COMP130014.02 编译

第十四讲：并行和优化

徐 辉
xuh@fudan.edu.cn

大纲

- 一、并行指令特化
- 二、并发程序和优化
- 三、GPU协同计算

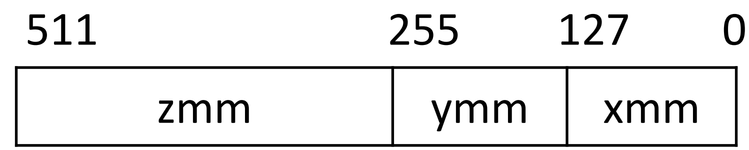
一、并行指令特化

并行指令特化：SIMD

- SIMD：Single Instruction Multiple Data
  - X86：SSE、AVX
  - ARM：Neon
- 典型应用场景：向量运算
- 编译时通过-m声明指令集类型，如-mavx2

X86（SSE/AVX）

- AVX扩展：Advanced Vector Extensions
  - 寄存器：ZMM0-ZMM31 (512bit)



- 指令：
  - 浮点数运算相关指令
    - Scalar模式：MOVSS、ADDSS、SUBSS
    - Packed模式：MOVAPS、MOVUPS、ADDPS、SUBPS...
  - 整数运算相关指令
    - PEXTRW、PINSRW、PMULHUW、PSADBW、PAVGB...

示例：向量运算

```
struct Vec {
    float a, b, c, d;
} x,y;

struct Vec avadd() {
    struct V z;
    z.a = x.a + y.a;
    z.b = x.b + y.b;
    z.c = x.c + y.c;
    z.d = x.d + y.d;
    return z;
}
```

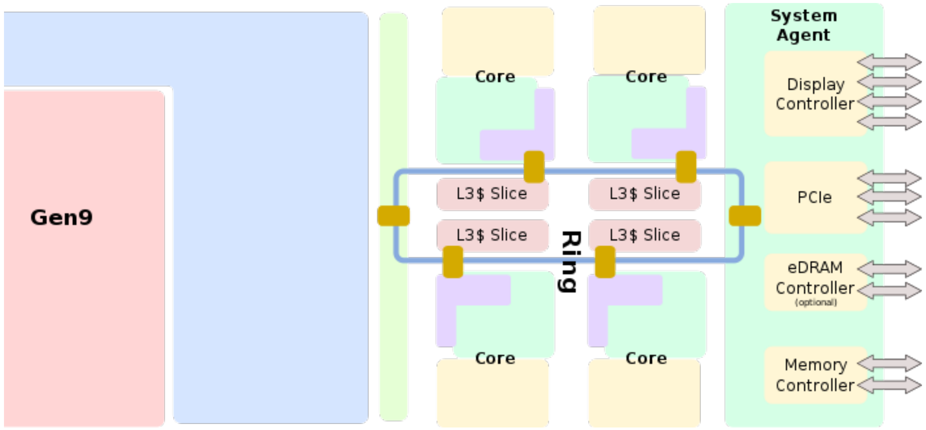
```
#: clang -mavx2 avx1.c -O2
```

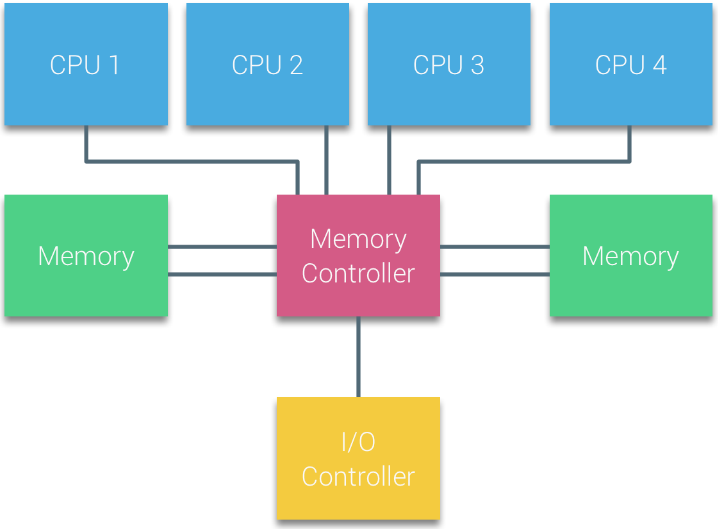
```
vmovups xmm0,XMMWORD PTR [rip+0x2f14]
vaddps xmm0,xmm0,XMMWORD PTR [rip+0x2f1c]
vpermilpd xmm1,xmm0,0x1
ret
```

二、并发程序和优化

并行计算

- 并行计算架构
  - 多核处理器（multicore）
    - 多线程并行计算
  - 多CPU（multiprocessor）
    - UMA：cache coherence
    - NUMA
  - 分布式系统
- 关键问题：
  - 任务分解：数据分块
  - 数据更新同步





https://en.wikichip.org/wiki/intel/microarchitectures/skylake_%28client%29

主要工具

- OpenMP：用于共享内存
  - 自动创建多线程
  - 线程同步、内存屏障
- MPI：Message Passing Interface
  - 用于分布式计算环境
  - 自动创建多个进程
  - 基于socket同步数据

