GPU硬件架构简介及CUDA编程基础

何小伟
中国科学院软件研究所
2023.4.2

大纲

- GPU硬件架构简介
- CUDA编程模型
- CUDA编程样例：规约算法（Reduce）
- CUDA编程样例：前缀和（Scan）
- C++模板
- 算法应用

GPU硬件架构：英伟达GPU架构演进

以前 — GPU
- **专用图形处理器**
- **可编程图形管线**
- Shadding Language(GLSL, HLSL)

2006 — Tesla
- 第一个实现**统一着色器模型**
- 第一款**支持 C 语言**的 GPU

2010 — Fermi
- 第一个完成的**GPU通用计算架构**
- 32 CUDA core

2012 — Kepler
- 相比于Fermi更快的性能
- 192个 CUDA 内核

2014 — Maxwell
- Warp Scheduler：4
- Dispatch Unit：8
- CUDA Core 128
- LD/ST Unit：32
- SFU：32

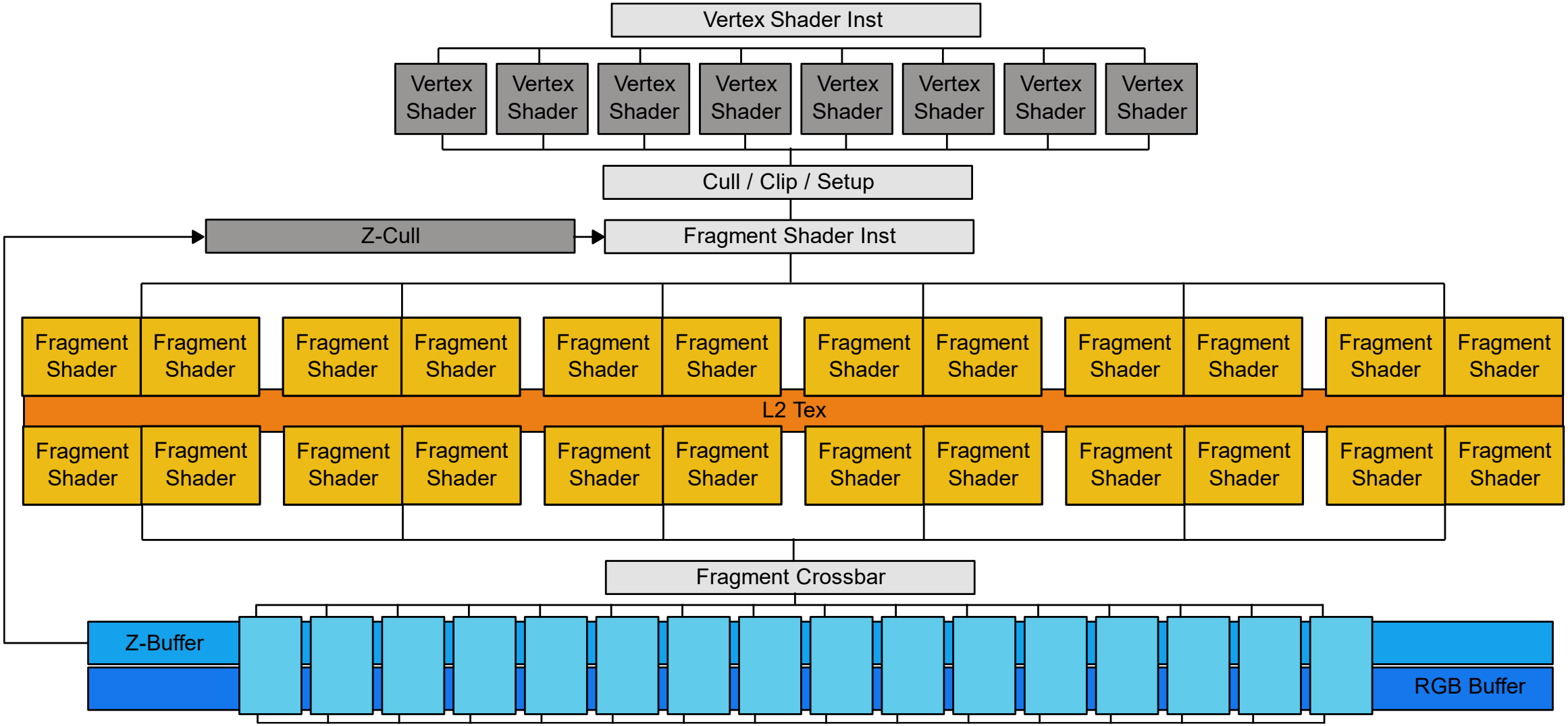
2016 — Pascal
- 深度学习加速
- CUDA Core支持FP16

2017 — Volta
- Warp Scheduler：1
- Dispath Unit：1
- FP64 Core：8
- **INT32 Core：16**
- **FP32 Core：16**
- **Tensor Core：2**
- LD/ST Unit：8
- SFU：4

2018 — Turing
- CUDA Core：64
- Tensor Core：8
- 移除FP64

2020 — Ampere
- AI训练相比Volta加速提升20倍
- 双精度运算提速2.5倍

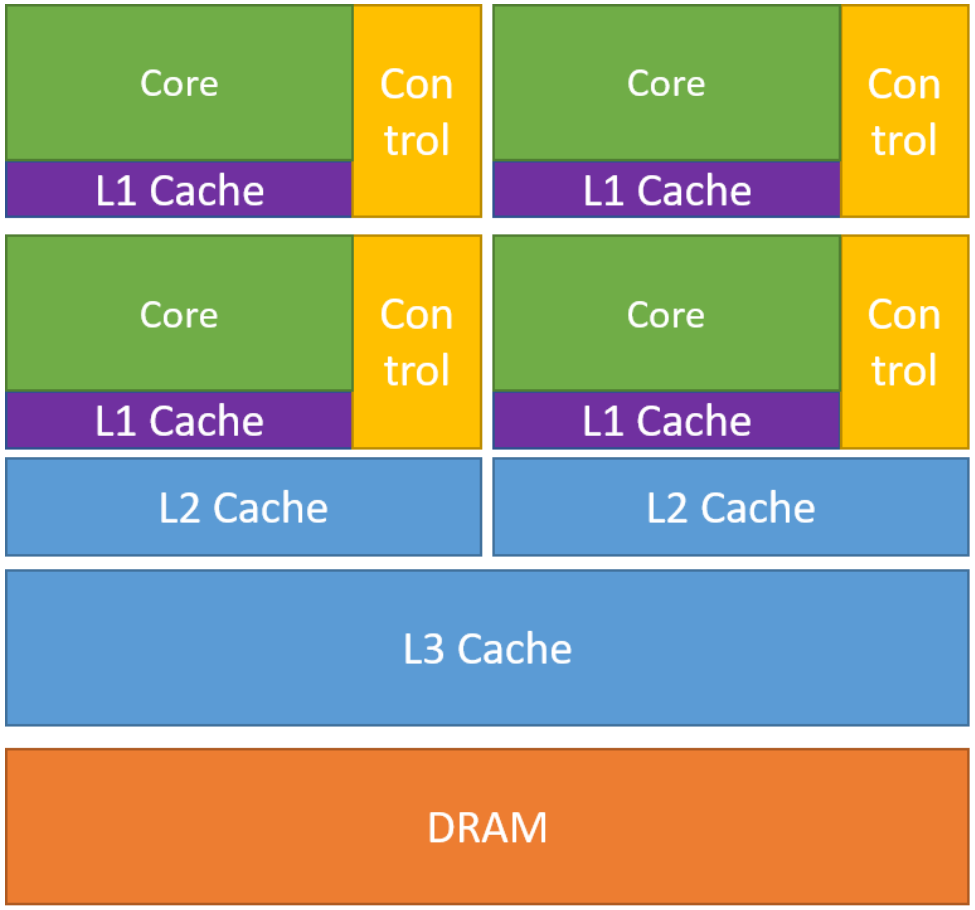
GPU硬件架构：传统图形渲染管线



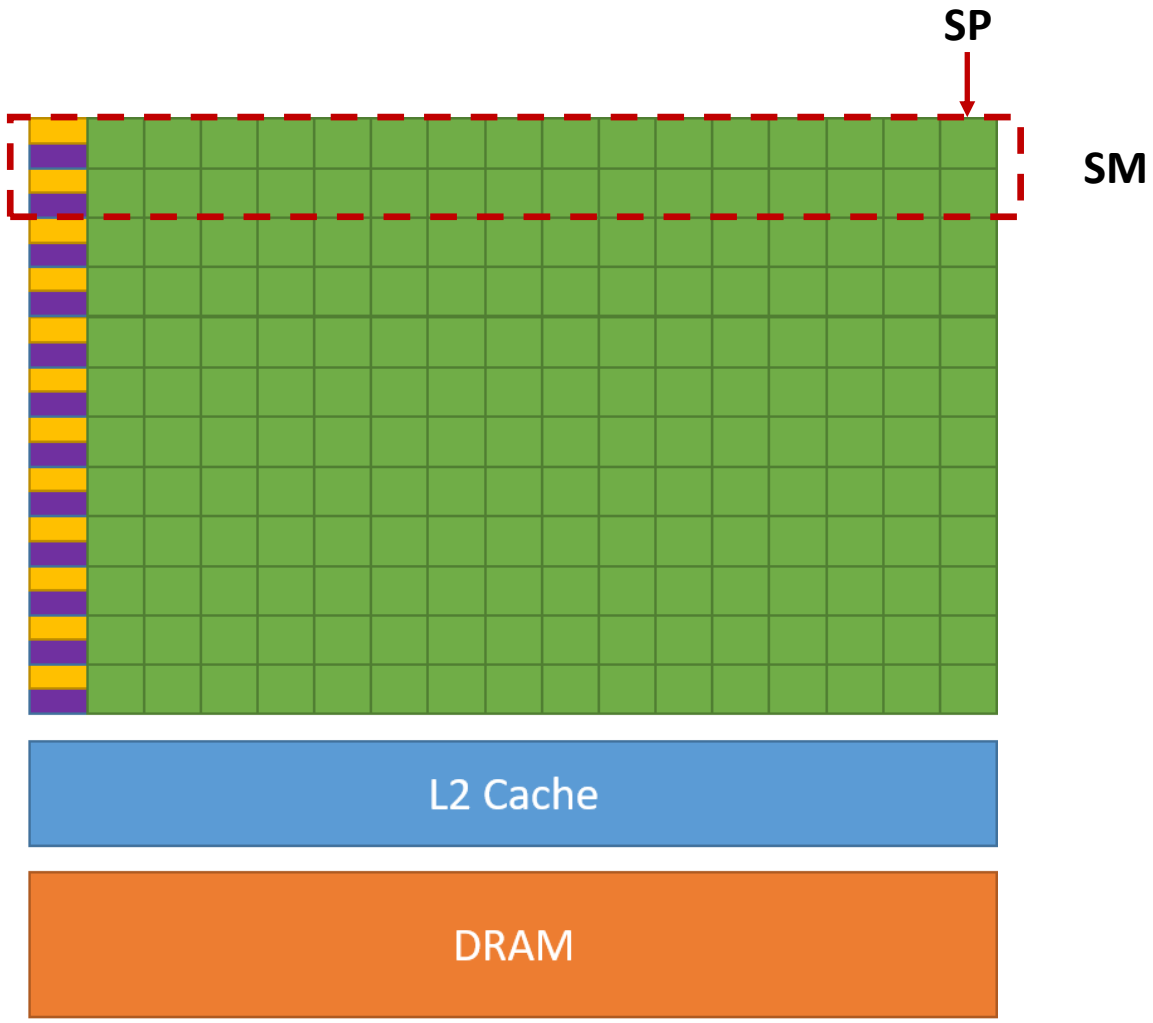
@ https://fabiensanglard.net/cuda/

GPU硬件架构

- **CPU vs. GPU**



CPU

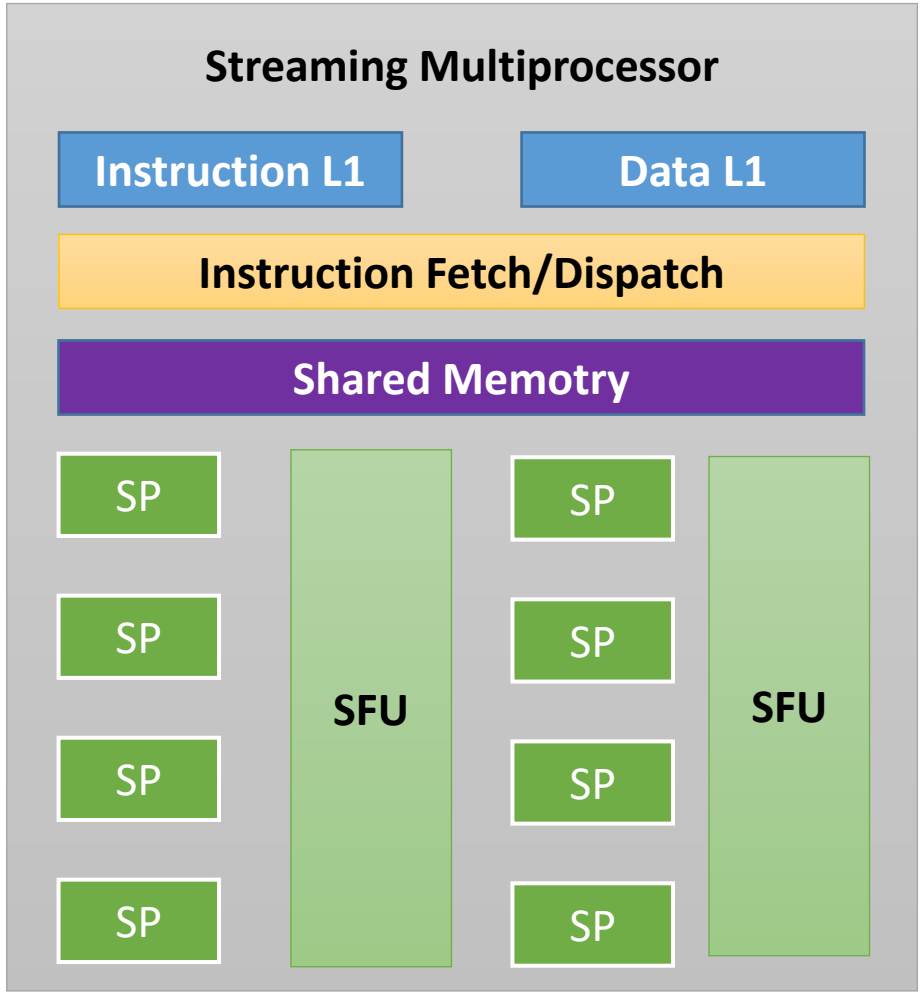


GPU

@ https://docs.nvidia.com/cuda/cuda-c-programming-guide/

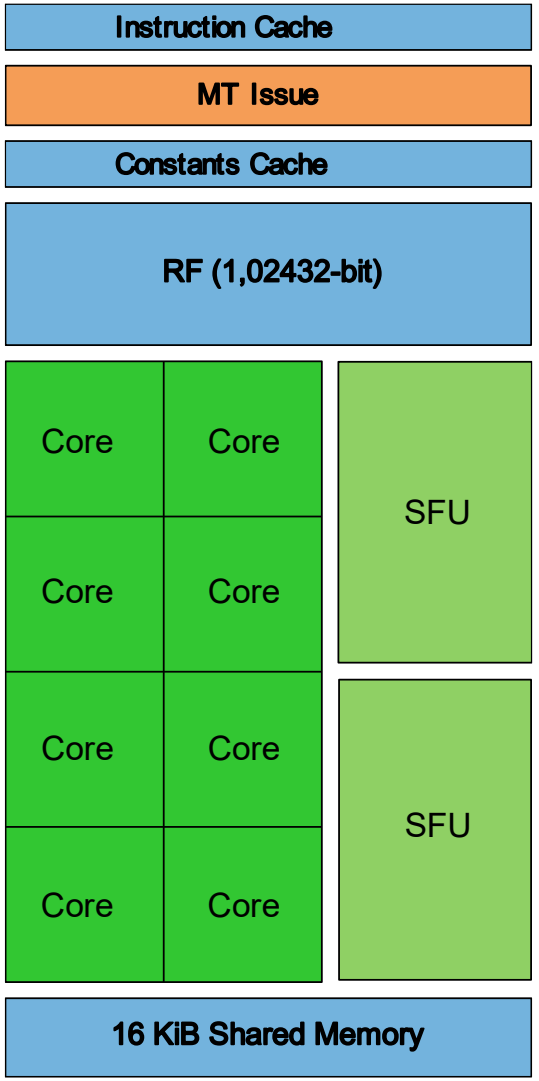
GPU硬件架构

- Streaming Multiprocessor（SM）
  - 包含多个SP，Fermi（32），Kepler（192）
  - Instruction Fetch/Dispatch
  - Register
  - Shared Memory
  - …
- Streaming Processor(SP)
  - 也叫CUDA Core
  - 执行具体指令和任务



