

chap2 并行硬件和软件

Basics of caching

2.2.1

缓存：访问速度更快的内存地址集合

CPU缓存：与CPU位于同一块芯片，或者位于其他芯片达能更快地访问

Cache mappings

2.2.2

规定：从主存中取出一行后应该放在缓存中的哪个位置

- **Full associative 全相联**：任意位置
- **Direct mapped 直接映射**：每行对应一个唯一的缓存位置
- **n-way set associative n路组相联**：每行可以放在n个缓存位置的其一

当有多个可以放入的位置（直接映射和n路组相联），还需要决策替换其中的哪一个

16行主存和4行缓存之间的分配示例：

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Table 2.1: Assignments of a 16-line main memory to a 4-line cache

Caches and programs

2.2.3

第一个嵌套循环比第二个的性能更好：

```

double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j]*x[j];

```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

c语言以行主序在主存中存储二维数组。

当期望访问数组的某一元素但不在缓存中时，就会从主存中按顺序取出缓存可容纳的一行进行替换。

如果不按行读取数组，而是按列读取数组，可能导致很快就会命中失败，需要再一次访问主存。

如果对数组按行读取可以增加命中率，减少读取主存的次数，从而性能更好。

Instruction Level Parallelism

2.2.5

让多个处理器部件或功能单元同时执行指令，以此来提高处理器的性能

两种实现指令级并行的方法：

- **Pipelining 流水线**：按阶段安排功能单元，上一阶段的功能单元输出是下一个单元的输入
- **Multiple issue 多发射**：复制出多个功能单元，每个单元同时执行不同的指令，没有依赖关系的指令才能同时执行

流水线例子：计算 $9.87 \times 10^4 + 6.54 \times 10^3$

涉及到下面7个计算步骤：取数、比较指数、指数移位对齐、相加、结果规格化、结果舍入、结果存储

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

假设每个operation需要1ns，需要循环1000次浮点数相加，最终需要7000ns

把以上的7个步骤分配到7个独立的硬件或功能单元，上一个单元的输出作为下一个单元的输入

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

时间5之后，流水线每个时间可以产生一个结果，而不是上图中7个时间产生一个结果

尽管完整的一次浮点数相加仍需要7ns，但是加入流水线后1000次相加只用1006ns

2.1 当讨论浮点数加法时，我们简单地假设每个功能单元都花费相同的时间。如果每个取命令与存命令都耗费2纳秒，其余的每个操作耗费1纳秒。

- 在上述假设下，每个浮点数加法要耗费多少时间？
- 非流水线 1000 对浮点数的加法要耗费多少时间？
- 流水线 1000 对浮点数加法要耗费多少时间？
- 如果操作数/结果存储在不同级的内存层级上，那么取命令与存命令所要耗费的时间可能会差别非常大。假设从一级缓存上取数据/指令要耗费2纳秒，从二级缓存上取数据/指令要耗费5纳秒，从主存取数据/指令要耗费50纳秒。当执行某条指令，取其中一个操作数时，发生了一次一级缓存失效，那么流水线会发生什么情况？如果又发生二级缓存失效，又会怎样？

a. 9ns

b. $9 \times 1000 = 9000\text{ns}$

c.

Time	F	C	Sh	A	N	R	St
0	1						
1	1						
2	2	1					
3	2		1				
4	3	2		1			
5	3		2		1		
6	4	3		2		1	
7	4		3		2		1
8	5	4		3		2	1
9	5		4		3		2
10	6	5		4		3	2
11	6		5		4		3

$1000 \times 2 + 7 = 2007\text{ns}$

d. 一级缓存取操作数失败 -> 取数时间变为 $2+5=7\text{ns}$

二级缓存取操作数失败 -> 取数时间变为 $2+5+50=57\text{ns}$

取数时间如果延长，流水线上后续相加任务需要进行等待，延长的时间会使得最终的流水线完成时间也会有相应的延长

Flynn's Taxonomy

2.3.1

Flynn分类法：按照计算机可以同时管理的指令流和数据流数目，对系统进行分类

- SISD：一次执行一条指令、存取一个数据项；冯诺依曼系统
- SIMD：单指令流 多数据流；把数据分散到不同的处理器，对它们执行相同的指令
- MISD：多指令流 单数据流
- MIMD：多指令流 多数据流；有一组独立的处理单元，每个处理单元都有自己的控制单元和ALU