并行性能提升上限：Amdahl's Law

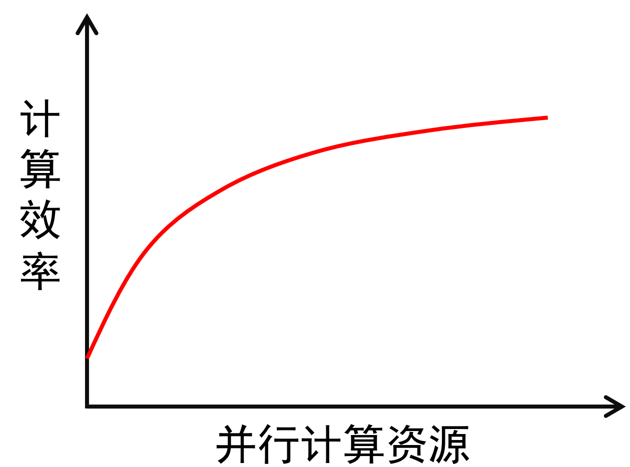
- 如果单线程运行一个任务需要n小时
- 其中可以并行的部分耗时m
- 如果增加线程数至s，则所需时间为：(n-m)+m/s
- 效率提升：

$$\frac{n}{(n-m)+m/s}$$

- 另p=m/n，则

$$\frac{1}{(1-p)+p/s}$$

$$\lim_{s\to\infty}\frac{1}{(1-p)+p/s}=\frac{1}{1-p}$$



并发程序架构

数据竞争问题：多线程的例子

- 加法不是原子操作：load-add-store (多条指令或micro ops)

```
#define NUM 100
int global_cnt = 0;

void *mythread(void *in) {
    for (int i=0; i<NUM; i++)
        global_cnt++;
}

int main(int argc, char** argv) {
    pthread_t tid[NUM];
    for (int i=0; i<NUM; i++){
        assert(pthread_create(&tid[i], NULL, mythread, NULL)==0);
    }
    for (int i=0; i<NUM; i++){
        pthread_join(tid[i], NULL);
    }
    assert(global_cnt==NUM*NUM);
}
```

多个线程并发访问

assertion fail

实验需关闭优化

原子访问

方式一：声明为原子变量类型

```
#define NUM 100
atomic_int global_cnt;

void *mythread(void *from) {
    //__atomic_fetch_add(&global_cnt, 1, __ATOMIC_SEQ_CST);
    for (int i=0; i<NUM; i++)
        global_cnt++;
}

int main(int argc, char** argv) {
    pthread_t tid[NUM];
     for (int i=0; i<NUM; i++){
        assert(pthread_create(&tid[i], NULL, mythread,
NULL)==0);
    }
    for (int i=0; i<NUM; i++){
        pthread_join(tid[i], NULL);
    }
    assert(global_cnt==NUM*NUM);
}
```

方式二：使用原子运算API

原子类型/访问的实现方式：原子指令

- X86架构：指令翻译时使用带lock前缀的指令
- ARM架构：ldrex/strex指令

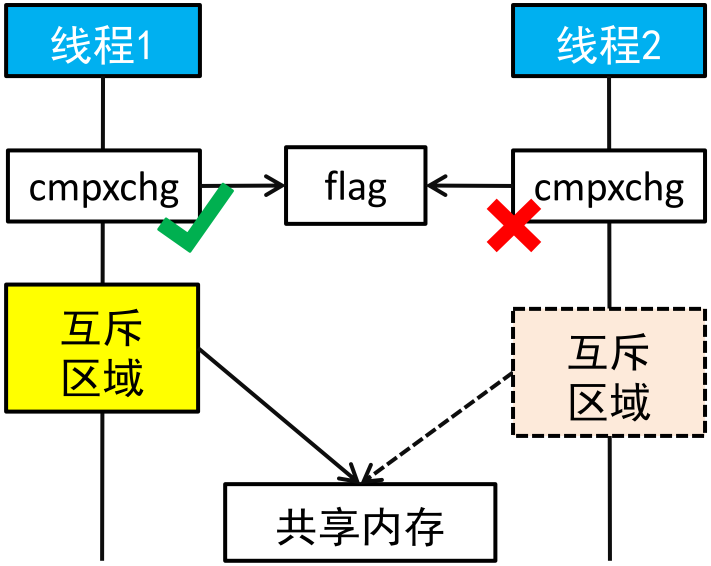
```
gef➤  disass mythread
Dump of assembler code for function mythread:
     0x00401150 <+0>:    push   rbp
     0x00401151 <+1>:    mov    rbp,rsp
     0x00401154 <+4>:    mov    QWORD PTR [rbp-0x10],rdi
     0x00401158 <+8>:    mov    DWORD PTR [rbp-0x14],0x0
     0x0040115f <+15>:   cmp    DWORD PTR [rbp-0x14],0x3e8
     0x00401166 <+22>:   jge    0x401182 <mythread+50>
     0x0040116c <+28>:   lock   add DWORD PTR [rip+0x2ed0],0x1
     0x00401174 <+36>:   mov    eax,DWORD PTR [rbp-0x14]
     0x00401177 <+39>:   add    eax,0x1
     0x0040117a <+42>:   mov    DWORD PTR [rbp-0x14],eax
     0x0040117d <+45>:   jmp    0x40115f <mythread+15>
     0x00401182 <+50>:   mov    rax,QWORD PTR [rbp-0x8]
     0x00401186 <+54>:   pop    rbp
     0x00401187 <+55>:   ret
```

如何实现一段连续指令的原子性

- 基于Compare and Set/Swap机制
  - X86_64：cmpxchg指令
  - 语言API支持：atomic_compare_exchange_strong

