共享内存

cpu-内存 板级连接, 高速访存/纠错限制了电缆长度, 尺寸不会大(失败了), 但芯片内部还保留下来, cpu互连(qpi)

服务器集群

90s: 服务器集群, 网络互连, 共享协议:一条总线, 冲突监听, 实现简单 ==> 后改用 机器--交换机--交换机--机器, 不用冲突监听, 速度快(光纤), 利用率低, 胖树: 类似B树, 跨一层速度变成1/4, 新的技术: torus, 三维结构, 每隔1,2,4...相连, 通讯时间logP

节点越多越快? ==> 通讯时间

超级计算机

比较解稠密矩阵的速度O(n^1.5), n个元素的矩阵

加速器

gpu, knl

众核: 每个核比较简单, 核多, 之前核复杂, 但是执行C简单

指令集, cache, 流水线 都可以简化

加速器? 牺牲可编程性

异构计算? 机器里什么都有(fpga, cpu, gpu,...)

太湖之光: 每个cpu自带存储器, 需要主动调配, 每一个cpu 4个大核, 每一个大核64个小核, 结构和knh一样

knc(一个环) knl(网格, 有cache一致性, 但是性能差) knh(大核带小核)

编程很困难, 按照汇编语言思考

功耗 多核

bill daddy: 功耗与数据搬运有关. 比特币不需搬数据, asic就很快

L1cache? 局部性原理 L2cache? 多核共享变量(少数情况), 共享空间(有的占用多 有的少)

内存层级

pageable, pin(连续的内存地址, 分配之后os不会使用它, dma加上映射表), device memory(每个gpu不同), L2cache(fermi数据隔一层), constant memory(速度很快, 调用过程中不能更改), shared memory(L1cache, 多核共享), register(放的东西少)

访存规则性, 减少长程访存, 快

主机

没讲什么有用的, pci, 课程服务器架构

众核产品

tilera, fermi, scc,

服务器登陆, 编程手册

ssh 162.105.86.43 test 2018par

ssh 192.168.1.199 student 2018par

CUDA_C_Programming_Guide_4.2

老师在讲啥

VPU: 向量处理单元, 80s出现

intel KNC

多核, 向量化, 超长指令, 单核功耗有上限, 每一个核cache很大, 如何整合不同核的cache? 锁住, 写别的cache, L1指令cache, datacache

高通量计算? 延迟比较高, 利用率很高(流水线), 访存队列(延迟),

dram的频率? 芯片和处理器交换数据的速度, 不是存数的速度, dram也有cache

众核cachemiss多? 预取指令, 不用cache也行, 先发访存指令, 再用

```
for(int i = 0; i < 100; i += 1){
    a[i] = b[i] + c[i];
}
```

会有预取和向量化, 但是复杂起来就不行

```
for(int i = 0;  i < 100; i += 1){
    int j = k[i];
    a[j] = b[j] + c[j];
}
```

这就不行, 自己写预取指令, load但是不等待, 不影响语义, 也可以load不用

核多, 性能高 ==> cache效果差

两个流水线, 最好用两个以上的线程

ANTON

D.E.Shaw, 蛋白质折叠

512个核

分子动力学, 大量迭代

```
x = f(y, z)
y = f(x, z)
z = f(x, y)
```

减少延迟, 存储器可以减少, L2可以去掉, 网络接口直连芯片, 不用排队

toroslink, 三位连接, 网络直连芯片, 环状总线

并行编程

OpenCL

HPF(高性能fortran)

map函数式语言, 高阶函数

MapReduce

并行算法-瓶颈是什么?

