

并行硬件和并行软件

- 冯诺依曼结构：主存、CPU
- 冯诺依曼瓶颈：主存和CPU之间的分离
- 当两个线程共属于一个进程时，它们共享进程的大多数资源

解决冯诺依曼瓶颈的三种改进措施：

- 缓存
 - 一次内存访问能存取一整块代码和数据；将部分数据块或者代码块存储在一个靠近CPU寄存器的特殊存储器里
 - 局部性原理：在不久的将来（时间局部性）访问邻近的区域（空间局部性）
 - 写直达（write-through）和写回（write-back）。在写回策略中，数据不是立即更新到主存中，而是将发生数据更新的高速缓存行标记成dirty。当发生高速缓存行替换的时候，标记为dirty的高速缓存行被写入主存
 - 全相连（任意位置）、直接映射（唯一位置）和n路组相连
 - 替换cache策略：最近最少使用

- 虚拟存储器

在主存中存放当前执行程序所需用到的部分，来利用时间和空间局部性；那些暂时用不到的部分存储在辅存的块中，称为交换空间

TLB：转译后备缓冲区，将页与物理地址直接对应起来，加快查找速度

虚拟内存通常采用写回的方法，因为写磁盘的代价太大

- 指令级并行ILP：流水线和多发射

流水线

多发射

- 复制功能单元来同时执行不同指令
- 编译时调度，静态多发射，运行时调度，动态多发射
- 一个支持动态多发射的处理器称为超标量（superscalar）

- 预测 (speculation) 用于找出能够同时执行的指令。预测错误需要回退
- 硬件多线程

线程级并行 (TLP) 通过尝试同时执行不同线程来提供并行性。与ILP相比, TLP是粗粒度的硬件多线程提供一种机制, 当前执行的任务被阻塞时, 系统能够继续其他有用的工作

- 细粒度: 处理器在每条指令后切换线程, 跳过被阻塞的线程
- 粗粒度: 只切换需要等待较长时间才能完成而被阻塞的线程

同步多线程 (Simultaneous Multithreading, SMT) 是细粒度多线程的变种, 它通过允许多个线程同时使用多个功能单元来利用 [超标量处理器](#) 的性能。可以指定“优先线程”

并行硬件

SIMD 单指令多数据流

向量处理器, 它的可扩展性有限制, 因为它不能处理不规则的数据和其他并行结构

- 向量寄存器
- 向量化和流水化的功能单元。SIMD
- 向量指令。对于向量加法, 可以只需要一次加载、一次加法、一次存储
- 交叉存储器。内存被分为内存体, 连续访问同一个内存体有时间延迟, 访问不同内存体可以无延迟访问
- 步长式存储器访问和硬件散射/聚集: 程序能够访问固定间隔的元素; 散射/聚集是对无规律间隔的数据进行读 (聚集) 和写 (散射)

图形处理单元 (GPU)

图形处理流水线、着色函数。对邻近元素使用着色函数会导致相同的控制流

MIMD 多指令多数据流

通常包含一组完全独立的处理单元或者核。它通常是异步的, 即各个处理器能够按照自己的节奏运行

MIMD有两种类型: 共享内存和分布式内存系统

共享内存:

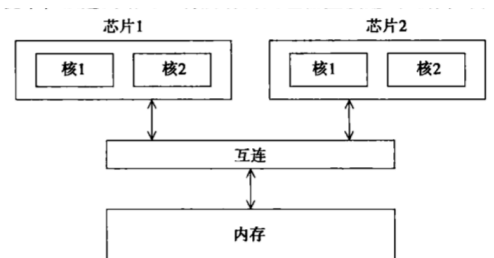


图 2-5 一个 UMA 多核系统

一致内存访问，每个核访问内存中任何一个区域时间相同

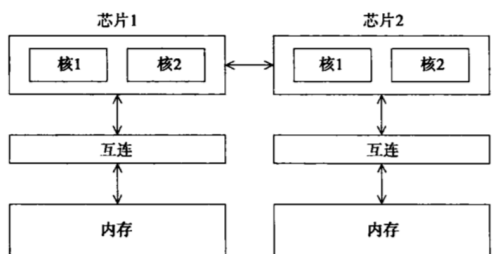


图 2-6 一个 NUMA 多核系统

非一致内存访问，存储容量更大，访问时间不一致

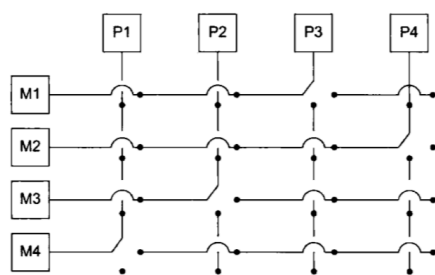
分布式内存系统：

最广泛使用的分布式内存系统成为集群。它们由一组商品化系统组成，通过商品化网络连接（如以太网）

互连网络 interconnection network

共享内存互连网络：

1.总线 2.交叉开关矩阵



分布式内存互连网络：

- 直接互连：环和环面网格

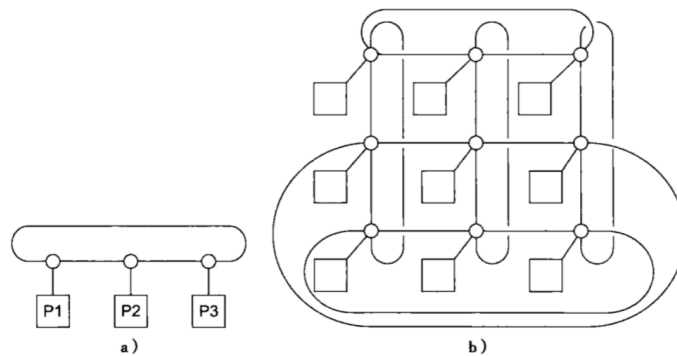


图 2-8 a) 一个环; b) 一个二维环面网格 (toroidal mesh)

等分宽度：用于衡量“同时通信的链路数目”或者“连接性”