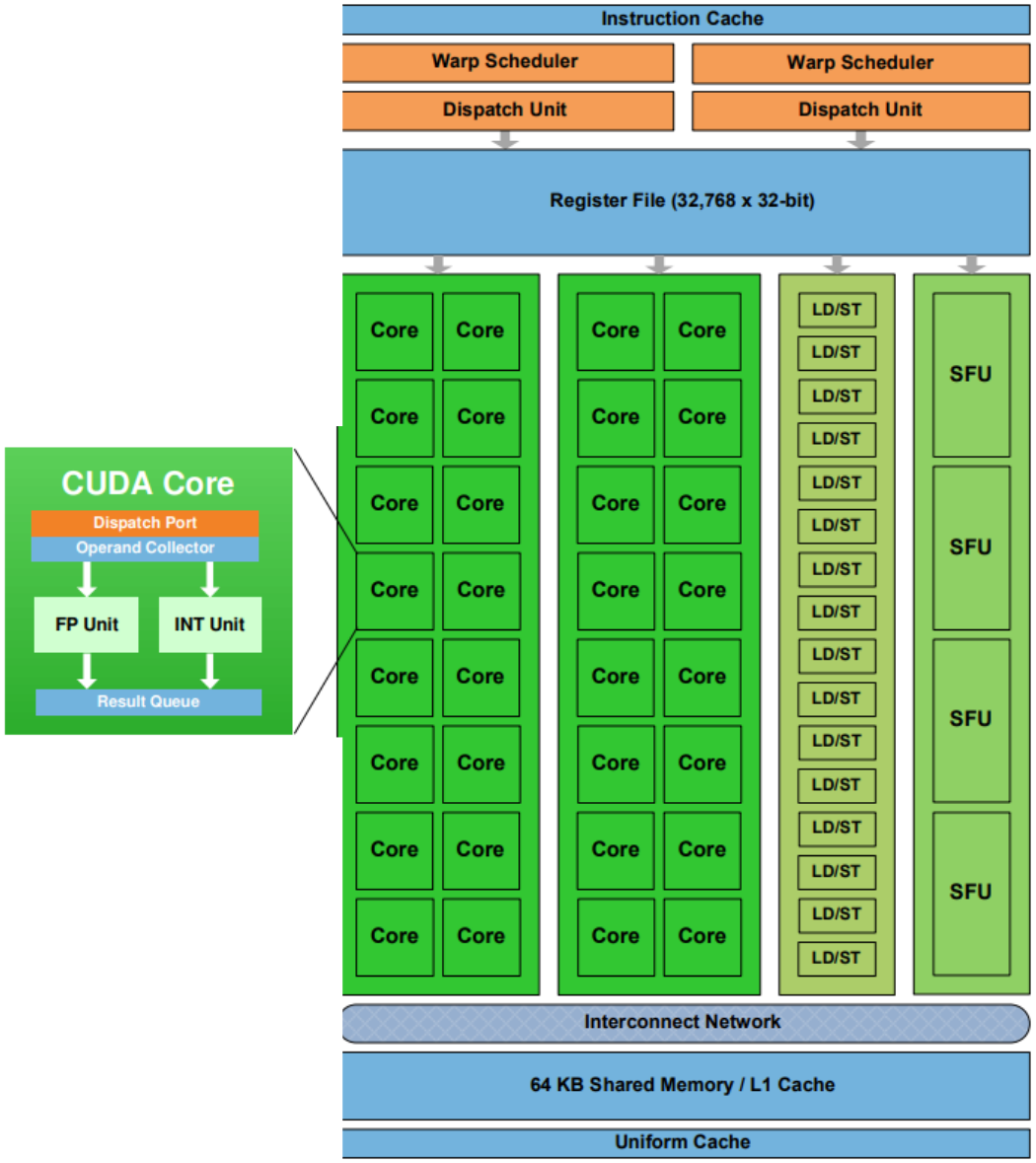
GPU硬件架构

- Tesla架构
  - Instruction cache
  - MT issue: Warp scheduler/Dispatch Unit
  - Constants Cache
  - Register File (RF)
  - CUDA Core
  - Special Function Unit (SFU)
  - Shared Memory



GPU硬件架构

- Fermi架构
  - SM包含
    - 2 个 Warp Scheduler
    - 2 个 Dispatch Unit
    - 32 个 CUDA core
    - 16 个 Load/Store units (LD/ST Unit)
    - 4 个 Special Function Units (SFU)
  - Cuda core
    - Integer Arithmetic Logic Unit (ALU)
    - Floating Point Unit (FPU)

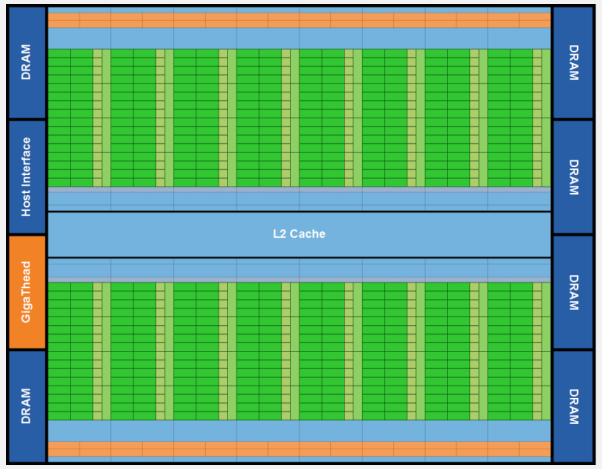


GPU硬件架构

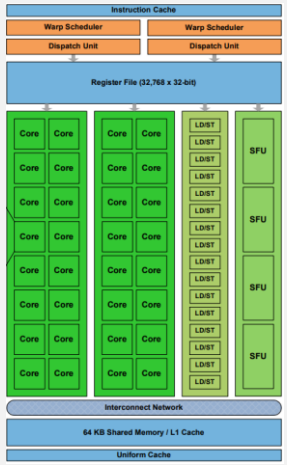
- Ampere架构
  - Compute Capability 8.0
  - 第三代Tensor Core 特性
  - 支持的数据类型有 FP16、BF16、TF32、FP64、INT8、INT4 和 INT1
  - 利用深度学习网络的细粒度结构化稀疏性

CUDA编程模型

硬件架构



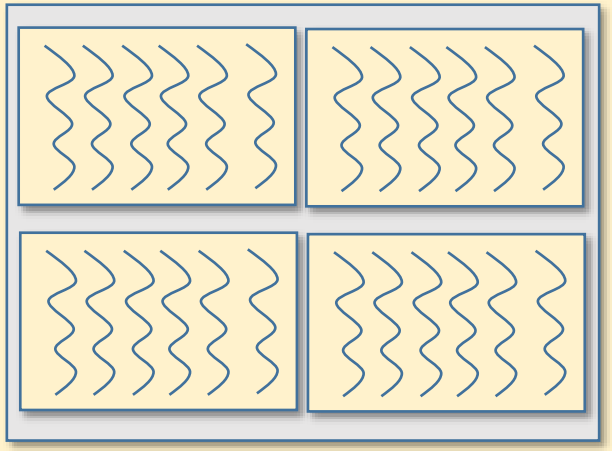
GPU



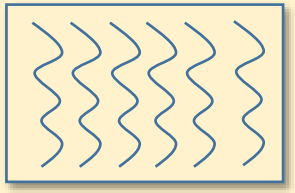
SM

SP

编程模型



Grid

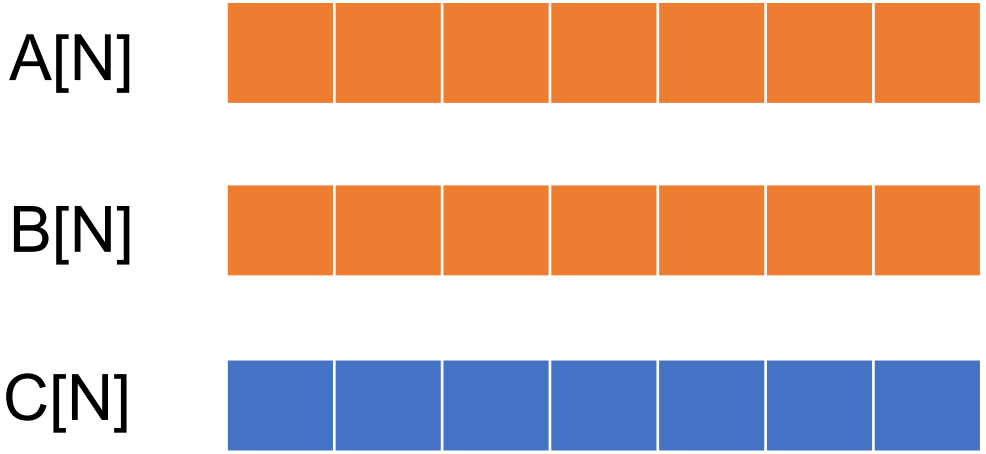


Block

Thread

CUDA编程模型

- 问题描述
  - 给出两个长度为N的数组A、B，将A和B中逐个元素相加存入数组C



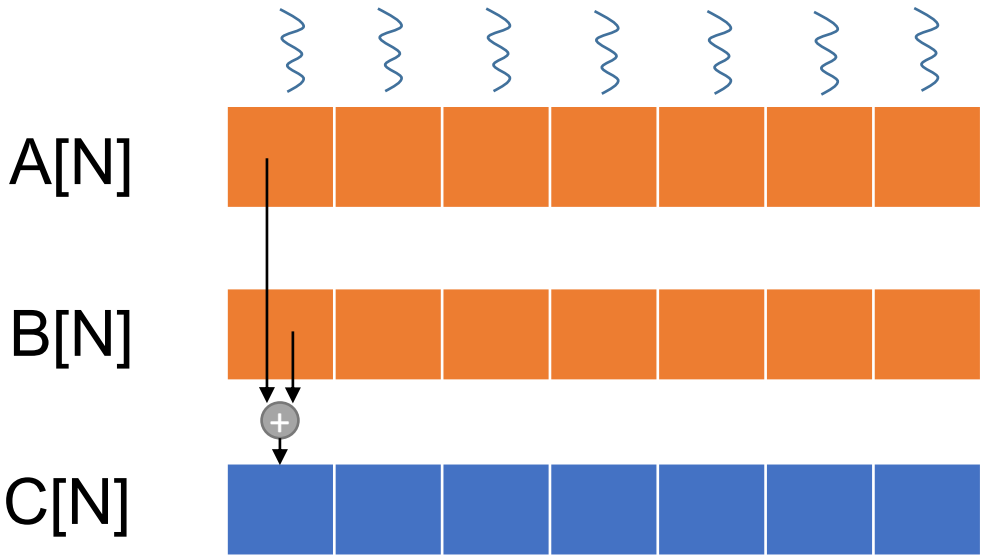
C++

```
int main()
{
        //Initialization

        for (int i = 0; i < N; i++);
                C[i] = A[i] + B[i];
}
```