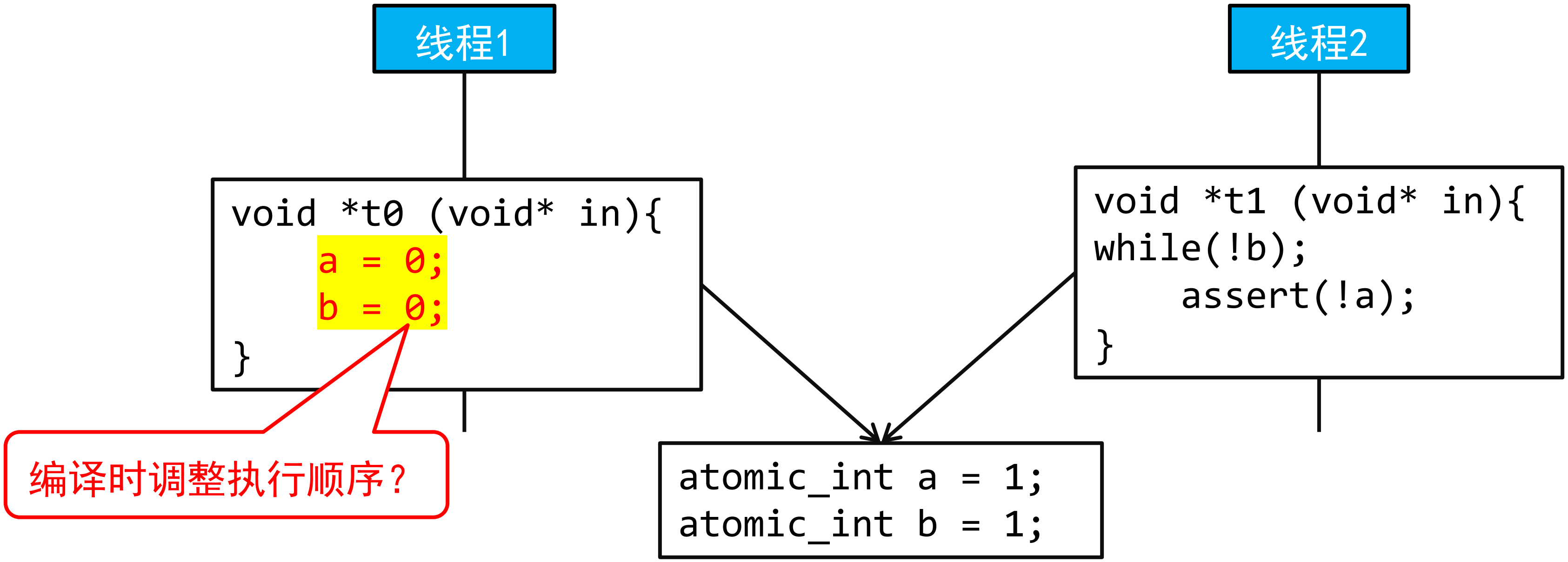
```
#based on rax
lock cmpxchg dst src
```

```
含义:
if(dst == eax) {
    dst = src;
    ZERO_FLAG = 1;
}
else {
    eax = dst;
    ZERO_FLAG = 0;
}
```



