[McCool 2012] Michael McCool,Arch D. Robison,James Reinders. “Structured Parallel

Programming Patterns for Efficient Computation”. Elsevier B.V. 2012.

# 对时间复杂度提出挑战

跨度 执行依赖

通信 内存 全局 < 组共享 < 局部 同步？ Reduce？

执行依赖 操作系统中的前驱图

Map（映射）

Reduce（归约）

# 工作窃取调度器

负载平衡 又称分散调度器 相对于中心调度器

# 指令级并行（ILP）

超标量（superscalar）体系结构

# 串行陷阱（Serial Trap）

串行陷阱：源自串行执行的假定的不必要的假定

# 算法策略模式（Algorithm Strategy Pattern）/算法骨架（Algorithm Skeleton）

# FIGURE 1.11 Overview of parallel patterns.

数据并行（Data Parallelism）

规则数据并行（Regular Data Parallelism）

线程并行（Thread Parallelism）

区分操作系统线程和硬件线程

向量并行（Vector Parallelism）

SIMD

SSE （x86 x86\_64）

NEON（armerbi-v7a arm64-v8a）

SIMT

Direct3D Vulkan 计算着色器

# 计算着色器（SIMT）属于向量并行

在分支较多的情况下，线程并行比向量并行有优势，因为指令寄存器独立

向量并行模拟线程并行较难，尤其在有分支的情况下

掩码（Masking）

假定向量并行支持 条件赋值 指令

打包（Packing）/流压缩（compaction）

类似于shuffle

将不同分支的元素分别打包到两个不同的向量

打包通常被认为可以将同一向量同时作为输入和输出，常将不同分支分别打包到向量的两端

查看HLSL汇编可知 条件运算符（? :）（汇编movc）比分支语句（if）（汇编 if）更高效

指令并行

单个核心有多个功能单元

超标量（superscalar）

硬件多线程

向量指令 SSE NEON 即SIMD（单指令多数据流）

内存层次

类似计算着色器中 局部 组内共享（shared） 设备（SRV UAV）

# 性能理论

延迟（Latency）完成单个任务所需的时间 单位 秒

吞吐量（Throughput） 在单位时间内完成的任务数 比特/秒

一些提升吞吐量的并行模式可能会增加延迟

比如串行阶段的流水线，

# 阿姆达尔（Amdahl）定律

# 古斯塔夫森(Gustafson)定律

# 省电

移动游戏应当并行？

# 工作跨度（Work Span）模型

Span = 关键路径（Critical Path）

# 陷阱

## 竞争条件

顺序一致性不一定成立

同一线程内 指令级并行 可能重新排序 可以用Memory Fence阻止

## 互斥量和锁

互斥量 可以保证 顺序一致性

## 扼杀伸缩性

## 缺乏局部性

## 负载不平衡

## 开销

# 控制模式

## 分叉结合（Fork-Join）

Clik Plus中

spawn 控制流分歧

sync 控制流结合

类似异步IO

支持嵌套？？？

## 映射（Map）

循环的一种（迭代次数事先已知，每次迭代都是独立的，计算仅依赖于迭代次数，数据读取使用迭代项作为容器的索引）

类似于计算着色器

不同的函数调用访问独立的数据元素

元素函数（elemental function）

向量 元素函数 等价性

元素函数 形参

常量 uniform //CBV SRV UAV

变量 varying //DispatchID

### TBB

float y[2000] = {};

tbb::parallel\_for(

tbb::blocked\_range<int>(0, 2000 ,4),

[&y](tbb::blocked\_range<int> r)//Tiling

{

for (int i = r.begin(); i != r.end(); ++i)//Tiling为SIMD向量化提供可能

{

y[i] = y[i] + 1.0f;//元素函数

}

}

);