<i>classic von Neumann</i> SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream <i>not covered</i>	(MIMD) Multiple instruction stream Multiple data stream

2.5 在冯·诺依曼系统中加入缓存和虚拟内存改变了它作为 SISD 系统的类型吗？如果加入流水线呢？多发射或硬件多线程呢？

硬件多线程：一种并行机制，确保当前线程被阻塞时，系统还可以继续运行其它线程

加入缓存和虚拟内存没有改变它的类型，因为这只引入了硬件上的改变，目的是为了提高系统性能和内存管理效率，但是没有改变指令流和控制流，所以仍然是 SISD 类型

如果流水线中同一时刻只存在一种指令，则是 SIMD 系统；如果流水线中同一时刻会对多个数据项执行不同的指令，则是 MIMD 系统 **???这里有疑问**

多发射是复制出多个功能单元，每个单元同时执行不同的指令，没有依赖关系的指令才能同时执行。每个处理单元都是独立的，指令流和数据流都不同，所以是 MIMD 系统

硬件多线程是一种并行机制，确保当前线程被阻塞时，系统还可以继续运行其它线程。这允许多个线程同时执行多个指令流，如果这些线程处理不同的数据流，则是 MIMD 系统

Cache coherence

2.3.5 缓存一致性

程序员对缓存什么时候更新没有直接控制，所以在多核系统中，如果不同处理器的缓存中存储了共享变量的副本，则一个处理器更新该变量时，其它的处理器也应该进行相应的更新

两种保证一致性的策略：

- **Snooping Cache Coherence 监听一致性**
- **Directory Based Cache Coherence 基于目录的一致性**

监听一致性

- 多个核共享一个总线，总线上传递的信号可以被其它核看到
- 某一个核更新它缓存中的变量副本时，会通过总线广播这一更新信息
- 如果其它核在监听总线，就可以得知该变量已被更新，并对自己缓存中的同一变量标记为非法
- 广播是针对所有核的，频繁广播会使得系统性能下降

基于目录的一致性

- 使用一个“目录”数据结构来存储每个内存行的状态
- 当一个变量被更新，查询目录，将包含该变量的内存行状态标记为非法
- 需要额外的存储空间，但变量更新时只需要和存储了这个变量核交流即可

Speedup of a parallel program

2.6.1

加速比S = 线性执行时间 / 并行执行时间

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

效率E = 加速比S / 处理器数量p

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}} \right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}$$

阿姆达尔定律：**加速比有上限**

假设：

1. 只能对串行程序的90%实施并行
2. 可并行部分加速比理想化为p，随进程数线性增加
3. $T_{\text{serial}} = 20s$

可并行部分的并行执行时间为：

$$T'_{\text{parallel}} = \frac{T'_{\text{serial}}}{S} = \frac{0.9 \times T_{\text{serial}}}{p} = \frac{18}{p}$$

不可并行部分的串行执行时间为：

$$(1 - 0.9) \times T_{\text{serial}} = 2$$

并行版本的总执行时间为：

$$T_{\text{parallel}} = \frac{18}{p} + 2$$

真实加速比为：

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{20}{\frac{18}{p} + 2}$$

随着p不断增加，加速比会趋近于10，但是不会超过10

超线性 Superlinear：加速比S > 处理器个数p

示例

- 单核程序cache有限，多核程序可以将所有数据放入cache（加入多核程序处理同一个数据）
- 并行DFS可以快速搜索目标结果

chap3 MPI

MPI reduce

3.4.2

在全局上进行operator指定的运算操作

```
int MPI_Reduce( 代替求和程序
    void*      input_data_p    /* in */, 当前局部值指针
    void*      output_data_p   /* out */, 最终值存储指针
    int        count           /* in */,
    MPI_Datatype datatype      /* in */,
    MPI_Op      operator       /* in */, 操作
    int        dest_process    /* in */, 拿到结果的进程
    MPI_Comm    comm           /* in */);
```

