Weiming HPC Training Camp
×
LCPU AI Infra Seminars

北京大学未名超算队
北京大学学生 Linux 俱乐部

2. Memory Abstraction & Hierarchy

Weiming Supercomputing Team
Linux Club of Peking University

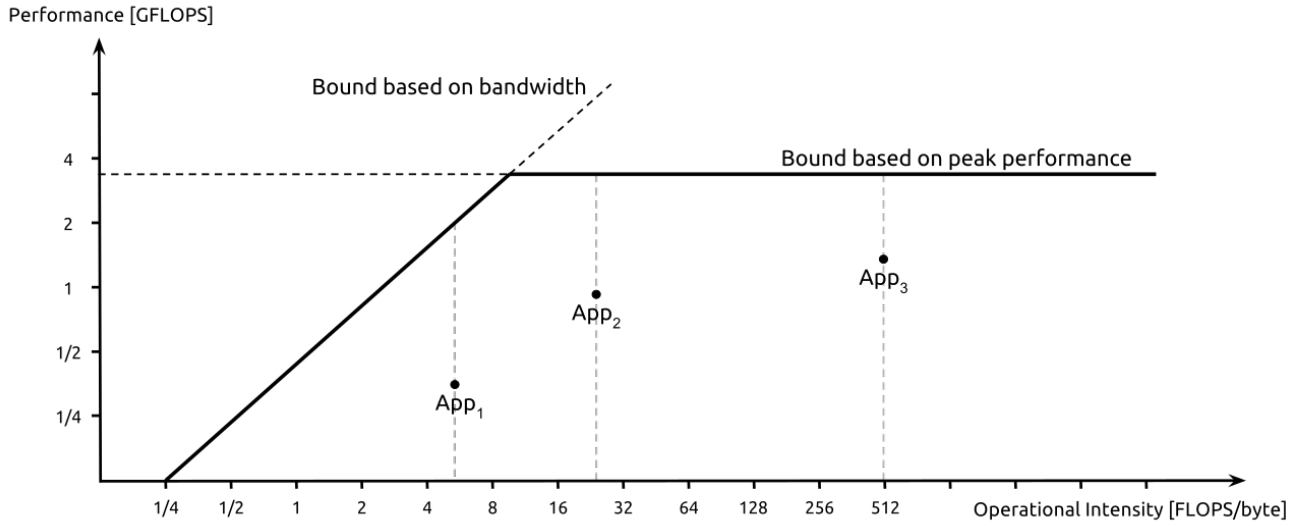
Linux俱乐部项目

性能限制 WHAT BOUNDS A KERNEL

性能限制 ROOFLINE MODEL

$$P = \min\left(P_{\text{peak}}, \text{BW} \times \text{I}\right)$$

$$\text{I} = \frac{\text{FLOPs}}{\text{Bytes from DRAM}}$$

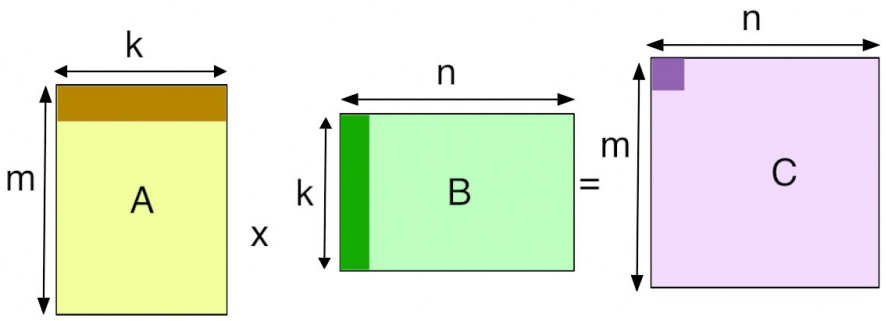


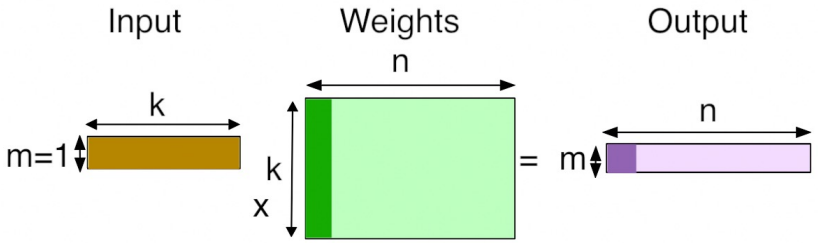
性能限制 ROOFLINE MODEL

GPU / 执行单元	峰值算力	HBM 带宽	Ridge point
V100 FP32 CUDA core	15.7 TFLOP/s	0.90 TB/s	17
A100 FP32 CUDA core	19.5 TFLOP/s	2.04 TB/s	10
H100 FP32 CUDA core	67 TFLOP/s	3.35 TB/s	20
V100 FP16 tensor core	125 TFLOP/s	0.90 TB/s	139
A100 FP16 tensor core	312 TFLOP/s	2.04 TB/s	153
H100 FP16 tensor core	989 TFLOP/s	3.35 TB/s	295
H100 FP8 tensor core	1979 TFLOP/s	3.35 TB/s	591

性能限制 ROOFLINE MODEL

Workload	FLOP	内存搬运 (Bytes)	I
Elementwise / SAXPY y=a*x+y	$2n$	$12n$ (读 x、读 y、写 y)	$1/6 \approx 0.17$
LayerNorm / Softmax / RMSNorm	$O(n)$	$O(n)$ (读一遍、写一遍)	$O(1)$
GEMM ($M \times K \cdot K \times N$)	$2MNK$	$2(MK + KN + MN)$	$(MNK)/(MK + KN + MN)$
GEMV (batch=1, $M \times K$)	$2MK$	$2(MK + K + M)$	≈ 1



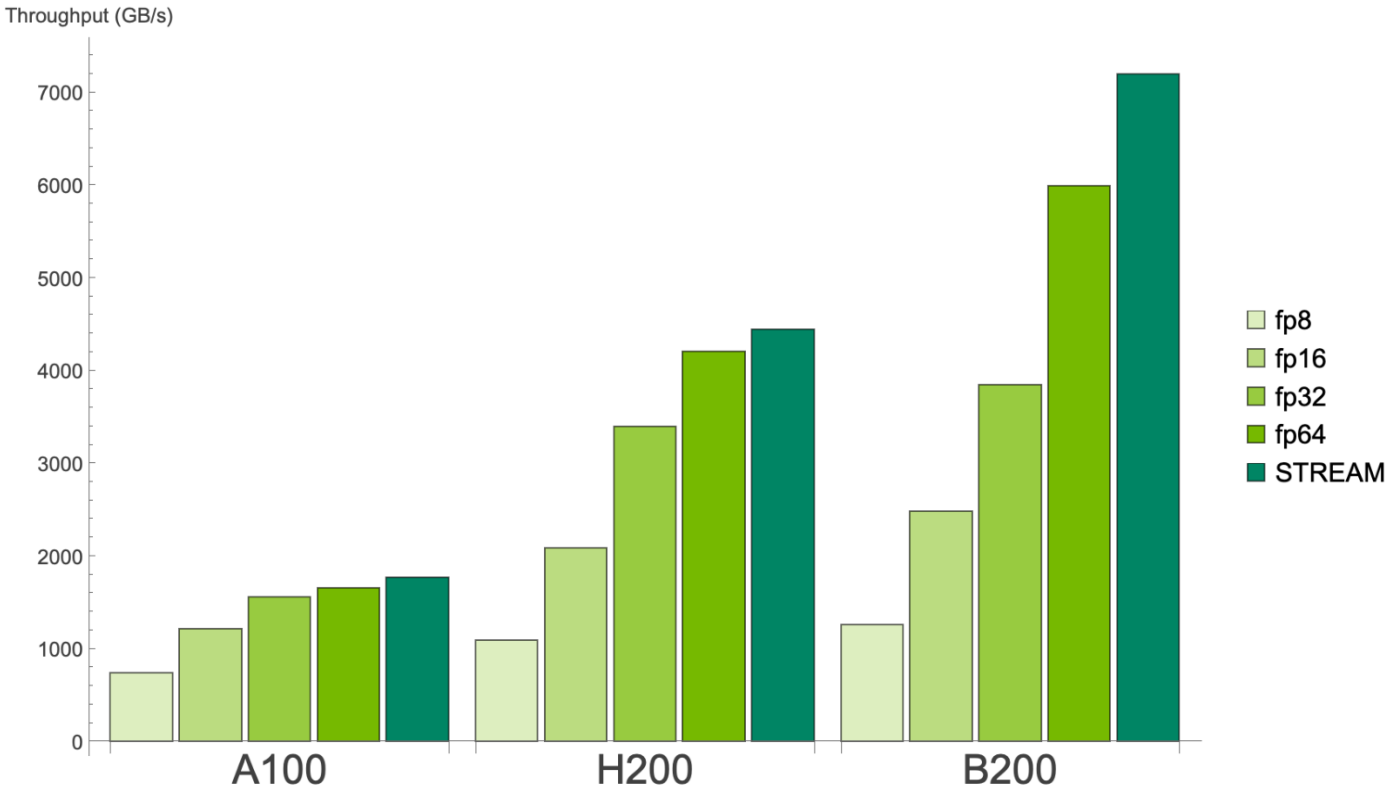


打满内存带宽 MEMORY BANDWITH

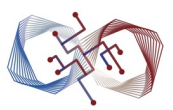
```cpp
template < typename scalar_t >
__global__ void axpy_naive(scalar_t a, const scalar_t * x, scalar_t * y, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n) {
        y[tid] = a * x[tid] + y[tid];
    }
}
```

- 完美coalesced
- 没有divergence, 没有同步
- 100% occupancy
- embarrassingly parallel,没有数据依赖