### HLSL 计算着色器

代码融合（Code Fusion）

缓存融合（Cache Fusion）

TBB 和affinity\_partitioner

## 工作桩（Work Pile）

元素函数可以产生新的工作

工作量事先并不确定，难以向量化

## 模板（Stencil）

像素着色器 ddx ddy

允许访问neighbour 可以使用tiling优化实现

需要防止超出边界！！！

## 集合(Collective)操作

组合器（Combiner）函数

### 归约（Reduce）

时间复杂度log2n 相对于串行，加速比为n/log2n

使用Tile优化

树形实现 组合器（Combiner）函数应当是可结合的（Associative）

浮点数累加——树形实现的精确度会提高 或者可以认为在严格意义上浮点运算并不是可结合的

#### SSE

SSE（4路SIMD）

向量化 组合器（Combiner）函数应当是可交换的（Commutative）

#include <xmmintrin.h>

alignas(16) float a[12] = { 1,2,3,4,5,6,7,8,9,10,11,12 };//输入数组

//4路SIMD

\_\_m128 tempv1 = ::\_mm\_setzero\_ps();//初始值0

for (size\_t i = 0U; i<12U; i += 4U)

{

tempv1 = ::\_mm\_add\_ps(tempv1, ::\_mm\_load\_ps(a + i));

}

//归约

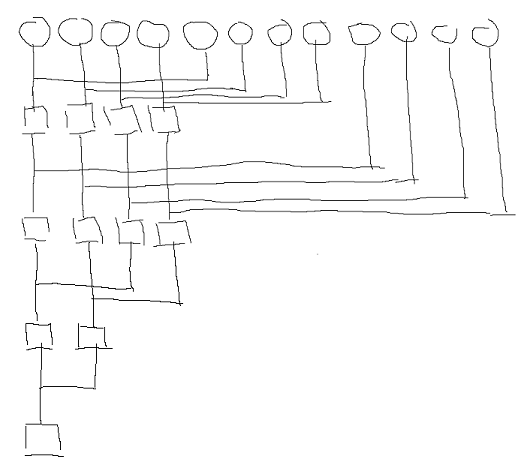
\_\_m128 tempv2 = ::\_mm\_shuffle\_ps(tempv1, tempv1, \_MM\_SHUFFLE(0, 0, 3, 2));//\_MM\_SHUFFLE宏从右往左看 z w \_ \_

tempv1 = ::\_mm\_add\_ps(tempv1, tempv2);//x+z y+w \_ \_

tempv2 = ::\_mm\_shuffle\_ps(tempv1, tempv1, \_MM\_SHUFFLE(0, 0, 0, 1));//y+w \_ \_ \_

tempv1 = ::\_mm\_add\_ss(tempv1, tempv2);//x+z+y+w \_ \_ \_

float accum;//求和结果78

::\_mm\_store\_ss(&accum, tempv1);

对空集合的归约有意义 组合器函数应当存在单位元（Identity）

区块（Tile） 双阶段过程 局部阶段 单个区块内使用 串行实现或向量化实现 全局阶段 区块间使用 树形实现

树形实现 空间复杂度O(n)

串行实现 空间复杂度O(1)

从空间复杂度出发，可能需要使用串行实现

#### TBB

parallel\_reduce

#### Vulkan 计算着色器

一：在区块内归约

#version 450 core

layout (set = 0, binding = 0) uniform samplerBuffer g\_InputArray;

layout (set = 1, binding = 0 , rgba32f) uniform imageBuffer g\_TileArray;

//GLSL ---- HLSL

//samplerBuffer vec4 ---- Buffer<float4>

//isamplerBuffer ivec4 ---- Buffer<int4>

//usamplerBuffer uvec4 ---- Buffer<uint4>

//imageBuffer vec4 ---- RWBuffer<float4>

//iimageBuffer ivec4 ---- RWBuffer<int4>

//uimageBuffer uvec4 ---- RWBuffer<uint4>

//vkCmdDispatch(XXX,(Width - 1U)/64U + 1U,1U,1U)

shared vec4 s\_CacheArray[64];

layout(local\_size\_x = 32, local\_size\_y = 1, local\_size\_z = 1) in;