SGEMM 单精度稠密矩阵乘法 O(n^{1.5})

DGEMM 双精度 O(n^{1.5})

FFT O(n log n) 通讯密集型, 带宽是瓶颈

NBody Brute force O(n^2) 树状方式组织粒子

小规模问题: 计算 / 访存

处理器个数增加? 总访存速度增加, 通讯, 蛋白质折叠? 延迟, 进入数据通道的延迟

gpu

和内存dma交换数据

整体程序在cpu上运行, 是串行程序, 会启动gpu的kernel

kernel<<<block, thread>>>

异构计算? 两种不同处理器计算

这节课

线程多? 不一定好

cuda: C语言不必关心cache

线程私有寄存器,

local memory, 编译时的一个概念, 物理上不存在

constant: 运行的时候不会变化, 但是每次运行开始前会变化

text memory: 二位cache, 一个点周围的点被读进来

GPU没有大的cache? 大量访存, 不同线程的数据共享很少

fma: cpu上, a*b* + *c* *和* *a*b的时间是一样的

芯片访存的延迟, dram里面也有一个小cache, 访存多就排队

warp: 32个线程, 所有GPU都是一样的

凝聚访问? 访问的内存是对齐的

```
if(x < 0){
    bar();
}else{
    foo();
}
```

如何执行? 两条指令所有线程都执行, 流水线是相同的, 只是有的人空转

错位访问内存就不凝聚, 访存会变成两条访问

x86精度会高点

```
dim3(32, 64)
```

二维的编号, 后一项相同的是同一个worp(一定是32个线程), 和c相反

```
dim3(64, 32)
```

后一项相同的分属两个worp

```
dim3(4, 64)
```

0, 0 => 3, 7 是同一个worp

是先排成线性, 再截断

同一个worp中, 同一个线程最好跳着访问数据, worp之间访存是顺序的

cudaMemcpy粒度? 大概8m会达到峰值性能

block规模和硬件有关, 有上限, block个数和问题的规模有关

所有线程共享代码, spmd(单程序多数据, 但是流水线的个数和线程差不多),

simd(单指令多数据流, 共用流水线), mimd(每一个线程自己的代码), stmd(worp之间是spmd的, 不同worp之间执行代码的不同点, 同一个worp之内simd)

block之间需要同步, 但是worp之内天然同步的

block之内线程可以合作, shared mem几十k, barrier sync都在一个点上

不同block之间很难合作, 但是volatile可以啊, 加上的话汇编加上一个load

block太多的话, 就先放上去一些

一个核组可以运行多个block

cudaMalloc开到globalmemory上, 没有分页的, 上面没有os

修饰符: *__device__* *和* __host__ 可以同时使用, 编译两次,

不能有函数指针, 没有本地变量, 参数不能变长

异步kernel调用? 调了kernel之后cpu还做事情

同步调用? 调完cpu等着

矩阵乘法, 优化

矩阵转置, 优化

解决流水线延迟? gpu是调用多个worp同时运行解决流水线延迟

内存层级

共享存储器可以用索引访问

```
__shared__ int a[16] //在共享存储器里面，一个block的线程都看得见

int a[16] //在寄存器里面，别人看不见

a[(int )x] //寄存器不能动态索引，共享存储器可以
a[7] //寄存器可以这样访问
```

但是寄存器访问比共享存储器快2-4倍, 最好只用它做通讯

共享存储器分路, 如果都想访问同一路, 就要排队

64并行就很快, 正好是两个worp, 但是要是有一个变元在shared mem里面, 就只能到60%

下面很重要

一些晶体管计算, 一些存储

没有存储的话, 做矩阵乘法就很吃力, 需要大量访存$O(n^3)$, 可以全读到芯片里面shared mem, 这样访存只有$O(n^2)$

KNL: 底层DDR-4 100+G/s, 芯片上堆叠的存储器, 400+G/s

GPU上面量就比较小, 需要充分利用, 如何做到? 分块, 控制粒度, 充分利用共享存储器

计算太快, 访存太慢, 如果都需要访存的话, 需要等很久

$\sum_i x[i] \times 2$, 每一次运算只有0.23个字节传进来, 但是计算一次需要2字节的数据(乘加算两个运算) ==> 计算比

访存要重复利用, 4 / 0.23, 需要18次重复利用

优化? 分块, 数据放到shared mem里面, 或者分块放到片上

在A里面取一行, B里面取一列, 读入data = 4(n + m)byte, 计算进行2mn flops(乘加算作两次), comp / data = $\frac{1}{2(\frac{1}{m}+\frac{1}{n})}$, 之前的 comp / data = 1 / 4, 比值达到$\frac{2}{\frac{1}{m}+\frac{1}{n}}$