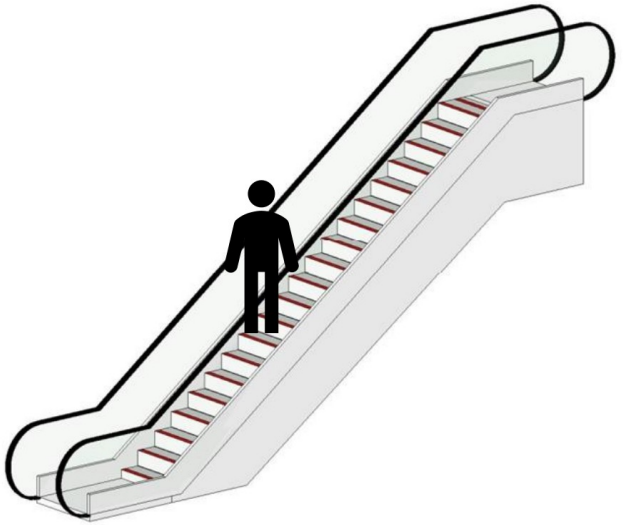
打满内存带宽 MEMORY BANDWITH



打满带宽 MEMORY BANDWITH: Little's Law

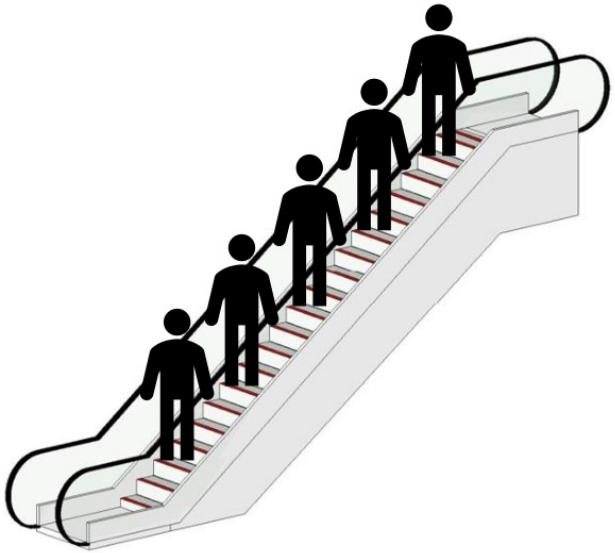


- 每个台阶1个人
- 2s一个台阶
  - 带宽：0.5人/s
- 20个台阶
  - 延迟 = 40s



打满带宽 MEMORY BANDWITH: Little's Law

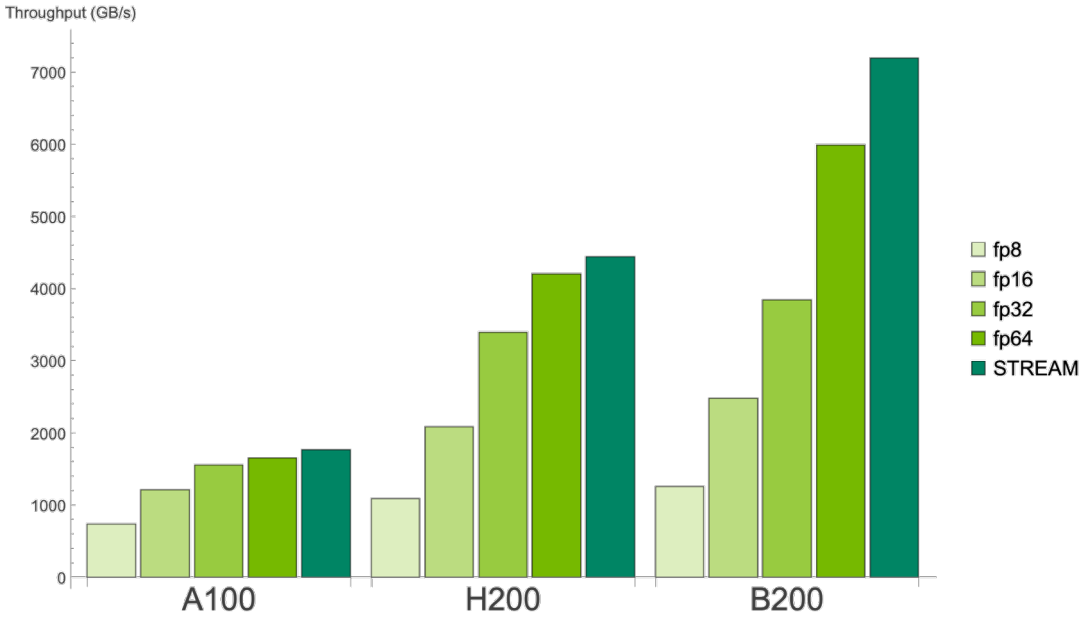
- 每个台阶1个人
- 2s一个台阶
  - 带宽：0.5人/s
- 20个台阶
  - 延迟 = 40s
- 需要多少人同时在台阶上才能占满带宽?
  - 带宽*延迟= 20人



打满带宽 MEMORY BANDWITH: Little's Law

在飞字节数 = 带宽 * 延迟

	A100 80GB	H100 SXM	B200
内存带宽 B	1.94 TB/s	3.35 TB/s	8 TB/s
DRAM 延迟 L	500 ns	600 ns	650 ns
SM 数量	108	132	148
全 GPU 在飞字节 $B \times L$	1.0 MB	2.0 MB	5.2 MB
每 SM 需要	9 KB	15 KB	35 KB
每 SM 最大驻留线程	2048	2048	2048
全 GPU 最大线程数	221 K	270 K	303 K
每线程需要	4.7 B	7.4 B	17 B

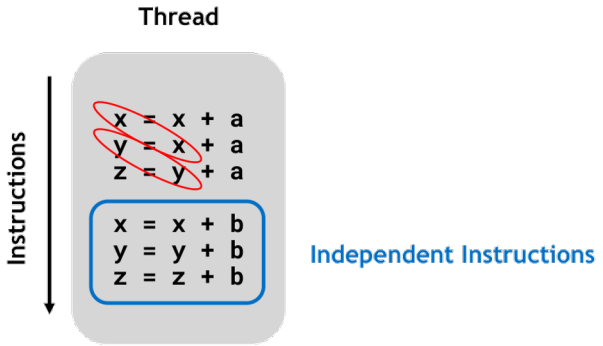


打满带宽 MEMORY BANDWITH: Little's Law

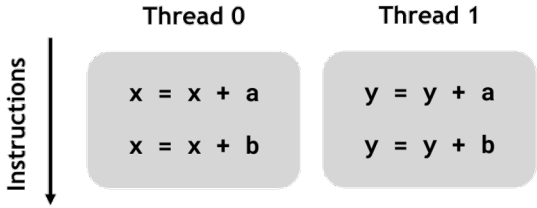
在飞字节数 = 带宽 * 延迟

$$\text{每 SM 在飞字节} = \underbrace{\text{驻留 warp 数}}_{\text{occupancy}} \times 32 \times \underbrace{\text{单指令字节数}}_{\text{向量化}} \times \underbrace{\text{“MLP“}}_{\text{展开}}$$

1. Improve **Instruction-Level Parallelism** (ILP).
   - Higher ILP -> more independent instructions per thread.



2. Improve **Thread-Level Parallelism** (TLP).
   - Higher TLP -> more threads -> more independent instructions per kernel.



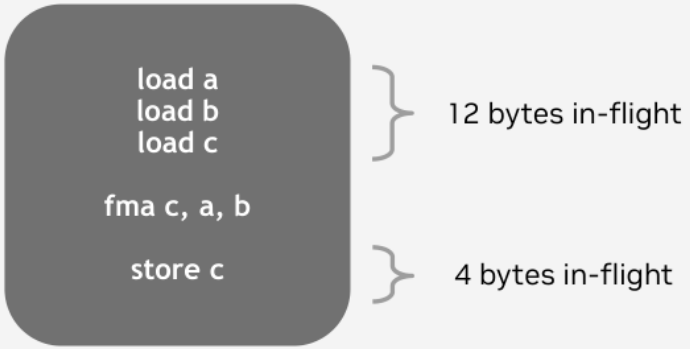
打满带宽 MEMORY BANDWITH：展开

- Assumptions
  - LDG/STG
    - Dependent Issue Rate: 1000 cycles
    - Issue Rate: 1 cycle
  - FP32 pipeline
    - Dependent Issue Rate: 4 cycles
    - Issue Rate: 2 cycles
  - 1 available warp per scheduler

Cycle	N	N+1	N+2		N+1002		N+1006
	LDG	LDG	LDG	(stall)	FFMA	(stall)	STG

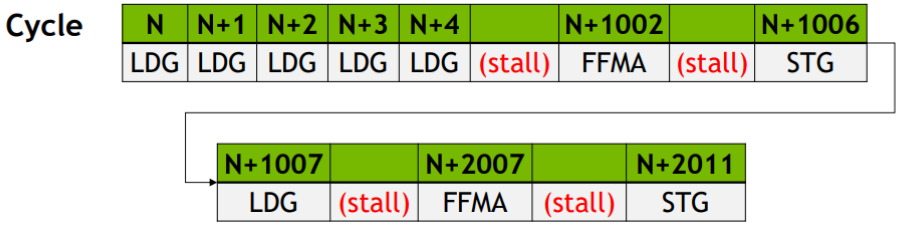
Total cycles = 1006

```
__global__
void kernel(const float * __restrict__ a,
            const float * __restrict__ b,
            float * __restrict__ c)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  c[idx] += a[idx] * b[idx];
}
```