CUDA编程模型

- 问题描述
  - 给出两个长度为N的数组A、B，将A和B中逐个元素相加存入数组C
  - 假设CUDA Core的数量为 M



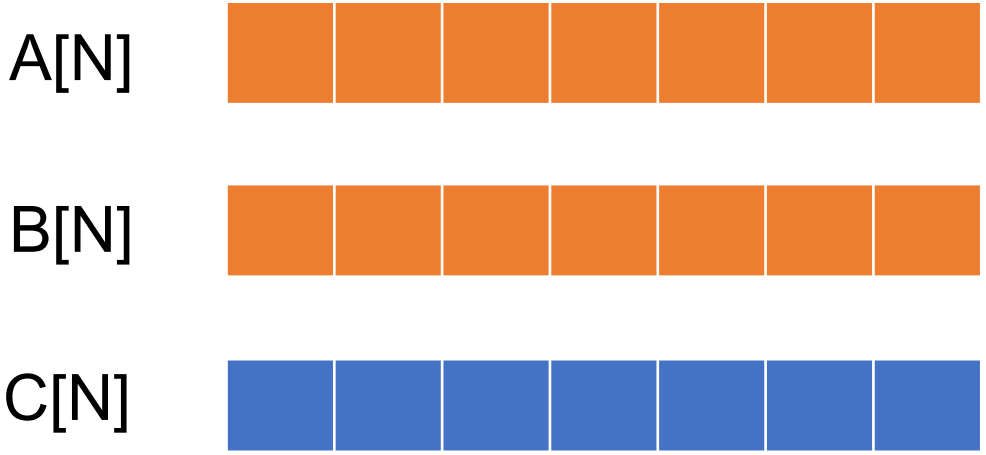
CUDA

```
// CUDA Kernel
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

$$N \leq M$$

CUDA编程模型

- 问题描述
  - 给出两个长度为N的数组A、B，将A和B中逐个元素相加存入数组C
  - 假设CUDA Core的数量为 M



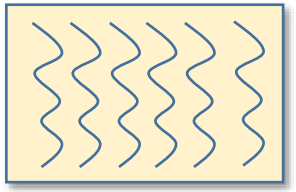
CUDA

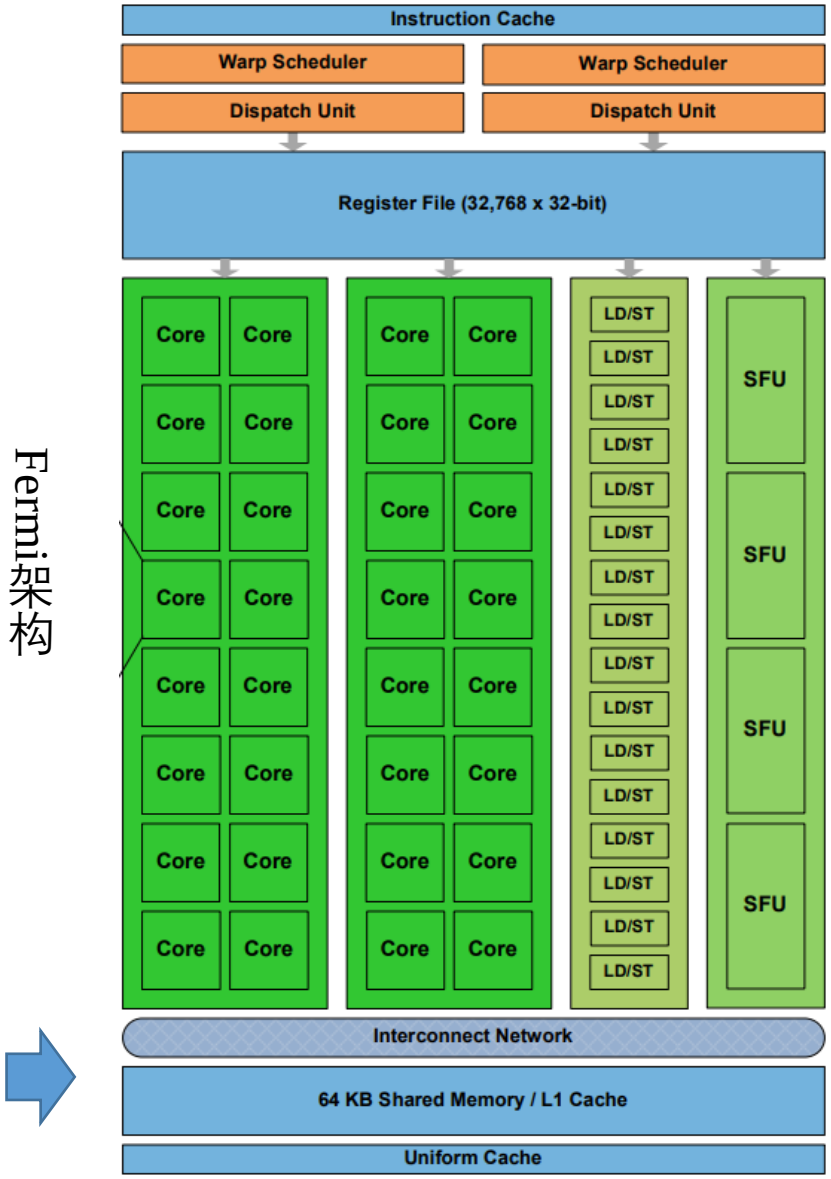
$N > M$

CUDA编程模型

- Warp
  - SM的基本执行单元
  - 一个warp一般包含32个并行thread，每个thread执行相同的指令
  - 一个warp按照单指令多线程(Single Instruction Multiple Threads)模式执行
  - 一个CUDA core执行一个thread

Block

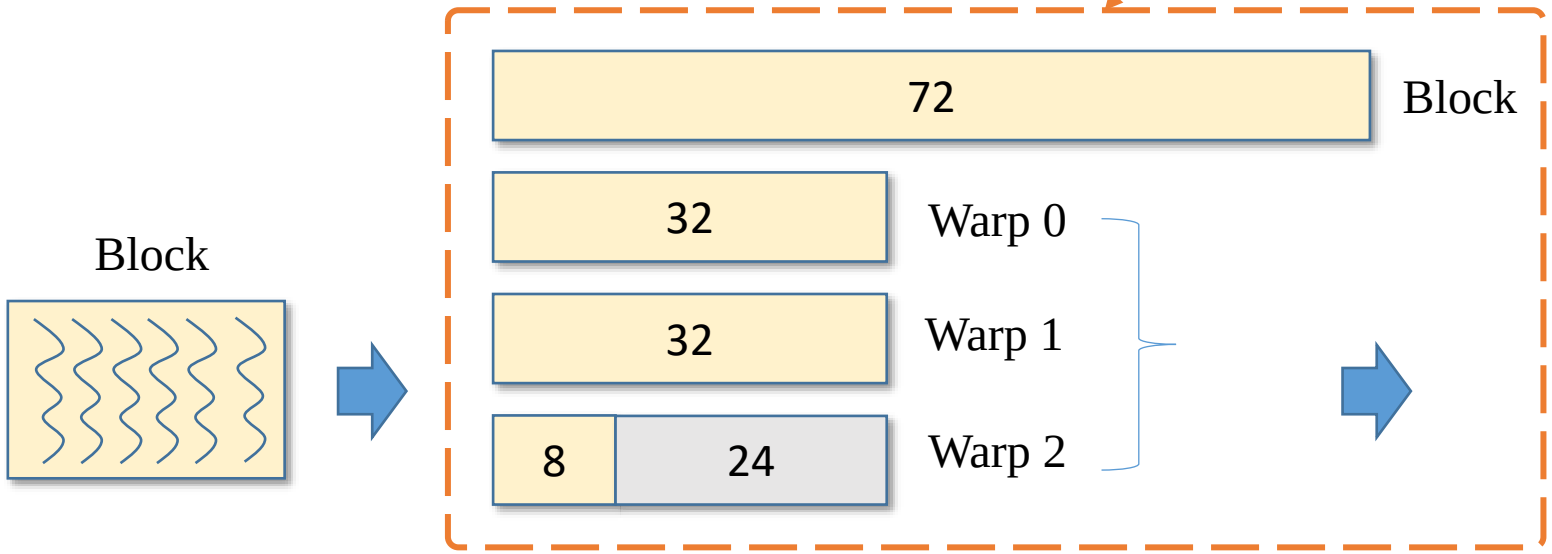


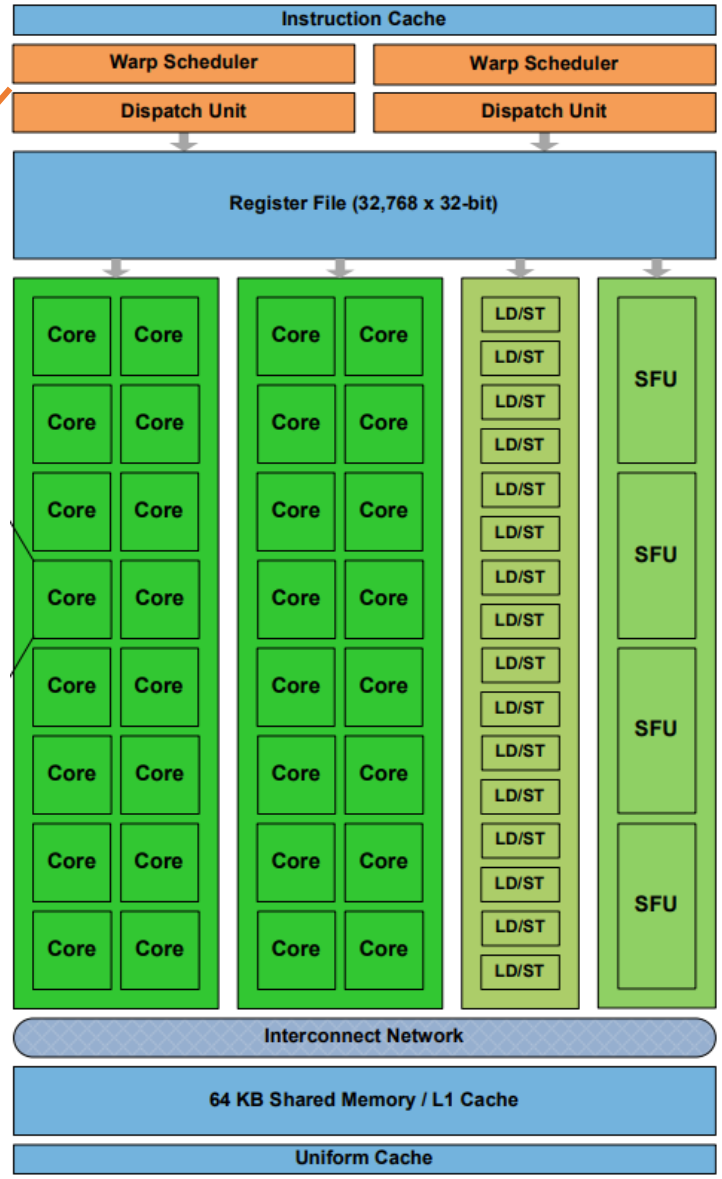


Fermi架构

CUDA编程模型

- Warp
  - 一个warp中的所有线程必须属于同一个block
  - Block包含线程数目最好是warp大小的整数倍

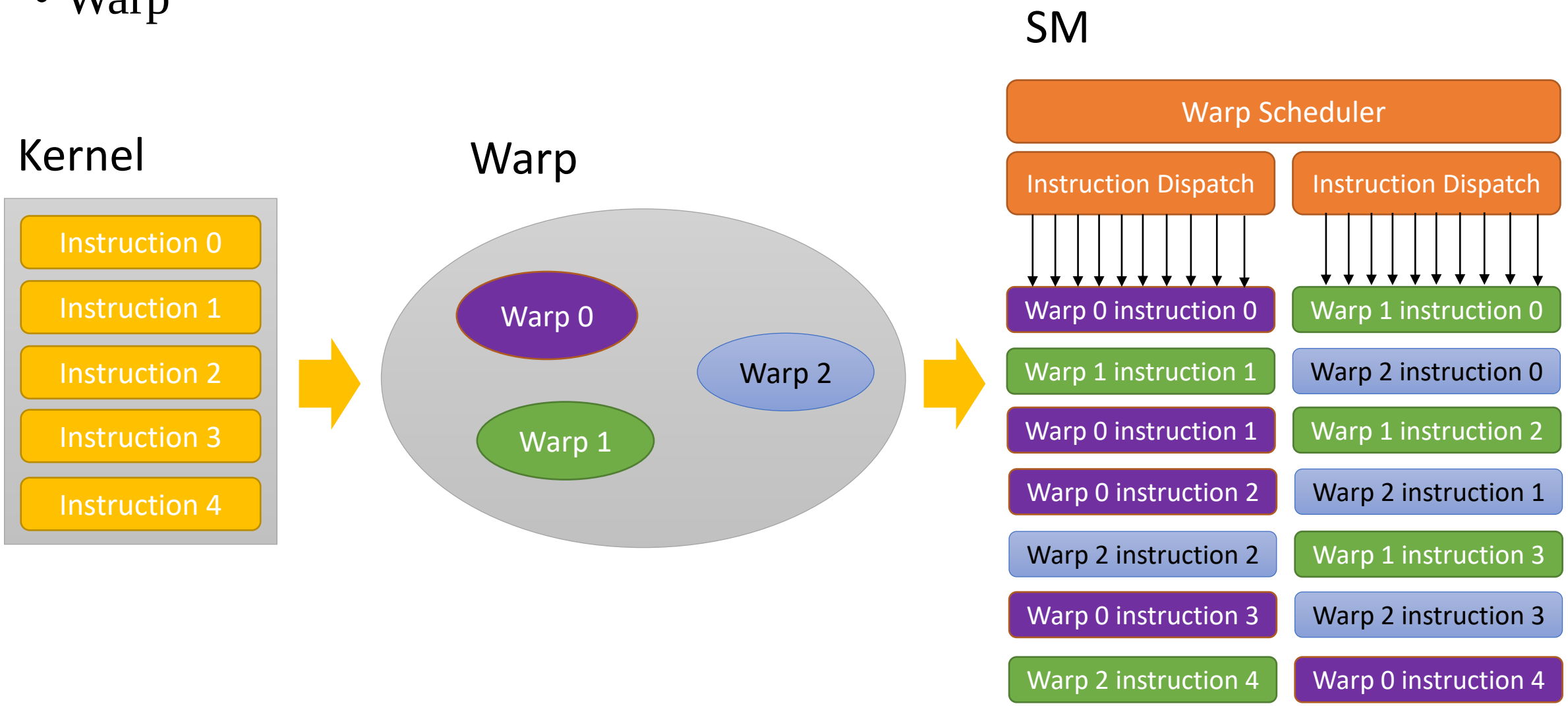




Fermi架构

CUDA编程模型

- Warp



CUDA编程模型

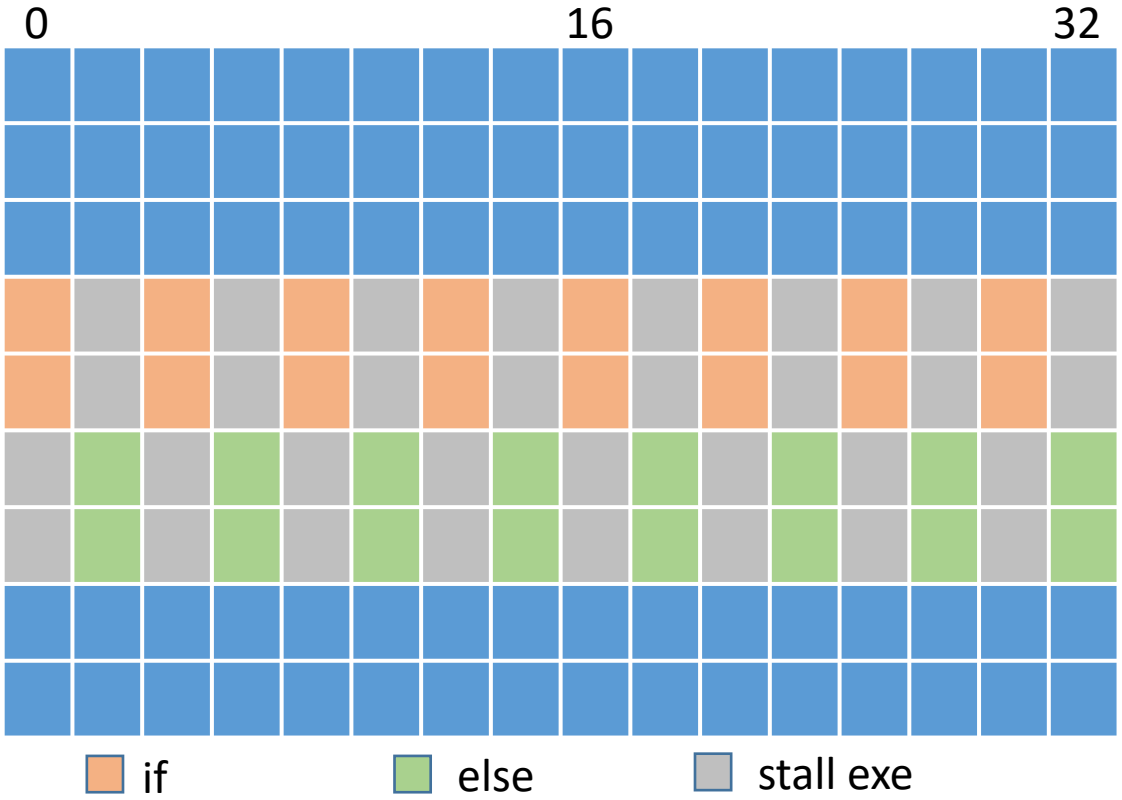
- Warp divergence
- Latency hiding
  - Algorithmic instruction latency
  - Memory instruction latency

CUDA编程模型

- Warp divergence
  - if…else, for, while
  - 分支阻塞 ⟶ 怎么解决？

CUDA kernel