Collective vs. Point-to-Point Communications

3.4.3 集合通信与点对点通信

集合通信：涉及通讯子中所有进程的通信

点对点通信：MPI_Send 和 Rec

- 通讯子中的所有进程需要调用同一个集合通信函数
 - 每个进程传递给集合通信函数的参数需要相容
 - dest_process 指定的进程才会使用 output_data_p 参数，但其它进程依旧需要往里面传入一个相应参数
 - 点对点通信通过标签tag和通讯子来匹配
- 集合通讯只通过通讯子和调用的顺序来匹配

Multiple calls to MPI_Reduce

假设3个进程都调用了MPI_Reduce来进行求和，并将目标进程设置为0号

Table 3.3 Multiple Calls to MPI_Reduce. (默认dest = 0)

Time	Process 0	Process 1	Process 2
0	a = 1; c = 2	a = 1; c = 2	a = 1; c = 2
1	MPI_Reduce(&a, &b, ...)	MPI_Reduce(&c, &d, ...)	MPI_Reduce(&a, &b, ...)
2	MPI_Reduce(&c, &d, ...)	MPI_Reduce(&a, &b, ...)	MPI_Reduce(&c, &d, ...)

b = 1 + 2 + 1 = 4

d = 2 + 1 + 2 = 5

Partitioning options

3.4.6 数据分发

分发向量/数组数据到不同进程时，需要对划分方法进行决策

- 块划分 Block partitioning：为每个进程分发连续的数据分量
- 循环划分 Cyclic partitioning：挨个给进程分发分量，一次发一个
- 块-循环划分 Block-cyclic partitioning：挨个给进程分发分量，一次发指定个数

Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

示例中进程数=3，等待分发的分量个数=12

chap4 pthread

共享内存的并行编程

Matrix-Vector Multiplication in pthreads

4.3 矩阵向量乘

a_{00}	a_{01}	\cdots	$a_{0,n-1}$
a_{10}	a_{11}	\cdots	$a_{1,n-1}$
\vdots	\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$
\vdots	\vdots		\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$

x_0
x_1
\vdots
x_{n-1}

 $=$

y_0
y_1
\vdots
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
\vdots
y_{m-1}

矩阵 $A \times$ 向量 $x =$ 向量 y

将目标向量 y 的计算，分发给不同的线程，通常分块划分 [\(见上\)](#)，每个线程平分 y 的分量进行计算

- 4.1 当讨论矩阵-向量乘法时，我们通常假设 m 和 n ，即矩阵的行数和列数，都能够被 t 整除， t 是线程的个数。但是，如果 m 和 n 不满足能被 t 整除的条件，那么用什么公式来分配数据？

```

1  # n为等待分发的分量个数，p为线程数目
2
3  # 计算商：每个线程计算的分量数目
4  quotient = n/p
5  # 计算余数：被剩下的分量数目
6  remainder = n%p
7
8  # 计算第一个分量下标 first_i
9  # 将剩余的分量分给标号小的线程
10 if rank < remainder:
11     # 这些线程比别人多算1个分量
12     n_count = quotient + 1
13     # 由于线程标号从0开始，所以前面有rank个线程计算
14     first_i = rank * n_count
15 else:
16     # 其余线程计算原定分量数目
17     n_count = quotient
18     # 多余的内容已被前面的线程算过，所以要加上余数
19     first_i = rank * n_count + remainder
20
21 # 计算最后一个分量下标 last_i
22 # 不减1的话，结果是下一个线程的第一个分量下标

```

chap5 openMp

共享内存的并行编程

_OPENMP

5.1.3

- 5.1 如果已经定义了宏 `_OPENMP`，它是一个 `int` 类型的十进制数。编写一个程序打印它的值。这个值的意义是什么？

```

1  #ifdef _OPENMP
2  #   include <omp.h>
3  #endif
4
5  int main(int argc, char* argv[]) {
6  # ifdef _OPENMP
7      printf("_OPENMP = %d\n", _OPENMP);
8  # else
9      printf("OPENMP not defined.");
10 # endif
11     return 0;
12 }
```

