z2)；
Kernel<<<dimGrid,dimBlock>>>(argv)；
```

2.3 Kernel 函数关键字

	执行设备	可被调用的设备
__device__ float DeviceFunc()	device	device(只能被__device__和__global__调用)
__global__ void KernelFunc()	device	host（只能被主函数和CPU上运行函数调用）
__host__ float HostFunc()	host	host

注：__global__ 只能返回void

注意：__host__ 是可以省略的。

3.CUDA中的各种内存的代码使用

1. 全局内存
2. 共享内存
3. 本地内存

3.1 本地变量

英文叫做local memory

```
/**********************
 * using local memory *
 **********************/

// a __device__ or __global__ function runs on the GPU
__global__ void use_local_memory_GPU(float in)
{
    float f;    // variable "f" is in local memory and private to each thread
    f = in;     // parameter "in" is in local memory and private to each thread
    // ... real code would presumably do other stuff here ...
}
```

```
/*
 * First, call a kernel that shows using local memory
 */
use_local_memory_GPU<<<1, 128>>>(2.0f);
```

3.2 全局变量


```
/**********************
 * using global memory *
 **********************/

// a __global__ function runs on the GPU & can be called from host
__global__ void use_global_memory_GPU(float *array)
{
    // "array" is a pointer into global memory on the device
    array[threadIdx.x] = 2.0f * (float) threadIdx.x;
}
```

```
 /*
  * Next, call a kernel that shows using global memory
  */
 float h_arr[128];   // convention: h_ variables live on host
 float *d_arr;       // convention: d_ variables live on device (GPU global mem)

 // allocate global memory on the device, place result in "d_arr"
 cudaMalloc((void **) &d_arr, sizeof(float) * 128);
 // now copy data from host memory "h_arr" to device memory "d_arr"
 cudaMemcpy((void *)d_arr, (void *)h_arr, sizeof(float) * 128, cudaMemcpyHostToDevice);
 // launch the kernel (1 block of 128 threads)
 use_global_memory_GPU<<<1, 128>>>(d_arr);  // modifies the contents of array at d_arr
 // copy the modified array back to the host, overwriting contents of h_arr
 cudaMemcpy((void *)h_arr, (void *)d_arr, sizeof(float) * 128, cudaMemcpyDeviceToHost);
 // ... do other stuff ...
```

global memory：就是在host定义，在device中使用的。

3.3 共享变量

```
/**********************
 * using shared memory *
 **********************/

// (for clarity, hardcoding 128 threads/elements and omitting out-of-bounds checks)
__global__ void use_shared_memory_GPU(float *array)
{
    // local variables, private to each thread
    int i, index = threadIdx.x;
    float average, sum = 0.0f;

    // __shared__ variables are visible to all threads in the thread block
    // and have the same lifetime as the thread block
    __shared__ float sh_arr[128];

    // copy data from "array" in global memory to sh_arr in shared memory.
    // here, each thread is responsible for copying a single element.
    sh_arr[index] = array[index];

    __syncthreads();    // ensure all the writes to shared memory have completed

    // now, sh_arr is fully populated. Let's find the average of all previous elements
    for (i=0; i<index; i++) { sum += sh_arr[i]; }
    average = sum / (index + 1.0f);

    // if array[index] is greater than the average of array[0..index-1], replace with average.
    // since array[] is in global memory, this change will be seen by the host (and potentially
    // other thread blocks, if any)
    if (array[index] > average) { array[index] = average; }

    // the following code has NO EFFECT: it modifies shared memory, but
    // the resulting modified data is never copied back to global memory
    // and vanishes when the thread block completes
    sh_arr[index] = 3.14;
}
```

shared global variable：线程块中的每个线程都会往这个变量中写数据。