```
__global__ void kerne1(float* c) {
        int tid = blockIdx.x * blockDim.x + threadIdx.x;
        float a = 0.0f;
        float b = 0.0f;
        if (tid % 2 == 0) {
                a = 100.0f;
        }
        else {
                b = 200.0f;
        }
        c[tid] = a + b;
}
```

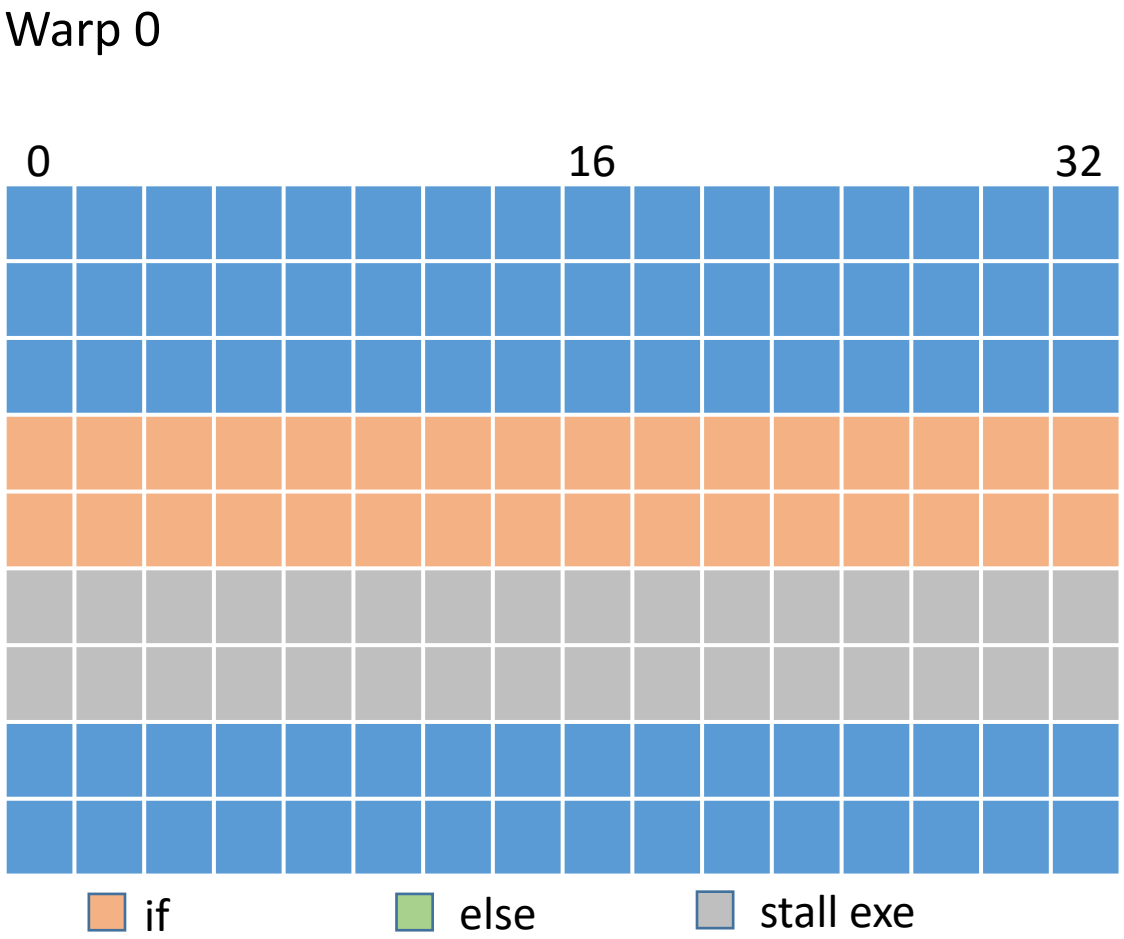


CUDA编程模型

- Warp divergence
  - if…else, for, while
  - 分支阻塞

CUDA kernel

```
__global__ void kerne1(float* c) {
        int tid = blockIdx.x * blockDim.x + threadIdx.x;
        float a = 0.0f;
        float b = 0.0f;
        if ((tid / warpSize) % 2 == 0) {
                a = 100.0f;
        }
        else {
                b = 200.0f;
        }
        c[tid] = a + b;
}
```



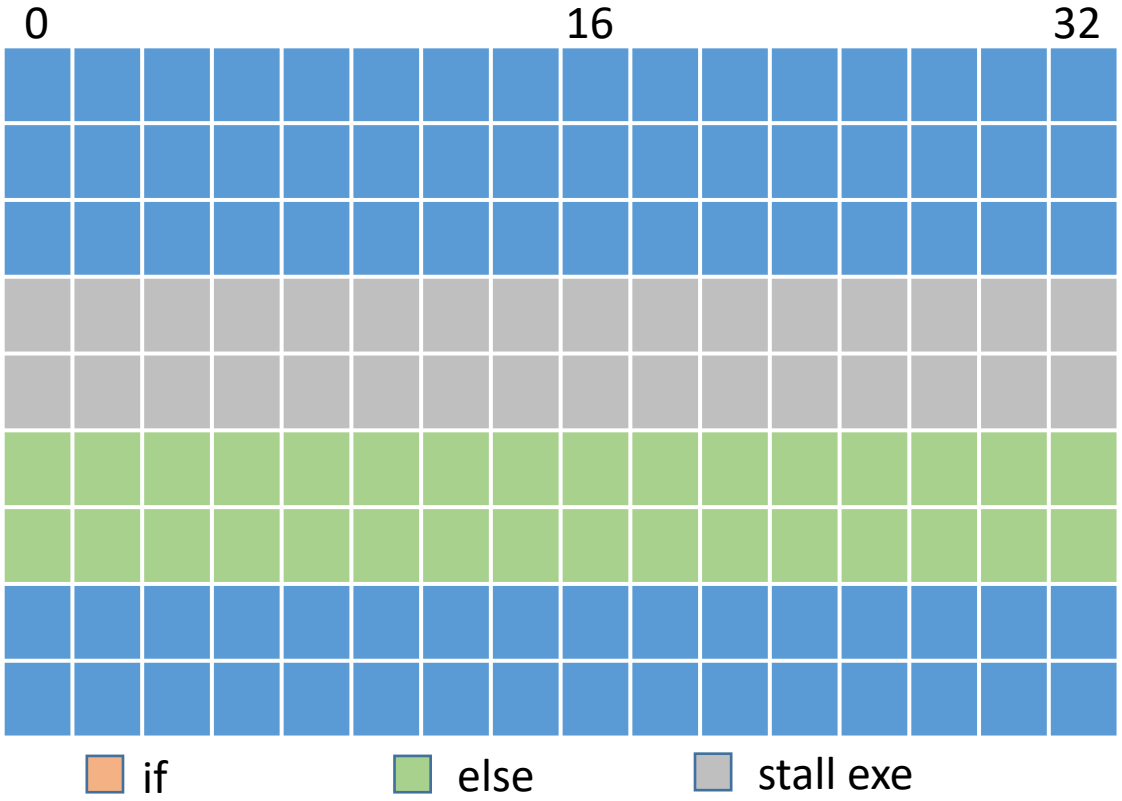
CUDA编程模型

- Warp divergence
  - if…else, for, while
  - 分支阻塞

CUDA kernel

```
__global__ void kerne1(float* c) {
        int tid = blockIdx.x * blockDim.x + threadIdx.x;
        float a = 0.0f;
        float b = 0.0f;
        if ((tid / warpSize) % 2 == 0) {
                a = 100.0f;
        }
        else {
                b = 200.0f;
        }
        c[tid] = a + b;
}
```

Warp 1

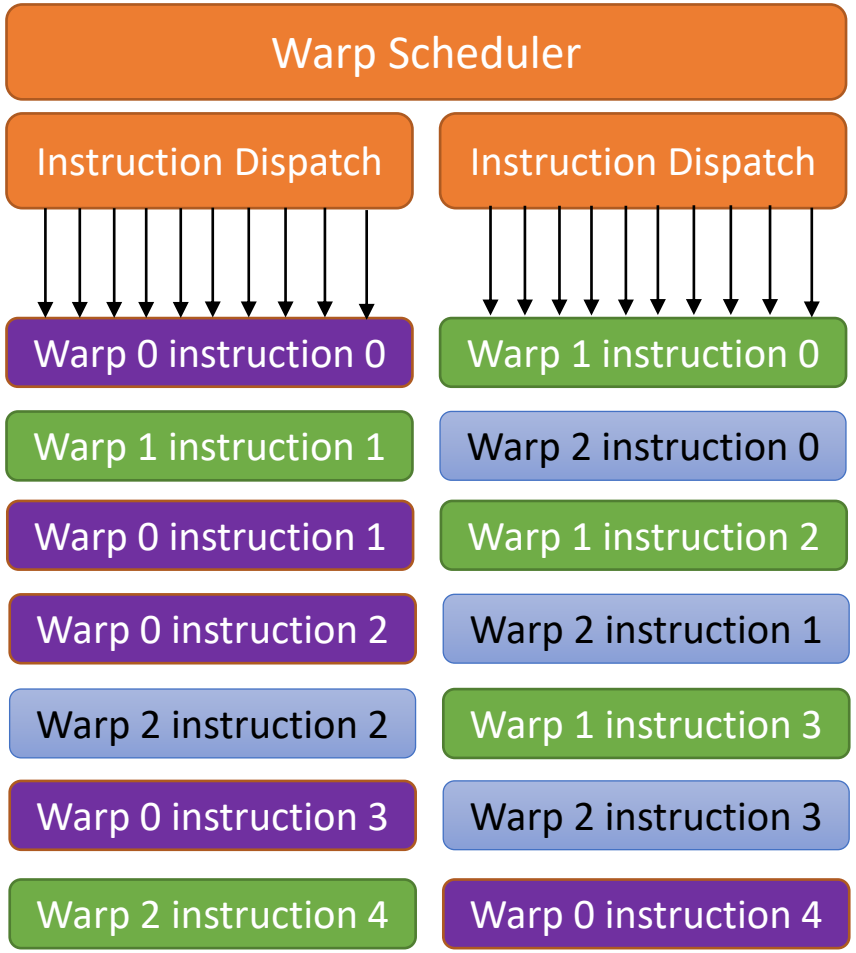


CUDA编程模型：Latency hiding

- Latency
  - Algorithmic instruction latency(**10-20 cycle**)
  - Memory instruction latency (**400-800 cycles**)
- Occupancy
  - 单个SM调度线程单位（wrap）中处于非等待状态的比例
  - 比例越高，运算单元整体的执行吞吐律就可能越高。

$$Occupancy = \frac{ative\ warps}{maximum\ warps}$$

SM



CUDA编程模型：Latency hiding

CUDA kernel

```
__global__ void kerne1(float* a, float* b) {
        int tid = blockIdx.x * blockDim.x + threadIdx.x;
        float c = a[tid] * b[tid];
}
```

Machine code

```
I0: LD R0 a[tid];
I1: LD R1 b[tid];
I2: MPY R2, R0, R1
```

Warps	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	c16	c17	c19	c20
w0	I0	I1															I2			
w1			I0	I1														I2		
w2					I0	I1													I2	
w3							I0	I1												I2
w4									I0	I1										
w5											I0	I1								
w6													I0	I1						
w7															I0	I1				

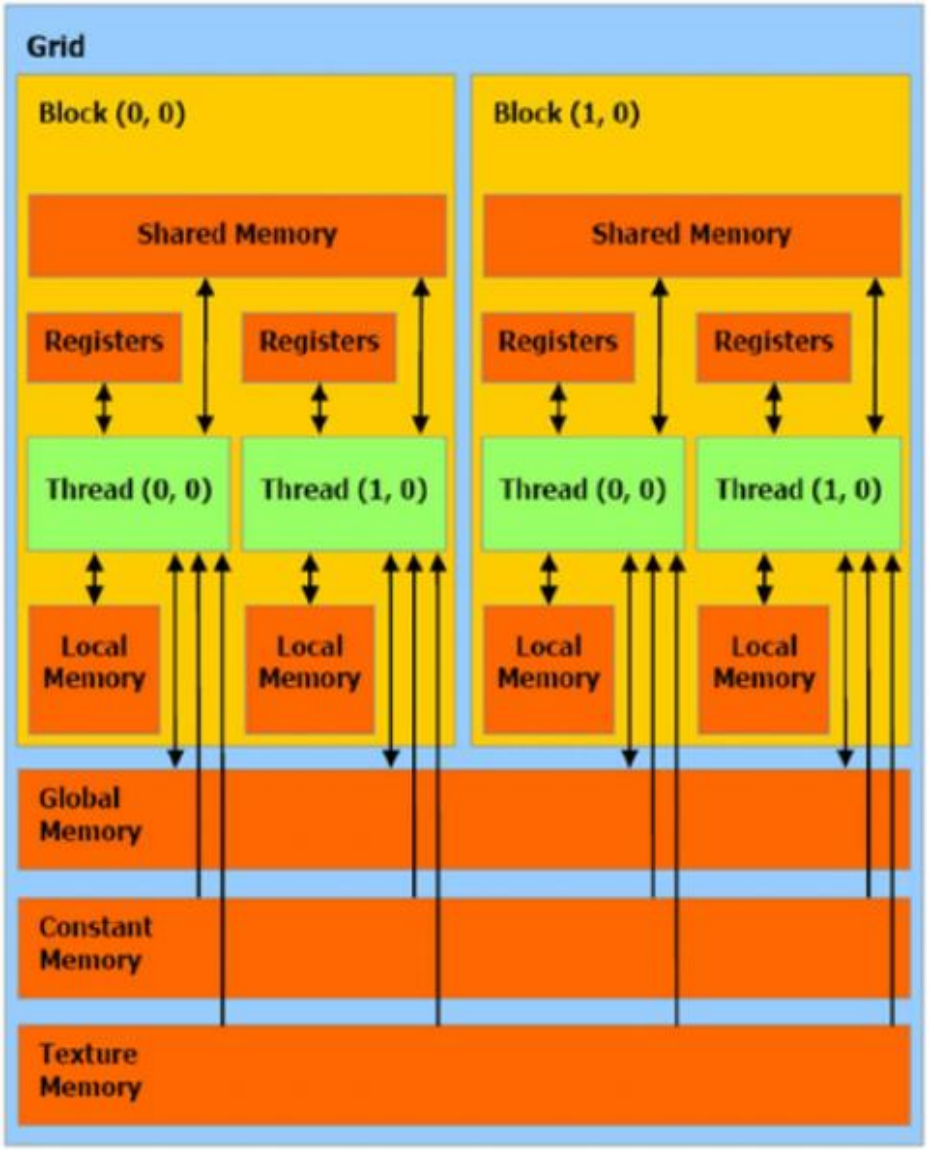
Clock Cycles

CUDA编程模型：Latency hiding

- 调整Block，提高Occupancy
  - Block中thread数目是warp的整数倍
  - Block尺寸避免太小，建议至少128以上
  - 确保Block数量远大于SM数量
  - 针对每个显卡，测试最佳配置

CUDA编程模型：Latency hiding

- 多级缓存结构，降低Memory instruction latency
  - Local **Registers**：每个线程特有，数量小，Kepler(255)
  - **Local Memory**: Registers不够时启用，慢，cached
  - **Shared Memory**：block内所有线程共享
  - **Constant Memory**: 只读，所有处理器共享
  - **Texture Memory**: 只读，所有处理器共享
  - **Global Memory**: 可读写, 所有处理器共享



CUDA编程模型：多级缓存结构

- 多级缓存结构

CUDA Code	Memory	Performance
int var;	register	1×
int local[20];	local	100×
__shared__ int sharedVar;	shared	1×
__device__ int globalVar;	global	100×
__constant__ int constVar;	constant	1×

- Make use of shared memory wherever possible

CUDA编程模型：多级缓存结构

- 多级缓存结构

CUDA Code	Memory	Scope	Lifetime
int var;	register	thread	thread
int local[20];	local	thread	thread
__shared__ int sharedVar;	shared	block	block
__device__ int globalVar;	global	grid	application
__constant__ int constVar;	constant	grid	application

CUDA编程样例：规约算法

- 输入：$a_0, a_1, \cdots\ a_N$
- 输出：$a_0 \oplus a_1 \ldots \oplus a_N$，其中$\oplus$代表符合结合律的操作符

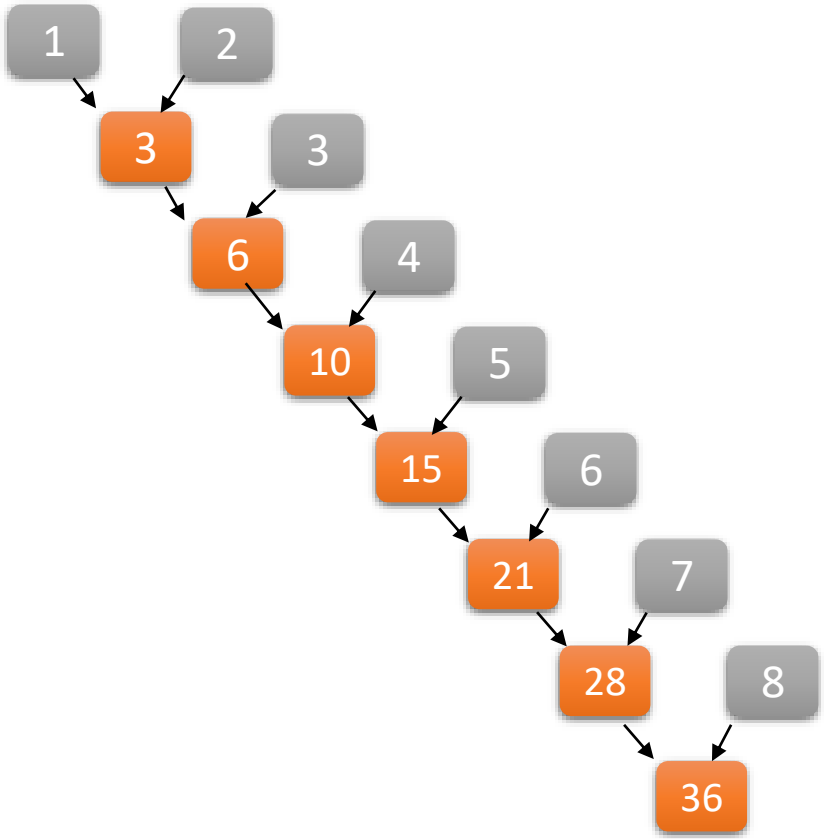
- 举例
  - 求和
  - 求最大值
  - 求最小值

CUDA编程样例：规约算法

- 输入：$a[8] = \{1, 2, 3, 4, 5, 6, 7, 8\}$;
- 输入：求和

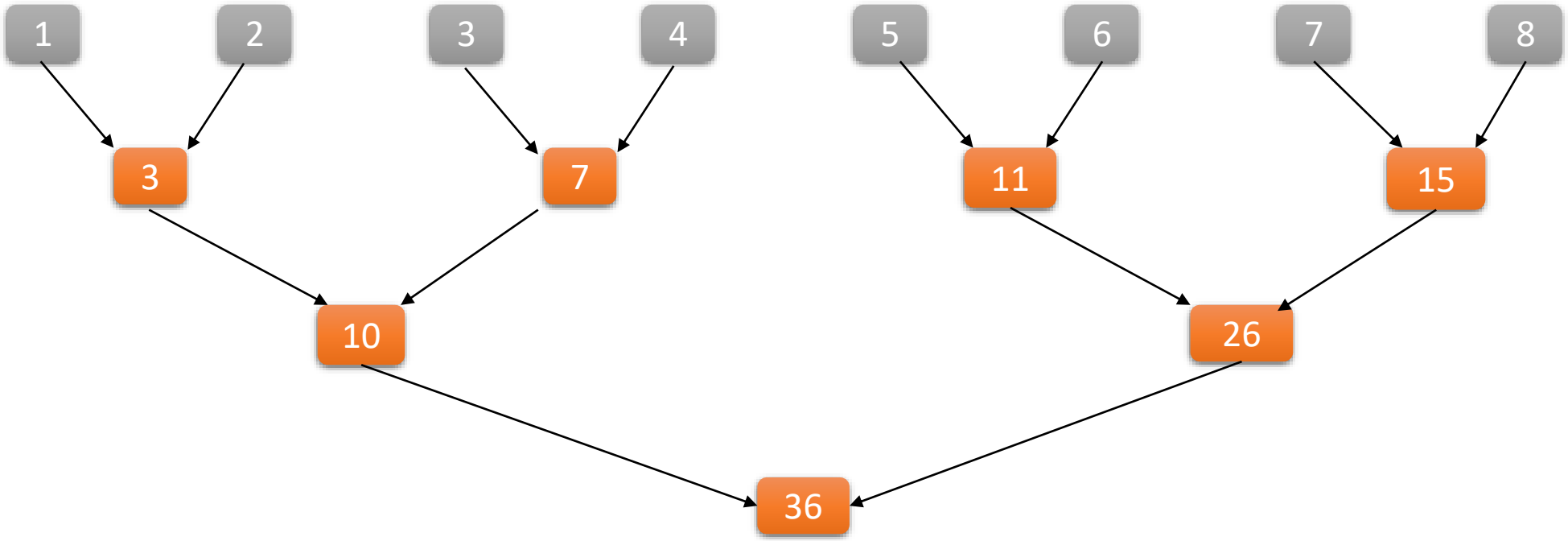
C++