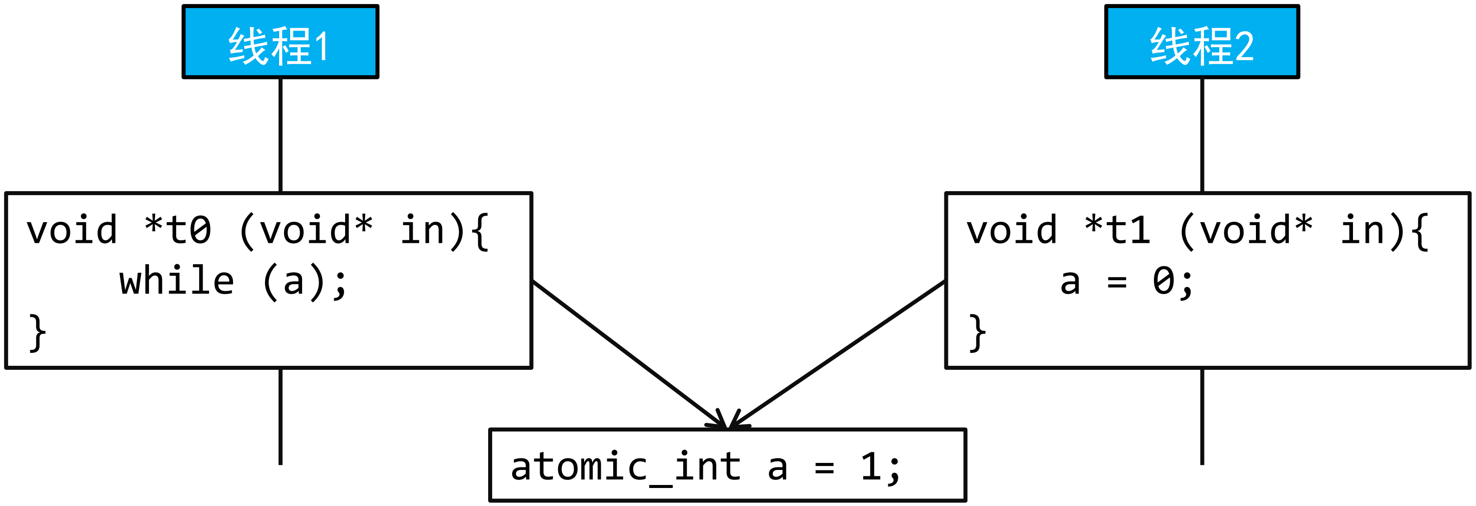
指令重排问题

- 指令执行的先后顺序不同会对其它线程产生影响



指令重排

- 编译优化可能会误将指令重排

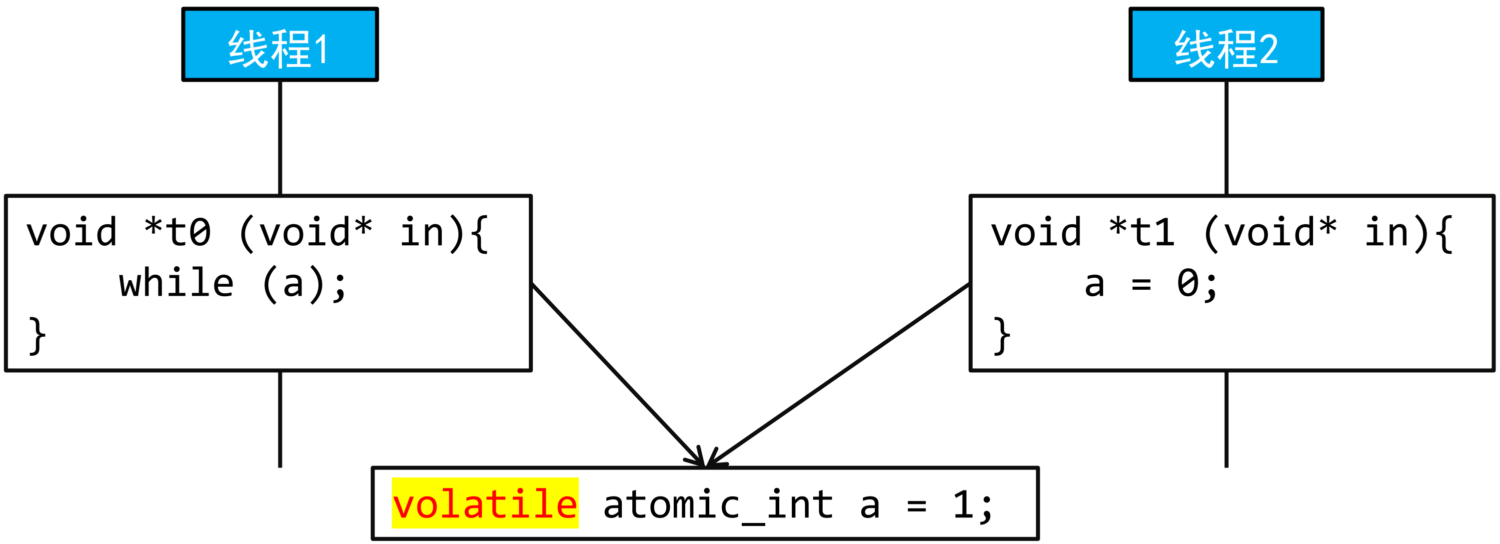



```
0x00401150 <+0>:    cmp    DWORD PTR [rip+0x2ee9],0x0  # <a>
0x00401157 <+7>:    je     0x401162 <t0+18>
0x00401159 <+9>:    nop    DWORD PTR [rax+0x0]
0x00401160 <+16>:   jmp    0x401160 <t0+16>
0x00401162 <+18>:   ret
```

死循环

易变内存访问：Volatile

- 丢弃寄存器中的值，重新从内存加载



内存屏障：Memory Barrier/Fence

- Happens-before：编译器确保屏障之前的指令在其之前完成
- 丢弃通用寄存器中的值，重新从内存加载
- 并发控制：mfence/sfence/lfence（CPU实现）

```
//空指令
#define barrier()  __asm__ __volatile__("":::"memory");
```

线程1

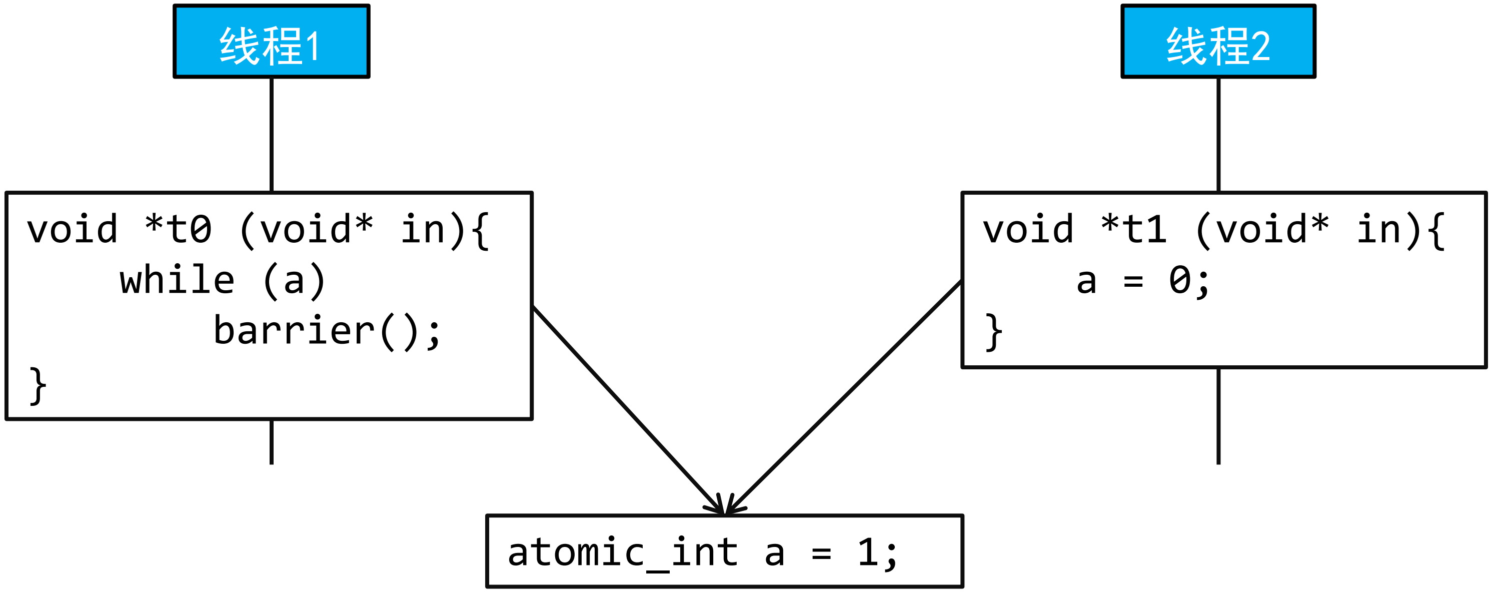
```
void *t0 (void* in){
    a = 0;
    barrier();
    b = 0;
}
```

