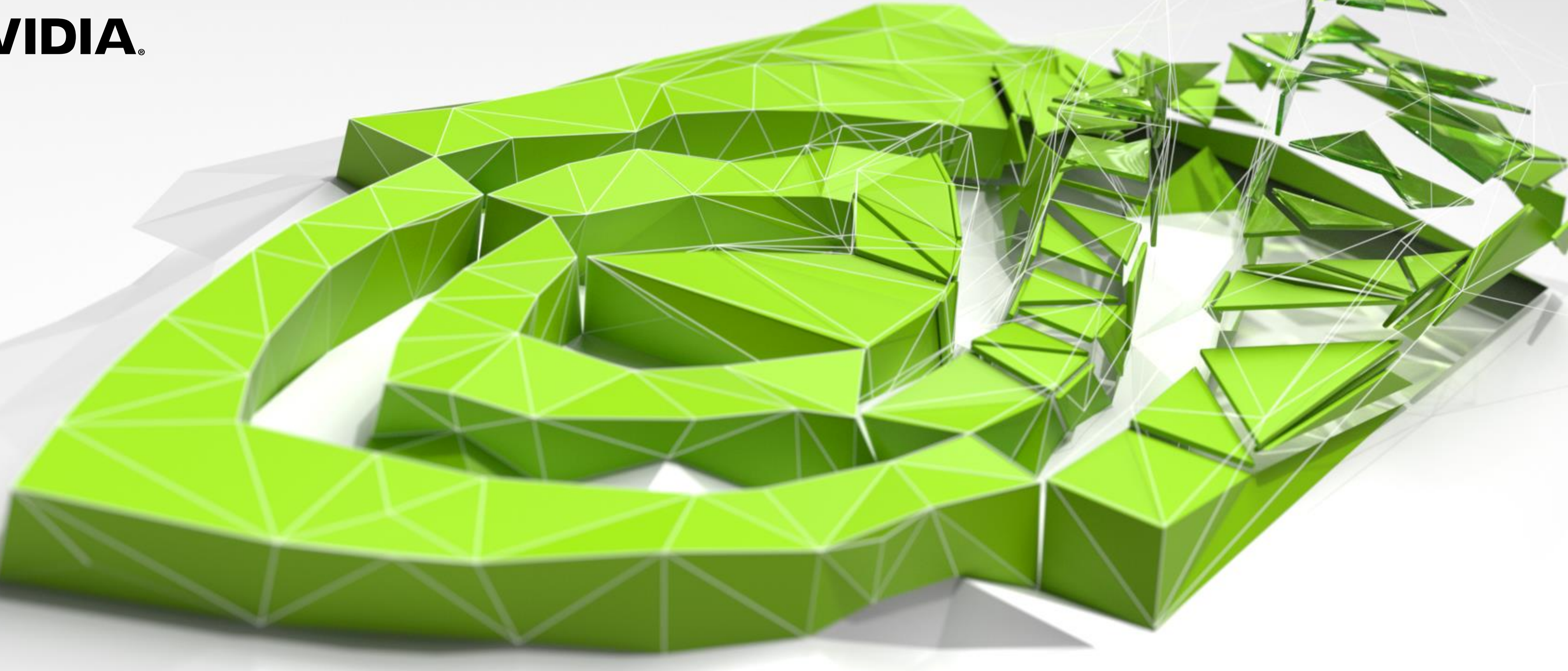
# CUDA kernel Optimization & libraries

GPU计算专家团队 工程师 郁凡(Fan YU) 2020.05.13

NVIDIA

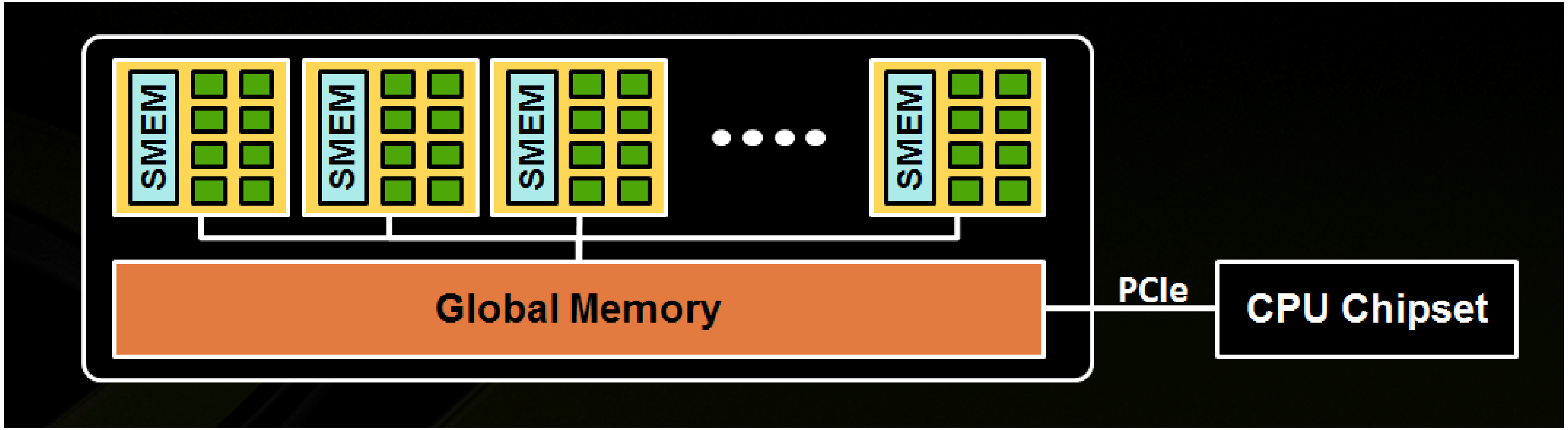


# Contents

1. GPU Architecture
2. Kernel Optimization
   - Measurement
   - Memory optimization
   - Latency optimization
   - Instruction optimization
3. CPU-GPU interaction optimization
4. Introduction to libraries
5. Summary

# 1. GPU architecture

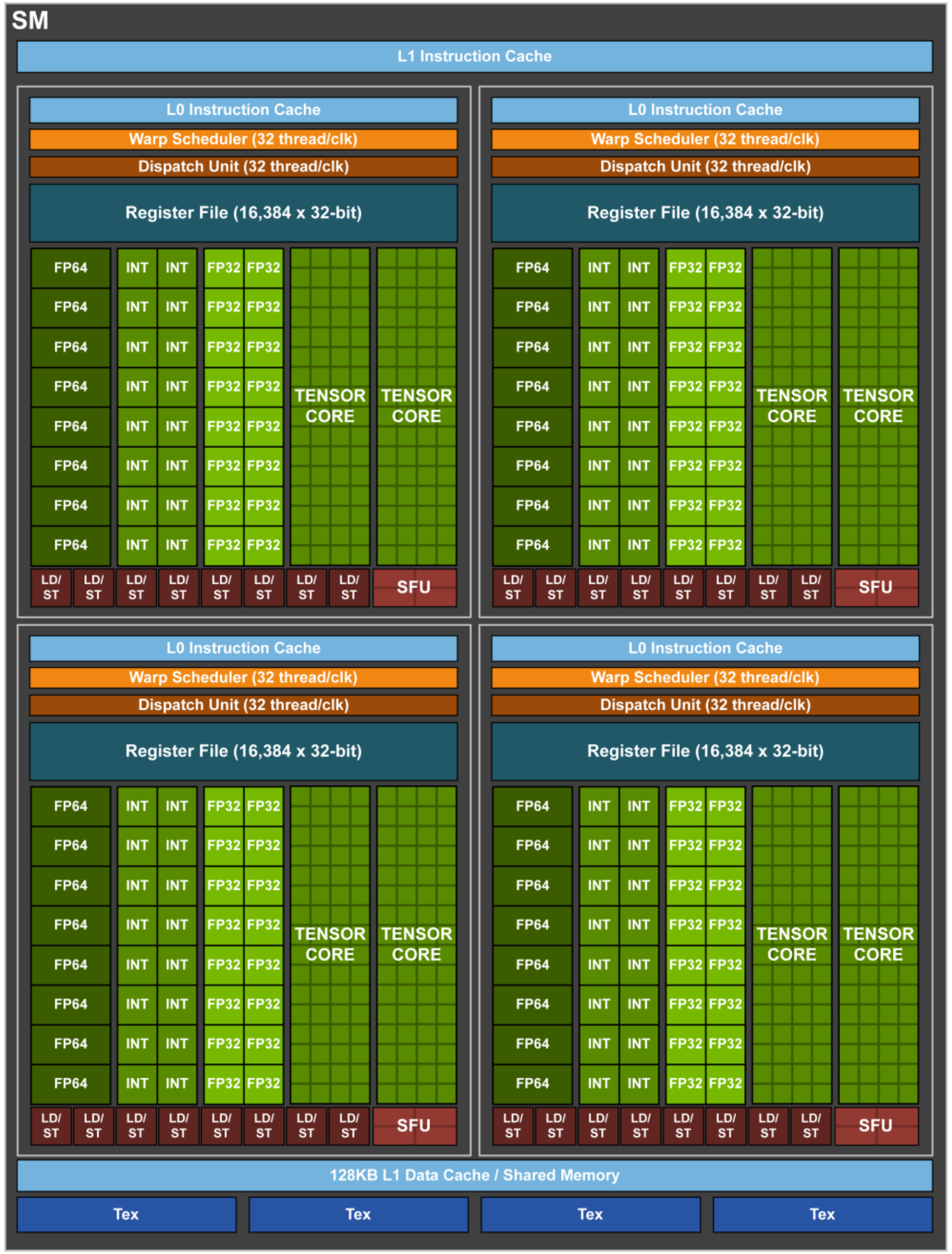
# GPU High Level View



- ❑ Streaming Multiprocessor (SM)
  - ❑ A set of CUDA cores
- ❑ Global memory

# VOLTA GV100 SM

| | GV100 |
|---|---|
| FP32 units | 64 |
| FP64 units | 32 |
| INT32 units | 64 |
| Tensor Cores | 8 |
| Register File | 256 KB |
| Unified L1/Shared memory | 128 KB |
| Active Threads | 2048 |