```
int sum(int* a)
{
    int ret = 0;
    for (int i = 0; i < N; i++)
    {
        ret += a[i];
    }
    return ret;
}
```



CUDA编程样例：规约算法

- 输入：$a[8] = \{1, 2, 3, 4, 5, 6, 7, 8\}$;
- 输入：求和

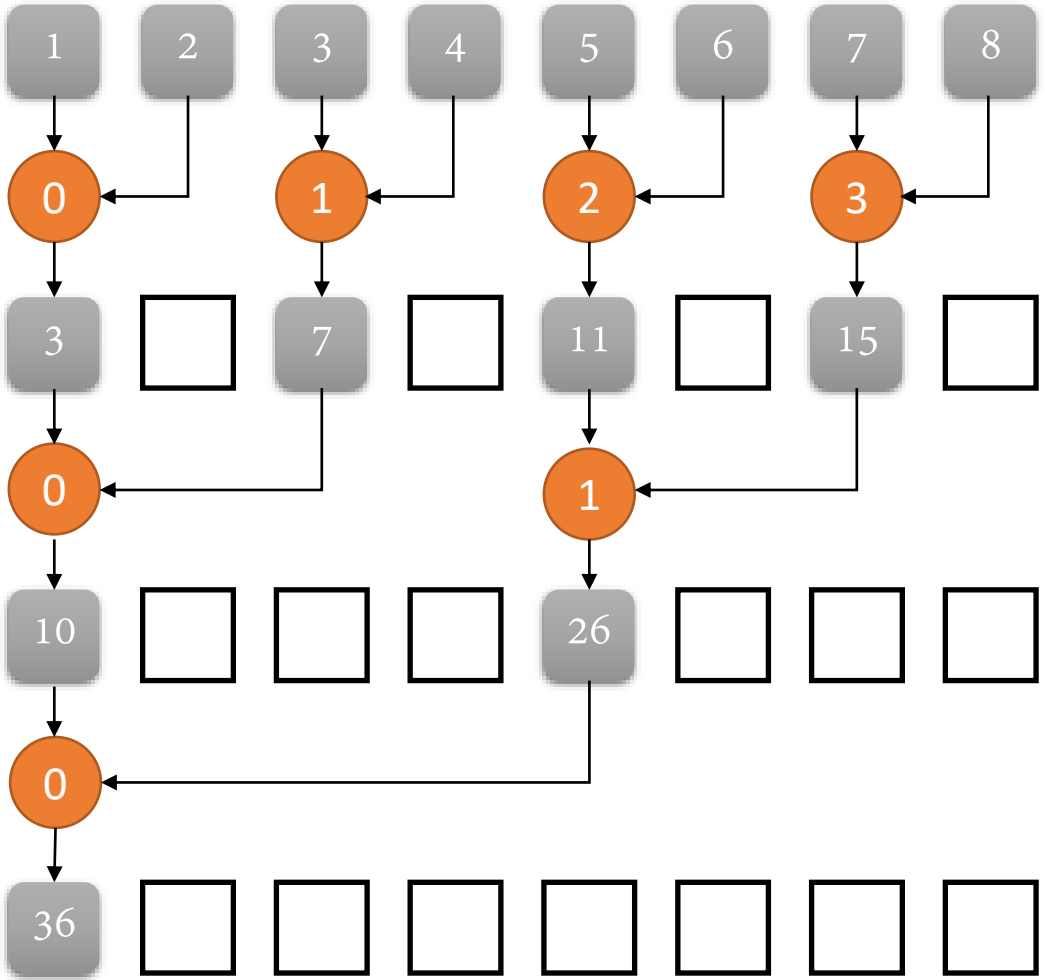


CUDA编程样例：规约算法

有哪些问题？

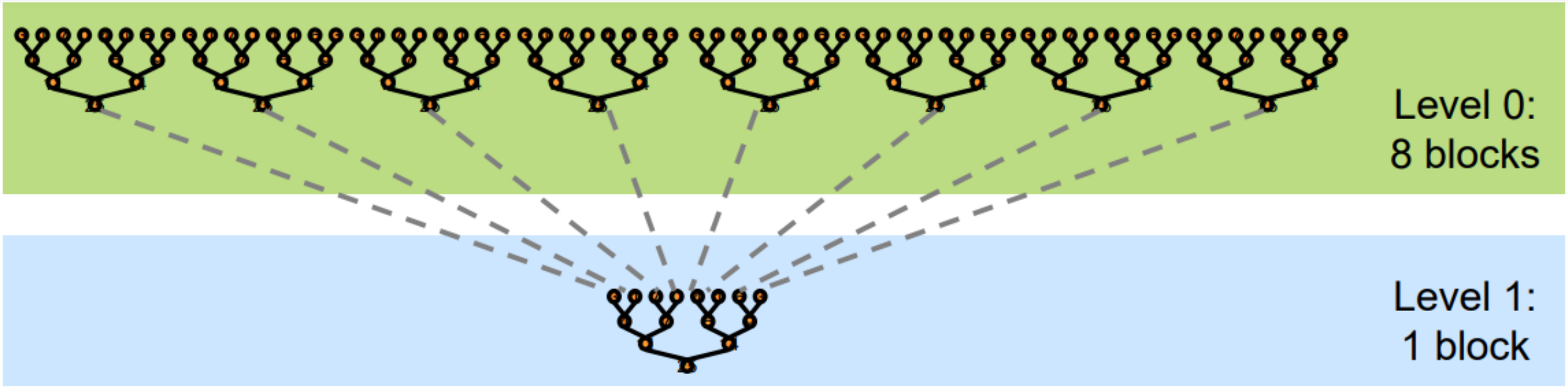
CUDA

```
__global__ void sum(int*  a, int N)
{
    const int tid = threadIdx.x;
    auto step_size = 1;
    int number_of_threads = N;
    while (number_of_threads > 0)
    {
        if (tid < number_of_threads)
        {
            const auto first = tid * step_size * 2;
            const auto second = first + step_size;
            a[first] += a[second];
        }
        step_size <<= 1;
        number_of_threads >>= 1;
    }
}
```



CUDA编程样例：规约算法

- 问题一：数据尺寸大于1024怎么办，block size ≤ 1024



CUDA编程样例：规约算法

- 问题二：如何降低延迟

CUDA

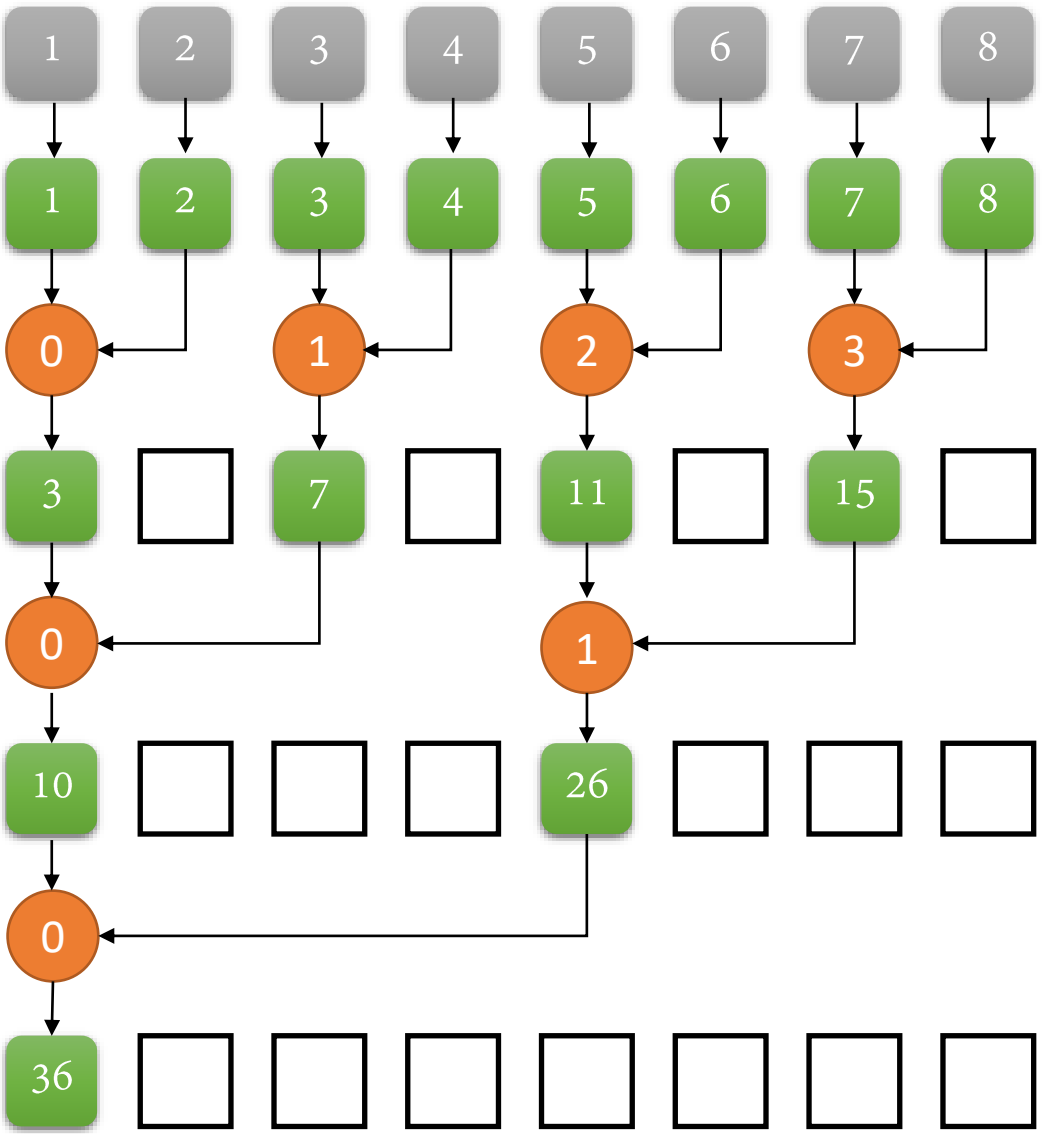
```
__global__ void sum(int*  a, int N)
{
        extern __shared__ int sharedMem[];

        uint tid = threadIdx.x;
        uint id = blockIdx.x* blockDim.x + threadIdx.x;

        sharedMem[tid] = a[id];