等分宽度的计算：去除最少的链路数从而将节点分成两份，每部分都有一半的节点，去除的链路数就是等分宽度。是基于**最坏的情况**来估计的

一个正方形二维环面网格的等分宽度为 $2\sqrt{p}$, p 为节点个数

全相连网络的等分宽度： $p^2/4$

超立方体的等分宽度： $p/2$

- 间接互连：交换器不一定与处理器直接连接。每个处理器有一个输入链路和输出链路，这些链路通过一个交换网络连接
- 交叉开关矩阵

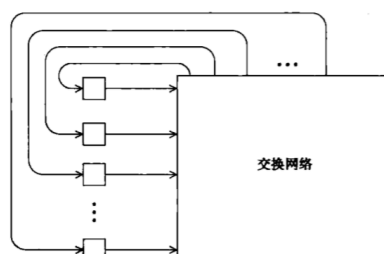


图 2-13 一个通用的间接网络

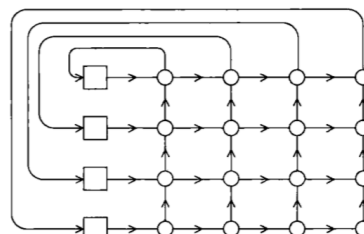


图 2-14 一个用于分布式内存的交叉开关矩阵互连网络

- omega网络

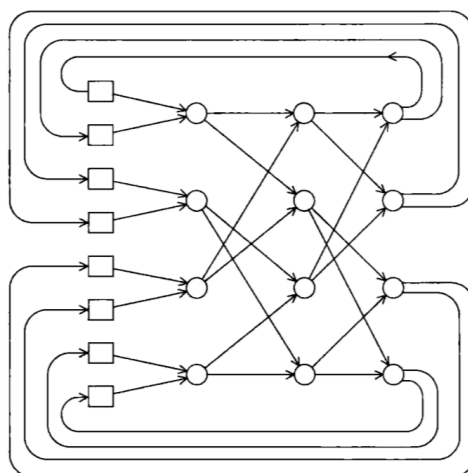


图 2-15 一个 omega 网络

- 延迟和带宽

Cache一致性

不同Cache中同一个变量的值可能不同，这和写回和写直达无关

如何保证一致性？

- 监听Cache一致性协议：对于基于总线的系统，当多个核共享总线时，总线上传递的信号能被所有核看到。因此，当核0更新Cache中的x副本时，它会广播整个Cache行已经更新，其他正在监听总线的核就可以将其标记为非法。这种协议是不可扩展的
- 基于目录的Cache一致性协议：使用一个叫做目录的数据结构。目录存储每个内存行的状态。这个数据结构一般是分布式的。当一个变量需要更新时，就查询目录，将所有包含该变量高速缓存行置为非法

伪共享：不会引发错误，但是会引起过多不必要的访存，降低程序的性能

并行软件

SPMD, Single Program Multiple Data: 单程序多数据流。SPMD不是在不同的核上运行不同的程序，相反，SPMD程序仅包含一段可执行代码，通过使用条件转移语句，让这一段代码在执行时表现得像是在不同处理器上处理不同的程序

任务并行：一个程序是将任务划分给各个进程来实现并行

负载均衡：分配的任务使得每个进程/线程获得大致相等的工作量

易并行的：能够通过简单地将任务分配给进程/线程来实现并行化

共享内存

- 动态线程：有一个主线程等待工作请求。当一个请求到达时，它派生出一个工作线程来执行该请求
- 静态线程：主线程派生出所有的线程，在工作结束前所有的线程都在运行

线程安全性：静态局部变量在函数调用后不会被销毁，类似于 `strtok` 这样的函数不是线程安全的，这意味着，如果它被多线程程序使用，会产生未知结果

分布式内存

单向通信：在消息传递中，一个进程必须调用一个发送函数，并且发送函数必须与另一个进程调用的接收函数相匹配。在单向通信或者远程内存访问中，单个处理器调用一个函数。在这个函数中，或者用来自另一个进程的值来更新局部变量，或者用来自于调用进程的值更新远端内存。这种方式能简化通信，并且降低了开销

性能

加速比和效率

线性加速比： $T_{parallel} = T_{serial}/p$

加速比：

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

效率：

$$E = \frac{S}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

Amdahl's law

除非一个串行程序的执行几乎全部并行化，否则，不论有多少个核，通过并行化所产生的加速比总是受限的

可扩展性

可扩展的：输入规模增大，效率不变

- 强可扩展的：如果在增加线程个数时，效率不变
- 弱可扩展的：如果在增加线程个数时，想要效率不变，需要以相同的速率增大问题的规模

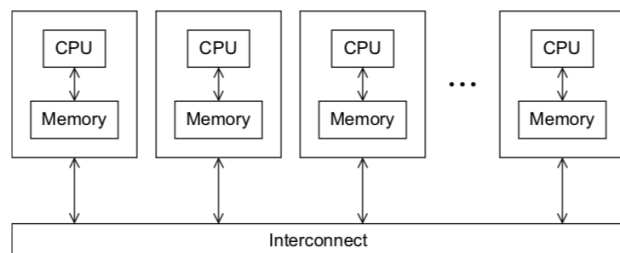
计时

墙上时钟：指的是代码从开始执行到执行结束的总耗费时间

分辨率：计时器的时间测量单位

用MPI进行分布式内存编程

本章将讨论如何用消息传递来对分布式内存系统进行编程



1. 预备知识

我们用MPI来编写一个简单 `hello world` 程序。我们指派其中的一个进程负责输出，其他的进程向它发送要打印的消息

```
#include <stdio.h>
int main(void) {
    printf("hello, world\n");
    return 0;
}
```

1.1 编译与执行

```
$mpicc -g -Wall -o mpi.hello mpi_hello.c
```

MPI程序

```

1 #include <stdio.h>
2 #include <string.h> /* For strlen */
3 #include <mpi.h> /* For MPI functions, etc */
4
5 const int MAX_STRING = 100;
6
7 int main(void) {
8     char greeting[MAX_STRING];
9     int comm_sz; /* Number of processes */
10    int my_rank; /* My process rank */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17        sprintf(greeting, "Greetings from process %d of %d!",
18                my_rank, comm_sz);
19        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20                 MPI_COMM_WORLD);
21    } else {
22        printf("Greetings from process %d of %d!\n", my_rank,
23               comm_sz);
24        for (int q = 1; q < comm_sz; q++) {
25            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
26                     0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
27            printf("%s\n", greeting);
28        }
29
30        MPI_Finalize();
31        return 0;
32    } /* main */