SM

L1 Instruction Cache

L0 Instruction Cache

Warp Scheduler (32 thread/clk)

Dispatch Unit (32 thread/clk)

Register File (16,384 x 32-bit)

FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32

TENSOR CORE TENSOR CORE

LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST SFU

L0 Instruction Cache

Warp Scheduler (32 thread/clk)

Dispatch Unit (32 thread/clk)

Register File (16,384 x 32-bit)

FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32

TENSOR CORE TENSOR CORE

LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST SFU

L0 Instruction Cache

Warp Scheduler (32 thread/clk)

Dispatch Unit (32 thread/clk)

Register File (16,384 x 32-bit)

FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32

TENSOR CORE TENSOR CORE

LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST SFU

L0 Instruction Cache

Warp Scheduler (32 thread/clk)

Dispatch Unit (32 thread/clk)

Register File (16,384 x 32-bit)

FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32
FP64 INT INT FP32 FP32

TENSOR CORE TENSOR CORE

LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST SFU

128KB L1 Data Cache / Shared Memory

Tex Tex Tex Tex

# GPU and Programming Model

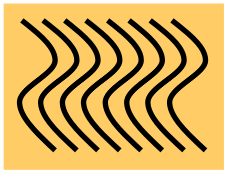
Software

GPU

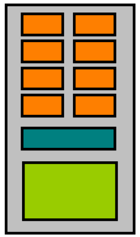
Thread

CUDA Core

**Threads are executed by cuda core**



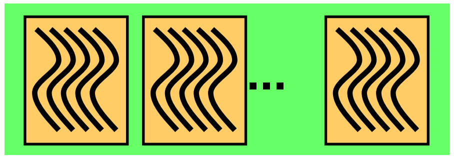
Thread Block



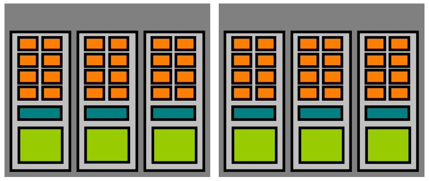
SM

**Thread blocks are executed on SM**

**Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources**



Grid

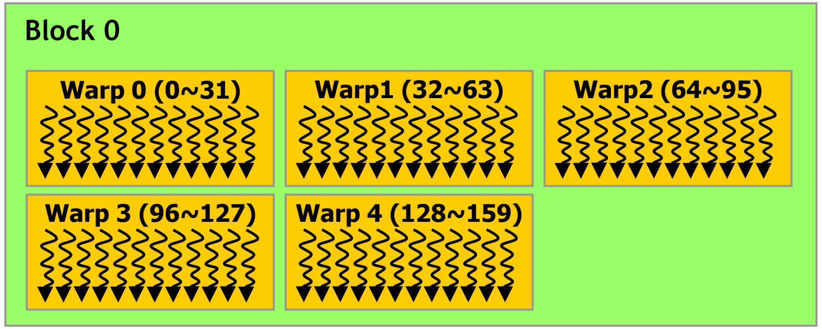


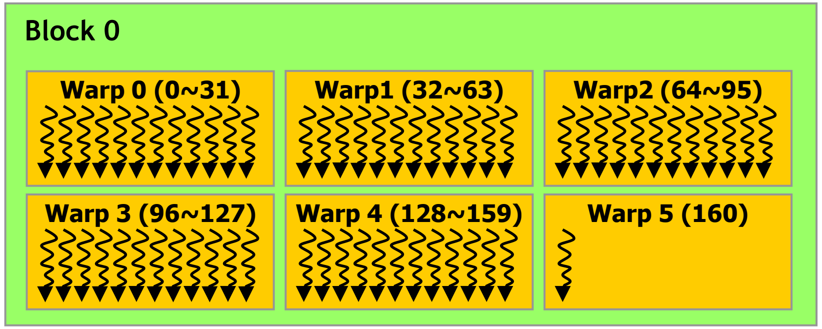
Device

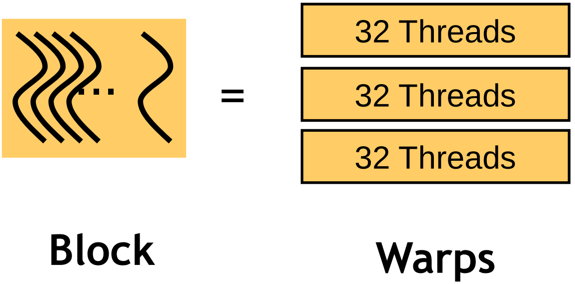
**A kernel is launched as a grid of thread blocks**

# Warp

- □ Warp is successive 32 threads in a block
- □ E.g. blockDim = 160
  - Automatically divided to 5 warps by GPU
- □ E.g. blockDim = 161
  - If the blockDim is not the Multiple of 32
    The rest of thread will occupy one more warp

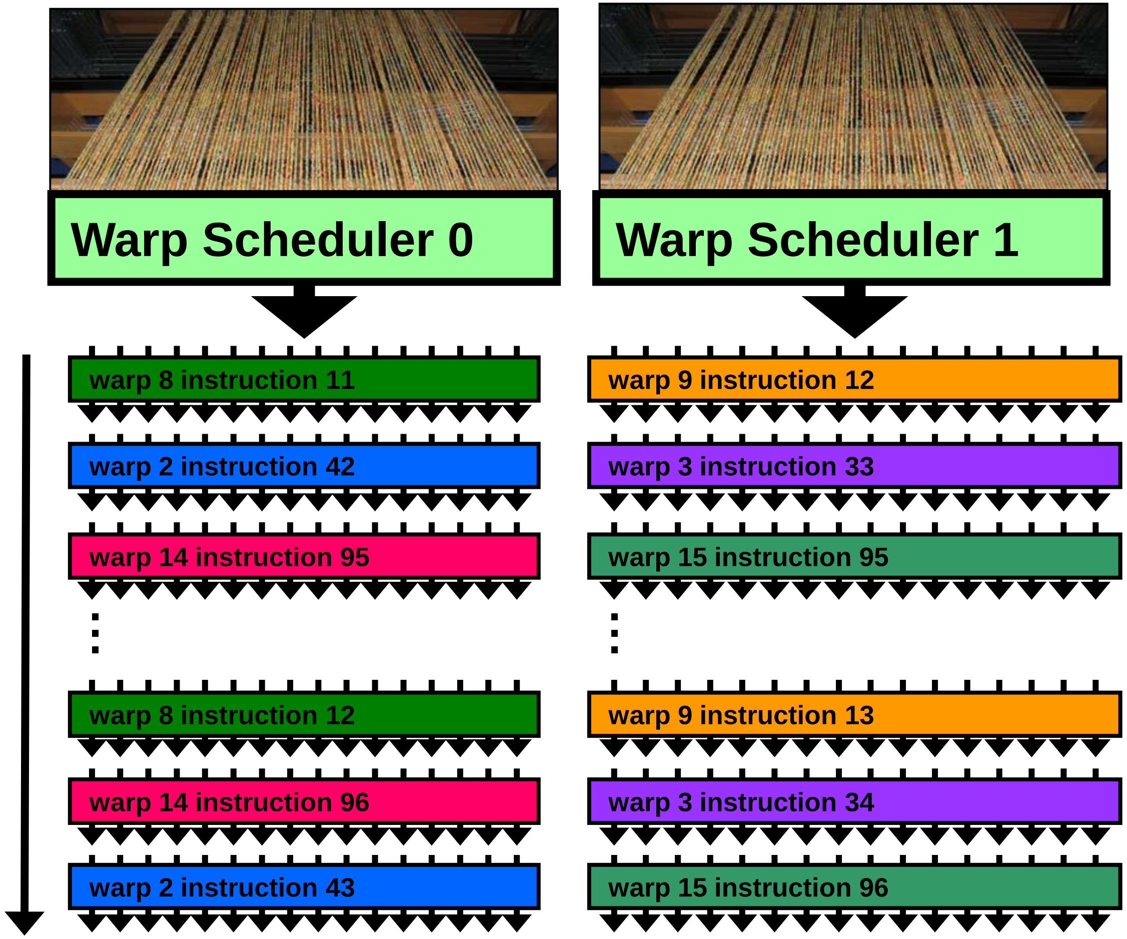






# Warp

- SIMT: Single Instruction Multi Thread
- The threads in the same warp always executing the same instruction
- Instructions will be issued to operation units by warp
- Latency is caused by the dependency between the neighbor instructions in the same warp
- In the waiting time, other instructions from other warps can be executed
- Context switching is free
- A lot of warps can hide memory latency