打满带宽 MEMORY BANDWITH：展开

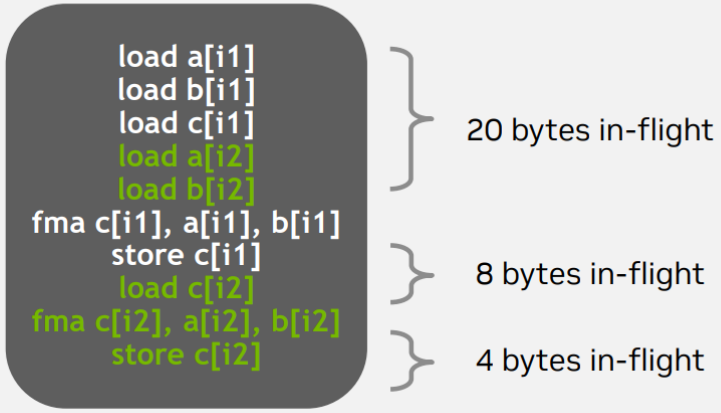
- Every thread computes 2 elements using a **grid stride.**



Total cycles = 2011

2x the amount of work in 2x more cycles!

```
__global__
void kernel(const float * __restrict__ a,
            const float * __restrict__ b,
            float * __restrict__ c)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  #pragma unroll 2
  for (int i = 0; i < 2; i++) {
    const int idx = tid + i * stride;
    c[idx] += a[idx] * b[idx];
  }
}
```



打满带宽 MEMORY BANDWITH：展开

- Every thread computes 2 elements using a **constant block stride.**

Cycle

N	N+1	N+2	N+3	N+4	N+5		N+1002	
LDG	LDG	LDG	LDG	LDG	LDG	(stall)	FFMA	(stall)

N+1004		N+1006		N+1008
FFMA	(stall)	STG	(stall)	STG

Total cycles = 1008

2x the amount of work in the ~same number of cycles!

```
#define THREAD_BLOCK_DIM 128

__global__
void kernel(const float * __restrict__ a,
            const float * __restrict__ b,
            float * __restrict__ c)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int off = 2 * THREAD_BLOCK_DIM * blockIdx.x + threadIdx.x;

  #pragma unroll 2
  for (int i = 0; i < 2; i++) {
    const int idx = off + i * THREAD_BLOCK_DIM;
    c[idx] += a[idx] * b[idx];
  }
}
```

```
load a[i1]
load b[i1]
load c[i1]
load a[i2]
load b[i2]
load c[i2]
fma c[i1], a[i1], b[i1]
fma c[i2], a[i2], b[i2]
store c[i1]
store c[i2]
```

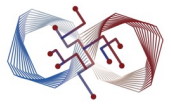
24 bytes in-flight

8 bytes in-flight

打满带宽 MEMORY BANDWITH: OCCUPANCY

- 4个smsp
- 每个都有独立warp scheduler
- 每个smsp最多驻留16个warp
- 每个cycle每个scheduler从eligible的warp中选择一个发射

打满带宽 MEMORY BANDWITH: OCCUPANCY

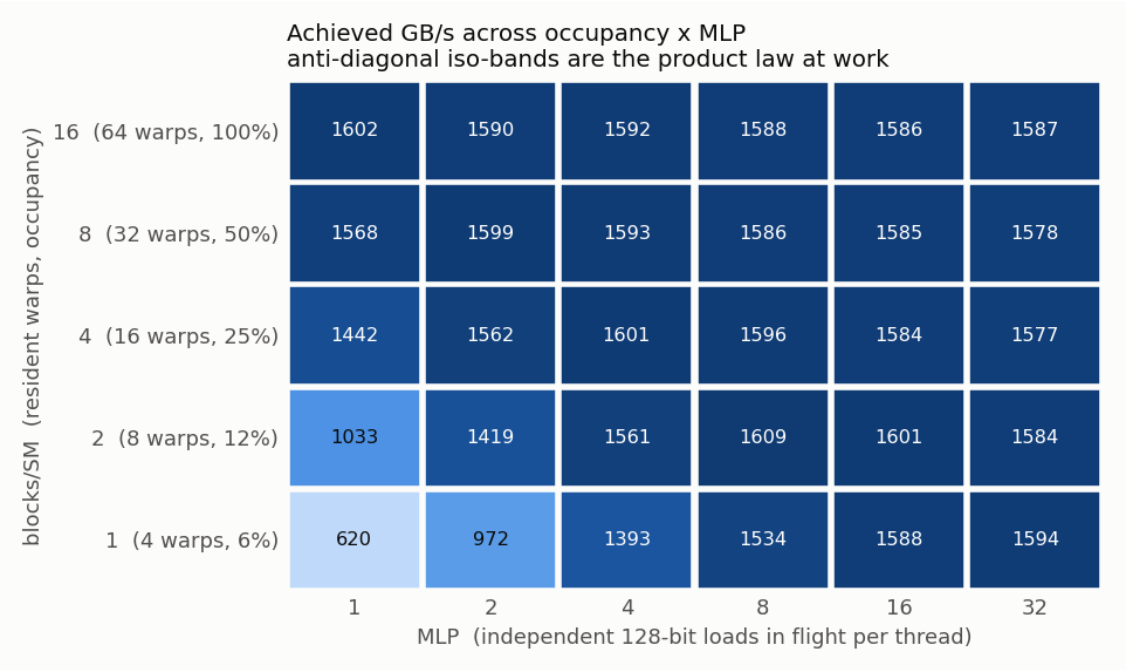


限制Occupancy的因素：

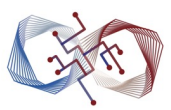
- Shared memory大小：每个SM 164KB（A100） 228KB (H100、B200)
- 寄存器数量：每个SM 65536个32bit寄存器

打满带宽 MEMORY BANDWITH: Little's Law

$$\text{每 SM 在飞字节} = \underbrace{\text{驻留 warp 数}}_{\text{occupancy}} \times 32 \times \underbrace{\text{单指令字节数}}_{\text{向量化}} \times \underbrace{\text{“MLP“}}_{\text{展开}}$$



打满带宽 MEMORY BANDWITH: 向量化访存

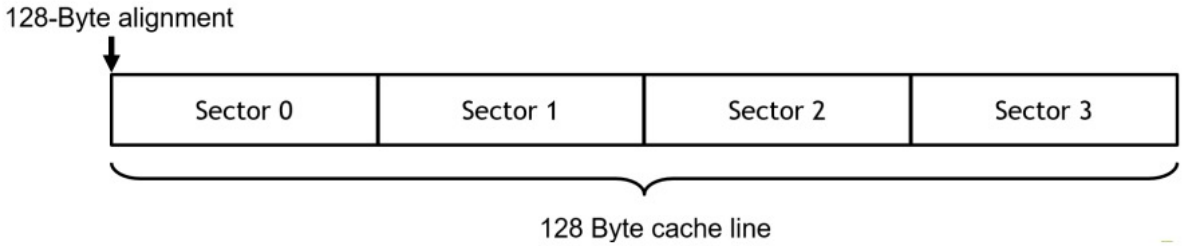


```cpp
// x and y must be 16-byte aligned (cudaMalloc allocations satisfy this).
__global__ void saxpy_vectorized(const float4* __restrict__ x,
                                 float4* __restrict__ y, float a, int n) {
  const int i = blockIdx.x * blockDim.x + threadIdx.x;
  const int n4 = n / 4;

  if (i < n4) {
    const float4 xv = x[i];
    const float4 yv = y[i];

    y[i] = make_float4(a * xv.x + yv.x, a * xv.y + yv.y,
                       a * xv.z + yv.z, a * xv.w + yv.w);
  }
}
```

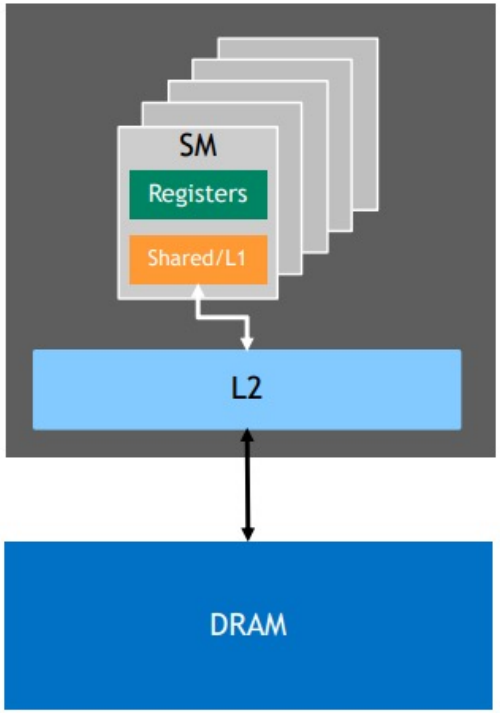
打满带宽 MEMORY BANDWITH: coalescing

- Minimum memory access granularity: **32 bytes** = **1 sector**
  - **L1 to L2:** 1 sector
  - **L2 to Global:** 2 sectors (default)
    - User can set a preferred granularity with `cudaDeviceSetLimit()` and `cudaLimitMaxL2FetchGranularity`.
    - Only a hint though!

- Cache line size: **128 bytes** = **4 sectors**
  - Cache "management" granularity = 1 cache line
    - Coalescing of requests.
    - Evictions.