```

启动程序：

```
$mpiexec -n <number of process> ./mpi_hello
```

1.3 MPI_Init 和 MPI_Finalize

MPI_Init：告知MPI系统进行所有必要的初始化设置。例如，分配存储空间，为进程指定进程号等

```

int MPI_Init(
    int* argc_p, // 指向argc和argv的指针 不使用时置为NULL
    char*** argv_p
);

```

MPI_Finalize：告知MPI系统MPI已经使用完毕，为MPI分配的任何资源可以释放了

```
int MPI_Finalize(void);
```

1.4 通信子、`MPI_Comm_size` 和 `MPI_Comm_rank`

通信子：一组可以互相发送消息的进程集合。`MPI_Init` 的一个目的，是在用户启动程序时，定义由用户启动的所有进程所组成的通信子。这个通信子称为 `MPI_COMM_WORLD`

```
// 返回通信子的进程数
int MPI_Comm_size(
    MPI_Comm comm,
    int* comm_sz_p
);
// 返回正在调用进程在通信子中的进程号
int MPI_Comm_rank(
    MPI_Comm comm,
    int* my_rank_p
);
```

1.5 SPMD程序

大部分的MPI的程序，都是编写一个单个程序，让不同进程产生不同动作。实现方式就是，让进程按照它们的进程号来匹配程序分支。这一方法叫做单程序多数据流（Single Program, Multiple Data, SPMD）

1.7 `MPI_Send`

```
int MPI_Send(
    void* msg_buf_p,
    int msg_size,
    MPI_Datatype msg_type, // 前三个参数定义消息内容
    int dest,
    int tag,
    MPI_Comm communicator // 后三个参数定义了消息目的地
);
```

C语言中的类型（int, char）不能作为参数传递给 `msg_type`，所以MPI定义了一个特殊的类型 `MPI_Datatype`，比如 `MPI_CHAR`

1.8 MPI_Recv

```
int MPI_Recv(
    void*      msg_buf_p,
    int        buf_size,
    MPI_Datatype msg_type,    // 前三个参数指定了用于接受消息的内存
    int        source,
    int        tag,
    MPI_Comm    communicator, // 后三个参数用于识别消息
    MPI_Status* status_p      // 大多数情况下不使用
);
```

source可以为MPI_ANY_SOURCE，可以按照进程完成工作的顺序来接收结果

tag可以为MPI_ANY_TAG，用于接收一个进程不同标签的信息

1.9 消息匹配

q号进程调用 MPI_Send 函数所发送的消息可以被r号进程调用 MPI_Recv 函数接收，如果：

- `recv_comm=send_comm`
- `recv_tag=send_tag`
- `dest=r`
- `src=q`
- `recv_type=send_type` ， 同时 `recv_buf_sz>=send_buf_sz`

为了使得用于接收的进程能够按照其他进程任务完成的顺序接收信息，可以指定 `source` 为 `MPI_ANY_SOURCE`

为了接收同一个进程不同的 `tag`，可以使用 `MPI_ANY_TAG`

1.10 status_p 参数

可以发现，接收者可以再不知道以下信息的情况下接收信息：

- 消息中的数据量
- 消息的发送者，或
- 消息的标签

那么，接受者是如何找出这些值的？`MPI_Recv` 的最后一个参数 `MPI_Status *` 是一个至少包含三个成员的结构，`MPI_SOURCE`、`MPI_TAG` 和 `MPI_ERROR`。通过前两个成员确定发送者和标签。用 `MPI_GET_COUNT` 可以找到接收到的数据量

1.11 `MPI_Send` 和 `MPI_Recv` 的语义

当我们将消息从一个进程发送到另一个进程时，会发生什么？发送进程组装消息。一旦组装完毕，发送进程可以缓冲消息，也可以阻塞。如果它缓冲消息，则MPI系统会把消息放置在自己的内部存储器里，并返回 `MPI_Send` 的调用。另一方面，如果系统发生阻塞，那么它将一直等待，直到可以开始发送消息，并不立即返回对 `MPI_Send` 的调用。因此，我们实际上并不知道消息是否已经发送出去。

`MPI_Send` 典型的实现方法有一个默认的消息“截止”大小。如果一条消息的大小小于“截止”大小，它将被缓冲；否则，`MPI_Send` 将被阻塞

`MPI_Recv` 总是阻塞的，直到接收到一条匹配的消息。因此，当 `MPI_Recv` 函数调用返回时，就知道一条消息已经存储在接收缓冲区中了。

MPI要求消息是不可超越的。即如果q号进程发送了两条消息给r号进程，那么q进程发送的第一条消息必须在第二条消息之前可用。但是，如果消息是来自不同进程的，消息的到达顺序是没有限制的

1.12 潜在的陷阱

`MPI_Recv` 的语义会导致MPI编程中一个潜在陷阱：如果一个进程试图接收消息，但没有相匹配的消息，那么该进程将会永远被阻塞在那里，即进程 **悬挂**

因此，在程序设计时，我们需要保证每条接受都有一条相匹配的发送

`MPI_Reduce`中的count参数，1时单个值全局求和，否则数组全局求和

`MPI_Allreduce`

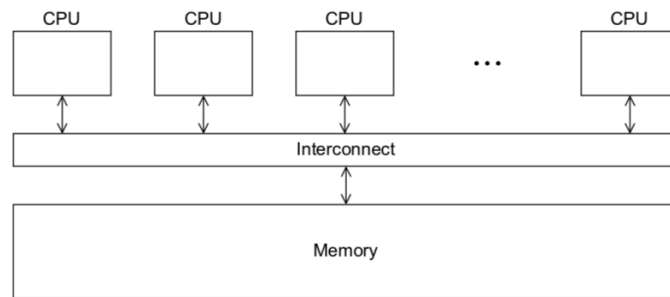
`MPI_Bcast`

广播比较耗费资源？

`MPI_Scatter`

用OpenMP进行共享内存编程

OpenMP是一个针对共享内存并行编程的API。它允许程序员只需要简单地声明一块代码应该并行执行，而由编译器和运行时系统来决定哪个线程具体执行哪个任务，这意味着OpenMP要求编译器支持某些操作



OpenMP被设计成用来对已有的串行程序进行增量式并行化，只需要对源代码进行少量改动就可以并行化许多串行 `for` 循环

1. 预备知识

在C和C++中，有一些特殊的预处理指令 `pragma`，它被用来允许不是基本C语言规范部分的行为。不支持 `pragma` 的编译器会忽略 `pragma` 指令提示的那些语句