线程2

```
void *t1 (void* in){
while(!b);
    assert(!a);
}
```

```
atomic_int a = 1;
atomic_int b = 1;
```

使用内存屏障


线程1
线程2
void *t0 (void* in){
while (a)
barrier();
}
void *t1 (void* in){
a = 0;
}
atomic_int a = 1;

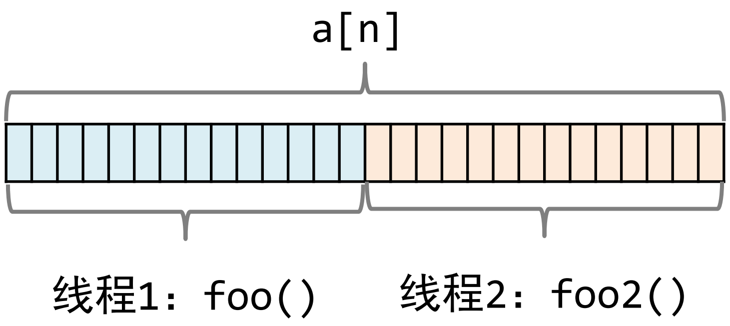
放松同步要求：Memory Ordering

- Sequential consistency
  - 默认模式，最强同步模式
- Acquire-release
  - 一般用于锁的实现
  - Release：当前线程的读和写操作保证在store fence前完成
  - Acquire：当前线程的读和写操作保证在load fence后开始
- Relaxed
  - 没有同步要求，仅保证原子性

```
//mfense/lfence/sfence
#define barrier()  __asm__ __volatile__("mfence":::"memory");
```

传统多线程

```
int test(){
    int a[n];
    for (int i = 0; i < n; i++) {
        a[i] = 2 * i;
    }
    return 0;
}
```




```
int paratest(){
    pthread_t t1;
    pthread_t t2;
    pthread_create(&t1, NULL, foo1, NULL);
    pthread_create(&t2, NULL, foo2, NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    return 0;
}
```

```
int a[n];
foo1(){
    for (int i=0; i<n/2; i++)
        a[i] = 2 * i;
}
foo2(){
    for (int i=n/2; i<n; i++)
        a[i] = 2 * i;
}
```

OpenMP应用举例

```
int test(){
    unsigned long long start = rdtsc();
    int a[100000];
    #pragma omp parallel for num_threads(2)
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }
    unsigned long long cycles = rdtsc()- start;
    printf("cycles = %d\n", cycles);
    return 0;
}
```

Assembly Code

```
push    rbp
mov     rbp, rsp
sub     rsp, 61A80h
mov     rax, offset _omp_outlined_
mov     rdi, offset unk_404058
mov     esi, 1
mov     rdx, rax
lea     rcx, [rbp+var_61A80]
mov     al, 0
call    ___kmpc_fork_call
xor     eax, eax
add     rsp, 61A80h
pop     rbp
retn
```

```
void __kmpc_fork_call (
       ident_t * loc, //源代码信息
       kmp_int32  argc,
       kmpc_micro  microtask,
       ...
)
```

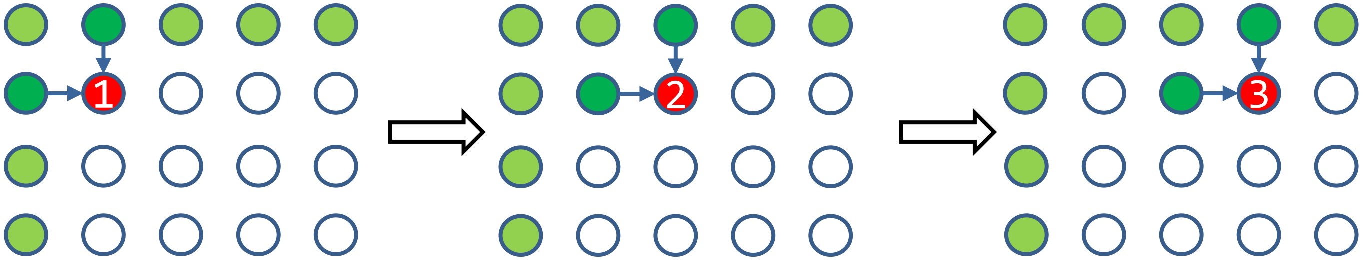
循环中的数据依赖问题：示例1

```c
int test(){
    int a[20];
    a[0] = 1;
    a[1] = 1;
    #pragma omp parallel for num_threads(4)
    for (int i = 0; i < 20; i++) {
        a[i] = a[i-1] + a[i-2];
    }
    return 0;
}
```