打满带宽 MEMORY BANDWITH: coalescing



Reads

Check if data is in L1 (if not, check L2)

Check if data is in L2 (if not, get from DRAM)

Unit of data moved: **full sector**

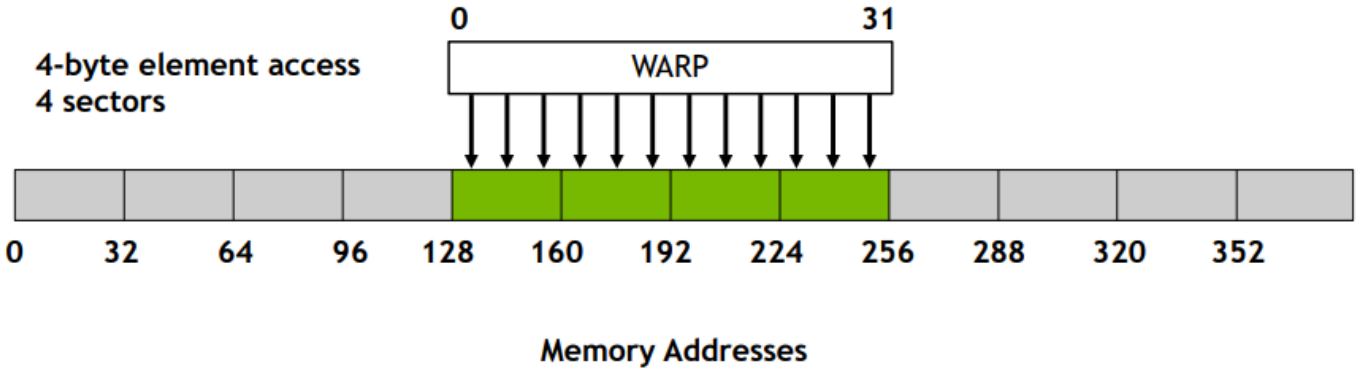
Writes

L1 is write-through: update both L1 and L2

L2 is write back: flush data to DRAM only when needed

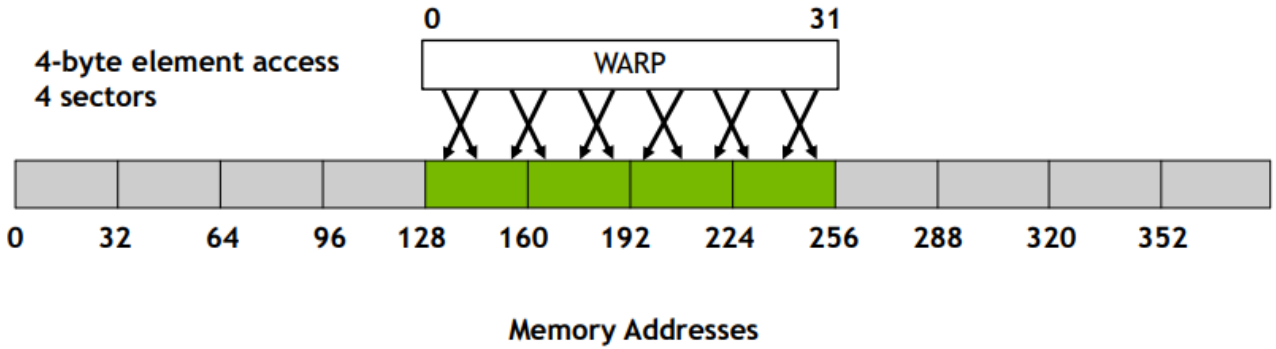
Unit of data moved: **partial sector***

打满带宽 MEMORY BANDWITH: coalescing



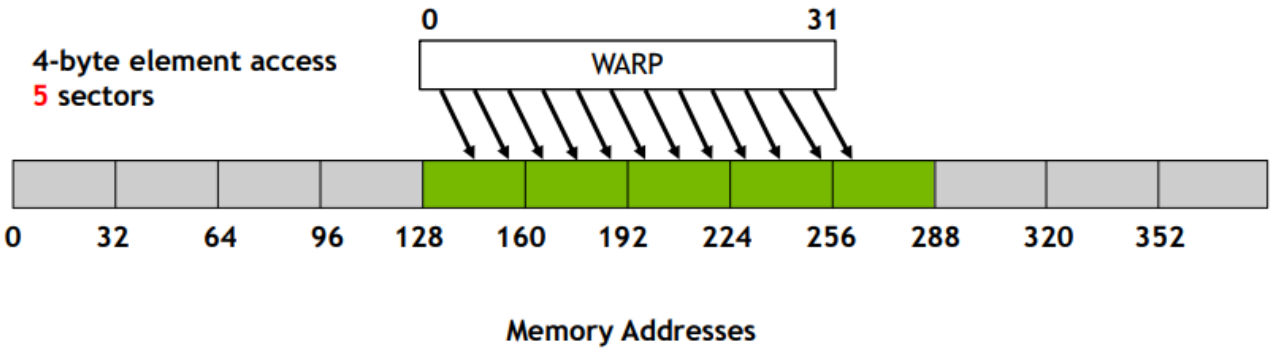
COALESCED!

打满带宽 MEMORY BANDWITH: coalescing

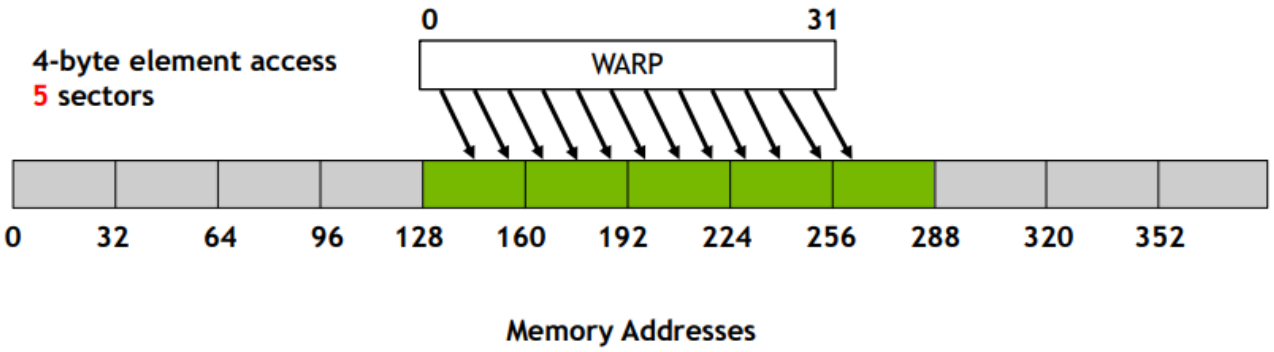


COALESCED!

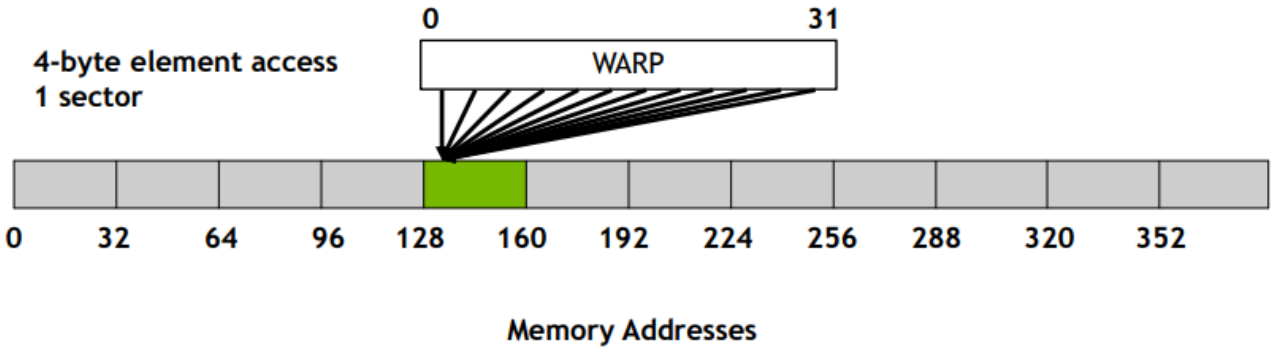
打满带宽 MEMORY BANDWITH: coalescing



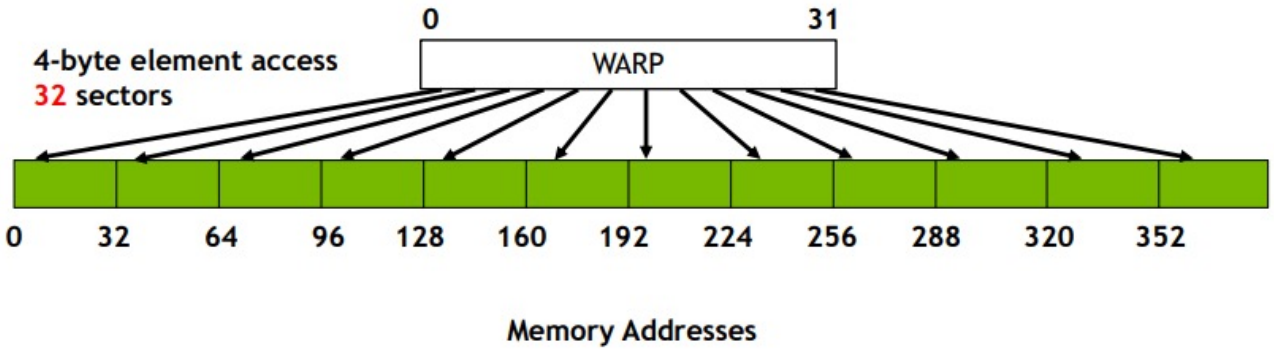
打满带宽 MEMORY BANDWITH: coalescing



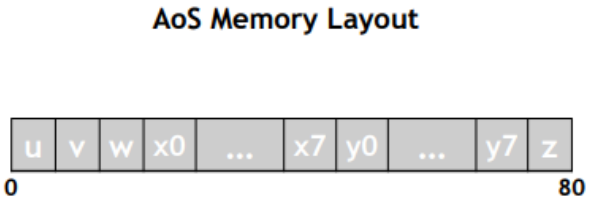
打满带宽 MEMORY BANDWITH: coalescing



打满带宽 MEMORY BANDWITH: coalescing



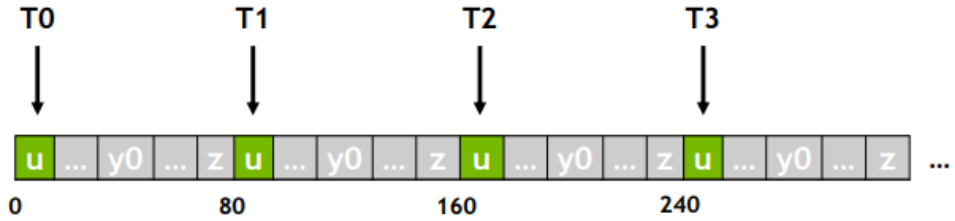
打满带宽 MEMORY BANDWITH: coalescing




```
struct Coefficients
{
  float u, v, w;
  float x[8], y[8], z;
};

__global__ void kernel(Coefficients *data)
{
  int i = cg::this_grid.thread_rank();

  data[i].u = data[i].u + 10.f;
  data[i].y[0] = data[i].y[0] + 10.f;
}
```