1.1 编译和运行OpenMP程序

编译时需要包含 `-fopenmp` 选项

```
$ gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

运行时需要明确线程的个数

```
./omp_hello 4
```

对于 `omp_hello` 这个程序而言，线程会竞争访问标准输出，因此不保证输出会按线程编号的顺序出现

1.2 程序

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 void Hello(void); /* Thread function */
6
7 int main(int argc, char* argv[]) {
8     /* Get number of threads from command line */
9     int thread_count = strtol(argv[1], NULL, 10);
10
11 # pragma omp parallel num_threads(thread_count)
12     Hello();
13
14     return 0;
15 } /* main */
16
17 void Hello(void) {
18     int my_rank = omp_get_thread_num();
19     int thread_count = omp_get_num_threads();
20
21     printf("Hello from thread %d of %d\n", my_rank, thread_count);
22
23 } /* Hello */
```

头文件为 `omp.h`

使用 `strtol` 函数来获得线程数，`argv[1]` 是命令行输入，`10` 表示 `argv[1]` 的基数

第十一行，`pragma` 后面的第一条指令是一条 `parallel` 指令，表明之后的结构化代码块应该被多个线程并行执行。每个线程有自己的栈和程序计数器。当一个线程完成了执行，就合并到启动它的进程中

线程可以启动的线程数会受到系统定义的限制

当线程从 `Hello` 调用中返回时，有一个隐式路障，这意味着完成代码块的线程将等待线程组中的所有其他线程完成代码块

调用 `omp_get_thread_num()` 和 `omp_get_num_threads` 可以分别获得每个线程的编号和线程组中的线程数

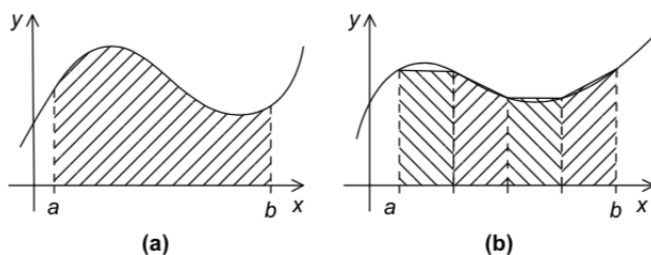
1.3 错误检查

上述程序存在一些潜在的错误：

1. 一定要检查命令行参数的存在
2. 假如编译器不支持OpenMP，那么它会忽略 `parallel` 指令，但是试图包含 `omp.h` 和调用其中的函数会引起错误。解决方法：检查预处理器宏 `_OPENMP` 是否定义

```
#ifndef _OPENMP
# include <omp.h>
#endif
```

2. 梯形积分法



将 $[a, b]$ 划分为 n 个子区间， $h = (b - a)/n$, $x_i = a + ih$ ，那么梯形面积的近似值为

$$h[f(x_0)/2 + f(x_1) + \dots + f(x_{n-1}) + f(x_n)/2]$$

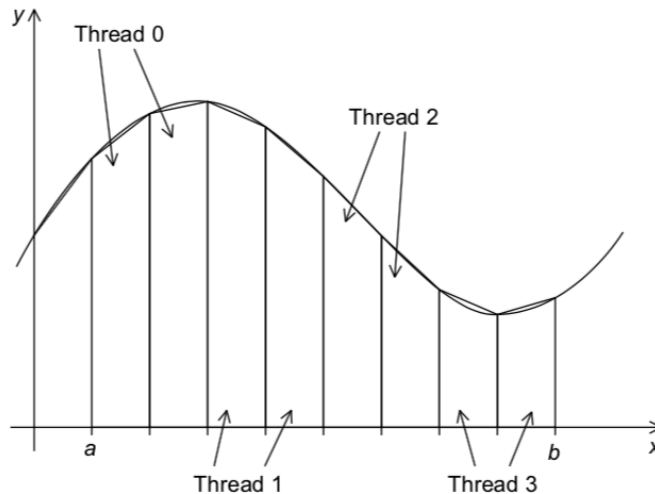
串程序序：

```
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

第一个OpenMP版本


该算法有两个任务：a. 单个梯形面积的计算 b. 梯形面积的求和

任务分配：



假如用一个共享变量作为所有线程的和，`global_result += my_result`，有可能会发生竞争，即多个线程试图访问同一个共享资源，并且至少其中一个访问是更新该共享资源。引起竞争条件的代码称为临界区。可以使用 `critical` 指令来实现互斥访问：

```
# pragma omp critical
global_result += my_result;
```

 代码请看书

3. 变量的作用域

在OpenMP中，一个能够被线程组中所有线程访问的变量拥有共享作用域，只能被单个线程访问的变量拥有私有作用域

在 `parallel` 块之前被声明的变量的缺省作用域是共享的，而在块中声明的变量（比如函数中的局部变量）是私有的

4. 归约子句

在上述梯形积分的程序中，我们使用的函数原型是：

```
void Trap(double a, double b, int n, double* global_result_p);
```

但是我们可能会定义以下函数，返回每个线程计算的结果

```
double Local_trap(double a, double b, int n);
```

求和变为：

```
global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
#   pragma omp critical
    global_result += Local_trap(a, b, n);
}
```

但是，因为指定的临界区，对 `Local_trap` 的调用一次只能被一个线程执行，这就相当于强制各个线程顺序执行梯形积分法

我们可以通过在 `parallel` 块中声明一个私有变量，并将临界区移到函数调用之后来避免这个问题

```
global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
    double my_result = 0.0;
    my_result += Local_trap(a, b, n);
#   pragma omp critical
    global_result += my_result;
}
```

OpenMP提供了一个更加清晰的方法来避免 `Local_trap` 的串行执行：将 `global_result` 定义为一个归约变量

归约操作符是一个二元操作，比如加法和减法，归约就是将相同的归约操作符重复地应用到操作数序列来得到一个计算的结果。所有操作的中间结果存在同一个变量里：**归约变量**。只需要在 `parallel` 中添加一个 `reduction` 子句