```
/*
 * Next, call a kernel that shows using shared memory
 */

// as before, pass in a pointer to data in global memory
use_shared_memory_GPU<<<1, 128>>>(d_arr);
// copy the modified array back to the host
cudaMemcpy((void *)h_arr, (void *)d_arr, sizeof(float) * 128, cudaMemcpyHostToDevice);
// ... do other stuff ...
```

4. CUDA同步操作

1. 原子操作
2. 同步函数
3. CPU/GPU 同步

4.1 原子操作

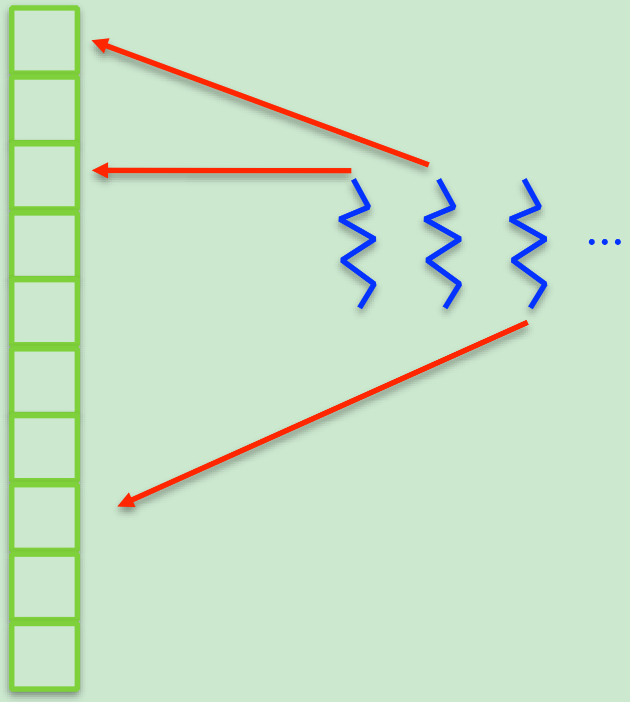
- 原子操作解决的问题：
- 对于有很多线程需要同时读取或写入相同的内存时，保证同一时间只有一个线程能进行操作。

我是一个并行的，但是现在变成串行的。这时候就需要院子操作！

- 原子操作的
- 1. 只支持某些运算(加、减、最小值、异或运算等，不支持求余和求幂等)和数据类型（整型）
- 2. 运行顺序不定 **谁先谁后是不确定的！**
- 3. 安排不当，会使速度很慢（因为内部是个串行的运行）

- 代码案例讲解：**atomics.cu**
- 让10000个线程增加10个数组元素(如果直接相加会出错)
- 采用原子内存操作来解决atomicAdd()

```
__global__ void increment_naive(int *g)
{
        // which thread is this?
        int i = blockIdx.x * blockDim.x + threadIdx.x;

        // each thread to increment consecutive elements, wrapping at ARRAY_SIZE
        i = i % ARRAY_SIZE;
        g[i] = g[i] + 1;
}

__global__ void increment_atomic(int *g)
{
        // which thread is this?
        int i = blockIdx.x * blockDim.x + threadIdx.x;

        // each thread to increment consecutive elements, wrapping at ARRAY_SIZE
        i = i % ARRAY_SIZE;
        atomicAdd(& g[i], 1);
}
```



同时进行了读取和写入操作！
两种方法解决：
1.屏障？
2.原子操作！

4.2 同步函数

- _syncthreads（ ）
- 线程块内线程同步
- 保证线程块内所有线程都执行到统一位置

注意是在线程块内的同步！

```
/**********************
 * using shared memory *
 **********************/

// (for clarity, hardcoding 128 threads/elements and omitting out-of-bounds checks)
__global__ void use_shared_memory_GPU(float *array)
{
    // local variables, private to each thread
    int i, index = threadIdx.x;
    float average, sum = 0.0f;

    // __shared__ variables are visible to all threads in the thread block
    // and have the same lifetime as the thread block
    __shared__ float sh_arr[128];