利用分块, 有暂存机制, 可以减少重复访问的次数, 比方说取 m = 64, n = 16, 达到26次重用

重用越多越好? 计算密度成为瓶颈了, 调参

cuda.ppt p7

计算$C = C + \alpha A + \beta B$

一个block64个线程, 每一个线程进行16个元素计算, 粒度

读一行是凝聚的, 但是读一列是跳着的, 不好, 一次读多行

把B的一块放到shared里面, 每一次读四行(ldb是一行的宽度), 这样多个线程就是凝聚访问

线程太多的话寄存器就少了

```
#pragma unroll //循环展开，循环变量编译时可知，很快，还要把数据相关拉开
```

为什么要同步? 读数据读到共享存储器里面, 进来的数据可能是别人用的, block内线程同步, 第二个同步保证转回来的时候不会产生坏数据

16 * 17? 错开一位, 减少bank conflict

ppt cuda p7

必须掌握

```
bs[16][17] // bank conflict，一次有32个线程访问shared mem，访问整个地址空间，0-31 属于32个bank，按cache理解就好，如果同时访问一个bank，需要排队，错开就很少排队了，但是所有人都查一个也不用排队
```

block中shared mem增加的话, 能调度的block数就减少了

增加block中线程数, 局部性差了, 访存延迟解决不了了

要有两个以上的block运行的话, 一个block访存的延迟被另一个block掩盖住了

现在在汇编上写了...

66%上限? 当时有1/3的单元在干别的事

Chien, TsingHua Univer.(ROC)

寄存器里面的数据读起来是最快的

上面的程序里面每一个B进行了16次计算(和16个A的元素乘)

反汇编写到寄存器里面又快了

compute capablility: 版本, block个数, therad个数

constant mem更快, 但是就发热了

其他的并行思想

- message passing
- mem sharing

SPMD, 单程序, 多数据, MPI

mpi安装在不同机器上, 用mpi-run运行

init之中所有机器握手, 知道自己是第几号进程, mpi_comm_rank获得自己的进程号

接受的长度一定要超过发送的最大长度

send 和recieve都是等待, 直到被接收/收到消息, 才能向下进行, 所以算是一个同步的过程

isend的话, 发出去就走了

irecieve的话, 看一下就走了, 但是之后要wait()(用test()实现), 保证能收到, 并且通讯和计算重叠

```
for (source = 1; source < p; course += 1){
    //
}
```

这样的话即便后面的线程消息到了也要锁在哪里

如果想要按消息到达的顺序接受, 用ANYSOURCE

tag 类似对讲机的通道, 如果收到不同通道的消息, 发送者挂起, 也有ANYTAG

收到的消息的tag和source都在states里面

下面的机制是轮询, 中断反而慢一点

RDMA: 表面上是邮件, 实际上是 cpu - mem - eth0(cache, 可以DMA) - ... - eth0(cache) - mem - cpu, 远程DMA, 先握手(TCP/IP)交换安全码, 有一部分mem不做分页机制(pin/page_lock), 做一个registration, 此后和对方的cpu没有关系了

put get就可以了, 单边通讯, 传输的内容先到网卡的cache里面, 再DMA到内存里面, 可以写对方的内存, 这一个过程对方的cpu并不知道, 对方的cpu可以查queue pair, 轮询它

如果send和recieve不配对就死锁了

双边通讯建立在单边通讯之上, 先把对方的地址传过去, 之后RDMA, 就不用memcpy了

小的消息先打包, 传到pin里面, 再解包, 放到用户指定的位置, 这就有一个memcpy

mpi_test相当于pooling

```
Isend()
wait() // ==> send()
```

```
wait() // ==> while(){test()}
```