}
```

Shared Memory



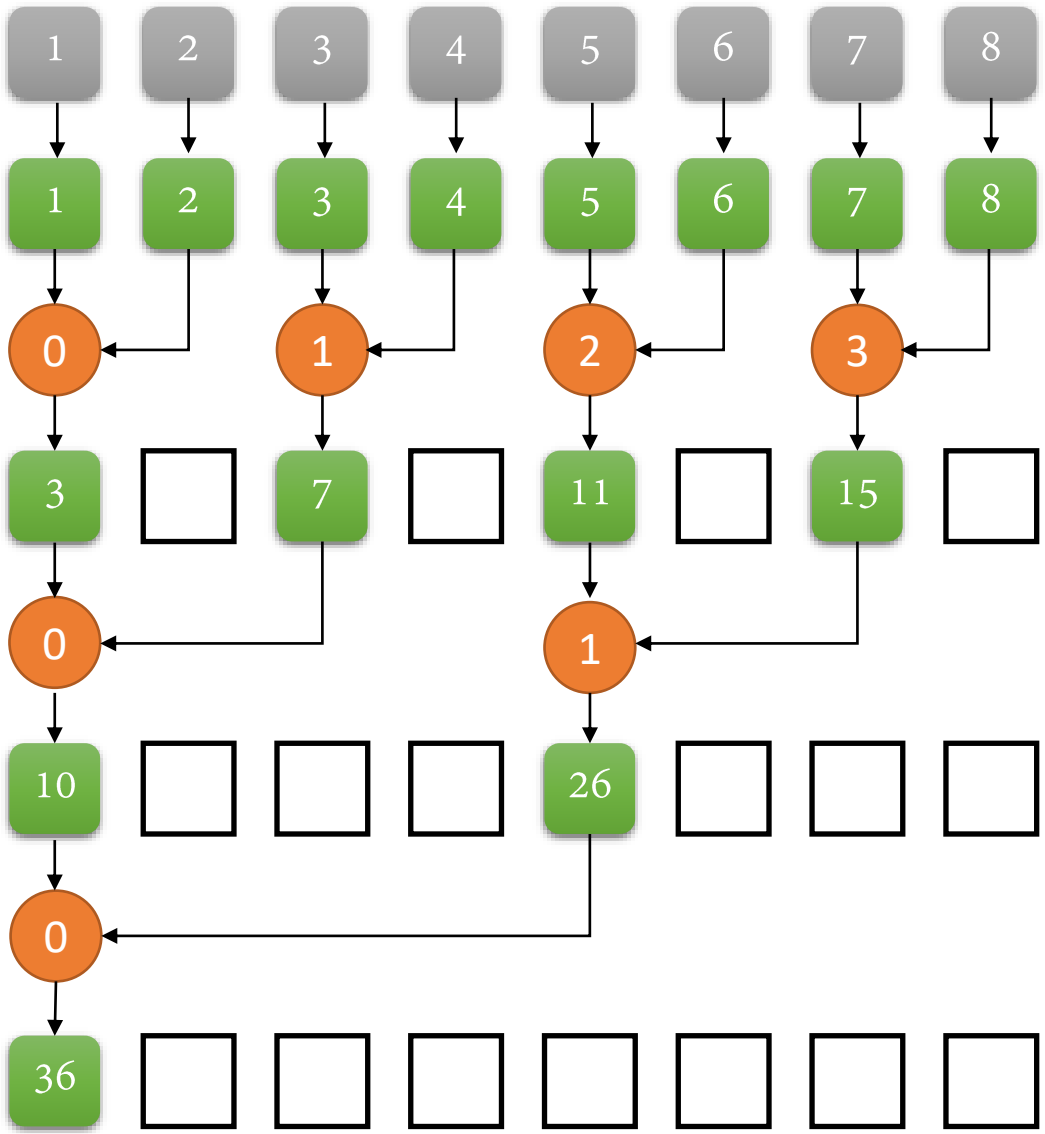
CUDA编程样例：规约算法

- 问题二：如何降低延迟

CUDA

```
__global__ void sum(int*  a, int N)
{
        ...
        for (int s = 0; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0)
        {
                sharedMem[tid] += sharedMem[tid + s];
        }
        __syncthreads();
        }
}
```

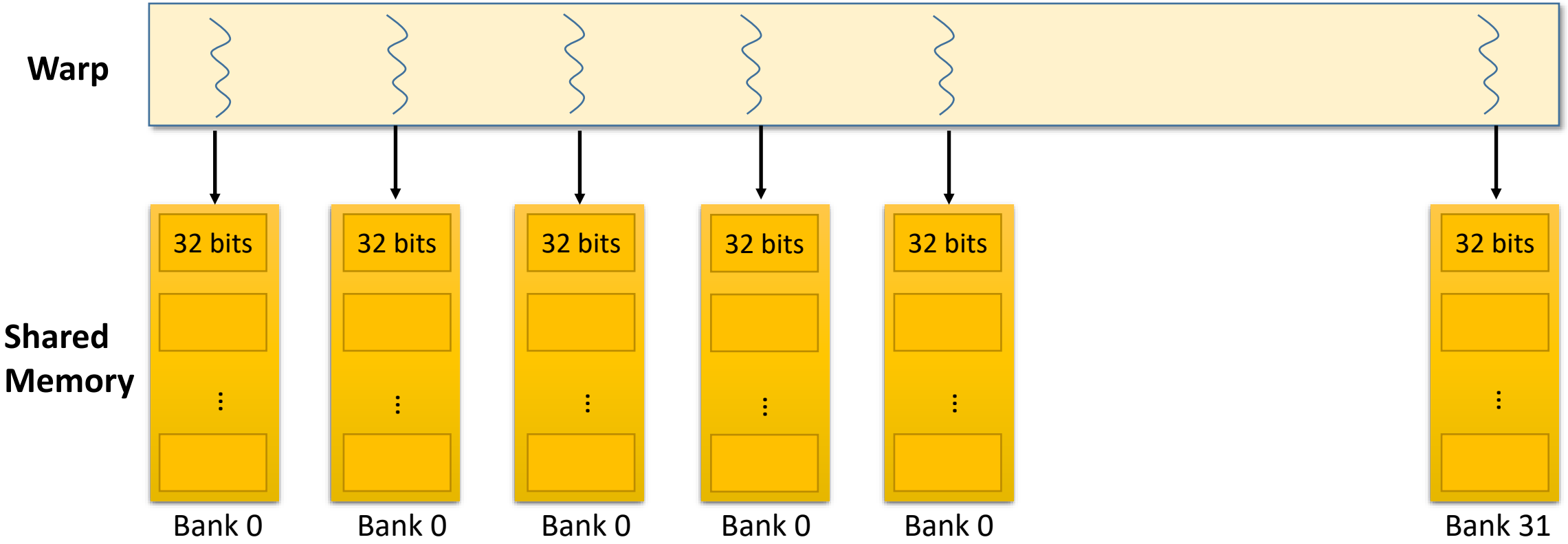
Shared Memory



Shared Memory Bank Conflicts

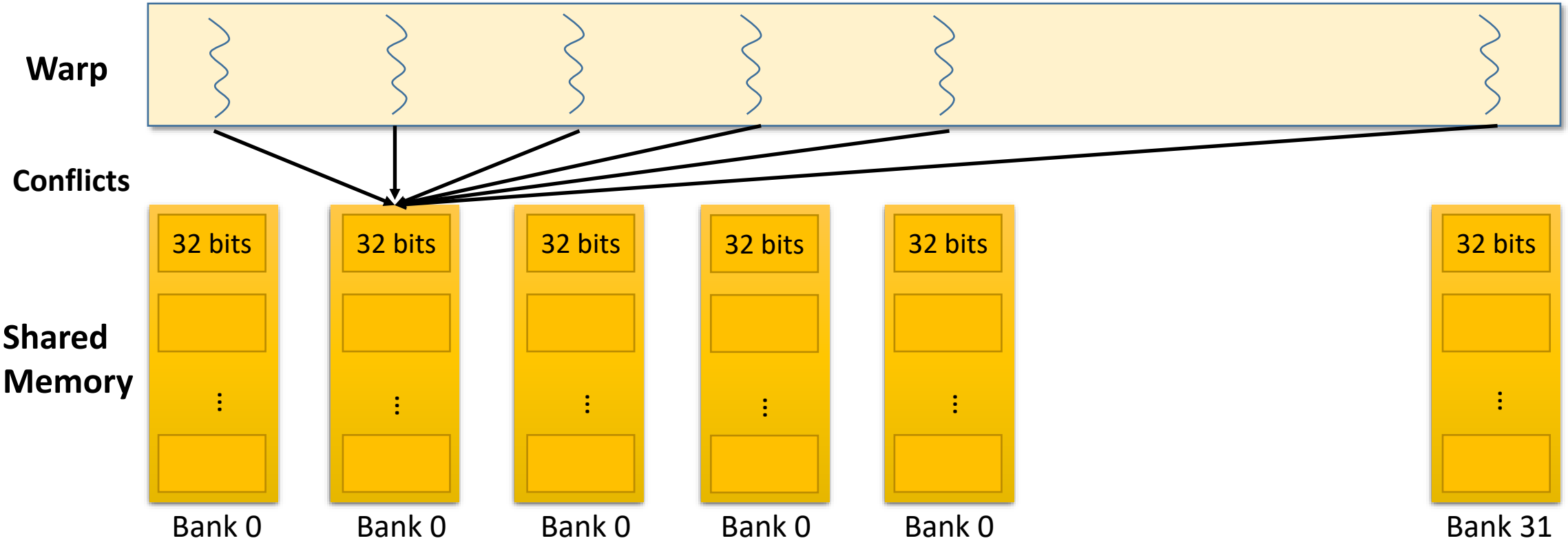
CUDA编程样例：规约算法

- 为了实现内存高带宽的同时访问，shared memory被划分成了可以同时访问的等大小内存块(banks)。
- Shared Memory 中连续的32位字被分配到连续的banks，每个clock cycle每个bank的带宽是32bits



CUDA编程样例：规约算法

- 为了实现内存高带宽的同时访问，shared memory被划分成了可以同时访问的等大小内存块(banks)。
- Shared Memory 中连续的32位字被分配到连续的banks，每个clock cycle每个bank的带宽是32bits



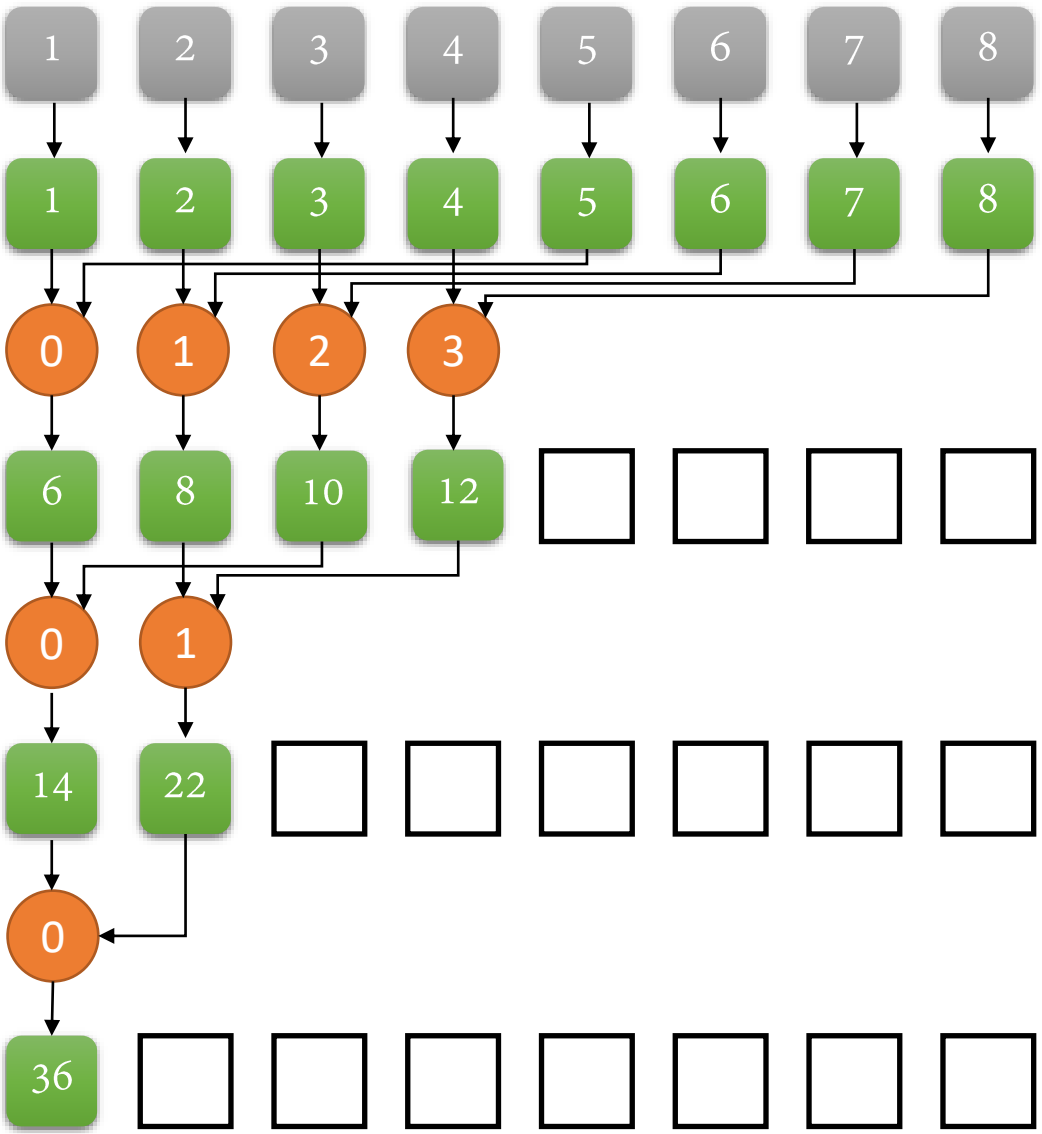
CUDA编程样例：规约算法

- 问题三：如何提升Occupancy

CUDA