`_OPENMP` 可以用于检查，只有编译器支持openmp时，其值才有意义，代表的是OPENMP标准的发布时间 201511

消除数据依赖

5.5.2

在循环中，计算可能依赖于先前循环的计算结果，但是openmp并不会检查依赖，这会造成错误

类似于“找规律”，让循环中的计算只依赖于迭代量 `i`，并得到一个公式

5.8 考虑下面的循环

```

a[0] = 0;
for (i = 1; i < n; i++)
    a[i] = a[i-1] + i;
```

在这个循环中显然有循环依赖，因为在计算 `a[i]` 前必须先算 `a[i-1]` 的值。请你找到一种方法消除循环依赖，并且并行化这个循环。

```

1  # pragma omp parallel for num_threads(thread_count) \
2      default(none) private(i) shared(a, n)
3  for (i=0;i<n;i++)
4      a[i] = i*(i+1)/2
```

设定了 `default(none)`：我们要显示声明变量的作用域

`private()`：其中包含循环中进程的私有变量

`shared()`：其中包含循环中进程之间共享的变量

奇偶排序

奇偶排序：将排序分为两种阶段——奇阶段和偶阶段

- 奇阶段：下标为计数的分量与右方分量比较，小的保留在左边
- 偶阶段：下标为偶数的分量与右方分量比较，小的保留在左边

5.6.2

简单并行奇偶排序：分别在奇阶段和偶阶段内部使用pragma for从句

```
1   for (phase = 0; phase < n; phase++) {
2       if (phase % 2 == 0) 偶阶段
3       # pragma omp parallel for num_threads(thread_count) \
4           default(none) shared(a, n) private(i, tmp)
5           for (i = 1; i < n; i += 2) {
6               if (a[i-1] > a[i]) {
7                   tmp = a[i-1];
8                   a[i-1] = a[i];
9                   a[i] = tmp;
10            }
11        }
12    else 奇阶段
13    # pragma omp parallel for num_threads(thread_count) \
14        default(none) shared(a, n) private(i, tmp)
15        for (i = 1; i < n-1; i += 2) {
16            if (a[i] > a[i+1]) {
17                tmp = a[i+1];
18                a[i+1] = a[i];
19                a[i] = tmp;
20            }
21        }
22    }
```

Program 5.4: First OpenMP implementation of odd-even sort.

简单版存在的问题：

- 每一个phase中，需要等待所有线程都执行结束，才能进入下一个phase
- 创建和结束线程的开销：在每一次外层循环开始或结束时，都会创建或合并pthread

改进：

```

1 # pragma omp parallel num_threads(thread_count) \
2   default(none) shared(a, n) private(i, tmp, phase)
3   for (phase = 0; phase < n; phase++) {
4     if (phase % 2 == 0)
5     # pragma omp for
6       for (i = 1; i < n; i += 2) {
7         if (a[i-1] > a[i]) {
8           tmp = a[i-1];
9           a[i-1] = a[i];
10          a[i] = tmp;
11        }
12      }
13    else
14    # pragma omp for
15      for (i = 1; i < n-1; i += 2) {
16        if (a[i] > a[i+1]) {
17          tmp = a[i+1];
18          a[i+1] = a[i];
19          a[i] = tmp;
20        }
21      }
22  }

```

Program 5.5: Second OpenMP implementation of odd-even sort.

- 在进入外层循环前使用 `parallel` 指令，创建好一些pthread
- 在进入内层循环前使用 `for` 指令，让程序只使用已经创建好的pthread
- 第一个版本中，`parallel for` 指令则会每次都创建新的

Cache伪共享

5.9

线程之间没有共享任何变量，共享了同一个缓存行，一个线程修改了此行中的变量，会使得整个缓存行在其它线程中标记为非法，其它线程就只能前往主存再次获取变量，这使得它们访问主存的行为看起来像是在共享变量