所以Isend, Ireceive, test就可以表示所有的例程

一个线程计算, 一个线程做pooling也可以

广播: 进程把信息传到所有进程, 所有进程都要执行这个指令, 把root数据段传到所有进程

reduce: 看ppt就好了, 基本上有交换律

如果send, receive传到本地地址, 直接做内存拷贝

scatter: 不同的数发给不同的进程, 实现上是一个二叉树的结构

all_gather: 所有数据给所有人

广播: p-1个消息, logp可以做完, 但是因为阻塞, 时间会长一点

all_gather: 每一个进程同时未完成的消息个数常数量级(小于8)就很好

all_to_all: 相当于做矩阵转置, 有一定粒度的转置

两个线程交换数据需要send_receive, 或者isend, 如果同时send, 死锁了

每一轮交换数据, 先和与自己异或1的数据交换, p个消息, 再异或2...

所以还是p^2的数据量

控制发送的数据不超过未接受的数据的+4或+8就很好了

CSP, 讲什么呢, 不动点方程的解, 无上界, 但不是无穷

看这个吧

```
/* 方程的解 */
COPY = μX* (in.0 -> out.0 -> X
            | in.1 -> out.1 -> X)
```

和

```
/* 平凡写法 */
COPY = (in.0 -> out.0 -> COPY
        | in.1 -> out.1 -> COPY)
```

一样

$COPY = |_{i<2} in.i -> out.i -> COPY$

$COPY = in?n -> out!n -> COPY$

这样就有值的计算了

画图也可以

类比到自动机也可以

x: A -> P(x) = y: B -> Q(y) \equiv (A = B & \forall x:A * P(x) = Q(x))

μX*F(X)* = *F*(μX*F*(X))

CCS? 操作语义

trace

```
<> 是单位元，^ 有结合律

traces是trace的集合

traces(coin -> STOP) = {<>, <coin>}

traces(μX* tick -> X) = {<>, <tick>, <tick, tick>, ...} = {tick}^*

{<>, <a, b>} 也不行，不能一次走两步，前缀封闭(prefix closure)

打括号里面还是写进程，不要写方程

{s | 存在n, s.t. s <= <coin, choc>^n}

traces(c -> P) = {<>} ∪ {<c> ^ t | t \in traces(P)}

traces(c -> P | d -> Q) = {t | t = <> or (t_0 = c, t` \in traces(P)) or (t_0 = d, t` \in traces(Q))}
```

完备性?

并行? P || Q, 定义成共识, 互相同意

GRCUST = coffee -> GRCUST | choc -> GRCUST | coin -> choc -> X

(GRCUST || VMCT) = μX*(coin -> choc -> X), 相当于取了一个交集

VMC: 投两磅出大的, 一磅出小的,

FOOLCUST = in2p -> large -> FOOLCUST | in1p -> large -> FOOLCUST

(FOOLCUST || VMC) = μX*(in2p -> large -> X | in1p -> STOP), 就死锁了

STOP是并行的零元, 并行也有结合律 RUN_αP: 每一步都可以进行所有的动作, 就是单位元了

c-> P || c -> Q = c -> (P || Q)

c -> P || d -> Q = STOP

(x | A -> P(x)) || (y | B -> Q(y))= (z | A ∩ B -> P(z) || Q(z))

哲学家就餐

α哲学家: 坐下, 站起, 拿起左叉 拿起右叉, 拿起右叉 拿起左叉

α叉子: 被左拿起, 被左放下, 被右拿起, 被右放下

哲学家: 坐下 -> 左叉 -> 右叉 -> 放左叉 -> 放右叉 -> 走人

叉子: 被右拿起 -> 被右放下 | 被左拿起 -> 被左放下

并行起来, 死锁了, 但是可以控制让哪些哲学家站起来, 哪些坐下

一个调度: αFOOT = {i.sitdown, i.getup}

U = {有人getup}, D = {有人sitdown}

FOOT_0 = {x : D -> FOOT_1}

FOOT_j = {x : D -> FOOT_j+1 | y : U -> FOOT_j-1}

FOOT_4 = {x : U -> FOOT_3}

有人坐下脚标就加一, 有人站起脚标就减一, 最多n - 1个人坐下, 就能吃饭了

非确定性

_ | | 这个符号, 是一个内部选择, 不知道怎么选

ch5d = in5p -> 1 + 2 + 2 or 1 + 1 + 1 + 2 -> ch5d

幂等律: p or p = p

交换律, 结合律, 分配律

f(p or q) = f(p) or f(q)

