主要内容：

1. Parallel Reduction 并行规约
2. Warp 分割
3. Memory Coalescing 访存合并
4. Bank 冲突
5. SM 资源动态分割
6. 数据预读
7. 指令混合
8. 循环展开

## 并行应用的重要的两部分：

有效的数据并行算法（1） + 针对GPU架构特性的优化（2） = 最优性能

1. parallel Reduction 并行规约

比如并行求和

2， 两种求数组和的parallel Reduction 设计

AnotherParallel.c

parallelSum.c

## Warp 分割

AnotherParallel.c

parallelSum.c

明白那种方式使得程序性能更好

## Memory Coalescing 访存合并

合并访存 还不是只能设置那份大使馆法国

## Bank 冲突的产生与消除

## SM资源动态分割

## 数据预读（数据预取）

在一次global memory读写操作和实际用到这个数据的语句中间，插入独立于以上数据的指令，可以隐藏访存延迟。

Float m = Md[i];

Float f = a \* b + b \* c;

Float f2 = m \* f;

从global memory预读数据可以有效地提升独立指令的数量，在global memory读取和使用两者之间。

Eg. 回顾瓦片化的matrix multiply

For (/\*…\*/)

{

//load current tile into shared memory

\_\_syncthreads();

// Accumulate dot product

\_\_syncthreads();

}

引入预读操作的瓦片化 matrix multiply

// load first tile into registers

For (/\*…\*/)

{

// Deposit registers into shared memory

\_\_syncthreads();

// load next tile into registers

//Accumulate dot product

/\* 这两个命令被同时执行 \*/ 如果有足够多的warp被调度，延时就可以被隐藏

\_\_syncthreads();

}

## 指令优化

如果发现代码性能受限于指令

如果不够仔细的话，计算密集型算法很容易受限于带宽。

典型情况，在存储器和执行配置优化完成后，担心指令化。

Eg.

除以2^n, 采用“>>n”

以2^n求模， 采用”&(2^n-1)”

避免double到float的类型自动转换

添加“f”到float常量（e.g, 1.0f）， 因为缺省的是double。浮点数容易自动转换为double…

运行时数学库和固有函数

两种类型的运行时数学函数库函数

Func()

慢但是精度高

\_\_func()

快但是精度低

其他固有函数

\_\_sincos(); \_\_rep(),…详细列表见 《CUDA编程指南》 附录C.2

-use-fast-math: func() 强制转换为 \_\_func()

## 循环展开

For(int k = 0; k < BLOCK\_SIZE; ++k)

{

Pvalue += Ms[ty][k] \* Ns[k][tx];

}

每轮循环包含其他指令：

更新循环计数器（++k）

分支（k < BLOCK\_SIZE）

地址运算（Pvalue += Ms[ty][k] \* Ns[k][tx]）

指令混合;

2条浮点运算指令（主要目的， 计算）

1条循环分支指令

2地址运算指令

1循环计数器自增指令

循环展开；

Pvalue += Ms[ty][0] \* Ns[0][tx] +

Ms[ty][1] \* Ns[1][tx] +

…………

Ms[ty][15] \* Ns[15][tx]；

不再有循环

不再有循环计数器更新

不再有分支

常量索引-不在有地址运算

自动实现：

#pragma unroll BLOCK\_SIZE // 编译选项 展开依据BLOCK\_SIZE

For(int k = 0; k < BLOCK\_SIZE; ++k)

{

Pvalue += Ms[ty][k] \* Ns[k][tx];

}

循环展开有什么缺点？

可扩展性不强