    // copy data from "array" in global memory to sh_arr in shared memory.
    // here, each thread is responsible for copying a single element.
    sh_arr[index] = array[index];

    __syncthreads();    // ensure all the writes to shared memory have completed

    // now, sh_arr is fully populated. Let's find the average of all previous elements
    for (i=0; i<index; i++) { sum += sh_arr[i]; }
    average = sum / (index + 1.0f);

    // if array[index] is greater than the average of array[0..index-1], replace with average.
    // since array[] is in global memory, this change will be seen by the host (and potentially
    // other thread blocks, if any)
    if (array[index] > average) { array[index] = average; }

    // the following code has NO EFFECT: it modifies shared memory, but
    // the resulting modified data is never copied back to global memory
    // and vanishes when the thread block completes
    sh_arr[index] = 3.14;
}
```

- **_threadfence()**
- 一个线程调用__threadfence后，该线程在该语句前对全局存储器或共享存储器的访问已经全部完成，执行结果对grid中的所有线程可见。

- **_threadfence_block()**
- 一个线程调用__threadfence_block后，该线程在该语句前对全局存储器或者共享存储器的访问已经全部完成，执行结果对block中的所有线程可见。

- **以上两个函数的重要作用是，及时通知其他线程，全局内存或者共享内存内的结果已经读入或写入完成了。**

这两个用的很少

4.3 CPU/GPU同步

- **cudaStreamSynchronize()/cudaEventSynchronize()**
- 主机端代码中使用cudaThreadSynchronize():实现CPU和GPU线程同步
- kernel启动后控制权将异步返回，利用该函数可以确定所有设备端线程均已运行结束；

5. 并行化高效策略（一）

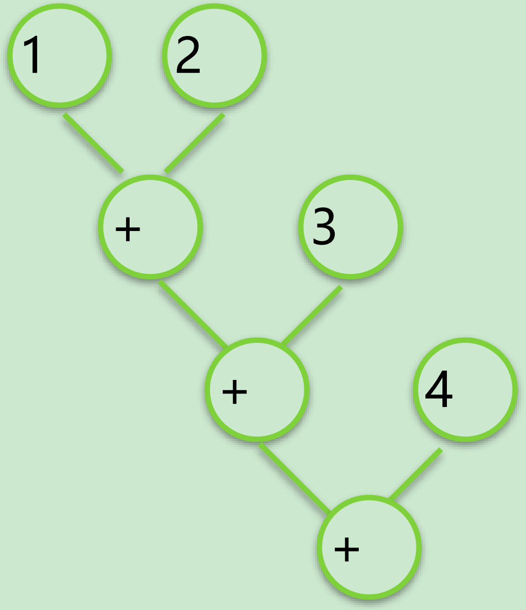
1. 归约Reduce（实例）
2. 扫描Scan（实例）

5.1 归约Reduce

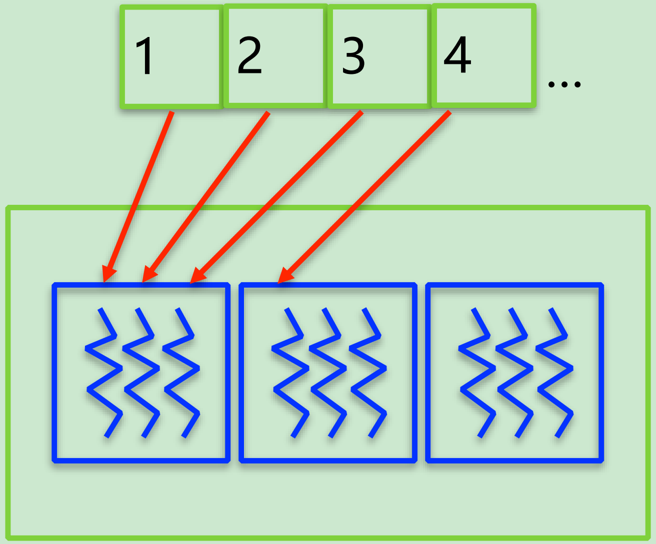
- 实例：做求和：1+2+3+4+···

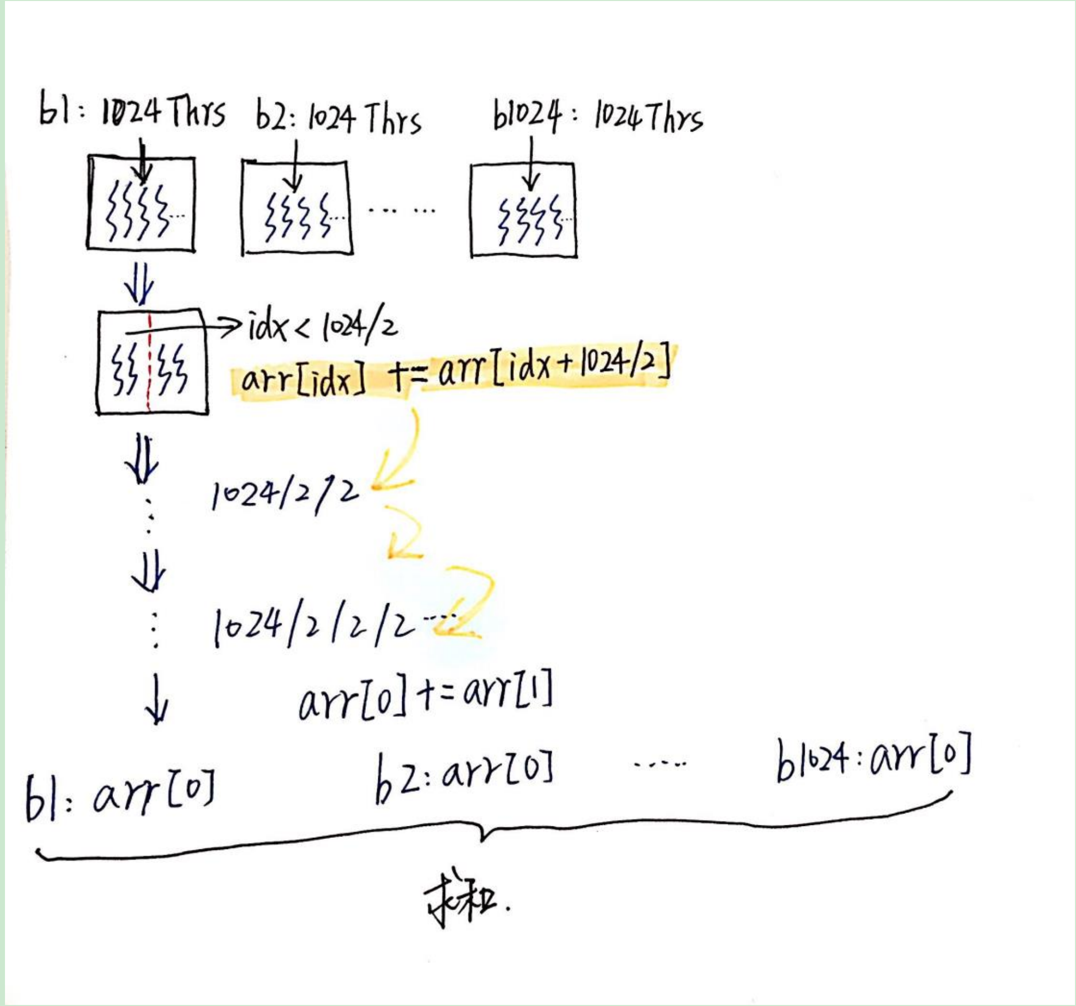
这个实际上就是将串行程序改为并行程序！

- 串行的做法：



- 并行化做法：


1
2
3
4
...


b1: 1024 Thrs
b2: 1024 Thrs
b1024: 1024 Thrs
idx < 1024/2
arr[idx] += arr[idx+1024/2]
1024/2/2
1024/2/2/2...
arr[0] += arr[1]
b1: arr[0]
b2: arr[0]
b1024: arr[0]
求和.

- 使用global memory