打满带宽 MEMORY BANDWITH: coalescing

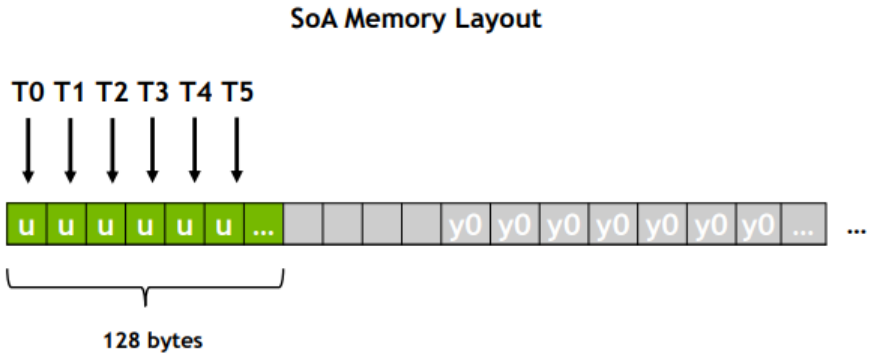
- When loading coefficients u and y[0]:
  - Successive threads in a warp read 4 bytes at 80-byte stride.




```
struct Coefficients
{
  float u, v, w;
  float x[8], y[8], z;
};

__global__ void kernel(Coefficients *data)
{
  int i = cg::this_grid.thread_rank();

  data[i].u = data[i].u + 10.f;
  data[i].y[0] = data[i].y[0] + 10.f;
}
```

打满带宽 MEMORY BANDWITH: coalescing

- Refactoring from **AoS** to **SoA** leads to coalesced memory accesses for u and `y[0]`.




```
struct Coefficients
{
  float *u, *v, *w;
  float *x0, …, *x7, *y0, … *y7, *z;
};

__global__ void kernel(Coefficients data)
{
  int i = cg::this_grid.thread_rank();

  data.u[i] = data.u[i] + 10.f;
  data.y0[i] = data.y0[i] + 10.f;
}
```

Implementation	Load Efficiency (%)	Store Efficiency (%)	Main Memory Bandwidth Utilization (%)	GPU Time (ms)
AoS	12.5	12.5	13.50	28.497
SoA	100	100	79.47	4.836

Reduce

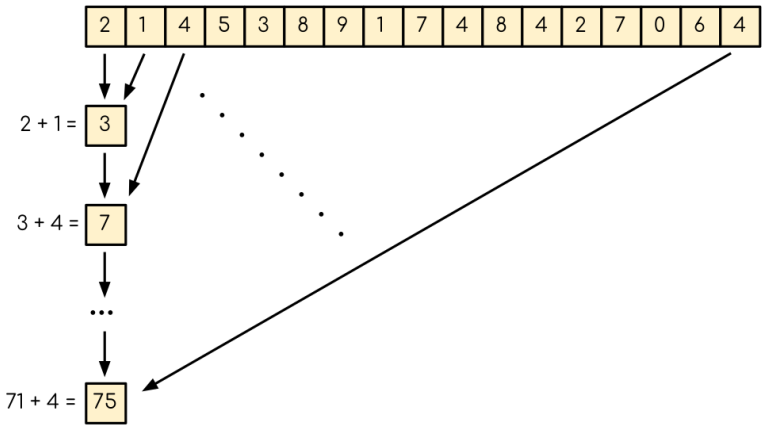
给定长度 N 的数组，求和：

$$\text{out} = \sum_{i=0}^{N-1} x_i$$

内存层级 MEMORY HIERARCHY

Reduce

```
1  __global__ void reduce_v1(
2  const float* x, float* out, int n)
3  {
4      int i = blockIdx.x * blockDim.x
5          + threadIdx.x;
6      if (i < n)
7          atomicAdd(out, x[i]);
8  }
```

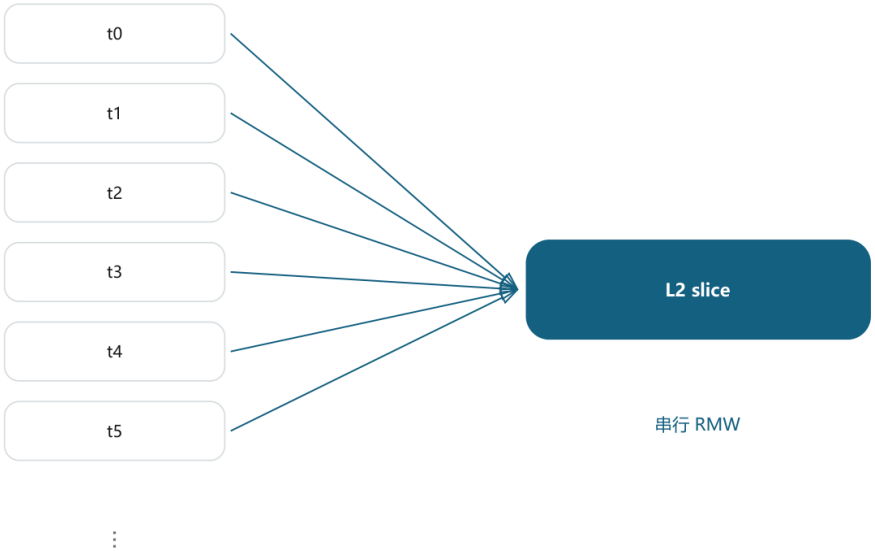


197 ms, 1.4 GB/s

内存层级 MEMORY HIERARCHY

Reduce

```
1  __global__ void reduce_v1(
2  const float* x, float* out, int n)
3  {
4      int i = blockIdx.x * blockDim.x
5          + threadIdx.x;
6      if (i < n)
7          atomicAdd(out, x[i]);
8  }
```

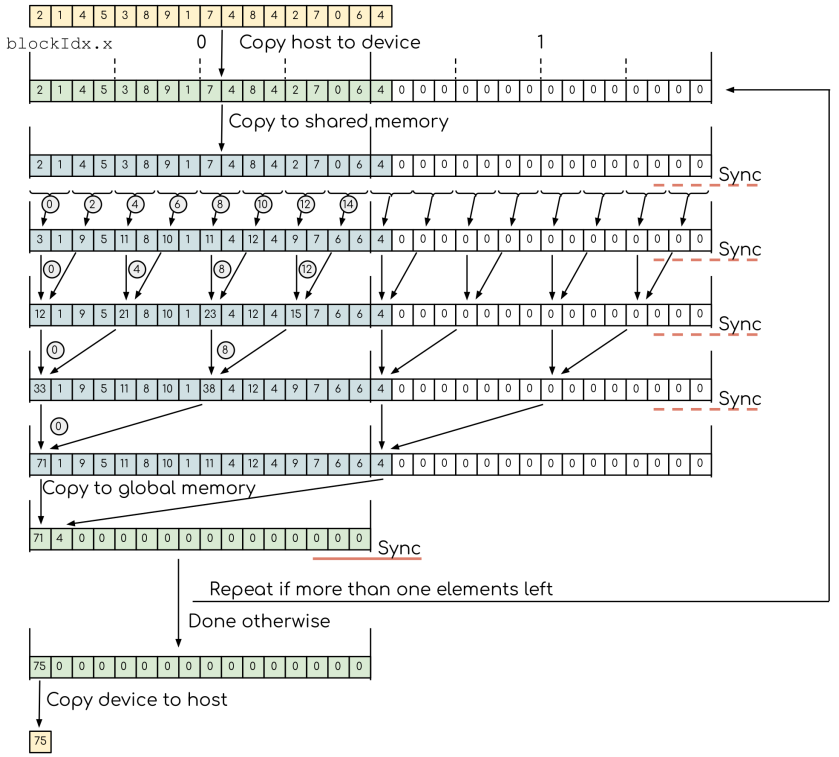


197 ms， 1.4 GB/s

内存层级 MEMORY HIERARCHY

Reduce

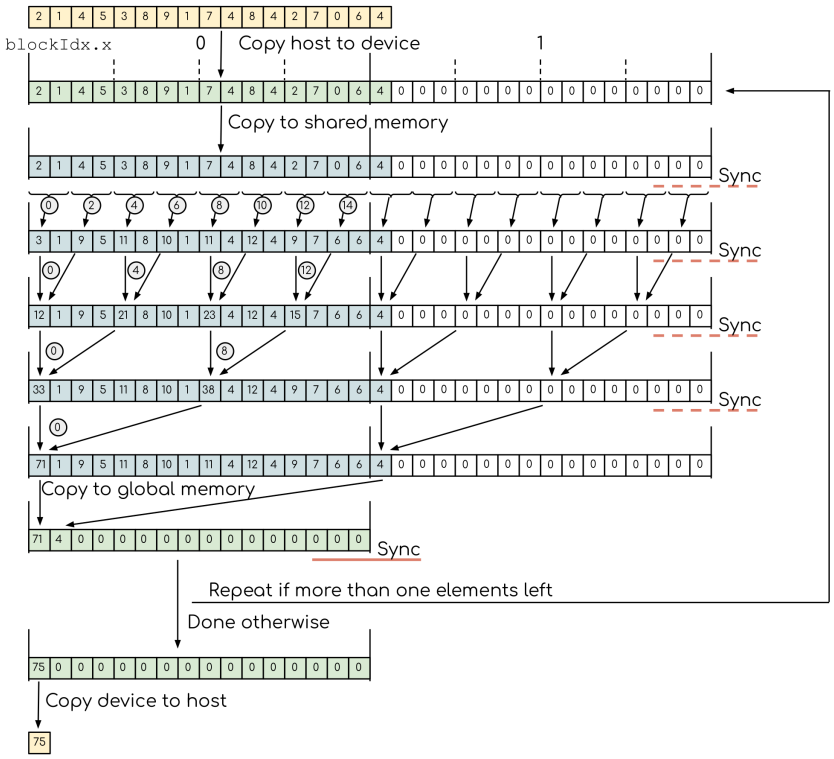
```
1  __global__ void reduce_v2(
2  const float* x, float* out, int n)
3  {
4      __shared__ float s[BLOCK];
5      int tid = threadIdx.x;
6      int i = blockIdx.x*blockDim.x + tid;
7      s[tid] = (i < n) ? x[i] : 0.0f;
8      __syncthreads();
9
10     for (int stride = 1;
11         stride < blockDim.x; stride *= 2) {
12         if (tid % (2*stride) == 0)
13             s[tid] += s[tid + stride];
14         __syncthreads();
15     }
16     if (tid == 0) atomicAdd(out, s[0]);
17 }
```