```
__global__ void sum(int*  a, int N)
{
        ...
        for (int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s)
        {
                sharedMem[tid] += sharedMem[tid + s];
        }
        __syncthreads();
        }
}
```

Shared Memory

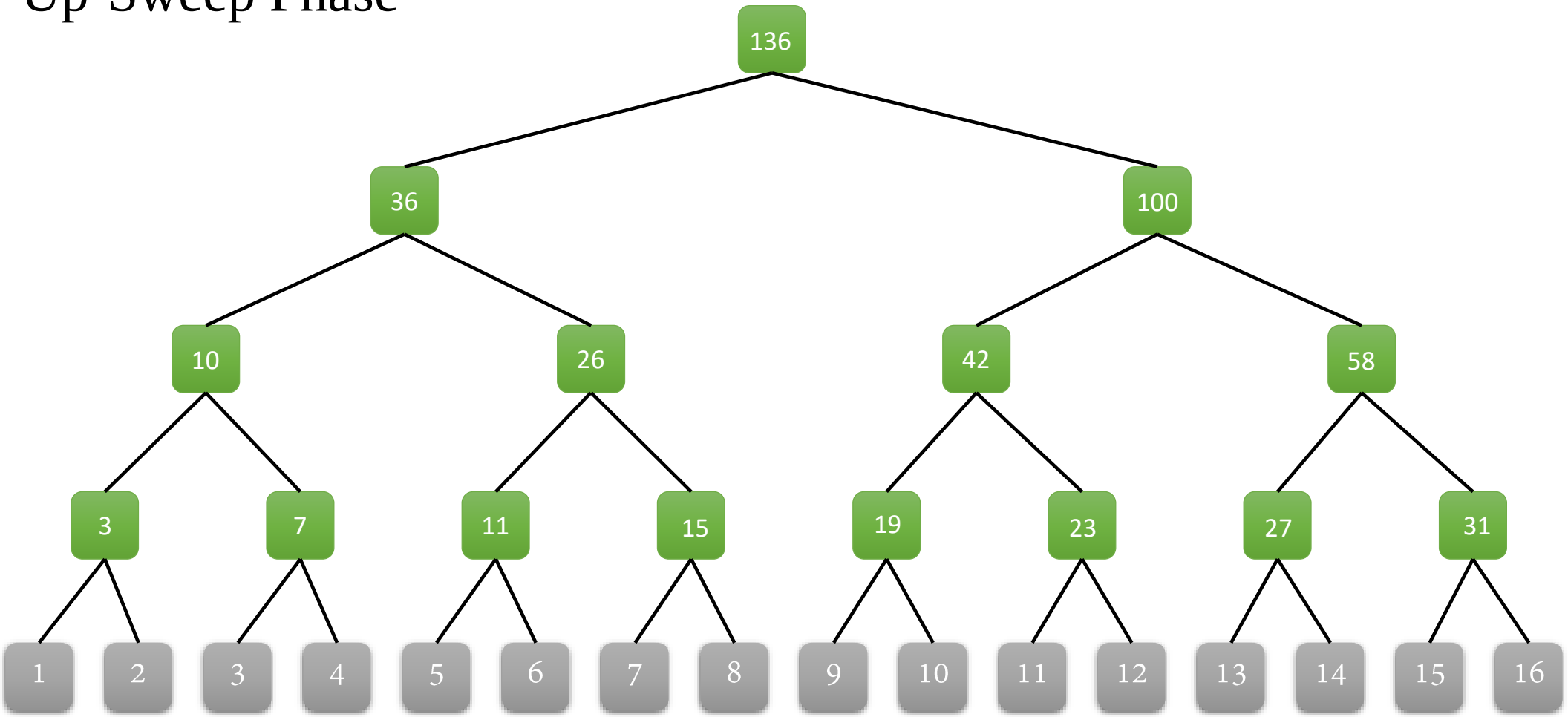


CUDA编程样例：前缀和(Scan)

- 输入：$a_0, a_1, \cdots, a_N$
- 输出：$a_0 = a_0, a_1 = a_0 \oplus a_1, \cdots, a_N = a_0 \oplus a_1 \oplus \cdots \oplus a_N$

CUDA编程样例：前缀和(Scan)

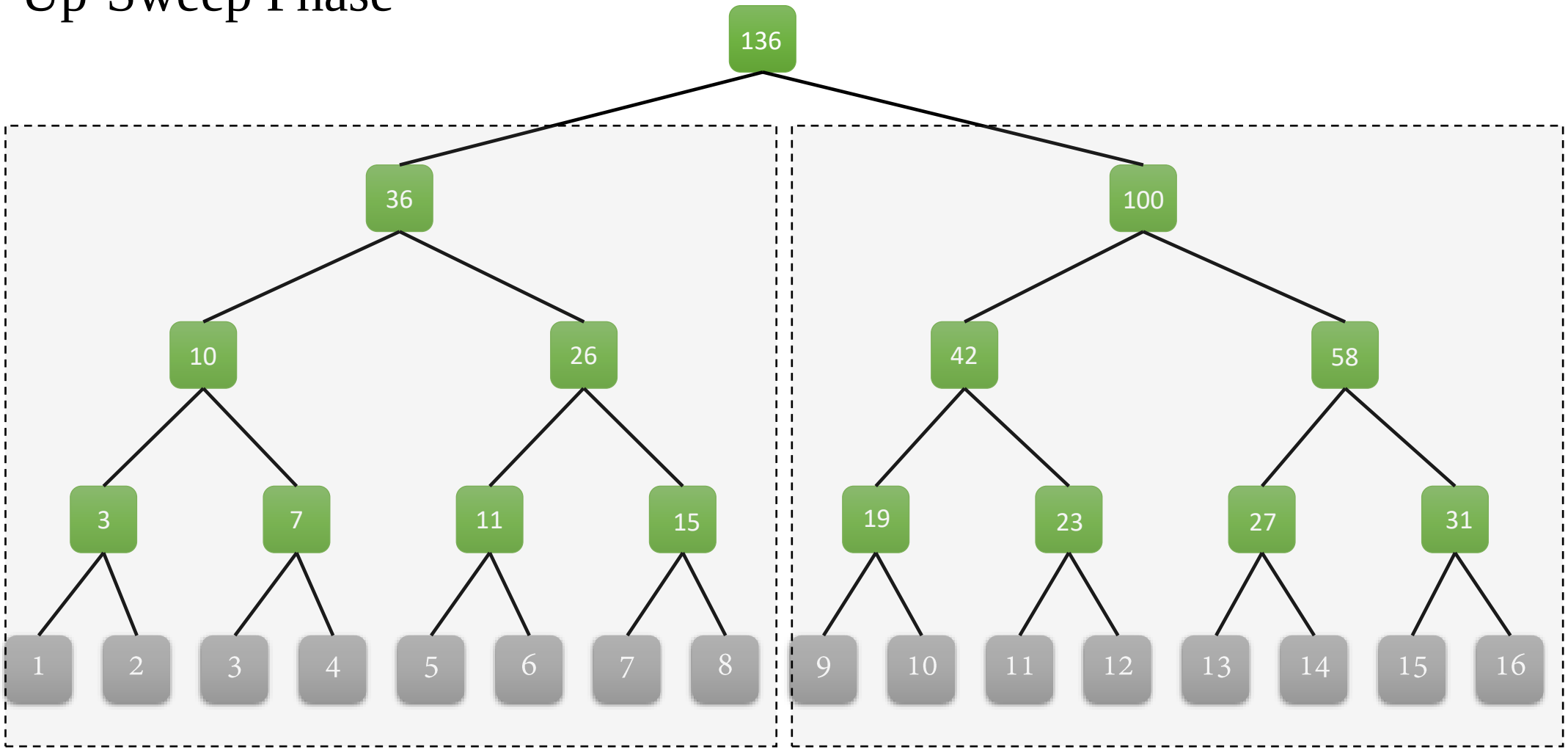
- Up-Sweep Phase



[Blelloch 1990]

CUDA编程样例：前缀和(Scan)

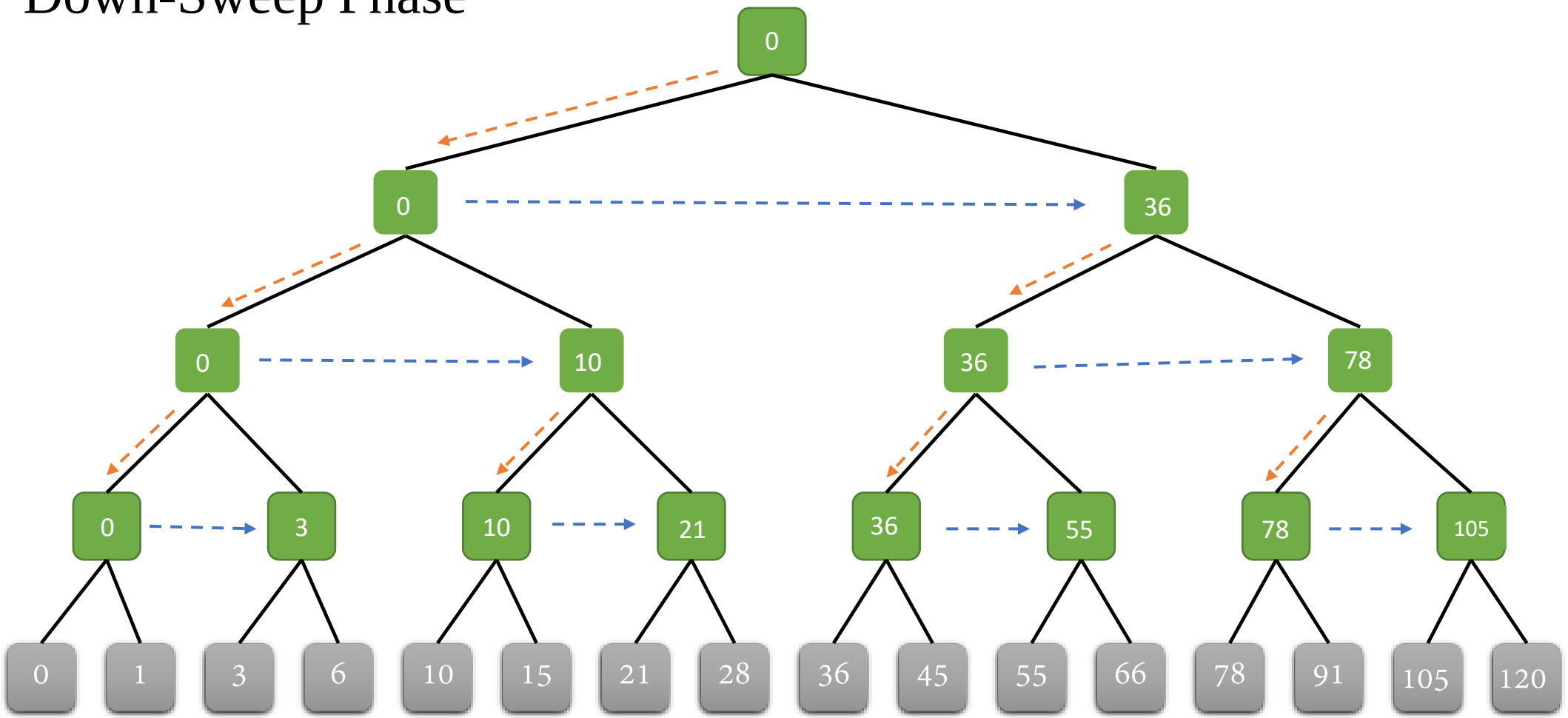
- Up-Sweep Phase



[Blelloch 1990]

CUDA编程样例：前缀和(Scan)

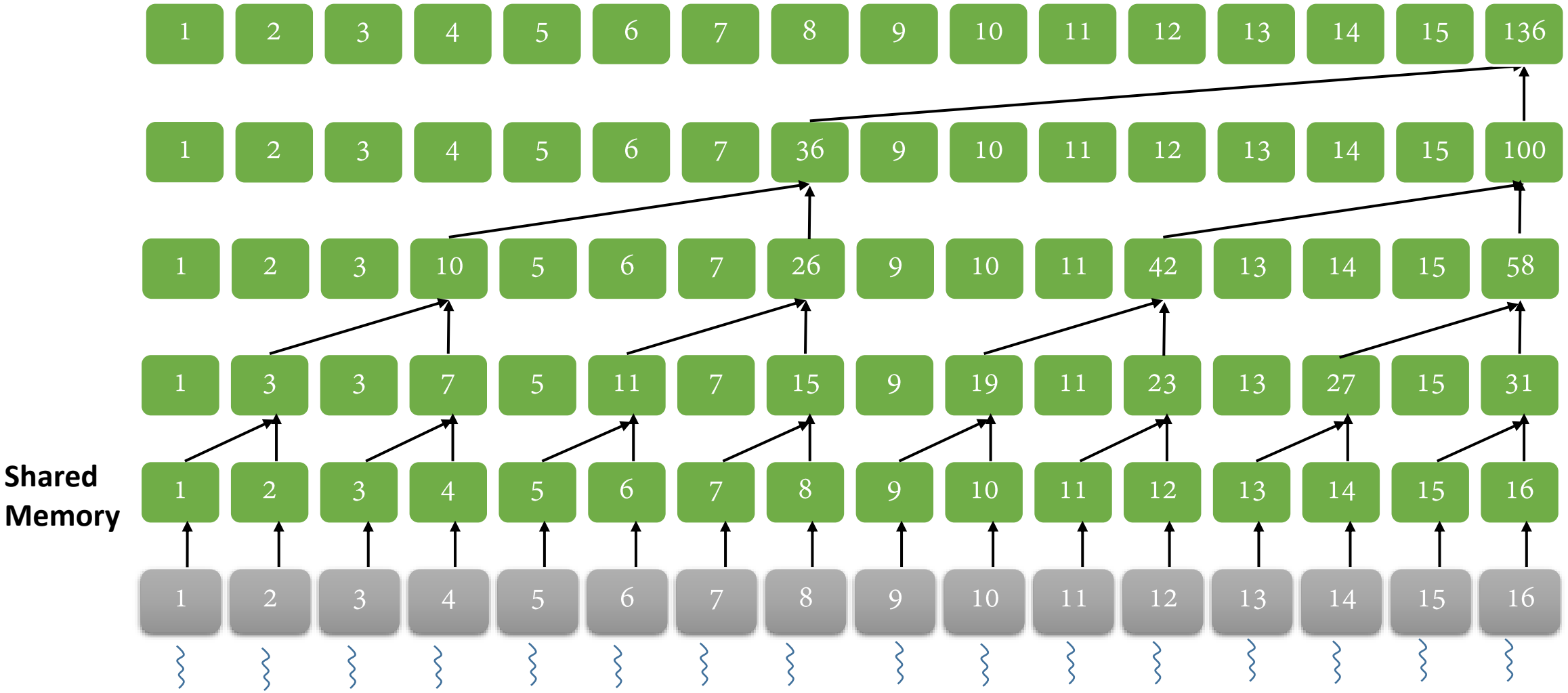
- Down-Sweep Phase



[Blelloch 1990]

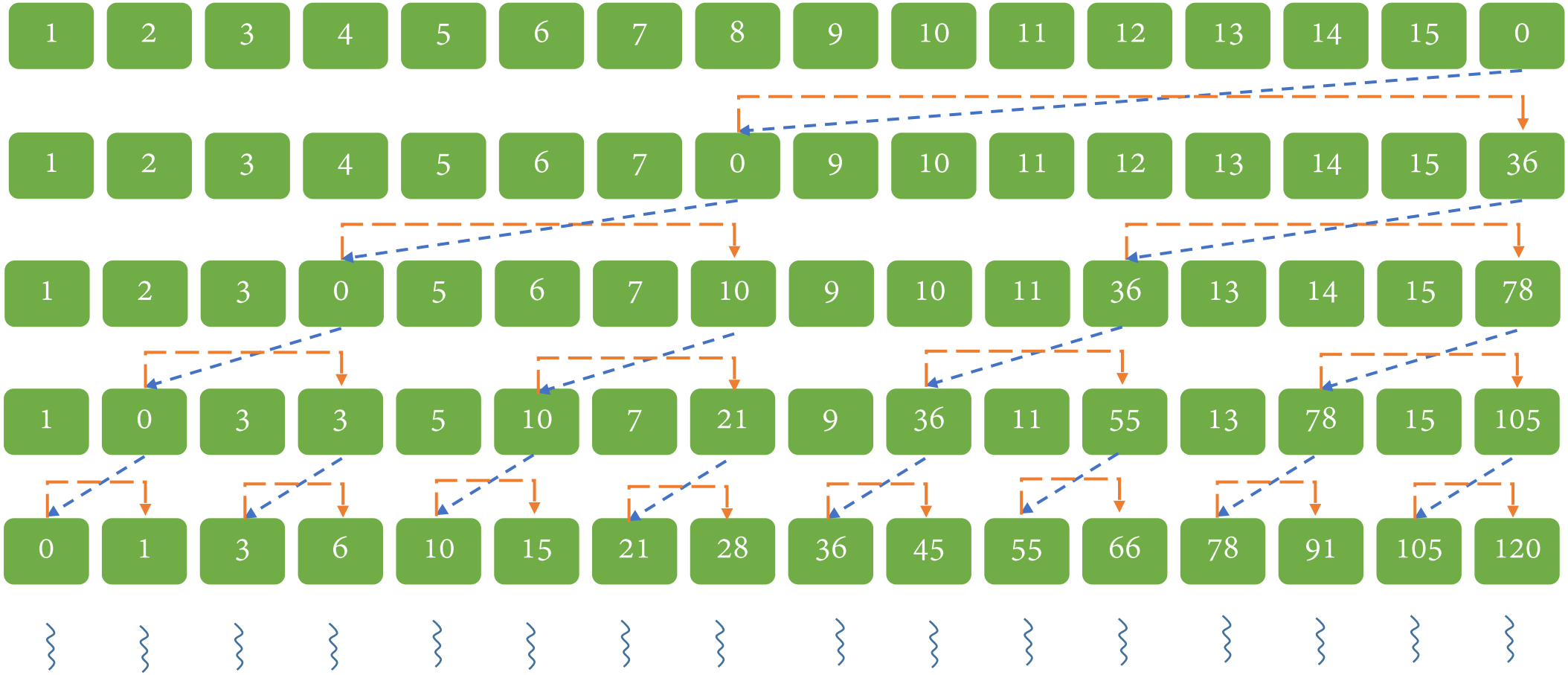
CUDA编程样例：前缀和(Scan)

- GPU实现：Up-Sweep Phase



CUDA编程样例：前缀和(Scan)

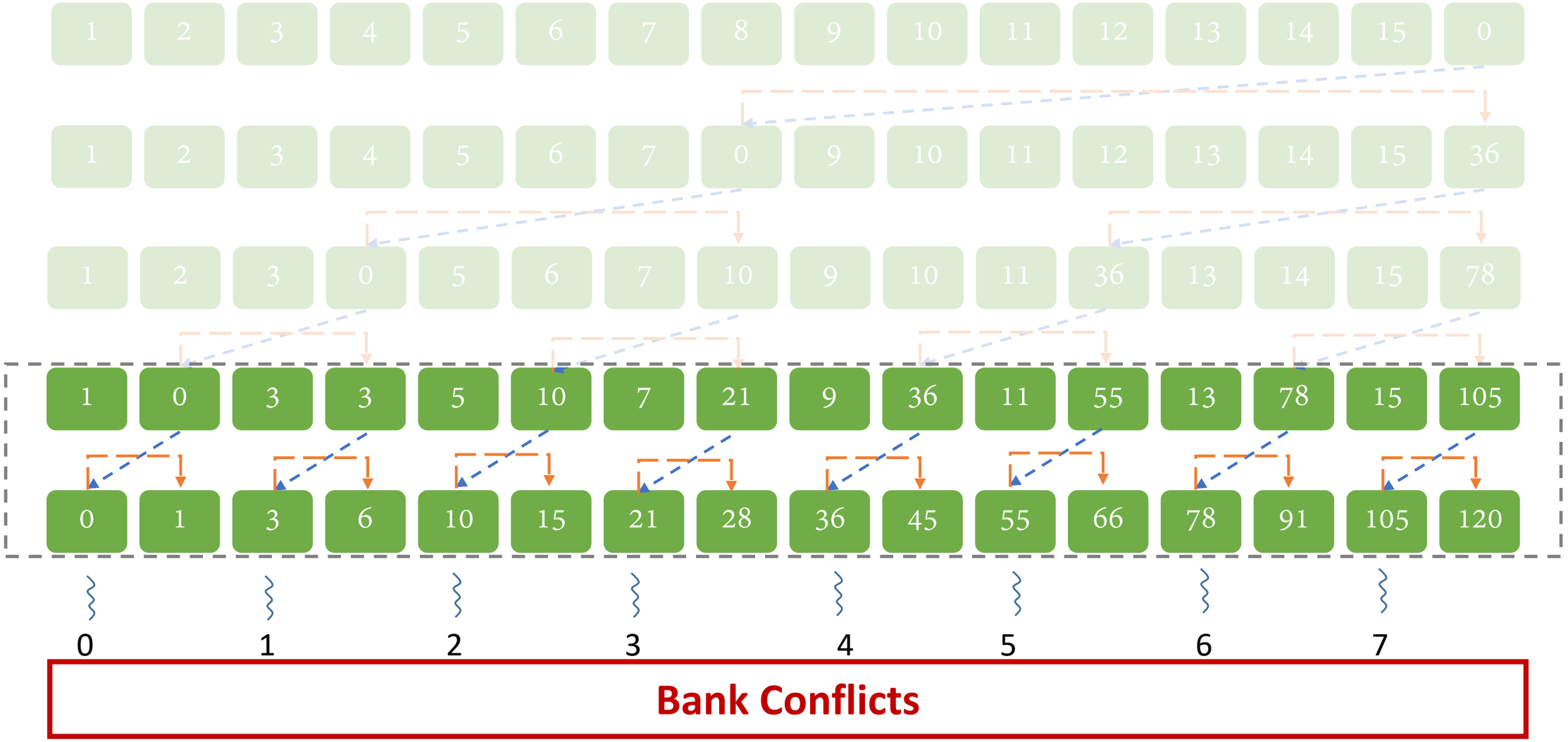
- GPU实现：Down-Sweep Phase



Occupancy低

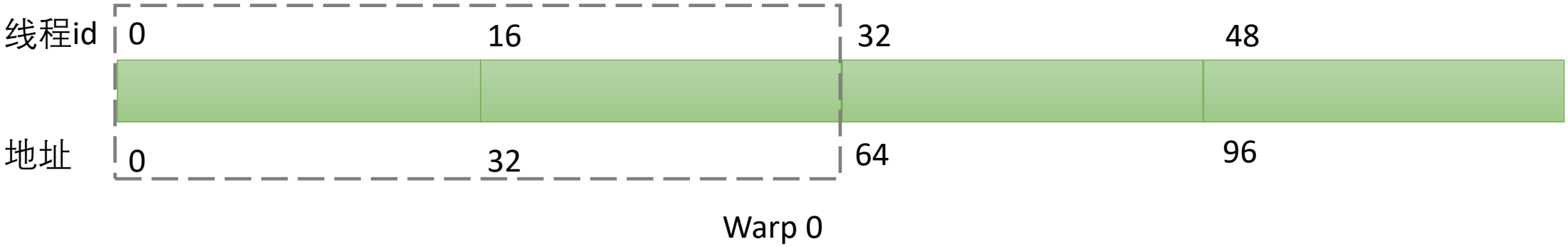
CUDA编程样例：前缀和(Scan)

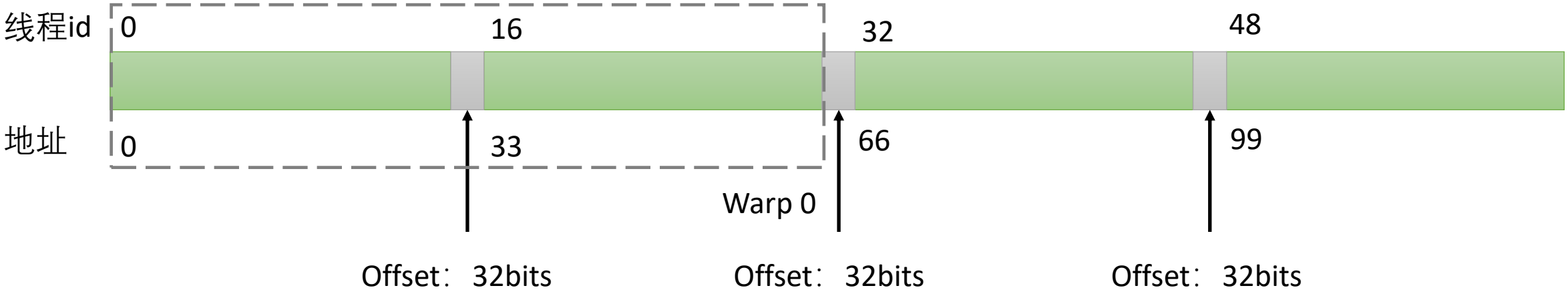
- GPU实现：Down-Sweep Phase



CUDA编程样例：前缀和(Scan)

- Bank Conflicts





C++模板

- 减少重复劳动
- 模板申明
  - 模板函数： template <typename T> int compare (T t1, T t2);
  - 模板类： template <typename T> class compare;
- 模板定义