(p_1 or p_2) || (q_1 or q_2)

μX*(a -> X or b -> X) != μX*(a -> X) or μX*(b -> X), 左面是abababab, 右面是aaaa or bbbb

内部选择就不能用状态机表示了

发送信息 c!v

接受消息 c?x

```
W(var) value
R(var) value // 这一次读出value，下一次不一定

x = x + 1 ==> P1: R(x) 0, W(x) 1
```

严格一致性: 任何读操作都读到最近一次写的结果, 依赖于全局时钟(物理时钟)

如果读写不是原子操作的话, 可能出现

```
W(x) 1
--------------------
       R(x) 0 R(x) 1
```

如果放一个互斥锁, 那就能严格一致了

顺序一致性: 任何一次并发 **执行** 顺序结果都一直, 好像操作按照顺序进行, 可以构造一个全序

```
W(x) 1
R(x) 1 R(x) 2
R(x) 1 R(x) 2
W(x) 2
```

就是有序的, 可以认为如下执行

```
W(x) 1
        R(x) 1          R(x) 2
        R(x) 1          R(x) 2
                W(x) 2
```

但是下面的就不行

```
W(x) 1
R(x) 1 R(x) 2
R(x) 2 R(x) 1
W(x) 2
```

顺序一致性应用: dijkstra, 反复运行, 反复更新且其他线程见过我终止的状态, 进入终止准备状态

弱一致性: 对同步变量的访问时顺序一致的, 之前的所有写操作完成之后才能读同步变量, 之前的所有读操作完成之后才能写同步变量

```
W(x) 1 W(x) 2
R(x) 0 R(x) 2 s R(x) 2
R(x) 1 s R(x) 2
```

s是同步指令, 同步指令之后, 读到的值就是准的, 写的操作没有义务广播出去, 没有同步指令, 最新的修改可能看不到, 读操作之前同步一下就好了, on demand

因果关系, 看信号量就好了, 假如有因果关系, 看到的顺序就一致, 如果没有因果关系, 看到的顺序随便了, 只是一个偏序

满足顺序一致一定因果一致

强一致性 > 顺序一致性 > 因果一致性

因果一致性可以没有全序, 但是偏序是有的, 没有因果, 看到的顺序随意

```
P1  w2   w1 // 有偏序

P1  w2
P2  w1  //只有偏序，别人看到的顺序随意
```

□是"总是"的意思, 时间线性离散的

```
HC => \Box HC_{ini} // 指时钟总在1-12之间
```

$HC_{ini} : \{< hr_i > | hr_0 \in \{1...12\}\}$

$\Box HC_{ini} : \{< hr_i > | hr_i \in \{1...12\} \ \forall i\}$

$HC := HC_{ini} \wedge \Box[HC_{nxt}]_{br}$

什么叫□? 譬如说HC_{ini}满足初始值小于12, 加一个□意味着在未来的所有时间点, 都有初始值小于12

需要rdy变量因为可能多次发送同一个值, 看到ready = 1之后把acknowledgement = 1(rdy, ack在01间翻转)

asynchinterface, 看看ppt就好了

unchanged(ack) : ack' = ack

注意到
$Next = (rdy = ack \wedge S) \vee (rdy \neq ack \wedge R) = (rdy = ack \vee R) \wedge (rdy \neq ack \vee S) = (rdy = ack => S) \wedge (rdy \neq ack => R)$

菱形: 未来存在一个时间点满足后面的命题

方块: 所有未来的时间点都满足后面的命题

方块+菱形: 类似下极限的理解