```
global_result = 0.0;
# pragma omp parallel num_threads(thread_count) reduction(+: global_result)
global_result += Local_trap(a, b, n);
```

当归约操作符是减法时会有一点问题，因为减法不满足交换律和结合律

当归约变量是一个 `float` 或 `double` 时，`(a+b)+c` 的结果可能不准确地等于 `a+(b+c)`

当一个变量被包含在 `reduction` 子句中时，变量本身是共享的。但是，线程组中的每个线程都创建自己的私有变量，在 `parallel` 块结束后进行汇总

5. `parallel for` 指令

运用OpenMP提供的 `parallel for` 指令，可以并行化串行梯形积分法。方法是直接在 `for` 循环前放置一条指令

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
# pragma omp parallel for num_threads(thread_count) \
    reduction(+: approx)
for (i = 1; i <= n-1; i++)
    approx += f(a + i*h);
approx = h*approx;
```

5.1 警告

`parallel for` 只会并行化具有典型结构的 `for` 循环（`for (...; ...; ...;)`），不并行 `while` 循环，且 `for` 循环只有一个出口，不能有 `break` 语句

5.2 数据依赖性

OpenMP编译器不检查被 `parallel for` 指令并行化的循环所包含的迭代间的依赖关系

5.5 关于作用域的更多问题

OpenMP提供子句 `default`，要求程序员明确在一个块中使用的每个变量和已经在块外声明的变量的作用域

```
double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    default(none) reduction(+:sum) private(k, factor) \
    shared(n)
for (k = 0; k < n; k++) {
    if (k % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    sum += factor/(2*k+1);
}
```

6. OpenMP：排序

奇偶变换排序

冒泡排序不容易被并行化

奇偶变换排序是一个和冒泡排序相似的算法，但是它更容易并行化。在“偶阶段”(`phase % 2 == 0`)，每个偶下标元素 `a[i]` 与它左边的元素 `a[i-1]` 相比较；在一个“奇阶段”里，每个奇元素下标元素与它右边的元素相比较。有定理证明，在 n 个阶段后，列表可以完成排序

```
for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) Swap(&a[i-1], &a[i]);
    else
        for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

对于上述算法，可以发现：外部 `for` 循环有依赖，内部 `for` 循环没有依赖

假如直接在两条内部 `for` 循环前加上 `# pragma omp parallel`，每次线程的创建和合并会有一定的开销

我们可以选择只创建一次线程，并在每次内部循环的执行中重用它们。在OpenMP中，用 `parallel` 指令在外部循环前创建 `thread_count` 个线程的集合，然后在内循环，直接使用 `for` 指令

```
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         }
```

7. 循环调度

在 `parallel for` 中，把各个循环分配给线程的操作是由系统完成的，而大部分的OpenMP实现只是粗略地使用块划分。不难想象，对于下列代码，块划分肯定不是最优的

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

其中，调用 $f(i)$ 所花费的时间与 i 成正比。一个更好的分配方案是采用循环划分

Thread	Iterations
0	0, n/t , $2n/t$, ...
1	1, $n/t + 1$, $2n/t + 1$, ...
\vdots	\vdots
$t - 1$	$t - 1$, $n/t + t - 1$, $2n/t + t - 1$, ...

在OpenMP中，将循环分配给线程称为调度，`schedule`子句用于在`parallel for`或者`for`指令中进行迭代分配：

7.1 `schedule` 子句

`schedule`子句的形式：`schedule(<type> [, <chunksize>])`

`type`可以是以下任意一个：

- `static`：迭代能够在循环执行前分配给线程
- `dynamic` 或 `guided`：迭代在循环执行时被分配给线程，因此在一个线程完成了它的当前迭代集合后，它从运行时系统中请求更多
- `auto`：编译器和运行时系统决定调度方式
- `runtime`：调度在运行时决定

`chunksize`是一个正整数，迭代块时在顺序循环中连续执行的一块迭代语句。只有`static`、`dynamic`和`guided`调度有`chunksize`

7.2 `static` 调度类型

对于`static`调度，系统以轮转的方式分配`chunksize`块给每个线程

Thread 0: 0, 3, 6, 9

`(static, 1)`: Thread 1: 1, 4, 7, 10

Thread 2: 2, 5, 8, 11

Thread 0: 0, 1, 6, 7

`(static, 2)`: Thread 1: 2, 3, 8, 9

Thread 2: 4, 5, 10, 11

Thread 0: 0,1,2,3

(static, 4) : Thread 1: 4,5,6,7

Thread 2: 8,9,10,11

7.3 dynamic 和 guided 调度类型

在 dynamic 调度中，迭代也被分成 chunksize 个连续迭代的块。每个线程执行一块，并且当一个线程完成一块时，它将从运行时系统请求另一块，直到所有的迭代完成

在 guided 调度中，每个线程执行一块，并且当一个线程完成一块时，它将从运行时系统请求另一块。然而，在 guided 调度中，当块完成后，新块的大小会变小，块的大小近似等于 剩下的迭代数除以线程数

7.4 runtime 调度类型

可以在命令行中指定调度的类型，需要设置环境变量

```
$export OMP_SCHEDULE="static, 1"
```

7.5 调度选择

dynamic 调度的系统开销要大于 static 调度，guided 系统开销最大

- 如果循环每次迭代需要几乎相同的计算量，那么可能默认的调度方式能提供最好的性能
- 如果随着循环的进行，迭代的计算量线性递增，那么采用比较小的 chunksize 的 static 调度可能会提供最好的性能
- 如果每次迭代的开销事先不确定，那么就要尝试多种不同的调度策略

8. 生产者和消费者问题

本节将讨论一个不适合用 parallel for 指令来并行化的问题

消息传递

为了在共享内存系统上实现消息传递，每一个线程有一个共享消息队列，当一个线程要向另一个线程发送消息时，它将消息放入目标线程的消息队列中。一个线程接收消息时只需从它的消息队列的头部取出消息

每个线程的伪代码：

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
    Send_msg();
    Try_receive();
}

while (!Done())
    Try_receive();
```

发送消息

访问消息队列并将消息入队，可能是一个临界区：我们有可能会使用一个单链表来实现消息队列，有一个指针指向队列队尾。那么，当一条新消息入队时，需要检查和更新这个队尾指针。如果两个线程试图同时进行这些操作，可能会发生错误

`Send_msg()` 函数的伪代码：它随机产生一个整数信息，并随机发送给队列中的一个

```
mesg = random();
dest = random() % thread_count;
# pragma omp critical
Enqueue(queue, dest, my_rank, mesg);
```