Warp Scheduler 0

warp 8 instruction 11

warp 2 instruction 42

warp 14 instruction 95

⋮

warp 8 instruction 12

warp 14 instruction 96

warp 2 instruction 43

Warp Scheduler 1

warp 9 instruction 12

warp 3 instruction 33

warp 15 instruction 95

⋮

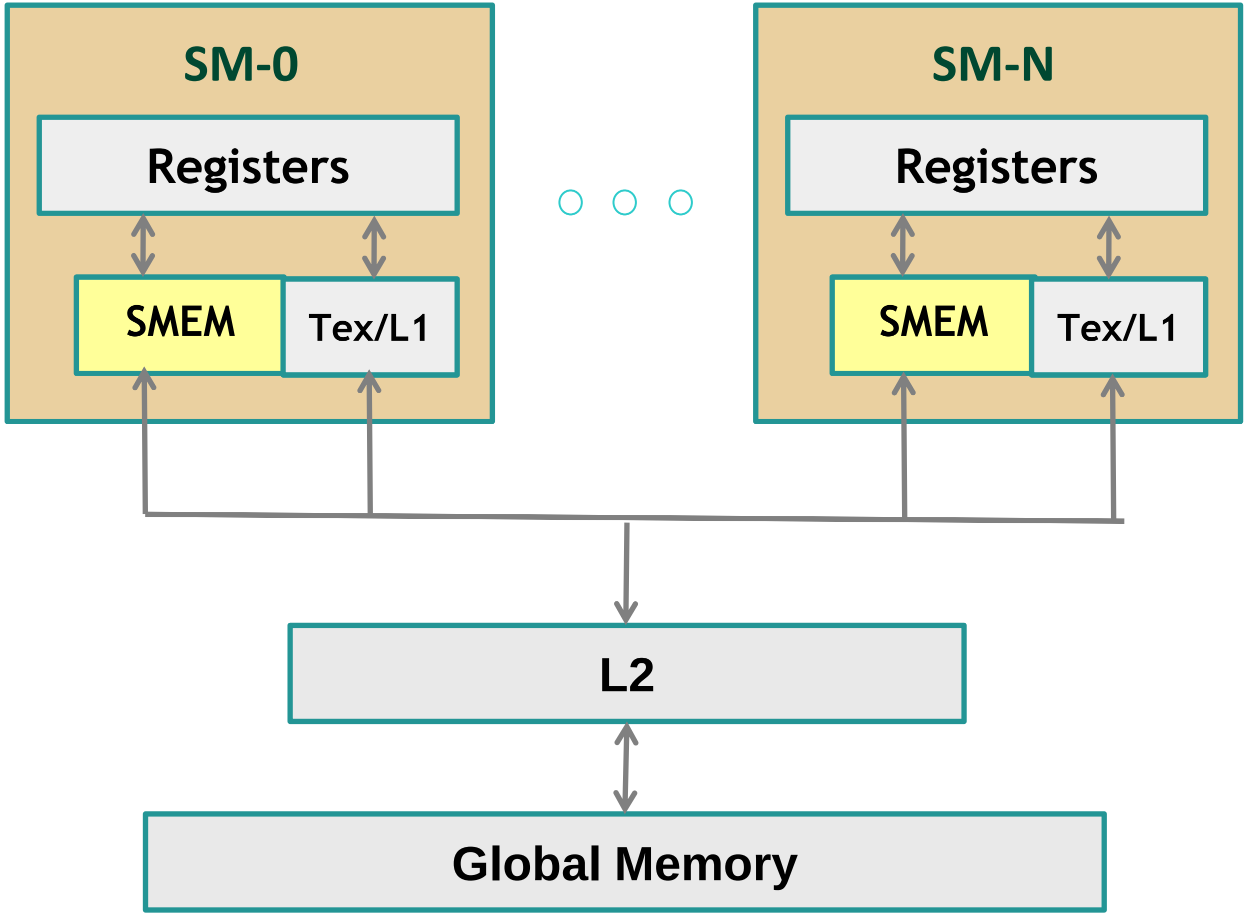
warp 9 instruction 13

warp 3 instruction 34

warp 15 instruction 96

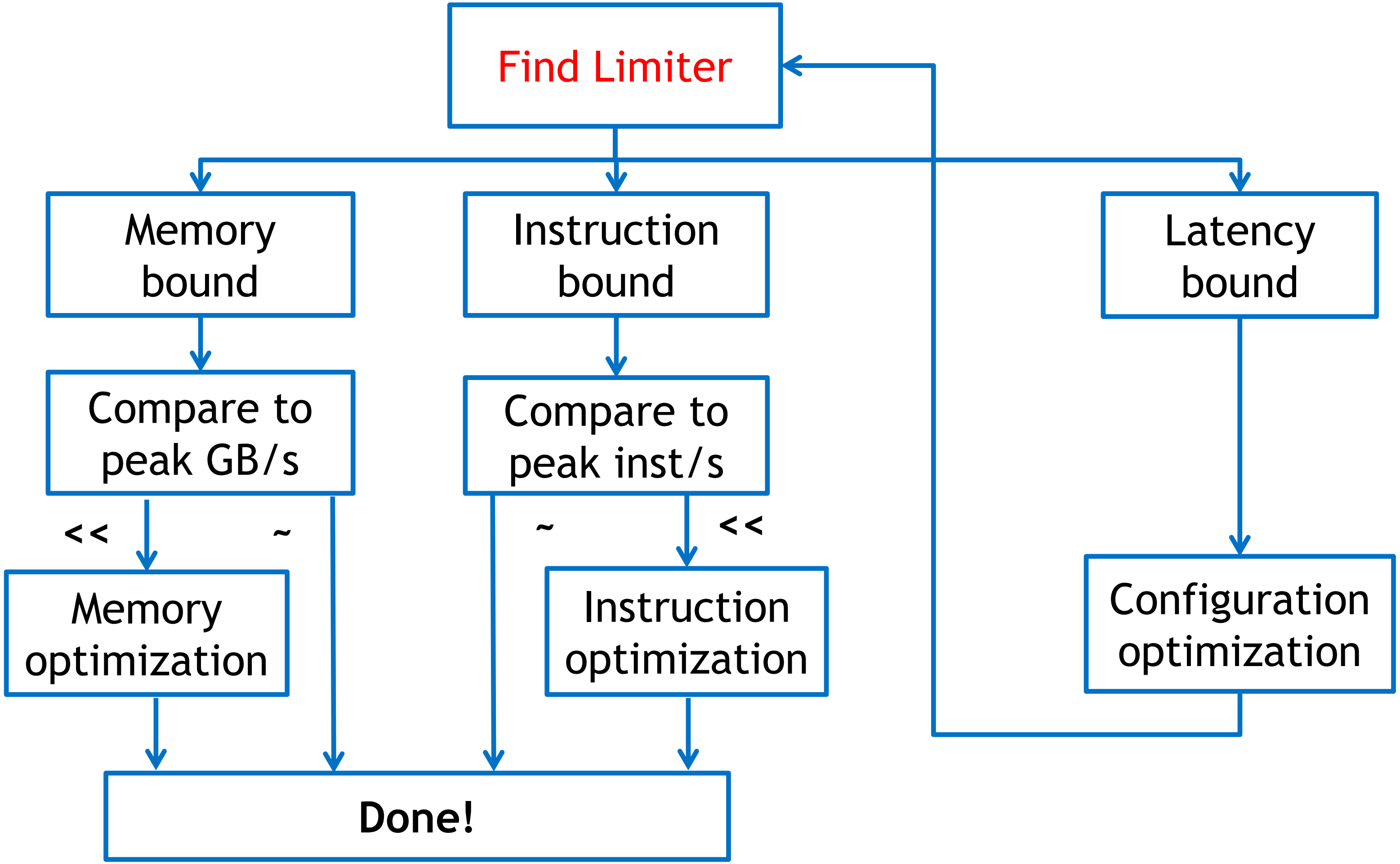
# Volta Memory Hierarchy

- ❑ Register 256 KB
  - Spills to local memory
- ❑ Caches
  - Shared memory
  - L2 cache 6M
  - L1 cache/Texture cache
- ❑ Global memory



# 2. Kernel Optimization

# Optimization Workflow



Find Limiter

Memory bound

Instruction bound

Latency bound

Compare to peak GB/s

Compare to peak inst/s

<<

~

~

<<

Memory optimization

Instruction optimization

Configuration optimization

Done!