矩阵向量乘法中的Cache问题

$Ax=y$

a_{00}	a_{01}	\cdots	$a_{0,n-1}$
a_{10}	a_{11}	\cdots	$a_{1,n-1}$
\vdots	\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$
\vdots	\vdots		\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$

$$\begin{matrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{matrix} = \begin{matrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{matrix}$$

```

1 # pragma omp parallel for num_threads(thread_count) \
2   default(none) private(i, j) shared(A, x, y, m, n)
3   for (i = 0; i < m; i++) {
4     y[i] = 0.0;
5     for (j = 0; j < n; j++)
6       y[i] += A[i][j]*x[j];
7   }

```

假设我们要计算的y的分量很少，可以在一个缓冲行放下；每个线程对该行中的某一个分量进行修改，虽然其它线程不读取这个分量，但是这一行的缓冲内容都会失效，其它线程就需要去主存中读取自己对应的分量

修改上面的矩阵乘法，使得每个线程私有存储自己处理的分量，计算完成后将其更新到共享变量中，这样就减少了修改共享变量的次数

```

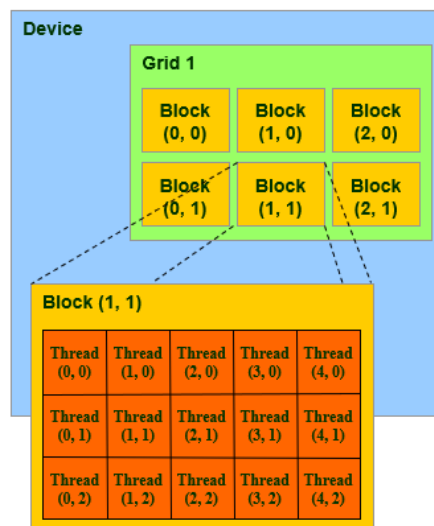
1 # pragma omp parallel num_threads(thread_count) \
2   default(none) private(i,j,sum) shared(A,x,y,m,n)
3   {
4   # pragma omp for
5   for (i=0; i<m; i++) {
6       sum = 0.0; #加入私有变量
7       for (j=0; j<n; j++)
8           sum += A[i][j]*x[j];
9       y[i] = sum
10   }
11 }

```

CUDA

CUDA架构

- **Threads:**
 - 3D IDs, unique within a block
- **Blocks:**
 - 3D IDs, unique within a grid
- **Built-in variables:**
 - threadIdx: idx within a block
 - blockIdx: idx within the grid
 - blockDim: block dimension
 - gridDim: grid dimension

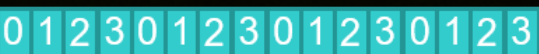


- 一次内核启动有一个网格 grid
- 每个网格包含若干个线程块 block
- 每个线程块包含若干个线程 thread

idx计算

当blockDim和gridDim都为1时

`blockDim.x = 4, gridDim.x = 4`

`threadIdx.x:` 


`blockIdx.x:` 

`idx = blockIdx.x*blockDim.x + threadIdx.x:` 

- `gridDim.x = 4`: 一共有4个block, `blockIdx.x` $\in \{0,1,2,3\}$
- `blockDim.x = 4`: 1个block里有4个thread, `threadIdx.x` $\in \{0,1,2,3\}$
- `idx = blockIdx.x*blockDim.x + threadIdx.x`

CUDA程序基础理解

长度为 N 的向量加法

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a, sizeof(float) * N);
    // ... allocate other arrays, fill with data

    // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

- `__global__` 放在核函数定义的开头
- 对于每个 `id < 向量长度N` 的线程, 计算一次对应 `id` 的分量加法
 - 线程 `id` 的计算: `idx = blockIdx.x * blockDim.x + threadIdx.x`
- 核函数调用时, 函数名之后、参数之前加入 `<<<线程块数目, 每个线程块内线程数目>>>`
- 给定线程块内的线程数 `p`, 则线程块数目 = $(N+p-1)/p$
 - 可以保证分量个数 `N` 无法整除线程数 `p` 时, 可以多分出1个线程块
- 使用函数管理GPU上的内存
 - `cudaMalloc()` 分配