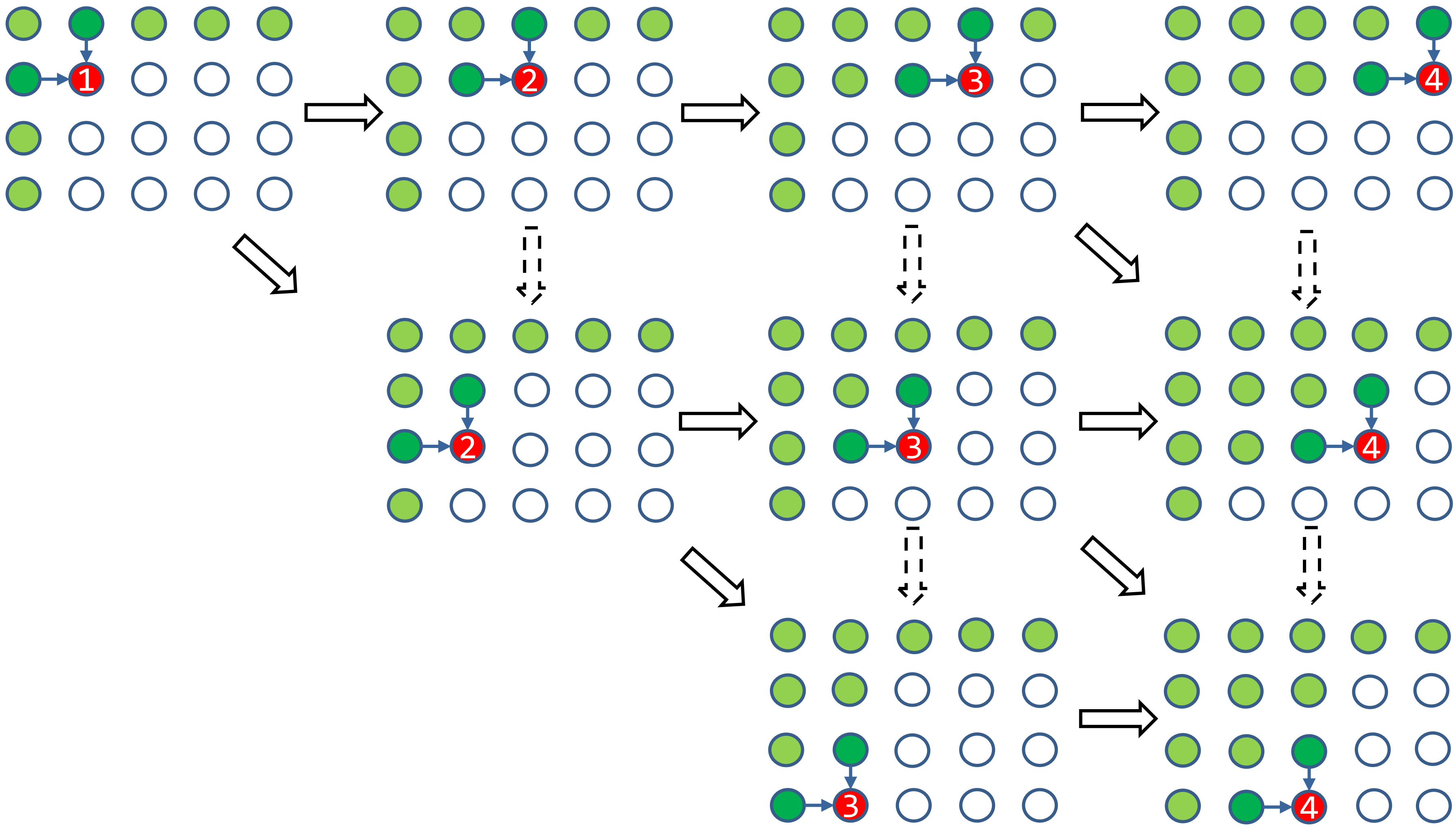
数据依赖，无法并行

循环中的数据依赖问题：示例2

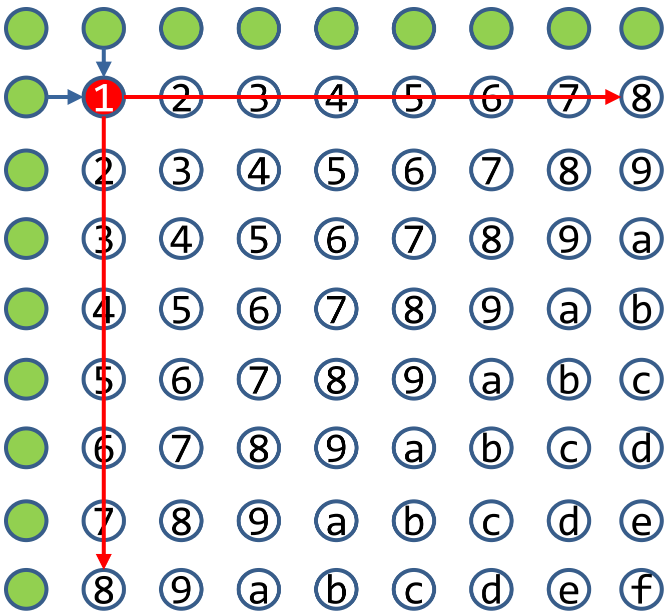
```
for(int i = 1; i < n; i++) {
    for(int j = 1; j < n; j++) {
        a[i][j] = a[i-1][j] + a[i][j-1];
    }
}
```