```
__global__ void global_reduce_kernel(float * d_out, float * d_in)
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int tid  = threadIdx.x;

    // do reduction in global mem
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
    {
        if (tid < s)
        {
            d_in[myId] += d_in[myId + s];
        }
        __syncthreads();        // make sure all adds at one stage are done!
    }

    // only thread 0 writes result for this block back to global mem
    if (tid == 0)
    {
        d_out[blockIdx.x] = d_in[myId];
    }
}
```

- 使用shared memory

```
__global__ void shmem_reduce_kernel(float * d_out, const float * d_in)
{
    // sdata is allocated in the kernel call: 3rd arg to <<<b, t, shmem>>>
    extern __shared__ float sdata[];

    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int tid  = threadIdx.x;

    // load shared mem from global mem
    sdata[tid] = d_in[myId];
    __syncthreads();            // make sure entire block is loaded!

    // do reduction in shared mem
    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
    {
        if (tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();        // make sure all adds at one stage are done!
    }

    // only thread 0 writes result for this block back to global mem
    if (tid == 0)
    {
        d_out[blockIdx.x] = sdata[0];
    }
}
```

5.2 扫描Scan

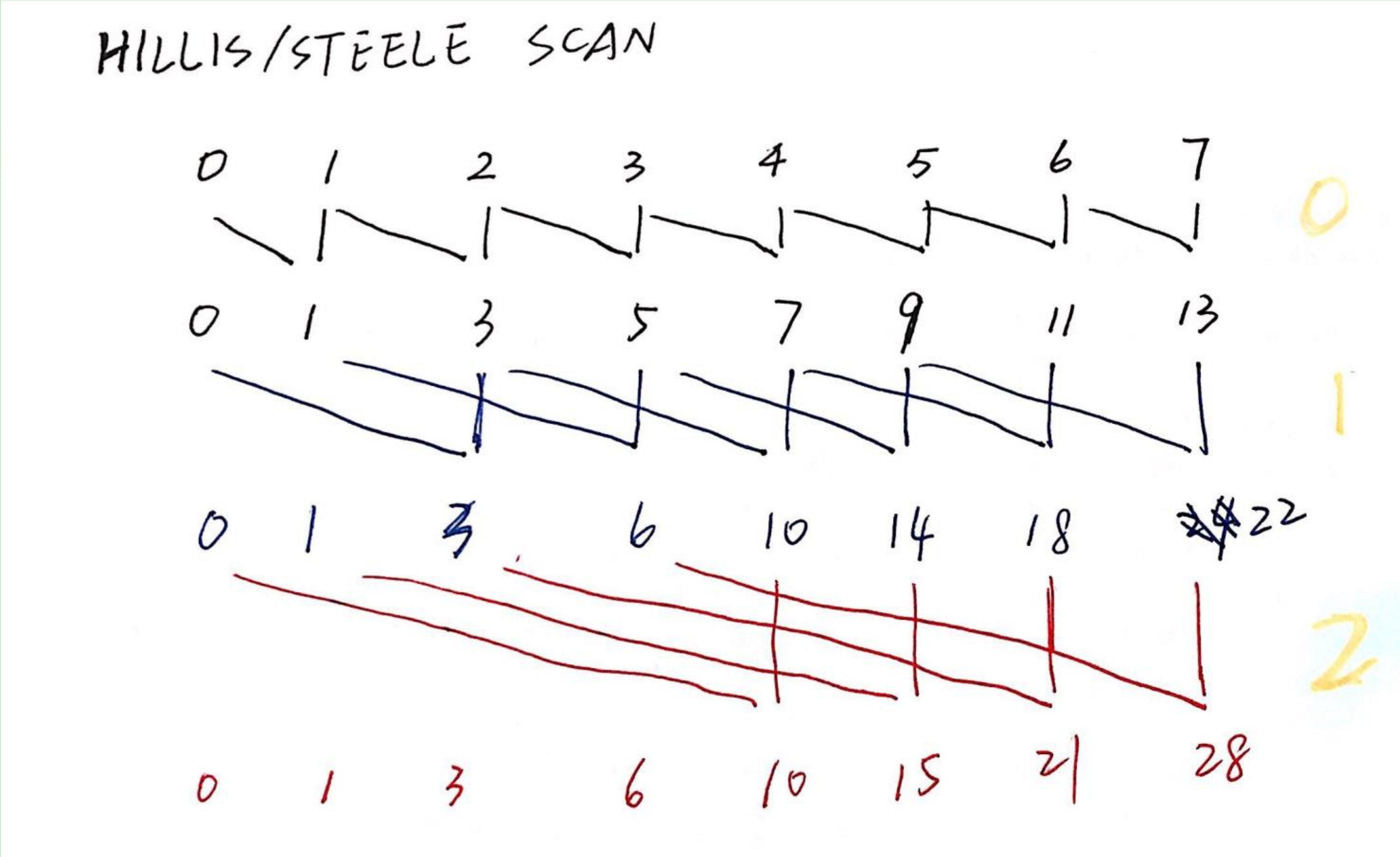
- 实例：

1	2	3	4

ADD

1	3	6	10

- 并行化算法思路：（Hillis Steele Scan）