接收消息

如果消息队列中至少有两消息，那么只要每次出队一条消息，那么出队操作和入队操作就不可能冲突。因此如果队列中至少有两消息，通过跟踪队列的大小就可以避免任何同步（如 `critical` 指令）

可以使用两个变量，`enqueued` 和 `dequeued`，队列中消息的个数为：`queue_size=enqueued-dequeued`

唯一能够更新 `dequeued` 的线程是消息队列的拥有者。在一个线程使用 `enqueued` 计算队列大小 `queue_size` 时，另外一个线程可以更新 `enqueued`。尽管这会造成队列大小计算的不一致，但是即使这种不一致发生，消息队列的拥有者只需要延迟一段时间后重新计算队列大小，就能得到正确的值

`Try_receive()` 函数的伪代码：

```
queue_size = enqueued - dequeued;
if (queue_size == 0) return;
    else if (queue_size == 1)
#       pragma omp critical
        Dequeue(queue, &src, &mesg);
    else
        Dequeue(queue, &src, &mesg);
        Print_message(src, mesg);
```

终止检测

讨论如何实现 `Done()` 函数

```
queue_size = enqueued - dequeued;
if (queue_size == 0)
    return TRUE;
else
    return FALSE;
```

上述实现是有问题的。如果线程u执行这段代码，那么很可能有些线程，比如v，在线程u计算出 `queue_size=0` 后向线程u发送一条消息。但是此时线程u已经终止了，那么这条v发送的消息就永远无法被u收到

增加一个计数器 `done_sending`，每个线程在 `for` 循环结束后（发送消息结束后）将计数器加1

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
    return TRUE;
else
    return FALSE;
```

启动

当程序开始执行时，主线程分配一个数组空间给消息队列，每个线程对应着一个消息队列，每个线程可以向其他任意的线程发送消息，因此这个数组应该被所有线程共享

这里有一个重要问题：在队列分配时，一个线程可能比其他线程更快地被分配，这个完成分配的线程可能会试图向还没有完成分配的线程发送消息。为了解决这个问题，OpenMP提供显式路障，保证所有线程都分配完成后，才开始发送消息

```
# pragma omp barrier
```

atomic 指令

这是之前的每个线程的伪代码：

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
    Send_msg();
    Try_receive();
}

while (!Done())
    Try_receive();
```

在 **for** 循环之后，需要对 **done_sending** 加1。显然，这个操作是临界区，可以用 **critical** 指令来保护它

然鹅，OpenMP提供了另外一种更加高效的指令：**atomic** 指令。它只能保护一条由C语言赋值语句形成的临界区，且语句必须是一下几种形式之一：

```
x <op> = <expression>; <op>是二元操作符+ * - / & ^ | << >>
x++;
++x;
x--;
--x;
```

临界区和锁

锁由一个数据结构和定义在这个数据结构上的函数组成，这些函数使得程序员可以显式地强制对临界区进行互斥访问


```

/* Executed by one thread */
Initialize the lock data structure;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure;

```

锁的数据结构被临界区的线程所共享。在一个线程进入临界区前，它尝试通过调用锁函数来上锁。执行完临界区的代码后，它调用解锁函数释放锁。

在消息传递程序中使用锁

我们想要确保的是对每个消息队列进行互斥访问，而不是对于一个特定的代码块

```

# pragma omp critical
/* q-p = msg-queues[dest] */ Enqueue(q-p, my-rank, mesg);

```

可以替换为 \Rightarrow

```

/* q-p = msg-queues[dest] */
omp_set_lock(&q-p->lock);
Enqueue(q-p, my-rank, mesg);
omp_unset_lock(&q-p->lock);

```

每个消息队列拥有不同的锁

critical 指令、**atomic** 指令、锁的比较

atomic 指令实现互斥访问最快

锁机制适用于需要互斥的是某个数据结构而不是代码块

使用 **critical** 指令保护临界区与使用锁机制保护临界区在性能上没有太大的差别

9. 缓存、缓存一致性、伪共享

缓存，是比主存更快的存储器。它考虑了时间和空间局部性

例如在矩阵乘法的例子中，矩阵维度相差较大时，容易发生读缺失或写缺失

缓存一致性是在"缓存行级别"上执行的。只要缓存行中某个变量改变了，且其他核的缓存也存储了该变量，那么整个缓存行都会被标记为不合法

伪共享：假设有不同缓存的两个线程访问同一个缓存行中的不同变量，还假设至少有一个线程更新了变量的值。尽管没有线程对共享变量进行了更新，但缓存控制器会将整个缓存行失效，从而迫使其他线程从主存中获取变量的值。这些线程之间没有共享任何变量（只是共享了同一个缓存行），但是它们访问主存的行为看起来好像它们共享了一个变量



详细内容请看书

10. 线程安全性

某些C函数通过声明 `static` 变量，在不同调用之间缓存数据。当多个线程调用该函数时，这可能导致错误。因为静态存储在多个线程间共享，一个线程可以写覆盖另外一个线程的数据。这样的函数不是线程安全的。

CUDA

这个部分书上没有对应章节，整理的是上课课件

[CUDA编程入门极简教程](#)

1. Heterogeneous Computing(异构计算)

- Host：指CPU和它的内存
- Device：指GPU和它的内存

在异构系统上进行的并行计算通常称为异构计算

处理流程

- 第一步：将输入数据从CPU内存拷贝至GPU内存
- 第二步：装载GPU程序并执行，可将数据缓存在芯片上提高性能
- 第三步：将输出结果从GPU内存拷贝至CPU内存

标准的C代码会在host上跑。NVIDIA编译器（nvcc）可以编译没有device代码的程序

Hello World

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1, 1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- 关键字 `__global__` 修饰的函数表明：这个函数在CPU中被调用，在GPU上运行
- `nvcc` 会把源代码分成host和device部分。device代码被NVIDIA编译器编译，host代码被标准编译器编译（如 `gcc`）
- `mykernel<<<1,1>>>`：三个尖括号表明在host调用device代码，这也被叫做 kernel launch，参数 `1,1` 的含义在之后解释

Memory Management

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b; // 使用指针表示变量  
}  
  
int main(void) {  
    int a, b, c; // host copies of a, b, c  
    int *d_a, *d_b, *d_c; // device copies of a, b, c  
    int size = sizeof(int);  
  