- `cudaFree()` 释放

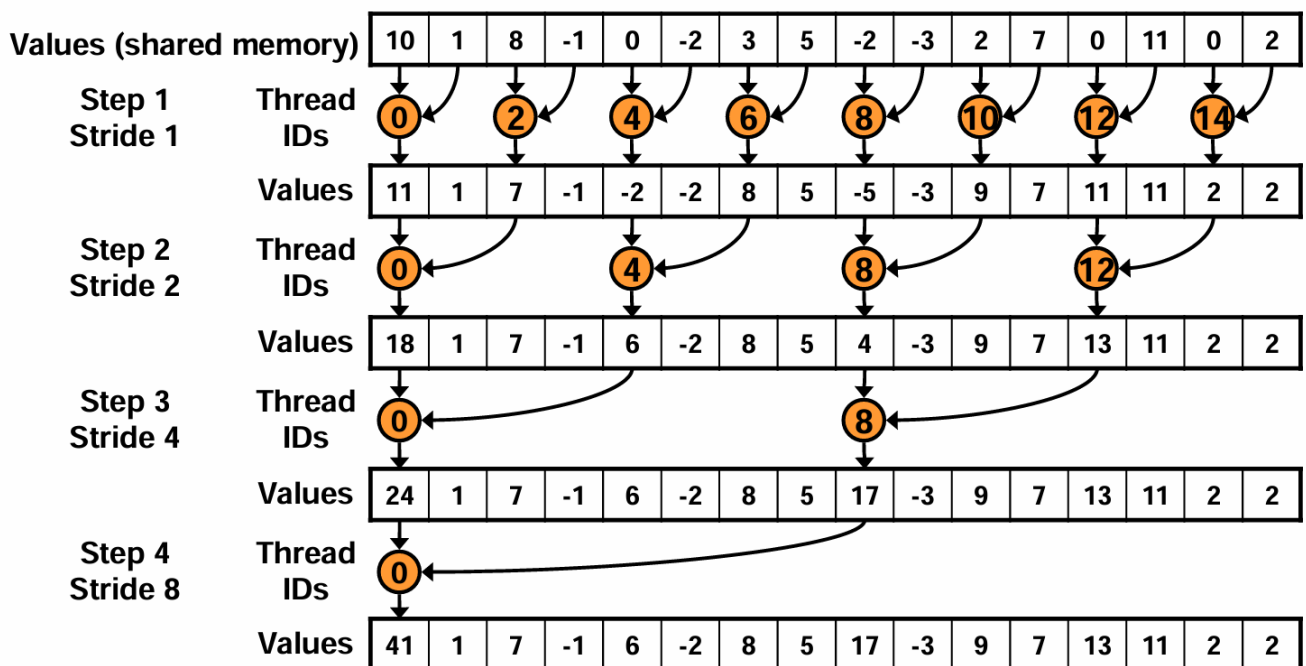
线程块内的线程同步

```
extern __shared__ float T[];
__device__ void
transpose (float* a, int lda){
    int i = threadIdx.x, j = threadIdx.y;
    T[i + lda*j] = a[i + lda*j];
    __syncthreads();
    a[i + lda*j] = T[j + lda*i];
}
```

26

- 使用 `__share__` 声明线程块内的共享变量
- 使用 `__syncthreads()` 同步对共享变量和全局变量的访问
 - 在写完共享变量之后调用
 - 所有线程需要执行相同的 `__syncthreads()`
- 不同线程块之间不能用这个方法同步

Reduction in CUDA



- s: 相加的两个变量的间隔 stride

Reduction #1: Interleaved Addressing



```
__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();
    }
```

Problem: highly divergent warps are very inefficient, and % operator is very slow

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

9

warp: 在一个block中, 一个warp包含32个并行thread, 并执行同一个指令

- warp divergence: 同一个warp中的所有thread必须执行相同指令, 在遇到控制流语句时, 如果线程进入了不同的分支, 则除了同一时刻正在执行的分支外, 其余分支都被阻塞了
- 模运算比简单的比较操作更复杂, 且会导致 warp 内线程在执行判断时的路径不一致

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();
}
```

- 通过线性计算出 index
- 分支判断时使用简单的小于比较