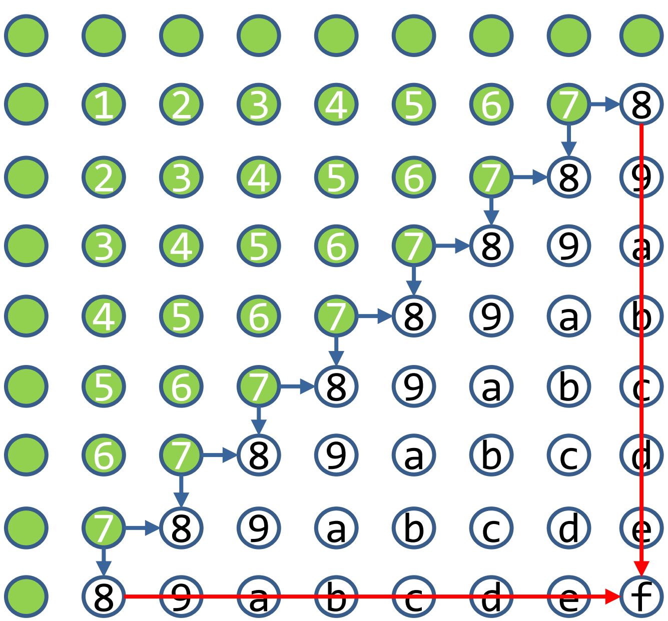
依赖分析



依赖优化：Polyhedral model




```
for(int i = 1; i < n-1; i++) {
    for(int j = 1; j < i+1; j++) {
        a[i-j+1][j] = a[i-j][j]
                    + a[i-j+1][j-1];
    }
}
```




```
for(int i = 1; i < n-1; i++) {
    for(int j = n-1; j > i-1; j--) {
        a[j-i+1][j] = a[j-i+1][j]
                    + a[j-i][j];
    }
}
```