//gl\_GlobalInvocationID

//gl\_WorkGroupID 区块间

//gl\_LocalInvocationID 区块内

void main()

{

//并行映射

s\_CacheArray[gl\_LocalInvocationID.x] = texelFetch(g\_InputArray, int(64U \* gl\_WorkGroupID.x + gl\_LocalInvocationID.x));//缓存到组共享内存

s\_CacheArray[gl\_LocalInvocationID.x + 32U] = texelFetch(g\_InputArray, int(64U \* gl\_WorkGroupID.x + gl\_LocalInvocationID.x + 32U));//缓存到组共享内存

//并行归约

s\_CacheArray[gl\_LocalInvocationID.x] += s\_CacheArray[gl\_LocalInvocationID.x + 32];

memoryBarrierShared();//内存依赖

barrier();//执行依赖

s\_CacheArray[gl\_LocalInvocationID.x] += (gl\_LocalInvocationID.x < 16) ? s\_CacheArray[gl\_LocalInvocationID.x + 16] : vec4(0.0f,0.0f,0.0f,0.0f);

memoryBarrierShared();//内存依赖

barrier();//执行依赖

s\_CacheArray[gl\_LocalInvocationID.x] += (gl\_LocalInvocationID.x < 8) ? s\_CacheArray[gl\_LocalInvocationID.x + 8] : vec4(0.0f,0.0f,0.0f,0.0f);

memoryBarrierShared();//内存依赖

barrier();//执行依赖

s\_CacheArray[gl\_LocalInvocationID.x] += (gl\_LocalInvocationID.x < 4) ? s\_CacheArray[gl\_LocalInvocationID.x + 4] : vec4(0.0f,0.0f,0.0f,0.0f);

memoryBarrierShared();//内存依赖

barrier();//执行依赖

s\_CacheArray[gl\_LocalInvocationID.x] += (gl\_LocalInvocationID.x < 2) ? s\_CacheArray[gl\_LocalInvocationID.x + 2] : vec4(0.0f,0.0f,0.0f,0.0f);

memoryBarrierShared();//内存依赖

barrier();//执行依赖

s\_CacheArray[gl\_LocalInvocationID.x] += (gl\_LocalInvocationID.x < 1) ? s\_CacheArray[gl\_LocalInvocationID.x + 1] : vec4(0.0f,0.0f,0.0f,0.0f);

memoryBarrierShared();//内存依赖

barrier();//执行依赖

//输出结果

if (gl\_LocalInvocationID.x == 0)

{

imageStore(g\_TileArray, int(gl\_WorkGroupID.x), s\_CacheArray[0]);

}

}

二：在区块间归约

#version 450 core

layout (set = 0, binding = 0) uniform samplerBuffer g\_TileArray;

layout (set = 1, binding = 0 , rgba32f) uniform imageBuffer g\_OutputArray;

//GLSL ---- HLSL

//samplerBuffer vec4 ---- Buffer<float4>

//isamplerBuffer ivec4 ---- Buffer<int4>

//usamplerBuffer uvec4 ---- Buffer<uint4>

//imageBuffer vec4 ---- RWBuffer<float4>

//iimageBuffer ivec4 ---- RWBuffer<int4>

//uimageBuffer uvec4 ---- RWBuffer<uint4>

shared vec4 s\_CacheArray[64];

//vkCmdDispatch(XXX,1U,1U,1U) //假设区块的数量不大于64

layout(local\_size\_x = 32, local\_size\_y = 1, local\_size\_z = 1) in;

//gl\_GlobalInvocationID

//gl\_WorkGroupID 区块间

//gl\_LocalInvocationID 区块内

void main()

{

//并行映射

s\_CacheArray[gl\_LocalInvocationID.x] = texelFetch(g\_TileArray, int(gl\_LocalInvocationID.x));//缓存到组共享内存

s\_CacheArray[gl\_LocalInvocationID.x + 32U] = texelFetch(g\_TileArray, int(gl\_LocalInvocationID.x + 32U));//缓存到组共享内存