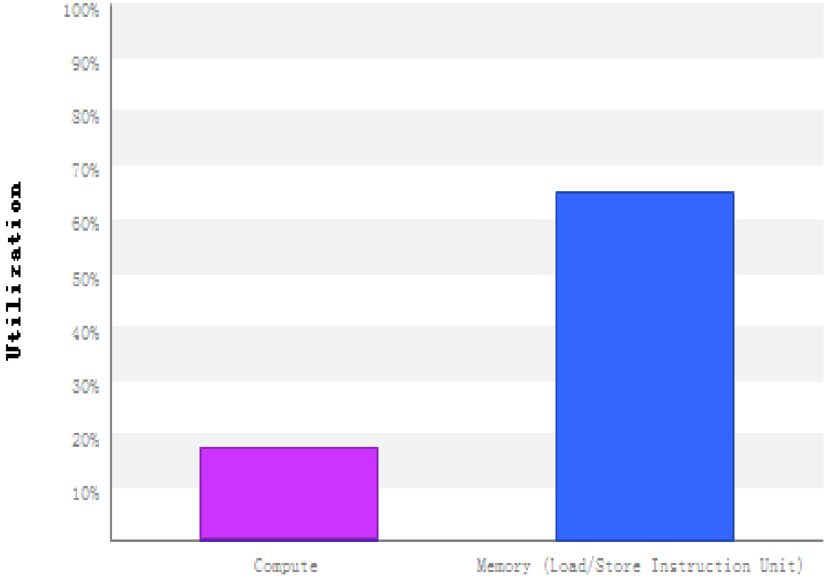
# Profiling Tools

- NVVP & nvprof (legacy)
  - NVIDIA Visual Profiler
    - Timeline
    - cuda kernel profiling
  - nvprof: command line tool
- Nsight System & Nsight Compute
  - Nsight System: system level timeline, including CPU and GPU utilization info
  - Nsight compute: cuda kernel profiling

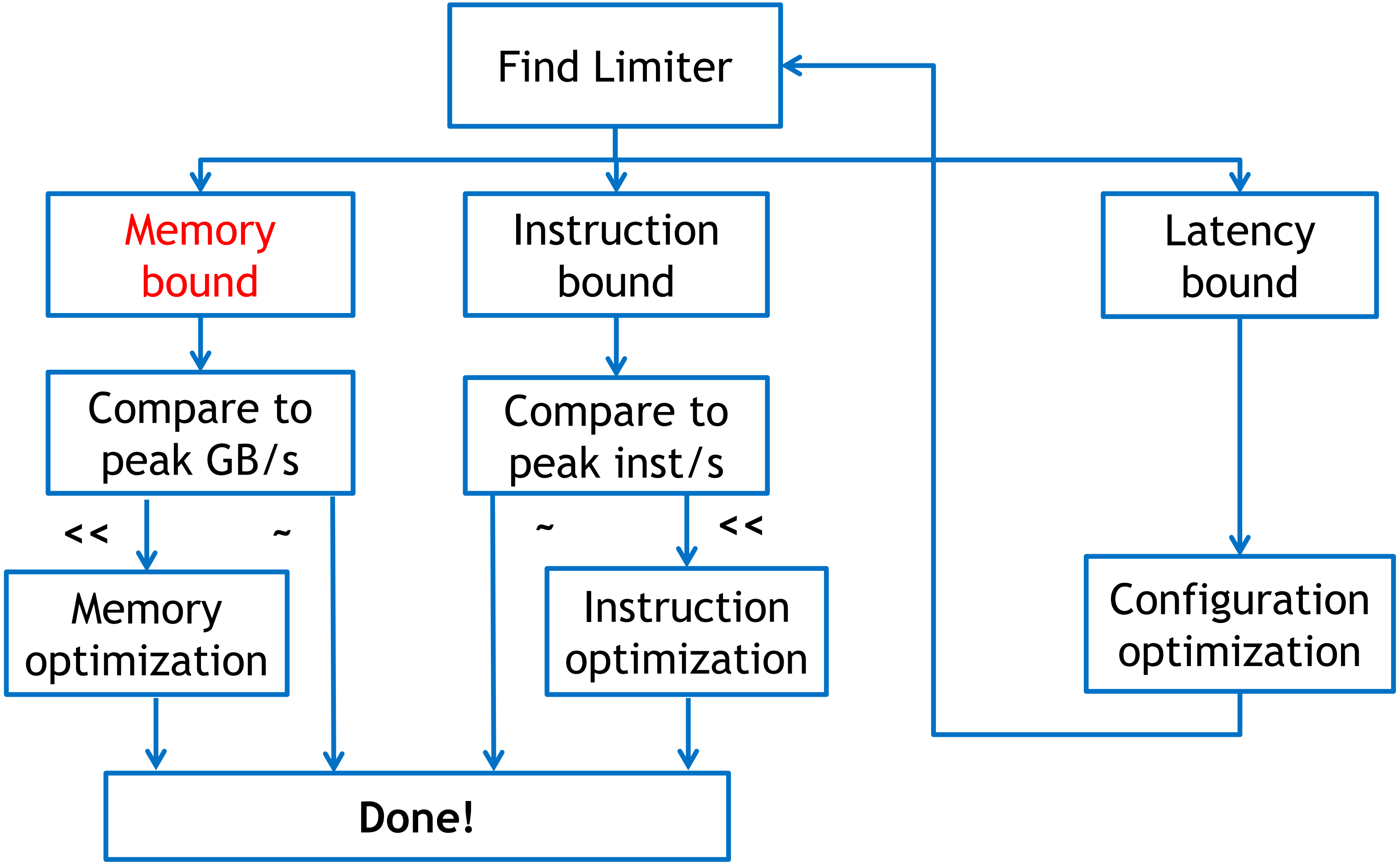
# General Optimization Strategies:

## Measurement

- Find out the limiting factor in kernel performance
  - Memory bandwidth bound
    - memory_utilization >> SM_utilization
  - Instruction throughput bound
    - SM_utilization >> memory_utilization
  - Latency bound
    - both SM_utilization and memory_utilization are low



# Optimization Workflow



# Memory Optimization

- If the utilization of memory resource is much larger than the utilization of SM and effective memory throughput is much lower than the peak

- Purpose: access only data that are absolutely necessary

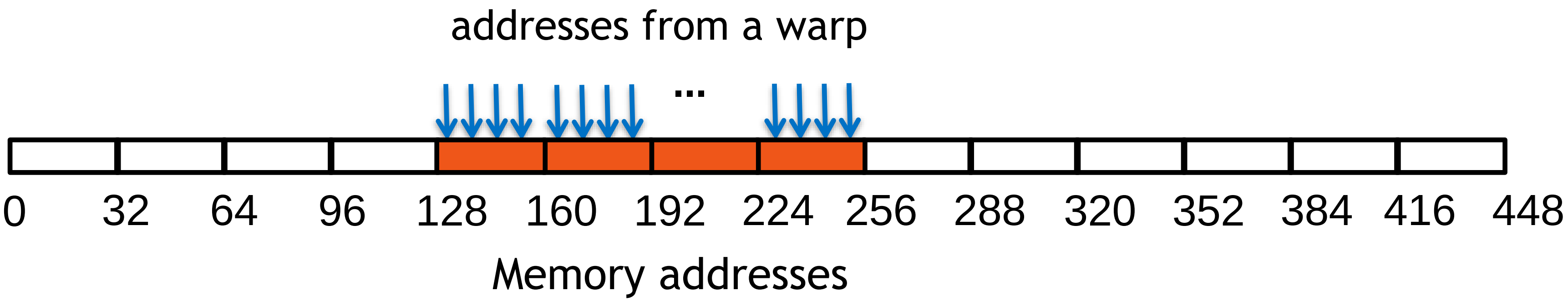
- Major techniques
  - Improve access pattern to reduce wasted transactions
  - Reduce redundant access: read-only cache, shared memory

# Reduce Wasted Transactions

- ❑ Memory accesses are per warp
- ❑ Memory is accessed in discrete chunks
  - Memory is transport in segments = 32B (same as for writes)
  - If a warp can’t take use all of the data in the segments, the rest memory transaction is wasted.

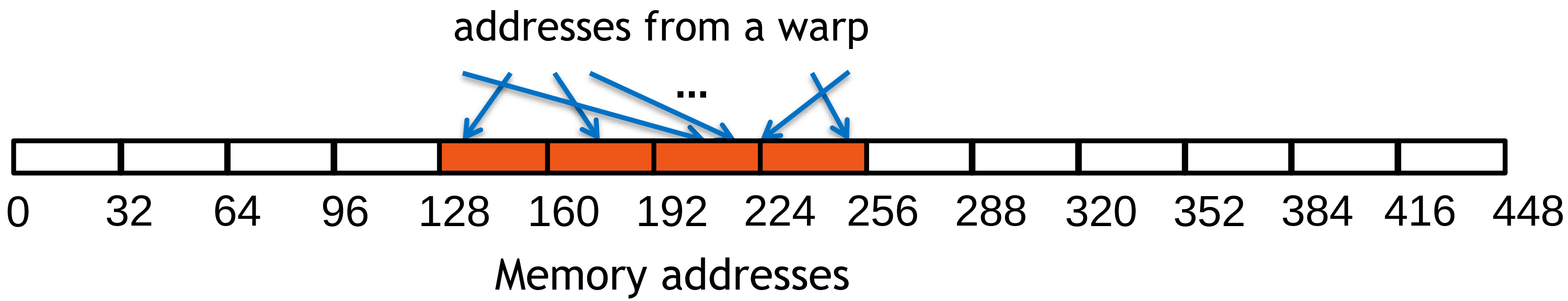
# Reduce Wasted Transactions

- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words
- **Addresses fall within 4 segments**
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss



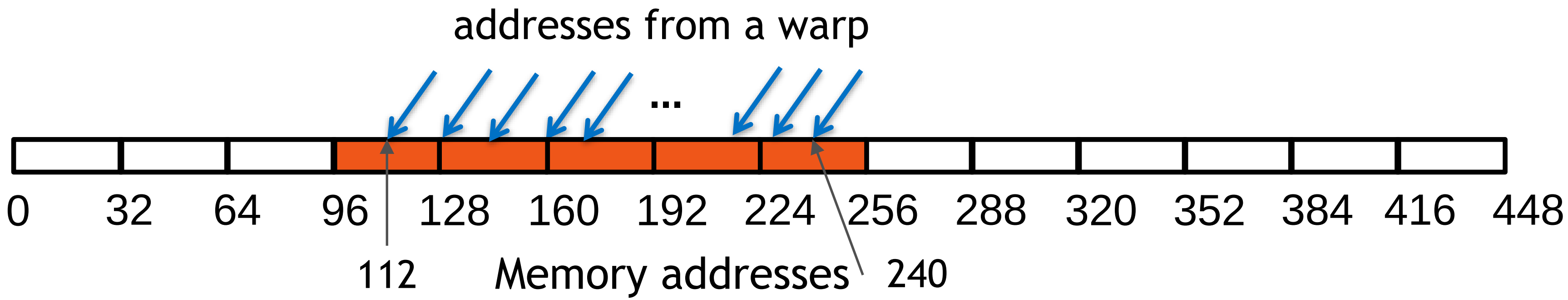
# Reduce Wasted Transactions

- **Scenario:**
  - Warp requests 32 aligned, permuted 4-byte words
- **Addresses fall within 4 segments**
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss



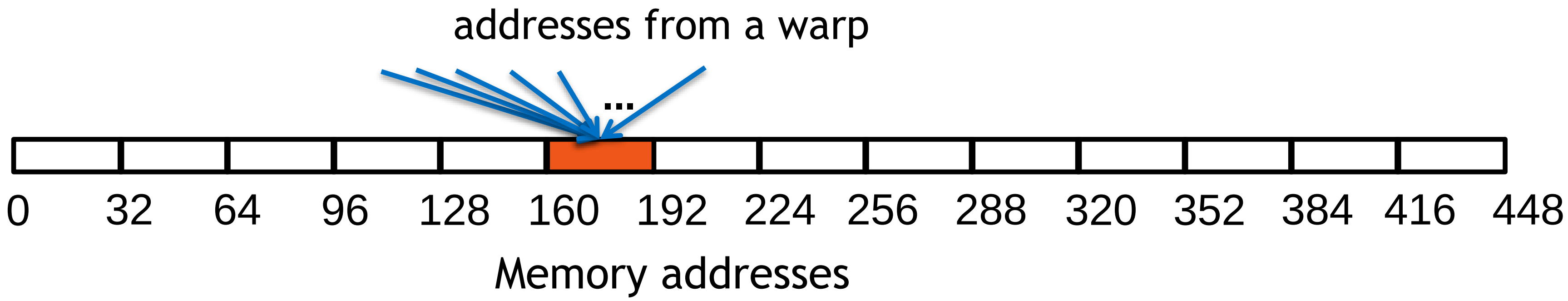
# Reduce Wasted Transactions

- Scenario:
  - Warp requests 32 consecutive 4-byte words, offset from perfect alignment
- Addresses fall within at most 5 segments
  - Bus utilization: at least 80%
    - Warp needs 128 bytes
    - At most 160 bytes move across the bus



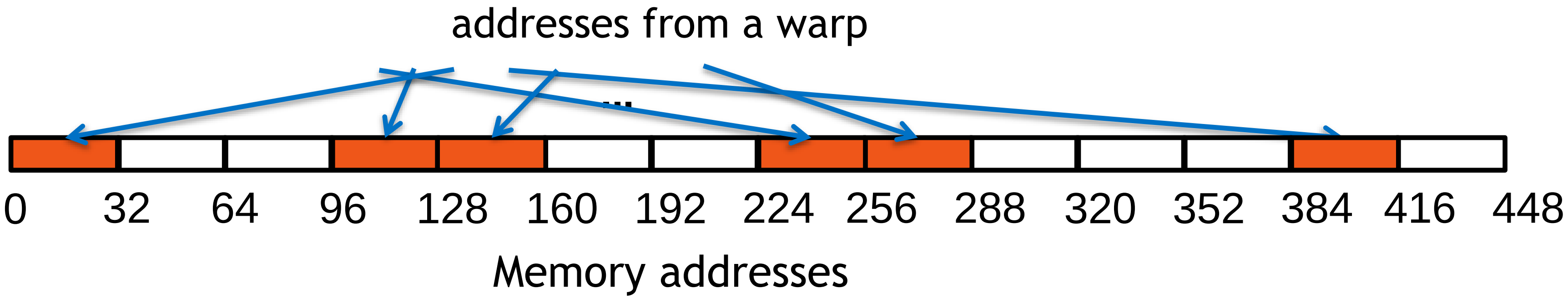
# Reduce Wasted Transactions

- ❑ Scenario:
  - ❑ All threads in a warp request the same 4-byte word
- ❑ Addresses fall within a single segment
  - Bus utilization: 12.5%
    - Warp needs 4 bytes
    - 32 bytes move across the bus on a miss



# Reduce Wasted Transactions

- Scenario:
  - Warp requests 32 scattered 4-byte words
- Addresses fall within N segments
  - Bus utilization: 128 / (N*32)
    - Warp needs 128 bytes
    - N*32 bytes move across the bus on a miss



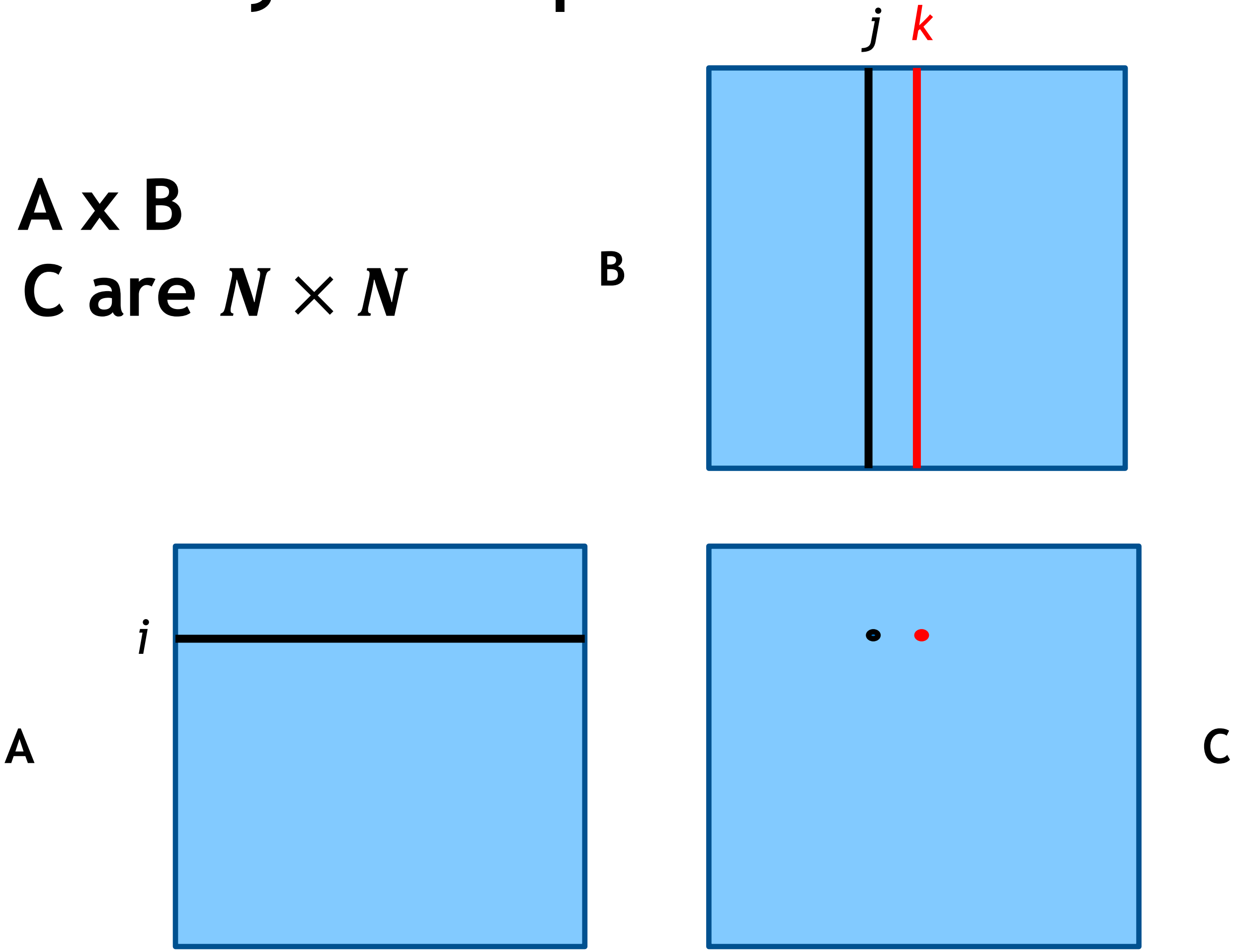
# Shared Memory

- Low latency: a few cycles
- High throughput
- Main use
    - Inter-block communication
    - User-managed cache to reduce redundant global memory accesses
    - Avoid non-coalesced access