三、GPU协同计算

GPU架构

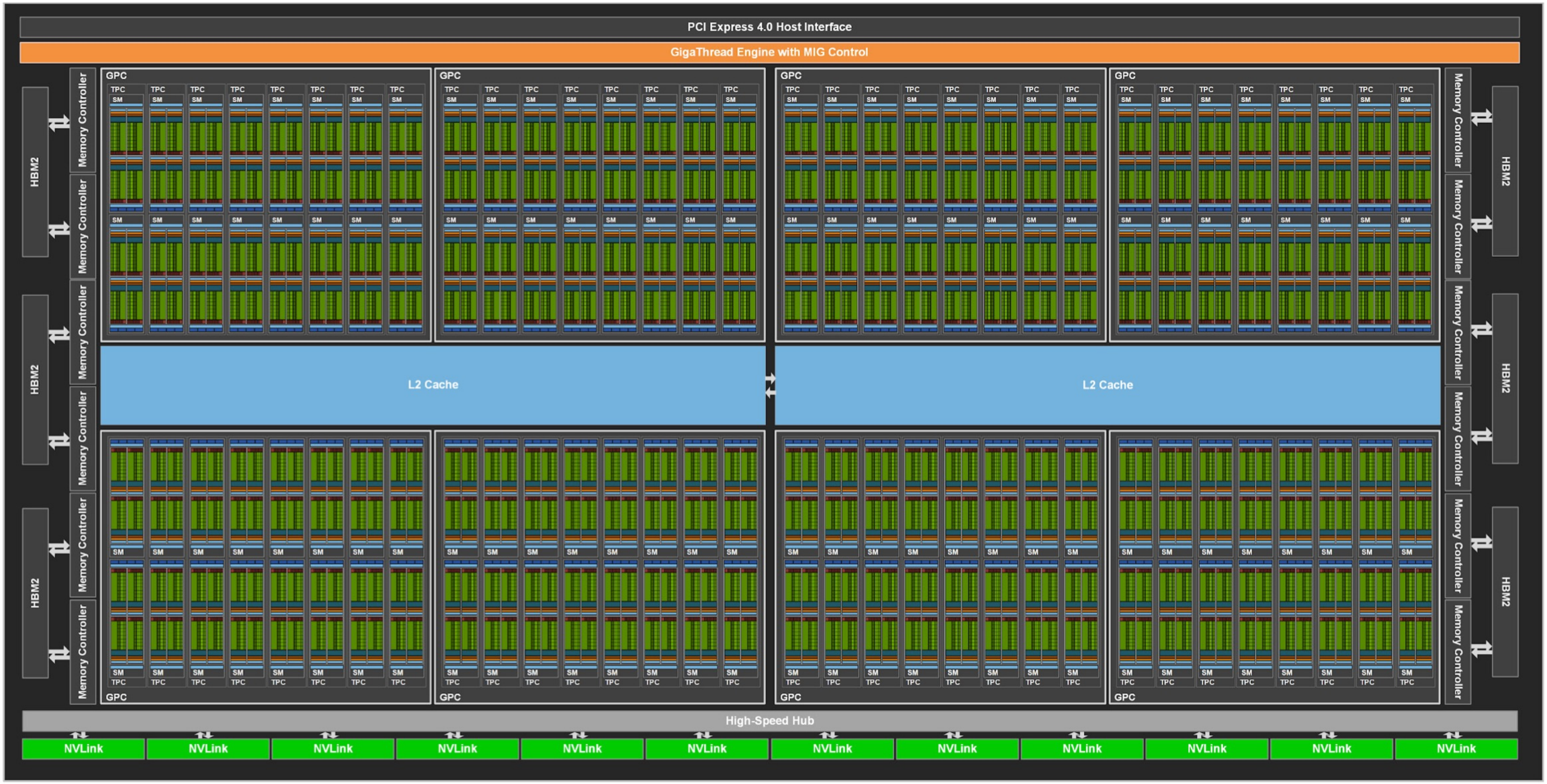


Figure 6. GA100 Full GPU with 128 SMs (A100 Tensor Core GPU has 108 SMs)

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

GPU架构：Streaming Multiprocessor

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

GPU/CUDA基本概念和术语

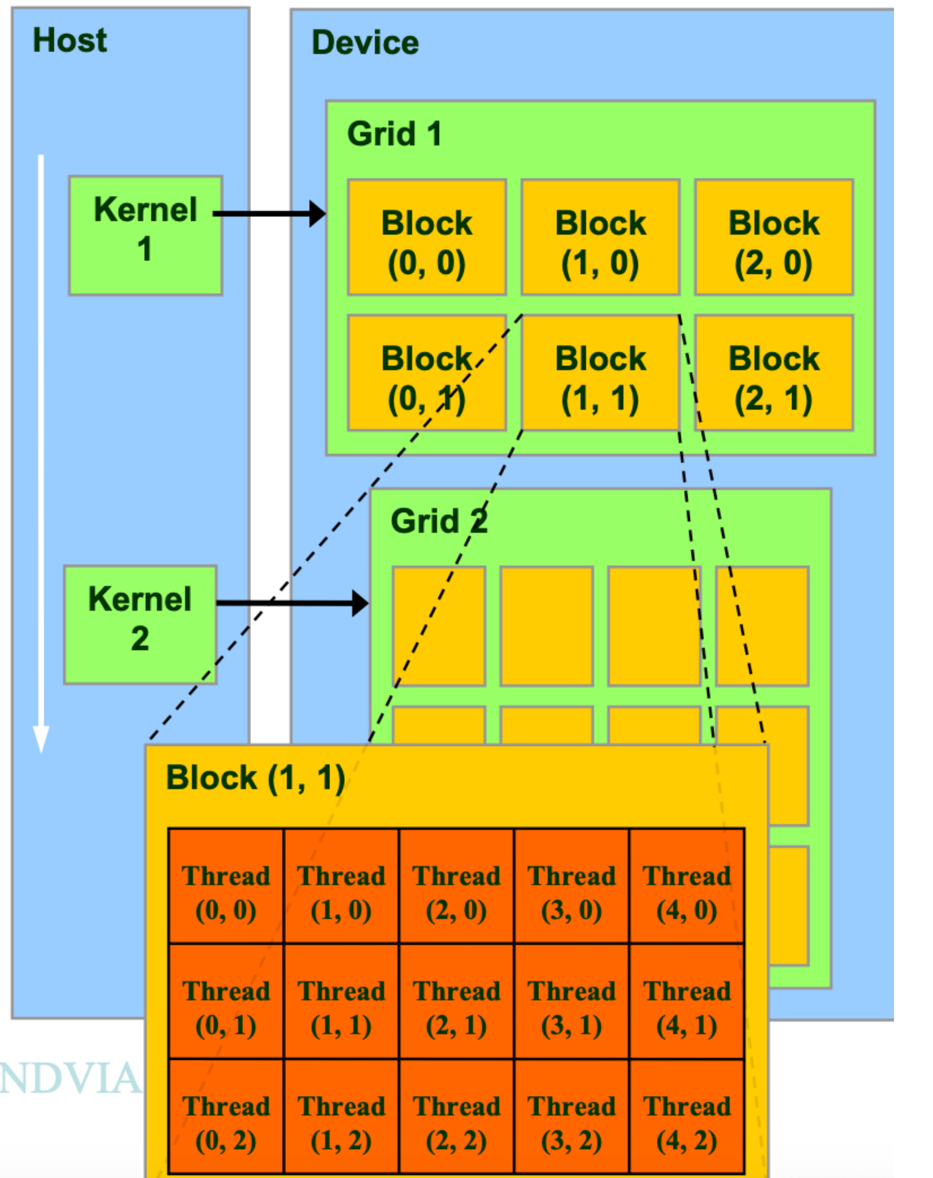
A kernel is executed as a grid of thread blocks

- All threads share data memory space

A thread block is a batch of threads that can cooperate with each other by:

- Synchronizing their execution
  - For hazard-free shared memory accesses
- Efficiently sharing data through a low latency shared memory

Two threads from two different blocks cannot cooperate



Courtesy: NDVIA

https://sites.cs.ucsb.edu/~tyang/class/240a17/slides/TopicGPU1.pdf

代码示例

```
__global__ void vectorAdd(const float* A, const float* B, float* C, int N) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < N) { C[i] = A[i] + B[i]; }
}

int main() {
    ...
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size); //分配GPU内存
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice); //数据传输：CPU=>GPU
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    //在GPU上调用内核函数
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost); //结果传输：GPU=>CPU
    ...
}
```