//并行归约

s\_CacheArray[gl\_LocalInvocationID.x] += s\_CacheArray[gl\_LocalInvocationID.x + 32];

memoryBarrierShared();//内存依赖

barrier();//执行依赖

s\_CacheArray[gl\_LocalInvocationID.x] += (gl\_LocalInvocationID.x < 16) ? s\_CacheArray[gl\_LocalInvocationID.x + 16] : vec4(0.0f,0.0f,0.0f,0.0f);

memoryBarrierShared();//内存依赖

barrier();//执行依赖

s\_CacheArray[gl\_LocalInvocationID.x] += (gl\_LocalInvocationID.x < 8) ? s\_CacheArray[gl\_LocalInvocationID.x + 8] : vec4(0.0f,0.0f,0.0f,0.0f);

memoryBarrierShared();//内存依赖

barrier();//执行依赖

s\_CacheArray[gl\_LocalInvocationID.x] += (gl\_LocalInvocationID.x < 4) ? s\_CacheArray[gl\_LocalInvocationID.x + 4] : vec4(0.0f,0.0f,0.0f,0.0f);

memoryBarrierShared();//内存依赖

barrier();//执行依赖

s\_CacheArray[gl\_LocalInvocationID.x] += (gl\_LocalInvocationID.x < 2) ? s\_CacheArray[gl\_LocalInvocationID.x + 2] : vec4(0.0f,0.0f,0.0f,0.0f);

memoryBarrierShared();//内存依赖

barrier();//执行依赖

s\_CacheArray[gl\_LocalInvocationID.x] += (gl\_LocalInvocationID.x < 1) ? s\_CacheArray[gl\_LocalInvocationID.x + 1] : vec4(0.0f,0.0f,0.0f,0.0f);

memoryBarrierShared();//内存依赖

barrier();//执行依赖

//输出结果

if (gl\_LocalInvocationID.x == 0)

{

imageStore(g\_OutputArray, 0, s\_CacheArray[0]);

}

}

### 映射—归约（Map-Reduce）

融合映射和归约

数量积

逐分量相乘——映射

相加——归约

#### TBB

alignas(16) float a[12] = { 1,2,3,4,5,6,7,8,9,10,11,12 };//输入数组

alignas(16) float b[12] = { 12,11,10,9,8,7,6,5,4,3,2,1 };//输入数组

float \*pA = a;

float \*pB = b;

float accum

= tbb::parallel\_reduce(

tbb::blocked\_range<int>(0, 12),

0.0f,//单位元

[pA, pB](

tbb::blocked\_range<int> & r, float initial)->float//区块内串行

{

for (int i = r.begin(); i != r.end(); ++i)

{

float temp = pA[i] \* pB[i];//映射

initial += temp;//归约

}

return initial;

},

[](const float Left, const float Right)->float//区块间并行

{

return Left + Right;

}

);

### 扫描（Scan）

在语义上可看作积分的离散版本（如果组合器是求和的话）

产生 序列 所有的 局部归约

计算量 n -> n-1 + n-2 + n–4 + n-8 + ... O(nlogn)