# Shared Memory Example: Matrix Multiplication

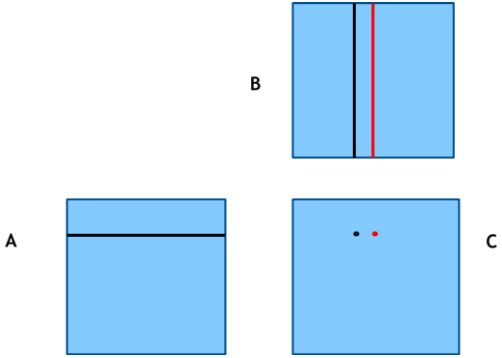
**C = A x B**

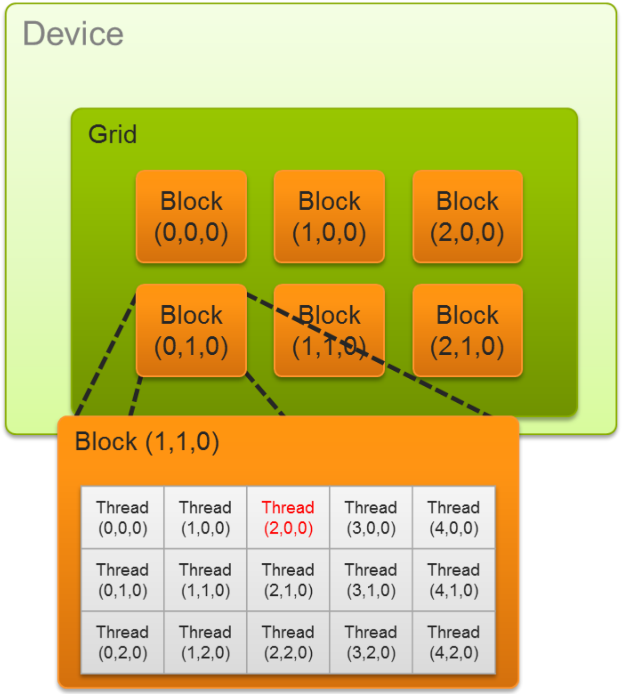
**Both A, B, C are $N \times N$**



**Every thread corresponds to one entry in C.**

# Naïve Kernel





| | | | | |
|---|---|---|---|---|
| Thread (0,0,0) | Thread (1,0,0) | Thread (2,0,0) | Thread (3,0,0) | Thread (4,0,0) |
| Thread (0,1,0) | Thread (1,1,0) | Thread (2,1,0) | Thread (3,1,0) | Thread (4,1,0) |
| Thread (0,2,0) | Thread (1,2,0) | Thread (2,2,0) | Thread (3,2,0) | Thread (4,2,0) |

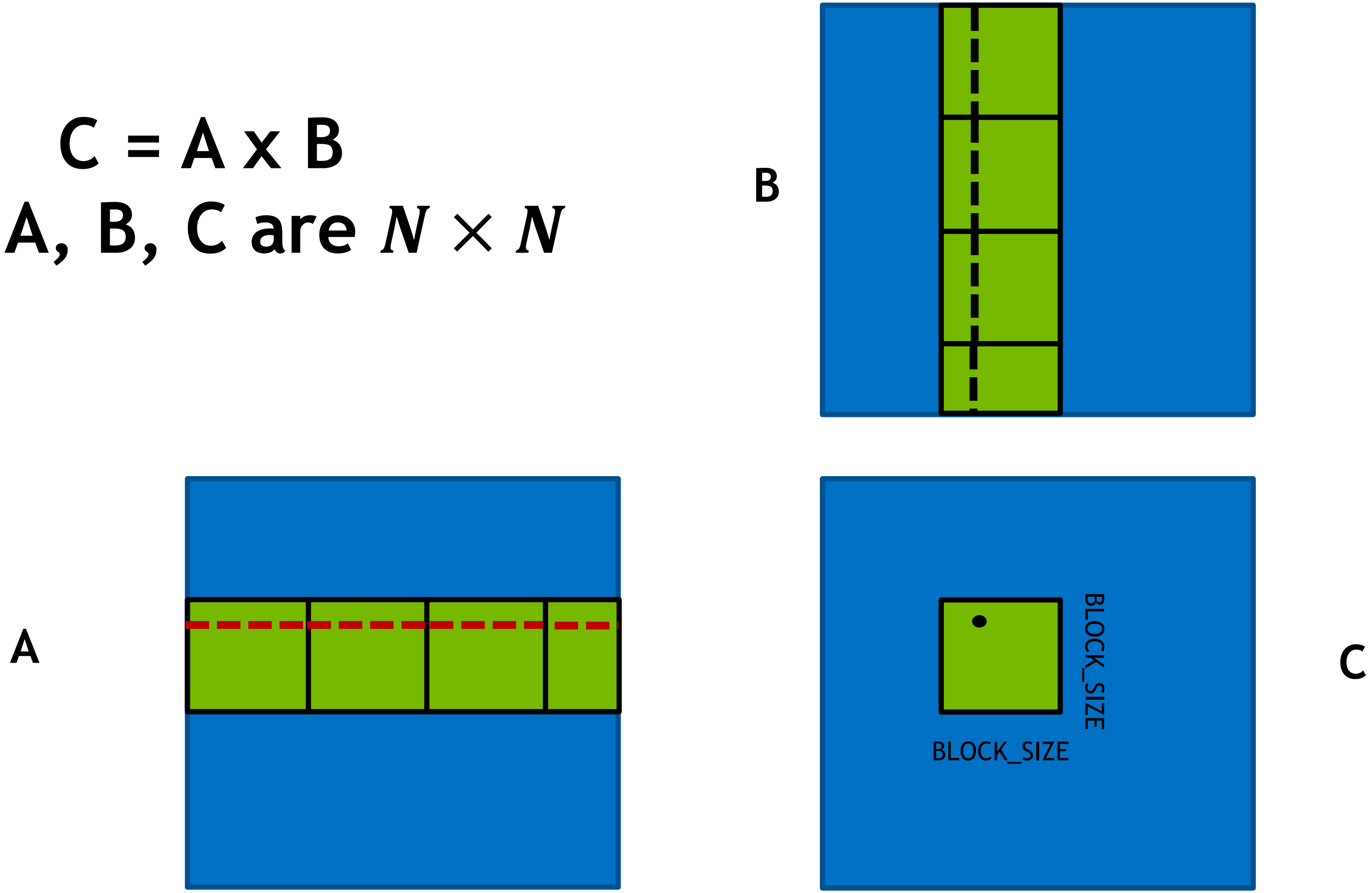
```
__global__ void simpleMultiply(float *a,
                               float *b,
                               float *c,
                               int N)
{
    int row = threadIdx.x + blockIdx.x * blockDim.x;
    int col = threadIdx.y + blockIdx.y * blockDim.y;
    float sum = 0.0f;
    for (int i = 0; i < N; i++)
    {
        sum += a[row * N + i] * b[i * N + col];
    }
    c[row * N + col] = sum;
}
```

**Every thread corresponds to one entry in C**

# Blocked Matrix Multiplication

**C = A x B**

**Both A, B, C are $N \times N$**



**Data reuse in the blocked version**

# Blocked and Cached Kernel

blockDim.x = BLOCK_SIZE
blockDim.y = BLOCK_SIZE

```
__global__ void sharedMatMult(float *a, float *b, float *c, int N)
{
    __shared__ float aTile[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float bTile[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.x * blockDim.x + threadIdx.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y;
    float sum = 0.0f;

    for (int k = 0; k < N; k += BLOCK_SIZE)
    {
        aTile[threadIdx.x][threadIdx.y] = a[row * N + threadIdx.y + k];
        bTile[threadIdx.x][threadIdx.y] = b[(threadIdx.x + k) * N + col];
        __syncthreads();

        for (int i = 0; i < BLOCK_SIZE; i++)
        {
            sum += aTile[threadIdx.x][i] * bTile[i][threadIdx.y];
        }
        __syncthreads();
    }
    c[row * N + col] = sum;
}
```

# Shared Memory Example: Matrix Multiplication

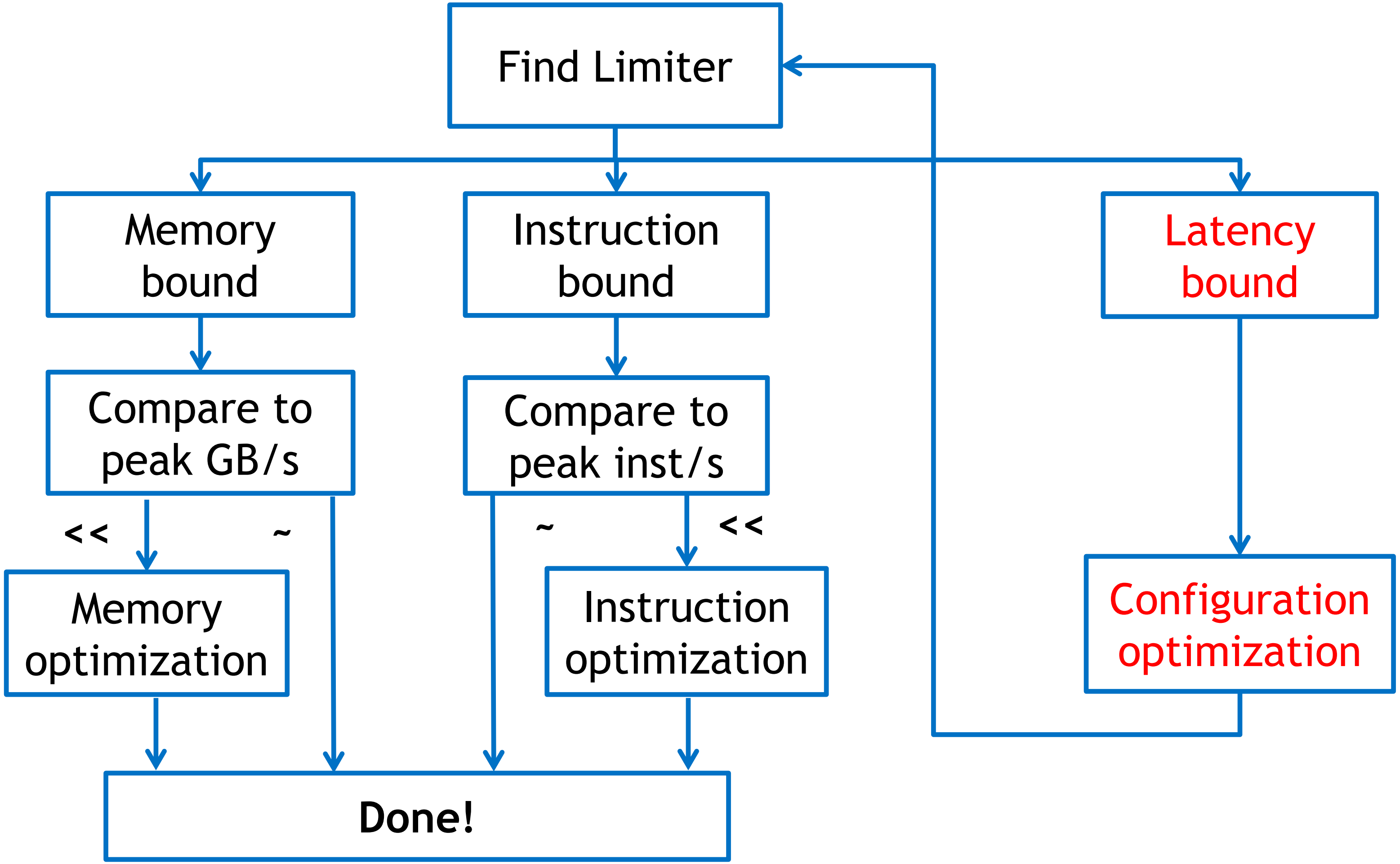
- ❑ Native implementation
  - ❑ global read = N * N * (2*N) = 2 * $N^3$
- ❑ Blocked implementation with shared memory
  - ❑ global read = N/BLOCK_SIZE * N/BLOCK_SIZE * (BLOCK_SIZE * N+ BLOCK_SIZE * N)