    // Allocate space for device copies of a, b, c
```

```

cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

// Setup input values
a = 2;
b = 7;

// copy input to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>>(d_a, d_b, d_c);

// copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}

```

- `add()` 是在device上运行的，所以 `a,b,c` 必须是指向device内存的指针
- CUDA处理内存的函数： `cudaMalloc()`， `cudaFree()`， `cudaMemcpy()`

<pre> cudaError_t cudaMemcpy (void * dst, const void * src, size_t count, enum cudaMemcpyKind kind) </pre> <p>Parameters:</p> <ul style="list-style-type: none"> <code>dst</code> - Destination memory address <code>src</code> - Source memory address <code>count</code> - Size in bytes to copy <code>kind</code> - Type of transfer 	<pre> enum cudaMemcpyKind </pre> <p>CUDA memory copy types</p> <p>Enumerator:</p> <table> <tr> <td><code>cudaMemcpyHostToHost</code></td> <td>Host -> Host.</td> </tr> <tr> <td><code>cudaMemcpyHostToDevice</code></td> <td>Host -> Device.</td> </tr> <tr> <td><code>cudaMemcpyDeviceToHost</code></td> <td>Device -> Host.</td> </tr> <tr> <td><code>cudaMemcpyDeviceToDevice</code></td> <td>Device -> Device.</td> </tr> </table>	<code>cudaMemcpyHostToHost</code>	Host -> Host.	<code>cudaMemcpyHostToDevice</code>	Host -> Device.	<code>cudaMemcpyDeviceToHost</code>	Device -> Host.	<code>cudaMemcpyDeviceToDevice</code>	Device -> Device.
<code>cudaMemcpyHostToHost</code>	Host -> Host.								
<code>cudaMemcpyHostToDevice</code>	Host -> Device.								
<code>cudaMemcpyDeviceToHost</code>	Device -> Host.								
<code>cudaMemcpyDeviceToDevice</code>	Device -> Device.								

2. Blocks

修改 `add<<<1,1>>>>()` 为 `add<<<N,1>>>>`，就可以实现并行。

将 `add` 修改为向量的加法

```

__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}

```

```

}

#define N 512
int main(void) {
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    int size = N * sizeof(int);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);

    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

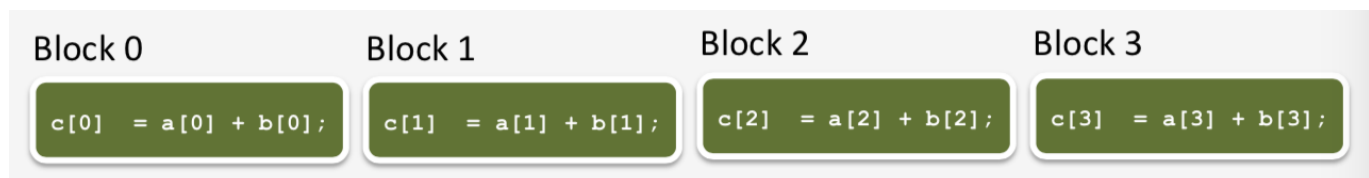
    add<<<N,1>>>(d_a, d_b, d_c);

    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}

```

- block: 线程块，一个线程块里包含很多线程
- grid: 多个线程块组成网格
- 使用 `blockIdx.x` 可以指明坐标



3. Threads

threads: 一个进程块(block)可以被划分为很多个线程

修改 `add()` 使用线程并行

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

此时在 `main()` 中，需要做对应的修改：

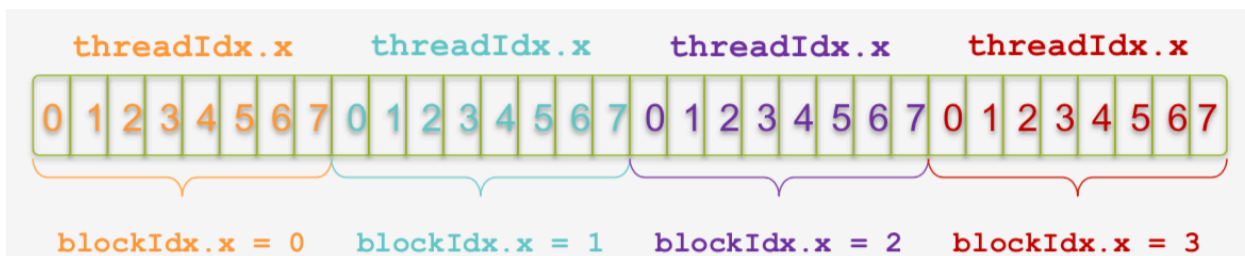
```
add<<<1,N>>>>(d_a, d_b, d_c);
```

4. Indexing

在上述加法的例子中，我们使用了：① 许多线程块，每个线程块一个进程 ② 一个线程块，块里有很多线程

我们可以修改程序，使得其同时使用多个线程块，每个线程块有多个线程

修改后，我们就需要同时使用 `blockIdx.x` 和 `threadIdx.x` 来确定一个线程的位置



```
int index = threadIdx.x + blockIdx.x * M; // M = 每个线程块的线程数
```

有一个内置变量 `blockDim.x`，它可以代替 `M`，即每个线程块的线程数

```
__global__ void add(int *a, int *b, int *c) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    c[index] = a[index] + b[index];  
}
```

在 `main()` 中，需要作出对应的修改：

```
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>>(d_a, d_b, d_c);
```

假如向量的大小不是 `blockDim.x` 的倍数怎么办? 需要在函数中添加条件检测

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n) {
        c[index] = a[index] + b[index];
    }
}
```

```
add<<<(N+M-1)/M, M>>>(d_a, d_b, d_c, N);
```