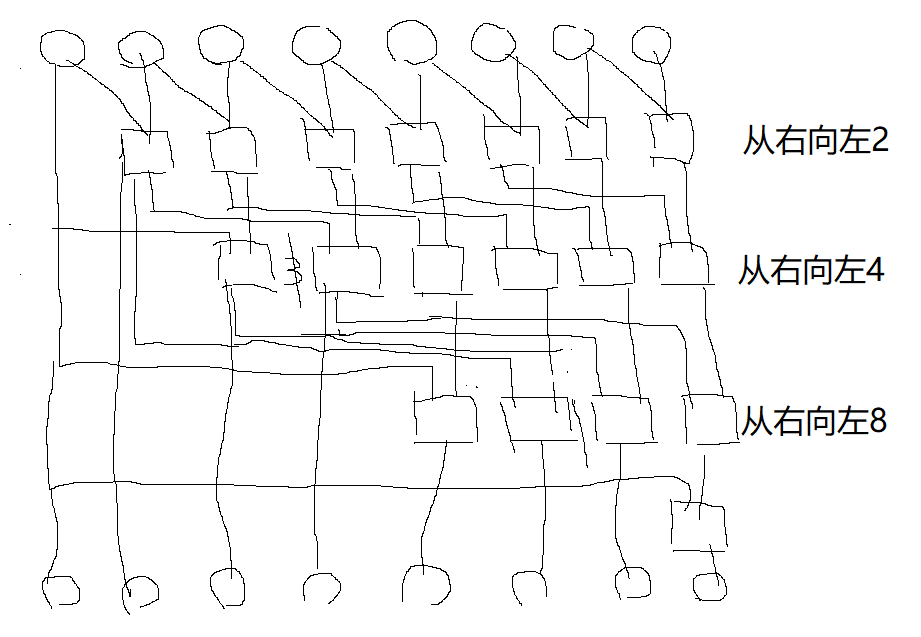
跨度 n –> log2n + 1 O(logn)

可用打包（pack）和发散（scatter）实现

应用 前缀和（Prefix Sum） SAT DOF

准确地说是 Inclusive Scan

一：在区块内扫描



二：在区块间归约

#### TBB

parallel\_scan

### 映射—扫描（Map-Scan）

## 循环（Recurrence）

# 数据管理模式

## 打包（Pack）

### SSE

\_mm\_shuffle\_ps

### 计算着色器

## 流水线（Pipeline）

吞吐量增加 但延迟增加（可能不适合VR）

延迟 同一个Task从开始的结束的时间 由于通信的开销导致延迟增加

吞吐量 单位时间内完成的Task个数

# 可能不适合VR(HMD设备)

生产者 消费者

通常用于 重叠计算和IO

渲染时，每一帧可看作数组中的一个项，因此适用流水线

与Map区别在于 不要求输入数据全部加载进内存

并行阶段的流水线

并行阶段与内部并行的串行阶段不同，并行阶段对流水线是可见的，流水线允许并行阶段是输出是乱序的

TBB将流水线中的阶段分为三种：

parallel：并行

serial\_out\_of\_order：串行乱序 对输入的项顺序没有限制

serial\_in\_order ：串行有序 与流水线中其它所有的串行有序 的顺序一致

渐近（Asymptotic）加速比（Speedup）

## 几何分解（Geometric Decomposition）

## 聚集（Gather）

readv ？？？

## 分散（Scatter）

writev ？？？

# TBB

分叉结合（fork-Join） 嵌套 递归

parallel\_invoke

映射（map）

parallel\_for parallel\_foreach

工作桩（work pile）

parallel\_do

规约（reduction）

parallel\_reduce 为使结果确定，组合函数应当是可结合的且可交换的

deterministic\_parallel\_reduce 结果一定是确定的（只要求组合函数可结合）

扫描（scan）

parallel\_scan为使结果确定，组合函数应当是可结合的且可交换的

流水线（Pipeline）

DAG（有向无环图）

parallel\_pipeline

预测选择 分支界定

TBB支持 任务取消

# 计算着色器

不支持嵌套但支持2个显式级别的并行（线程组 线程）

在OpenCL中 在CPU上 线程组对应Core 线程对应SIMD航道或超线程

映射（map）

一般认为，不同线程组间的指令寄存器独立，但同一线程组内部共用指令寄存器，使用掩码的方式模拟分支

由于同一线程组可能被作为多个分块（chunk）实现，因此线程组内的仍需要用barrier来同步

聚集

对计算着色器而言 吞吐量比延迟更重要

分散

# Vulkan/Direct3D12

## 内存屏障/资源屏障

CPU中 超标量体系结构

Winnt.h MemoryBarrier();