```
template <typename T>
int compare( T& t1, T& t2)
{
    if (t1 > t2)
        return 1;
    if (t1 == t2)
        return 0;
    if (t1 < t2)
        return -1;
}
```

```
template <typename T>
class compare
{
private:
    T _val;
public:
    explicit compare( T& val) : _val(val) { }
    bool operator==( T& t){
        return _val == t;
    }
};
```

C++模板

- 减少重复劳动
- 模板类型

```
__global__ void sum(int*  a, int N)
{
        ...
}
```

```
__global__ void sum(float*  a, int N)
{
        ...
}
```

忌：代码重复

模板函数

```
template<typename T>
__global__ void sum(T*  a, int N)
{
}
```

实例化

int

```
__global__ void sum(int*  a, int N)
{
}
```

float

```
__global__ void sum(float*  a, int N)
{
}
```

double

```
__global__ void sum(double*  a, int N)
{
}
```

自定义类型/自定义实现

```
__global__ void sum(double*  a, int N)
{
}
```

C++模板

- 模板特化
  - 模板参数在某种特定类型下的具体实现称为模板的特化

```
template <typename T, int Dim>
class Vector
{
public:
        DYN_FUNC Vector() {};
        DYN_FUNC ~Vector() {};
};
```

全特化

```
template <>
class Vector<float, 2>
{
public:
        DYN_FUNC Vector();
        DYN_FUNC ~Vector();
private:
        T x, y;
};
```

```
template <>
class Vector<float, 3>
{
public:
        DYN_FUNC Vector();
        DYN_FUNC ~Vector();
private:
        T x, y, z;
};
```

C++模板

- 模板特化
  - 模板参数在某种特定类型下的具体实现称为模板的特化

```
template <typename T, int Dim>
class Vector
{
public:
        DYN_FUNC Vector() {};
        DYN_FUNC ~Vector() {};
};
```

偏特化

```
template <typename T>
class Vector<T, 2>
{
public:
        DYN_FUNC Vector();
        DYN_FUNC ~Vector();
private:
        T x, y;
};
```

```
template <typename T>
class Vector<T, 3>
{
public:
        DYN_FUNC Vector();
        DYN_FUNC ~Vector();
private:
        T x, y, z;
};
```

C++模板:规约算法模板函数

```
template<typename T, typename Function>
T Reduce(T* pData, uint num, T* pAux, Function func, T v0)
{
    uint n = num;
    uint sharedMemSize = REDUCTION_BLOCK * sizeof(T);
    uint blockNum = cudaGridSize(num, REDUCTION_BLOCK);
    T* subData = pData;
    T* aux1 = pAux;
    T* aux2 = pAux + blockNum;
    T* subAux = aux1;
    while (n > 1) {
        KerReduce<T, REDUCTION_BLOCK, Function> << <blockNum, REDUCTION_BLOCK, sharedMemSize >> > (subData, n, subAux, func, v0);
        n = blockNum;
        blockNum = cudaGridSize(n, REDUCTION_BLOCK);
        if (n > 1) {
            subData = subAux; subAux = (subData == aux1 ? aux2 : aux1);
        }
    }

    T val;
    if (num > 1)
        cudaMemcpyAsync(&val, subAux, sizeof(T), cudaMemcpyDeviceToHost);
    else
        cudaMemcpyAsync(&val, pData, sizeof(T), cudaMemcpyDeviceToHost);

    return val;
}
```

src/Core/Backend/Cuda/Algorithm/Reduction.cu

C++模板:规约算法模板函数

- 求和 `Reduce(val, num, m_aux, PlusFunc<T>(), (T)0);`
- 求最大值 `Reduce(val, num, m_aux, MaximumFunc<T>(), -(T)REAL_MAX);`
- 求最小值 `Reduce(val, num, m_aux, MinimumFunc<T>(), (T)REAL_MAX);`

src/Core/Backend/Cuda/Algorithm/Reduction.cu

算法应用

- 场景一：SPH邻域查找

碎片化严重，效率极低

邻域查找

C++ STL

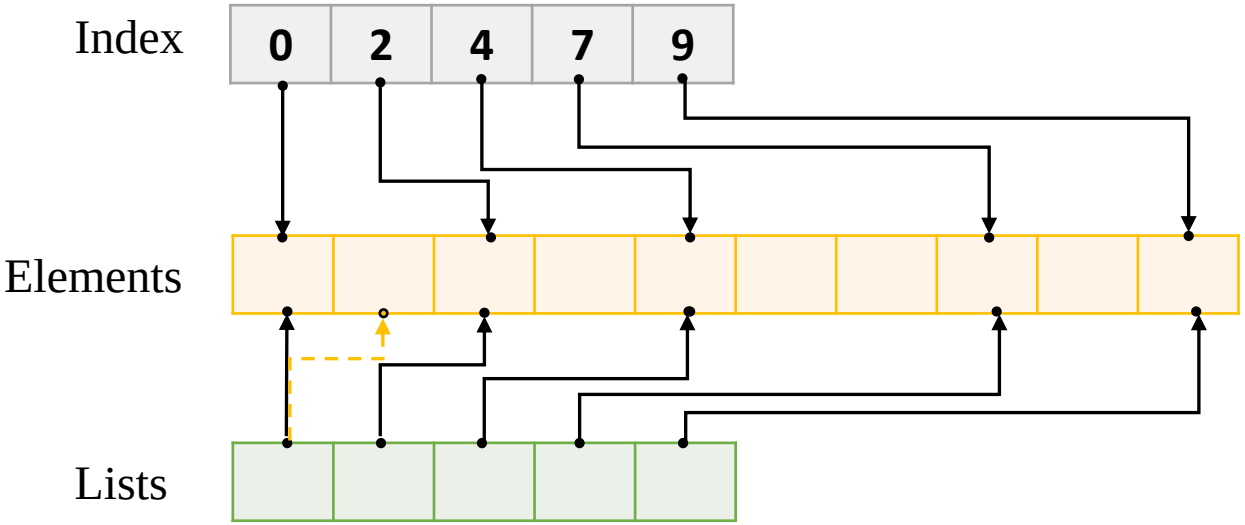
```
std::vector<std::vector<int>> neighbors;
```

算法应用

- 场景一：SPH邻域查找

```
template<class ElementType>
class ArrayList<ElementType, DeviceType::GPU>
{
public:
    ArrayList(){};
    ~ArrayList() {};

    bool resize(const DArray<uint>& counts);
private:
    DArray<uint> mIndex;
    DArray<ElementType> mElements;
    DArray<List<ElementType>> mLists;
};
```



算法应用

- 场景一：SPH邻域查找

2	2	5	3	3	7	4	4	5	3	2	2	1	1	5	2

```
template<class ElementType>
class ArrayList<ElementType, DeviceType::GPU>
{
public:
    ArrayList(){};
    ~ArrayList() {};

    bool resize(const DArray<uint>& counts);
private:
    DArray<uint> mIndex;
    DArray<ElementType> mElements;
    DArray<List<ElementType>> mLists;
};
```

```
template<class ElementType>
bool ArrayList<ElementType, DeviceType::GPU>::resize(const DArray<uint>& counts)
{
    assert(counts.size() > 0);

    if (mIndex.size() != counts.size())
    {
        mIndex.resize(counts.size());
        mLists.resize(counts.size());
    }

    mIndex.assign(counts);

    Reduction<uint> reduce;
    uint total_num = reduce.accumulate(mIndex.begin(), mIndex.size());

    Scan<uint> scan;
    scan.exclusive(mIndex);

    //printf("total num 2 = %d\n", total_num);

    mElements.resize(total_num);

    parallel_allocate_for_list<sizeof(ElementType)>(mLists.begin(), mElements.begin(), mElements.size()

    return true;
}
```

src/Core/Backend/Cuda/Array/ArrayList.inl

算法应用

- 场景二：已知点云，求Bounding Box

```
Reduction<Vec3f> reduce;
Vec3f hiBound = reduce.maximum(points.begin(), points.size());
Vec3f loBound = reduce.minimum(points.begin(), points.size());
```

算法应用

- 场景三：输入一个数组，去掉重复元素

0	2	5	3	3	7	4	4	5	3	2	2	1	1	0	2

0	0	1	1	2	2	2	2	3	3	3	4	4	5	5	7	Sort

1	0	1	0	1	0	0	0	1	0	0	1	0	1	0	1	Reduce

0	1	1	2	2	3	3	3	3	4	4	4	5	5	6	6	Exclusive scan

0	1	2	3	4	5	7

Homework： git checkout homework(Tests/Homework)

Question