假如向量太大怎么办? 让每个线程处理多个值

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    while (index < n){
        c[index] = a[index] + b[index];
        index += blockDim.x * gridDim.x;
    }
}
```

Thread Hierarchy

Thread: Distributed by CUDA runtime(identified by threadIdx)

Warp: A scheduling unit of up to 32 threads

Block: A user defined group of 1 to 512 threads

Grid: A group of one or more blocks. A grid is created for CUDA kernel

CUDA Warp

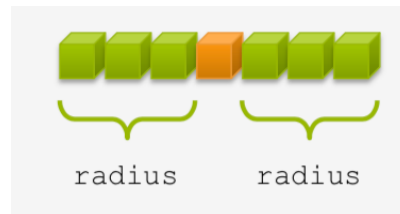
CUDA利用了单指令多线程 (SIMT)。线程束(warp) 包含32个线程, 它把接收到的单指令广播给它所有的线程。但是每个线程都包含自己的指令地址计数器和寄存器状态, 也有自己的独立执行路径

Streaming Multiprocessor

假如一个核启动了很多线程，但是没有多核的支持，那么在物理层也是无法并行的

GPU硬件的一个核心组件是SM。它可以并发地执行数百个线程。当一个kernel被执行时，它的grid中的线程块被分配到SM上，一个线程块只能在一个SM上被调度。SM的基本执行单元是线程束(warp)，所以被分配的线程块会进一步被划分为线程束。根据上述讨论，线程束有自己独立的执行路径，这会导致性能下降

5. Shared memory



假如我们要计算橙色方块的输出，它的输出为相邻7个方块值的总和

假设每个线程需要计算一个橙色方块的输出。那很可能一个方块的值要被不同的线程读入很多次（7次）

可以使用共享内存，使用 `__shared__` 声明

CUDA Memory Hierarchy

Local Memory: per thread memory for automatic variables and register spilling

Shared Memory: per block low-latency memory to allow for intra-block data sharing and synchronization.

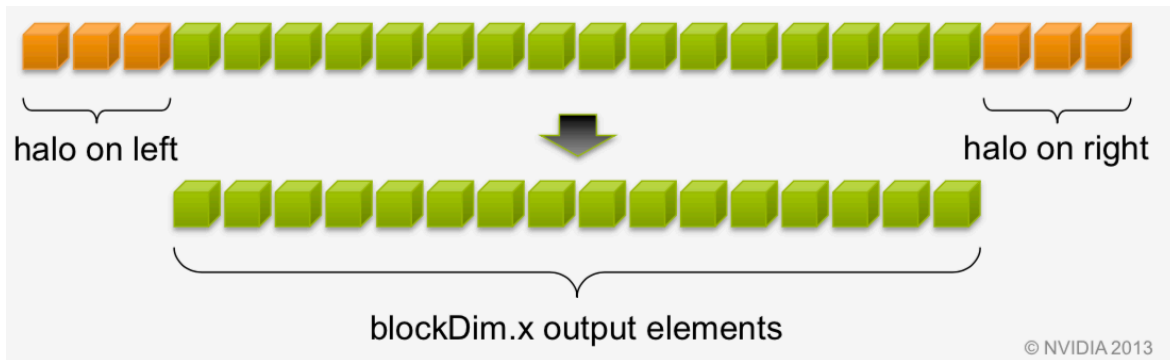
Threads can safely share data through this memory and can perform barrier synchronization through

`__syncthreads()`

Global Memory: device level memory that may be shared between blocks or grids

共享内存实现

把需要共享的内存进行缓存



把所有方块读入共享内存；计算绿色方块的输出值；把输出值存回全局内存

在计算之前，需要在两侧添加多余的方块

```
__global__ void stencil_1d(int *in, int *out) {  
    // 所有绿色的方块构成in数组 大小和out数组相同  
    // [绿色方块左侧边界-RADIUS, 绿色方块右侧边界+RADIUS]  
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];  
    // global index  
    // 每个线程计算一个方块  
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;  
    // local index  
    // 共享内存和in不一样大 它的左侧多了三个(RADIUS 个)橙色方块 需要偏移  
    int lindex = threadIdx.x + RADIUS;  
  
    // read input elements into shared memory  
    // 每个线程至少把自己对应位置的元素读进来  
    temp[lindex] = in[gindex];  
    // 对于线程0, 1, 2, 还要把橙色方块读进来  
    if (threadIdx.x < RADIUS) {  
        temp[lindex - RADIUS] = in[gindex - RADIUS];  
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
    }  
  
    // apply the stencil  
    int result = 0;  
    for (int offset = -RADIUS; offset <= RADIUS; offset++)  
        result += temp[lindex + offset];  
  
    out[gindex] = result;  
}
```

但是上述程序存在一个问题：有可能某个线程需要计算的值，还未被其他进程读入

6. __syncthreads()

使用 `__syncthreads()` 可以解决上述问题。它可以使同一个block中的所有线程同步

```
__global__ void stencil_1d(int *in, int *out) {
    // [绿色方块左侧边界-RADIUS, 绿色方块右侧边界+RADIUS]
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    // global index
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    // 在共享内存中的位置
    int lindex = threadIdx.x + RADIUS;

    // read input elements into shared memory
    // 为什么只读了三个数?因为每个线程都会读
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    __syncthreads(); // 使得所有线程同步

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

    out[gindex] = result;
}
```

7. Asynchronous operation

kernel launch是异步的，这意味着，在host中调用device上的代码后，host不会等待kernel执行，而是马上执行下一步。所以在需要用到计算结果之前，CPU需要同步

- `cudaMemcpy()` : Blocks the CPU until the copy is completed. Copy begins when all preceding CUDA calls have completed.
- `cudaMemcpyAsync()` : Asynchronous, does not block the CPU
- `cudaDeviceSynchronize()` : Blocks the CPU until all preceding CUDA calls have completed

8. Handling errors

所有的CUDA API会返回错误码 `cudaError_t`，说明错误发生于当前的API调用或者之前的异步操作 (e.g. kernel)

获取最后发生的错误码: `cudaError_t cudaGetLastError(void)`

获取一个描述错误的字符串: `char *cudaGetErrorString(cudaError_t)`

9. Managing devices

应用可以查询GPU并且选择对应的GPU

```
cudaGetDeviceCount(int *count);
cudaSetDevice(int device);
cudaGetDevice(int *device);
// cudaDeviceProp是一个结构体,里面包含GPU的信息
cudaGetDeviceProperties(cudaDeviceProp *prop, int device);
```

多个线程可以共用一个GPU，一个线程可以控制多个GPU

Optimizing Code for CUDA

- 有可能你对问题的分解会造成线程饥饿
- 使用共享内存来降低延时（与系统内存交换信息是很慢的）
- 对于不同的线程块，不存在内置的同步方法

Constant Memory

目的：降低内存的traffic

最大为64KB