GCC \_\_sync\_synchronize

# 多线程渲染

Demo做法

用于BasePass，场景节点（可关联二级命令队列，描述符集合）可以作为一维数组索引

可视锥体剔除后将指针添加到该一维数组中

信号量

初值为0 事件

初值为1 互斥量

# Sample用了二级命令队列，同时访问同一个一级命令队列，是否需要同步？？？

唤醒工作线程

每个工作线程处理若干场景节点并填充一个命令列表（同时计算动画）

切换场景节点时，vkCmdBindDescriptorSets切换描述符集合，vkCmdExecuteCommands执行命令队列

在工作线程产生命令的同时主线程也产生一个命令列表

等待工作线程

将左右cmd组合成一个submit并vkQueueSubmit

双缓冲 在另外一个CommandAllocator中填充命令

用于LightPass，所有Light存放在一个数组中

# SSE2

http://software.intel.com/en-us/node/683911

#include <xmmintrin.h>

8个128位（16字节）寄存器 XMM0-XMM7

数据类型

\_\_m64

\_\_m128 可存放4个float

\_\_m128d 可存放2个double

\_\_m128i 可存放16个char 8个short 4个int 2个longlong

访问\_\_m128i

4个int \_\_mm\_extract\_epi32(src,index)

# CUDA

执行

SM 流式多处理器

Wrap 一个SM内部 对 千兆线程调度器 隐藏

TLP 线程级别并行Thread-Level Parallelizaton

ILP 指令级别并行 Instruction-Level parallelism

flatten if

HLSL中flatten语句 类似常规的 三元条件运算符 ? :

unroll while

内存

分为片上（On-Chip）和板上（On-Board）两种

数据局部性 时间局部性 空间局部性

Texture Cache 2D空间局部性

但L2 支持非Pattern

# 基于围栏的资源管理

https://msdn.microsoft.com/en-us/library/windows/desktop/dn899125(v=vs.85).aspx

在NUMA中，CPU写入到Upload Heap中的Buffer，再由GPU的复制队列CopyTextureRegion

CPU根据伙伴堆算法在Upload Buffer中分配相应的BlockHeader

CPU向Upload Buffer写入数据，向GPU发出CopyTextureRegion并添加一个Fence，将项{Event（用于等待Fence） BlockHeader}添加到偏移队列中

当CPU无法从伙伴堆中获取足够的BlockHeader时，对偏移队列{ Event BlockHeader }进行轮询（WaitForMultiplyObjectsEx），释放相关的BlockHeader并再次尝试

UploadBufferSubAllocate的伪代码大致如下：

查询所有Fence（以0时间等待）释放已完成的BlockHeader

if 伙伴堆分配成功

进行Upload操作

else

以INFINITE时间等待Fence释放已完成的BlockHeader

由于图形队列也需要等待复制队列完成Copy，因此Event可以和资源存放在同一个结构中{Event Mesh ... }，而偏移队列用指针引用该结构{ {Event Mesh ...}指针, BlockHeader }

在UMA中，CPU直接用WriteToSubresource写入

在Vulkan中，CPU用vkGetImageSubresourceLayout，map后根据偏移写入，但文档指出这么多并不适用于VK\_IMAGE\_TILING\_OPTIMAL；因为Vulkan不支持对UMA的上述优化

# ————————————————————————————————————————

# TBB

Alexey Kukanov, Michael J. Voss. "The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks". Intel Technology Journal 2007.

## 任务调度器

Yannis Minadakis. "Using Tasking to Scale Game Engine Systems". GDC 2011.

<http://software.intel.com/en-us/articles/using-tasking-to-scale-game-engine-systems>

## 内存分配器

伪共享（False Sharing） 不同线性访问同一高速缓存行（Cache Line）

# SIMD SSE2/NEON

# 计算着色器 HLSL/GLSL