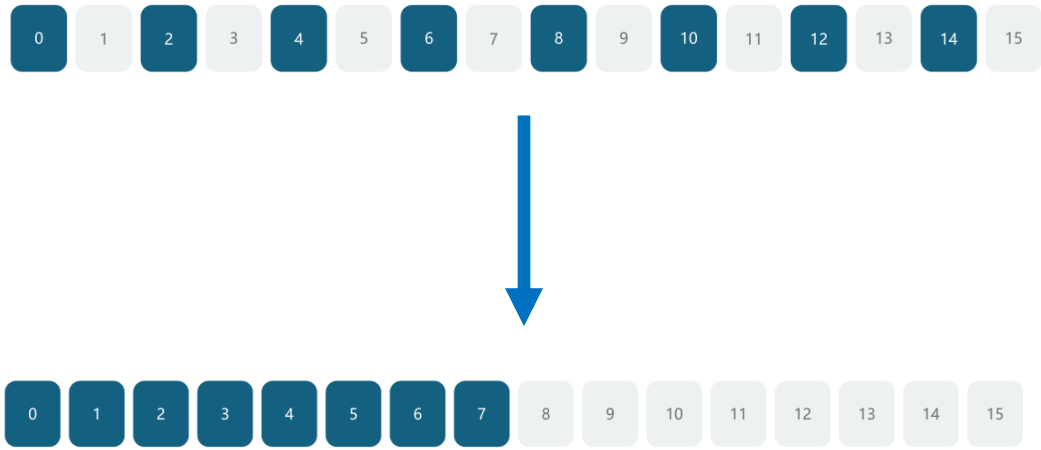


1.67 ms， 160 GB/s， 118x

内存层级 MEMORY HIERARCHY

Reduce

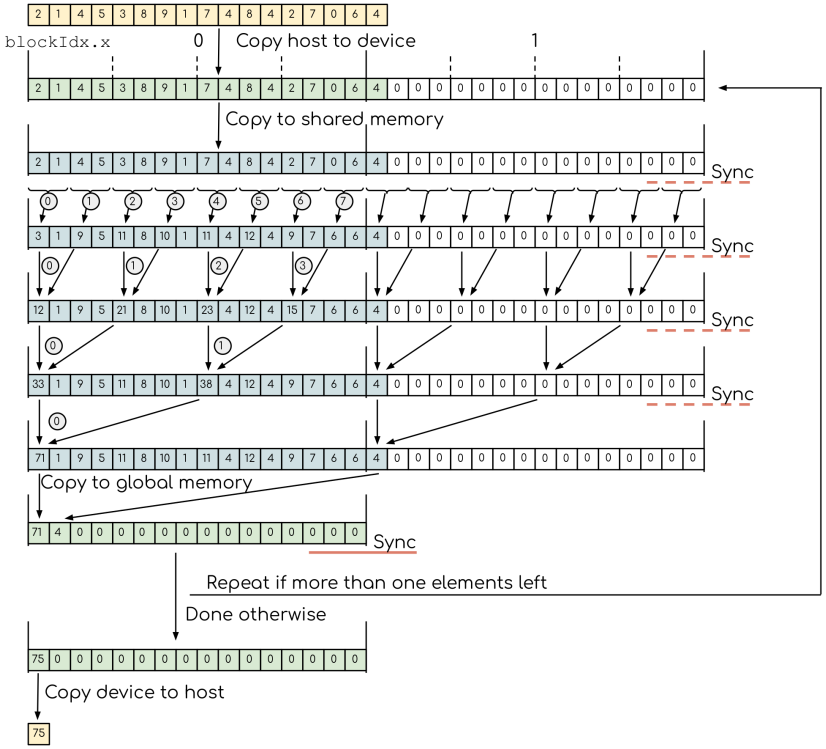




内存层级 MEMORY HIERARCHY

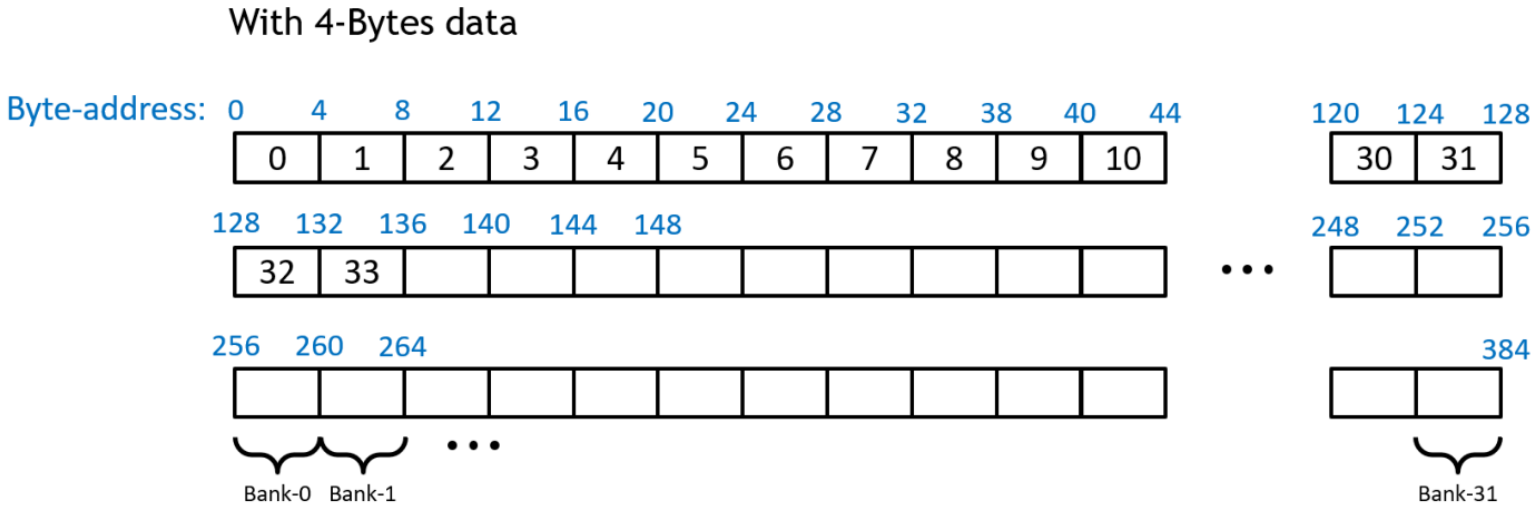
Reduce

```
1  __global__ void reduce_v2(
2  const float* x, float* out, int n)
3  {
4      __shared__ float s[BLOCK];
5      int tid = threadIdx.x;
6      int i = blockIdx.x*blockDim.x + tid;
7      s[tid] = (i < n) ? x[i] : 0.0f;
8      __syncthreads();
9
10     for (int stride = 1;
11             stride < blockDim.x; stride *= 2) {
12           int idx = 2 * stride * tid;
13           if (idx < blockDim.x)
14               s[idx] += s[idx + stride];
15           __syncthreads();
16       }
17     if (tid == 0) atomicAdd(out, s[0]);
18 }
```

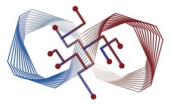


0.87 ms，308 GB/s，1.9x

打满带宽 MEMORY BANDWITH: bank conflict



打满带宽 MEMORY BANDWITH: bank conflict

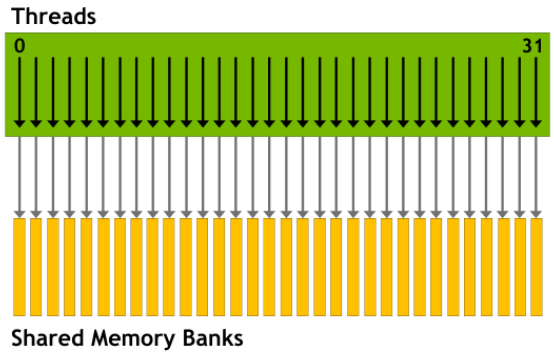


Coalesced access

(No bank conflicts)

shmem[threadIdx.x] = data[tid]

4-byte data

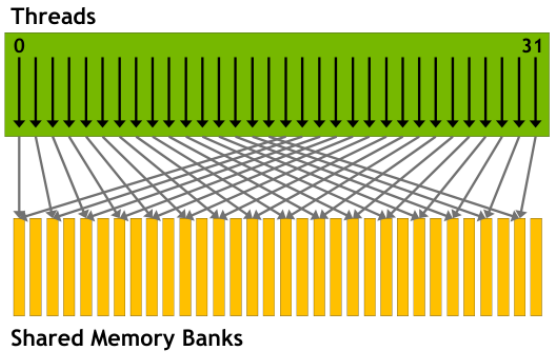


Conflict access

(2-way bank conflicts)

shmem[threadIdx.x * 2] = data[tid]