    = 2 * $N^3$/BLOCK_SIZE

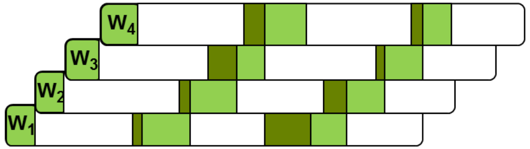
# Summary of Memory Optimization

- ❑ Memory is transport in segments = 32B
- ❑ Improve access pattern to reduce wasted transactions
- ❑ Read-only cache can help the data load from global memory
- ❑ Shared memory is on-chip memory, fast, can be accessed by all threads in a block.

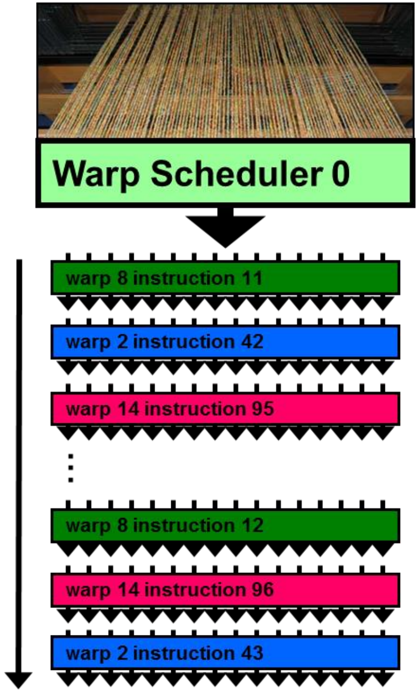
# Optimization Workflow



# Latency Optimization



- When the code is latency bound
  - Both the memory and instruction throughputs are far from the peak
- Latency hiding: switching threads / ILP
  - A thread blocks when one of the operands isn't ready
- Purpose: have enough warps to hide latency
- Major techniques: increase active warps



# Occupancy & Active Warps

- ❑ Occupancy: ratio of active warps per SM to the maximum number of allowed warps
  - Maximum number: 64 in Volta GV100 Per SM(or pre-volta), but **32** in Turing
- ❑ We need the occupancy to be high enough to hide latency
- ❑ Occupancy is limited by resource usage

# Dynamical Partitioning of SM Resources

- Shared memory is partitioned among blocks
- Registers are partitioned among threads: <= 255
- Thread block slots: <= 32
- Thread slots: <= 2048 (1024 in Turing)
- Any of those can be the limiting factor on how many threads can be launched at the same time on a SM

# How do the SM resources affect the occupancy

- ❑ If the block size is 32, what is the upper limit of occupancy?
- ❑ If each block uses 32 KB shared memory and block size is 64, what is the upper limit of occupancy?
- ❑ If each thread uses 64 registers, what is the upper limit of occupancy?

# Enough Block and Block Size

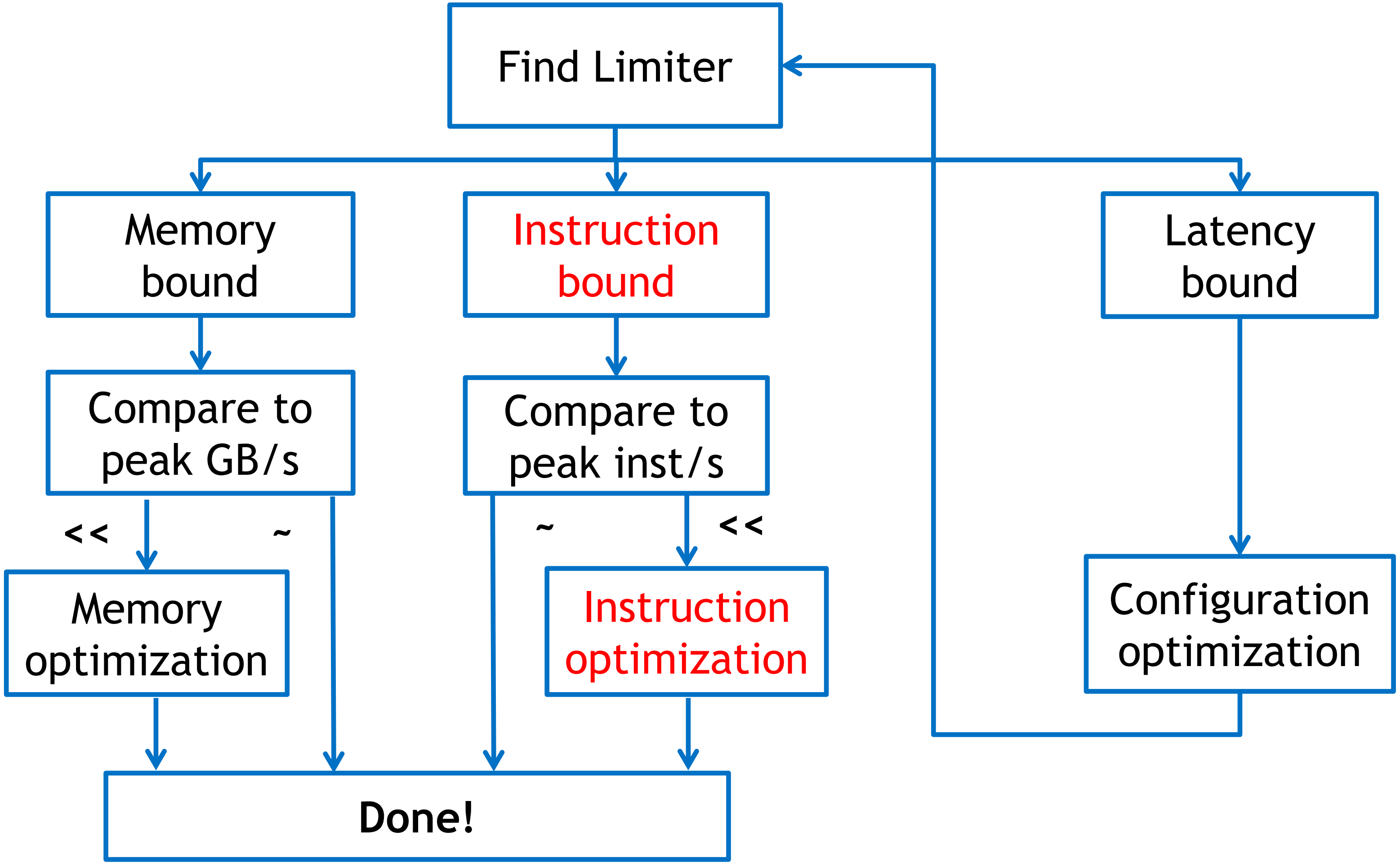
- ❑ Enough Block: # of blocks >> # of SM (for GV100 80 SM) to scale well to future device.
- ❑ Enough Block size: Min 64. Generally use 128 or 256. But use whatever is best for your app.
- ❑ Depends on the problem, do experiments!

# Occupancy Optimizations

- Know the current occupancy: NVIDIA Visual profiler / Nsight Compute
- Adjust resource usage to increase occupancy
  - Change block size
  - Limit register usage
    - Compiler option –maxrregcount=n: per file
    - __launch_bounds__: per kernel
  - Dynamical allocating shared memory: third parameter in launch configuration.

```
__global__ void
__launch_bounds__(maxThreadsPerBlock,
minBlocksPerMultiprocessor)
MyKernel(...)
{
    ...
}
```

# Optimization Workflow



# Instruction Optimization

- If you find out the code is instruction bound
  - Compute-intensive algorithm can easily become memory-bound if not careful enough
  - Typically, worry about instruction optimization after memory and execution configuration optimizations
- Purpose: reduce instruction count
  - Use less instructions to get the same job done
- Major techniques
  - Use high throughput instructions
  - Reduce wasted instructions: branch divergence, etc.