```
__global__ void global_scan(float* d_out,float* d_in){
  int idx = threadIdx.x;
  float out = 0.00f;
  d_out[idx] = d_in[idx];
  __syncthreads();
  for(int interpre=1;interpre<sizeof(d_in);interpre*=2){
    if(idx-interpre>=0){
      out = d_out[idx]+d_out[idx-interpre];
    }
    __syncthreads();
    if(idx-interpre>=0){
      d_out[idx] = out;
      out = 0.00f;
    }
  }
}
```

本周作业

- 把Hillis Steele Scan算法使用共享内存实现, 在homework_scan.cu中实现，并运行成功，上传代码与结果截图。

```
__global__ void global_scan(float* d_out,float* d_in){
  int idx = threadIdx.x;
  float out = 0.00f;
  d_out = d_in;
  for(int interpre=1;interpre<sizeof(d_in);interpre*=2){
    if(idx-interpre>=0){
      out = d_out[idx]+d_out[idx-interpre];
    }
    __syncthreads();
    if(idx-interpre>=0){
      d_out[idx] = out;
      out = 0.00f;
    }
  }
}

//TODO:[homework] use shared memory to complete the scan algorithm.
//![Notice]remember to modify the kernel loading.
__global__ void shmem_scan(float* d_out,float* d_in){

}
```

法律声明

【声明】本视频和幻灯片为炼数成金网络课程的教学资料，所有资料只能在课程内使用，不得在课程以外范围散播，违者将可能被追究法律和经济责任。

炼数成金逆向收费式网络课程

- Dataguru（炼数成金）是专业数据分析网站，提供教育，媒体，内容，社区，出版，数据分析业务等服务。我们的课程采用新兴的互联网教育形式，独创地发展了逆向收费式网络培训课程模式。既继承传统教育重学习氛围，重竞争压力的特点，同时又发挥互联网的威力打破时空限制，把天南地北志同道合的朋友组织在一起交流学习，使到原先孤立的学习个体组合成有组织的探索力量。并且把原先动辄成千上万的学习成本，直线下降至百元范围，造福大众。我们的目标是：低成本传播高价值知识，构架中国第一的网上知识流转阵地。
- 关于逆向收费式网络的详情，请看我们的培训网站 http://edu.dataguru.cn

Thanks

FAQ时间