4-byte data

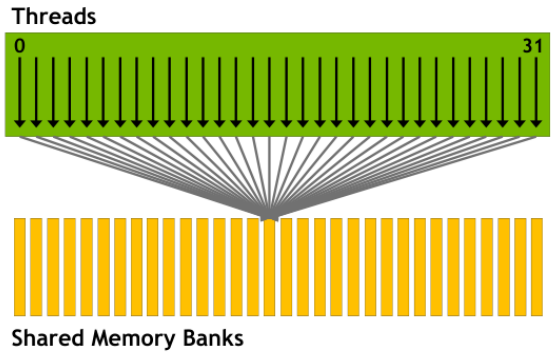


Broadcast access

(No bank conflicts)

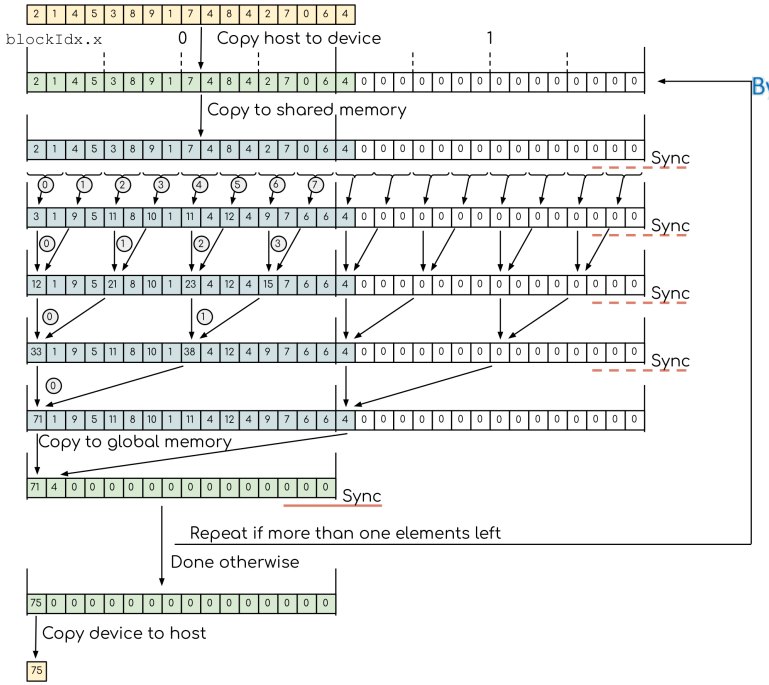
data = shmem[0]

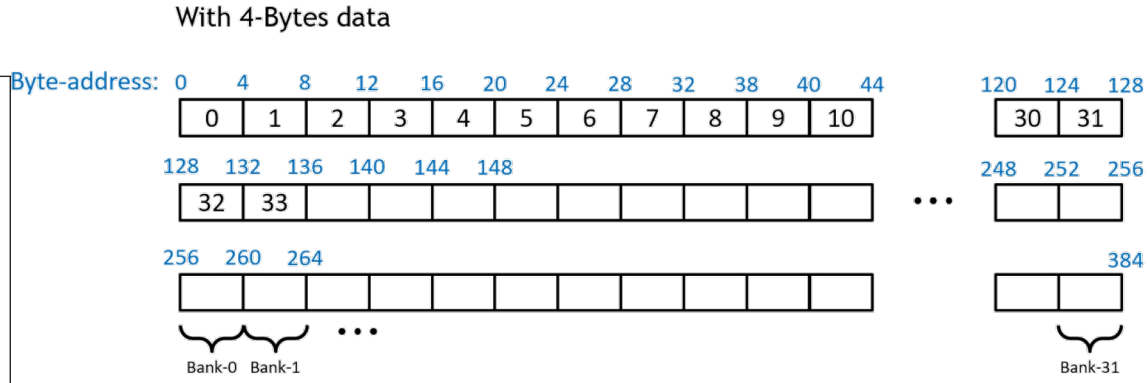
4-byte data

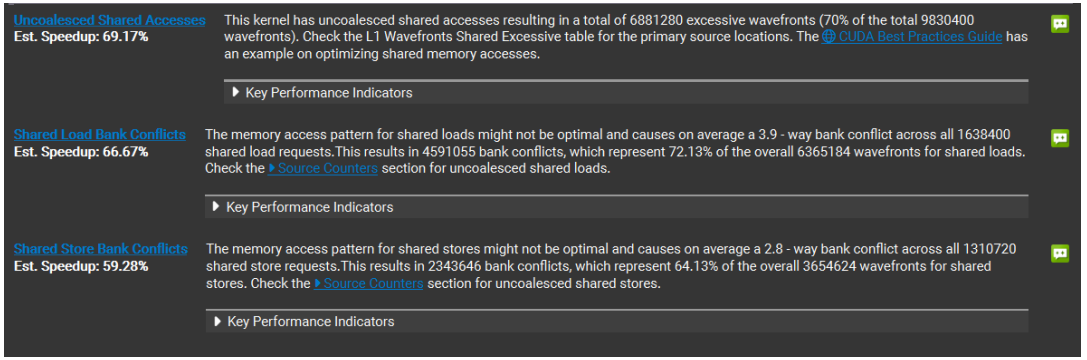


内存层级 MEMORY HIERARCHY

Reduce





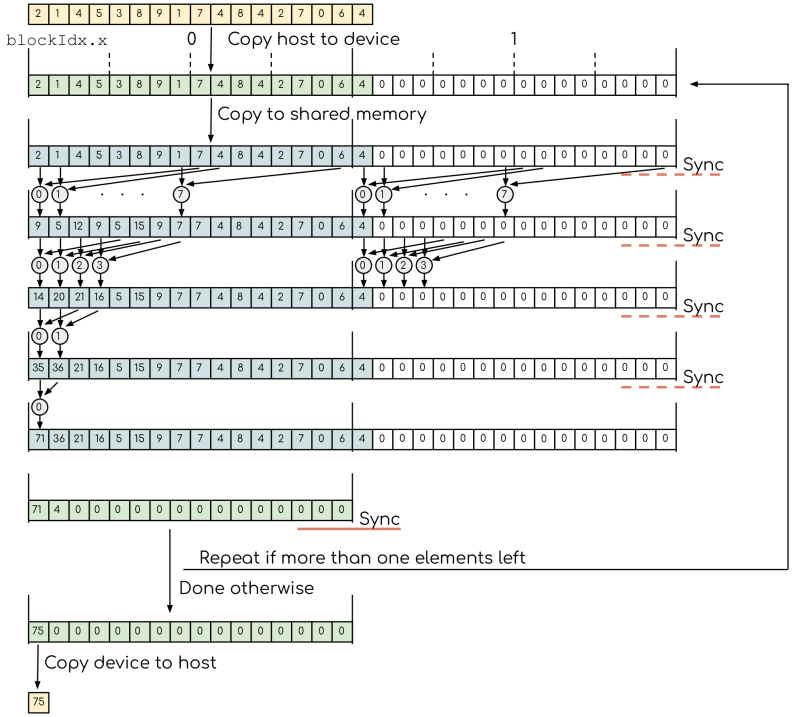


内存层级 MEMORY HIERARCHY

Reduce

```cpp
__global__ void reduce_v4(const float* __restrict__ x, float* out, int n) {
    __shared__ float s[BLOCK];
    int tid = threadIdx.x;
    int i   = blockIdx.x * blockDim.x + tid;
    s[tid] = (i < n) ? x[i] : 0.0f;
    __syncthreads();
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride) s[tid] += s[tid + stride];
        __syncthreads();
    }
    if (tid == 0) atomicAdd(out, s[0]);
}
```

Compute (SM) Throughput [%]	62.22
Memory Throughput [%]	70.38
L1/TEX Cache Throughput [%]	72.44
L2 Cache Throughput [%]	18.93
DRAM Throughput [%]	18.01



0.78 ms, 344 GB/s

内存层级 MEMORY HIERARCHY

Reduce

```cpp
__global__ void reduce_v5(const float* __restrict__ x, float* out, int n) {
    __shared__ float s[BLOCK];
    int tid = threadIdx.x;
    float sum = 0.0f;
    for (int i = blockIdx.x * blockDim.x + tid; i < n; i += gridDim.x * blockDim.x)
        sum += x[i];
    s[tid] = sum;
    __syncthreads();
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (tid < stride) s[tid] += s[tid + stride];
        __syncthreads();
    }
    if (tid == 0) atomicAdd(out, s[0]);
}
```

之前：`gridDim.x = ceil(n / blockDim.x)`
现在：`gridDim.x = SM数 * 16`

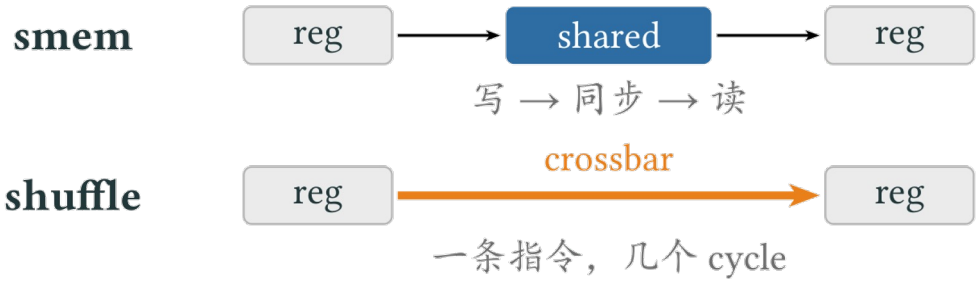
0.25 ms，1074 GB/s

内存层级 MEMORY HIERARCHY

Reduce

折半归约到 stride ⩽ 16 时， 参与的线程全在一个warp 内
warp 内的 32 个 lane 可以直接交换寄存器，不需要经过 shared memory。

```cpp
__device__ __forceinline__ float warp_reduce_sum(float v) {
#pragma unroll
    for (int off = 16; off > 0; off >>= 1)
        v += __shfl_xor_sync(0xffffffff, v, off);
    return v;
}
```



Reduce: Warp Shuffle Primitives

折半归约到 stride ⩽ 16 时， 参与的线程全在一个warp 内
warp 内的 32 个 lane 可以直接交换寄存器，不需要经过 shared memory。

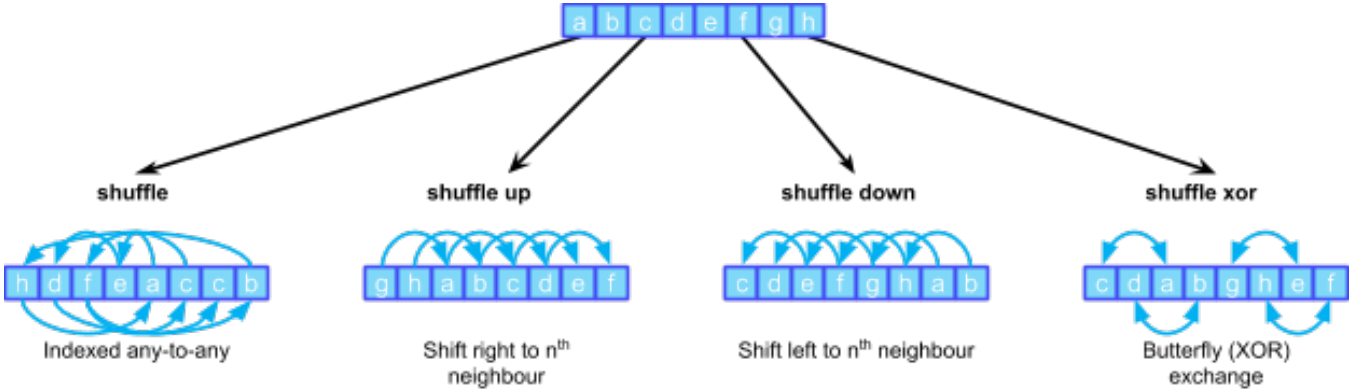
```
T   __shfl_sync(unsigned mask, T var, int srcLane, int width);
T   __shfl_up_sync(unsigned mask, T var, unsigned delta, int width);
T   __shfl_down_sync(unsigned mask, T var, unsigned delta, int width);
T   __shfl_xor_sync(unsigned mask, T var, int laneMask, int width);
```

指令	lane i 接收	用途
__shfl_sync	lane: srcLane (可指定)	广播 / 置换
__shfl_up_sync	lane: i − delta	Prefix sum, stencil
__shfl_down_sync	lane: i + delta	Reduce (to lane 0)
__shfl_xor_sync	lane: i xor laneMask	Butterfly Reduce (to all lanes)

Reduce: Warp Shuffle Primitives

折半归约到 stride ⩽ 16 时， 参与的线程全在一个warp 内
warp 内的 32 个 lane 可以直接交换寄存器，不需要经过 shared memory。

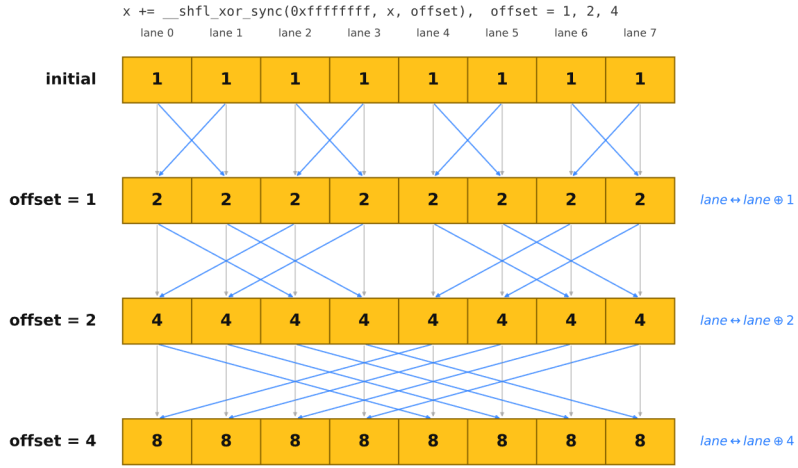
指令	lane i 接收	用途
__shfl_sync	lane: srcLane (可指定)	广播 / 置换
__shfl_up_sync	lane: i − delta	Prefix sum, stencil
__shfl_down_sync	lane: i + delta	Reduce (to lane 0)
__shfl_xor_sync	lane: i xor laneMask	Butterfly Reduce (to all lanes)

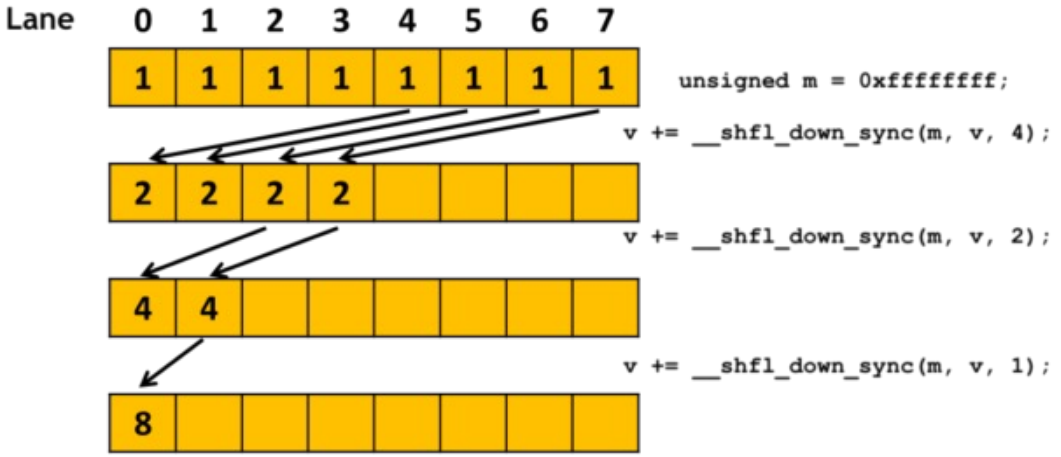


内存层级 MEMORY HIERARCHY

Reduce: Butterfly Shuffle & Shuffle Down

折半归约到 stride ⩽ 16 时， 参与的线程全在一个warp 内
warp 内的 32 个 lane 可以直接交换寄存器，不需要经过 shared memory。





内存层级 MEMORY HIERARCHY

折半归约到 stride ⩽ 16 时， 参与的线程全在一个warp 内
warp 内的 32 个 lane 可以直接交换寄存器，不需要经过 shared memory。

mask: 决定参与 shfl 的lanes

- 写 0xffffffff 表示全部 32 个 lanes
- 分支lanes与mask不符时：未定义行为
  - __activemask()

width: 切分warp成子段，shfl 在子段内进行

- 默认为32 (waspSize)
- 设为 8 时warp分成 4 段，每段 8 个 lanes在段内shfl
  - 例如每 8 个 lane 处理一行

内存层级 MEMORY HIERARCHY

```cpp
__global__ void reduce_v6(const float* __restrict__ x, float* out, int n) {
    float sum = 0.0f;
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < n;
         i += gridDim.x * blockDim.x)
        sum += x[i];

    sum = warp_reduce_sum(sum);

    __shared__ float warp_sum[32];
    int lane = threadIdx.x & 31;
    int wid  = threadIdx.x >> 5;
    if (lane == 0) warp_sum[wid] = sum;
    __syncthreads();

    if (wid == 0) {
        sum = (lane < blockDim.x / 32) ? warp_sum[lane] : 0.0f;
        sum = warp_reduce_sum(sum);
        if (lane == 0) atomicAdd(out, sum);
    }
}
```

0.247 ms，1087 GB/s

内存层级 MEMORY HIERARCHY

```cpp
__global__ void reduce_v6(const float* __restrict__ x, float* out, int n) {
    float sum = 0.0f;
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < n;
         i += gridDim.x * blockDim.x)
        sum += x[i];

    sum = warp_reduce_sum(sum);

    __shared__ float warp_sum[32];
    int lane = threadIdx.x & 31;
    int wid  = threadIdx.x >> 5;
    if (lane == 0) warp_sum[wid] = sum;
    __syncthreads();

    if (wid == 0) {
        sum = (lane < blockDim.x / 32) ? warp_sum[lane] : 0.0f;
        sum = warp_reduce_sum(sum);
        if (lane == 0) atomicAdd(out, sum);
    }
}
```

Compute (SM) Throughput [%]	15.37
Memory Throughput [%]	54.74
L1/TEX Cache Throughput [%]	14.94
L2 Cache Throughput [%]	51.45
DRAM Throughput [%]	54.74

内存层级 MEMORY HIERARCHY

```cpp
__global__ void reduce_v7(const float4* __restrict__ x4, float* out, int n4) {
    float sum = 0.0f;
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < n4;
         i += gridDim.x * blockDim.x) {
        float4 v = x4[i];
        sum += v.x + v.y + v.z + v.w;
    }
    sum = warp_reduce_sum(sum);

    __shared__ float warp_sum[32];
    int lane = threadIdx.x & 31;
    int wid  = threadIdx.x >> 5;
    if (lane == 0) warp_sum[wid] = sum;
    __syncthreads();

    if (wid == 0) {
        sum = (lane < blockDim.x / 32) ? warp_sum[lane] : 0.0f;
        sum = warp_reduce_sum(sum);
        if (lane == 0) atomicAdd(out, sum);
    }
}

```

0.17 ms， 1572 GB/s

内存层级 MEMORY HIERARCHY

Reduce

版本	时间	带宽	vs V1	warp 指令
V1 global atomic	197.4 ms	1.4 GB/s	1×	—
V2 smem, 散开	1.67 ms	161 GB/s	118×	1.55 亿
V3 连续线程	0.87 ms	308 GB/s	227×	6167 万
V4 sequential	0.78 ms	344 GB/s	253×	5276 万
V5 grid-stride	0.25 ms	1074 GB/s	790×	560 万
V6 warp shuffle	0.247 ms	1087 GB/s	799×	460 万
V7 + float4	**0.17 ms**	**1572 GB/s**	**1156×**	185 万