# Reduce Instruction Count

- Use float if precision allow
  - Adding “f” to floating literals (e.g. 1.0f) because the default is double

- Fast math functions
  - Two types of runtime math library functions
    - func(): slower but higher accuracy (5 ulp or less)
  - __func(): fast but lower accuracy (see prog. guide for full details)
  - -use_fast_math: forces every func() to __func ()

# Control Flow

- Divergent branches:
  - Threads within a single warp take different paths
  - Example with divergence:
    - `if (threadIdx.x%2 == 0) {...} else {...}`
    - Branch granularity < warp size
  - Different execution paths within a warp are serialized

- Different warps can execute different code with no impact on performance

- Avoid diverging within a warp
  - Example without divergence:
    - `if ((threadIdx.x/WARP_SIZE)%2 == 0) {...} else {...}`
    - Branch granularity is a whole multiple of warp size

# 3. CPU-GPU Interaction Optimization

# Minimizing CPU-GPU data transfer

- Host<->device data transfer has much lower bandwidth than global memory access.
  - 16 GB/s (PCIe x16 Gen3) vs  250 GB/s & 10.6 T inst/s (GP100)
- Minimize transfer
  - Intermediate data can be allocated, operated, de-allocated directly on GPU
  - Sometimes it’s even better to recompute on GPU
- Group transfer
  - One large transfer much better than many small ones
  - Overlap memory transfer with computation

# Streams and Async API

- Default API:
  - Kernel launches are asynchronous with CPU
  - Memcopies (D2H, H2D) block CPU thread
  - CUDA calls are serialized by the driver
- Streams and async functions provide:
  - Memcopies (D2H, H2D) asynchronous with CPU
  - Ability to concurrently execute a kernel and a memcopy
  - Concurrent kernel
- Stream = sequence of operations that execute in issue-order on GPU
  - Operations from different streams can be interleaved
  - A kernel and memcopy from different streams can be overlapped

# Pinned (non-pageable) memory

- Pinned memory enables:
  - memcopies asynchronous with CPU & GPU

- Usage
  - cudaHostAlloc / cudaFreeHost instead of malloc / free

- Note:
  - pinned memory is essentially removed from virtual memory
  - cudaHostAlloc is typically very expensive

# Overlap kernel and memory copy

Requirements:

D2H or H2D memcopy from pinned memory

Device with compute capability ≥ 1.1 (G84 and later)

Kernel and memcopy in different, non-0 streams

Code:

```
cudaStream_t   stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream2>>>(…);
```

**potentially overlapped**

# 4. Introduction to libraries

# cuDNN

- ❑ NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks.
- ❑ Provide function for following DNN application:
  - ❑ Convolution forward and backward, including cross-correlation.
  - ❑ Pooling forward and backward.
  - ❑ Softmax forward and backward.
  - ❑ Neuron activations forward and backward.(ReLU, Sigmoid, Tanh)
  - ❑ Tensor transformation functions.
  - ❑ LRN, LCN and batch normalization forward and backward

# cuDNN

- CPU interface(as normal C/C++ function).
  - Easy to use.
  - Minimal code modification to use GPU.
- Context-based:
  - Can be assign to target stream.
  - Can be overlapped with other GPU tasks.
- Optimal performance.
  - Highly-optimized implementation.
  - Tensor-core optimized.

# cuDNN

- ❑ Common usage:
  - ❑ Call cudaSetDevice() to bind the current thread to a CUDA device.
  - ❑ Initialize a handle to the library context by calling cudnnCreate() on this device.
  - ❑ Optionally bind the context to a CUDA stream using cudnnSetStream() (otherwise default stream).
  - ❑ Call cuDNN APIs and pass the context handle to them explicitly.
  - ❑ Release the resources associated with the context using cudnnDestroy() when finish using cuDNN.
- ❑ For individual cuDNN API usage, refer to the documents.

# NCCL

- NVIDIA Collective Communications Library is a library of multi-GPU collective communication primitives that are topology-aware.
- GPU-oriented: 1 rank per GPU (traditional CPU-oriented library: 1 rank per process).
- Easy-to-use: Very similar to MPI library.
    - Support communication:
        - AllReduce
        - Broadcast
        - AllGather
        - ReduceScatter
        - Send/Receive(preview)

# NCCL

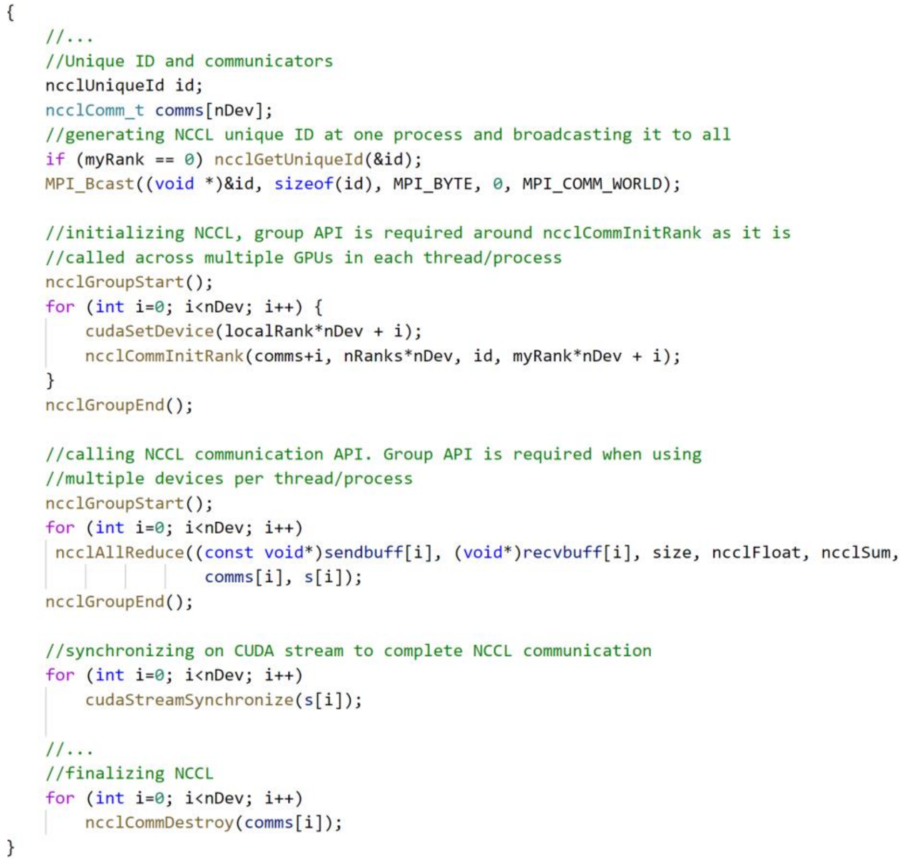
- ❑ NCCL Usage:
    - ❑ Generate a Unique ID for all ranks to setup communicator.
        - ❑ Don’t need it if using ncclCommInitAll when only one-process is involved.
        - ❑ Need third-party communication library to communicate the ID.
    - ❑ Initialize a communicator on every GPU(rank) involved in the communication with the unique ID.
    - ❑ Using the communicators to do the communication.
    - ❑ Destroy the communicator when no longer needed.

# NCCL

- Group mechanism:
  - Many operations in NCCL requires synchronization between ranks.
    - Communicator initialization and collective communication.
  - Not a problem for CPU-oriented libraries, since all ranks operate concurrently.
  - When a thread/process operate on multiple GPUs, it is a deadlock problem.
  - Using group mechanism to batch operations from all GPUs together to avoid deadlock.

# NCCL

- Simple NCCL example:



```
{
    //...
    //Unique ID and communicators
    ncclUniqueId id;
    ncclComm_t comms[nDev];
    //generating NCCL unique ID at one process and broadcasting it to all
    if (myRank == 0) ncclGetUniqueId(&id);
    MPI_Bcast((void *)&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD);

    //initializing NCCL, group API is required around ncclCommInitRank as it is
    //called across multiple GPUs in each thread/process
    ncclGroupStart();
    for (int i=0; i<nDev; i++) {
        cudaSetDevice(localRank*nDev + i);
        ncclCommInitRank(comms+i, nRanks*nDev, id, myRank*nDev + i);
    }
    ncclGroupEnd();

    //calling NCCL communication API. Group API is required when using
    //multiple devices per thread/process
    ncclGroupStart();
    for (int i=0; i<nDev; i++)
     ncclAllReduce((const void*)sendbuff[i], (void*)recvbuff[i], size, ncclFloat, ncclSum,
                    comms[i], s[i]);
    ncclGroupEnd();

    //synchronizing on CUDA stream to complete NCCL communication
    for (int i=0; i<nDev; i++)
        cudaStreamSynchronize(s[i]);

    //...
    //finalizing NCCL
    for (int i=0; i<nDev; i++)
        ncclCommDestroy(comms[i]);
}
```

# 5. Summary and Reference

# Summary

- GPU Architecture
  - Warp is successive 32 threads in a block.
  - On-chip memory (Shared memory, L1/Texture cache) has much lower latency compared with off-chip global memory.

- Memory Optimization
  - Improve access pattern to reduce wasted transactions (coalesced access).
  - Use read-only cache or shared memory to reduce the access to global memory.

# Summary

- ❑ Configuration Optimization
  - Enough block and block size, appropriate SM resource assignment to ensure enough active warps and high occupancy.
- ❑ Instruction Optimization
  - Use high throughput instructions if possible.
  - Avoid branch divergence to reduce wasted instructions.
- ❑ CPU-GPU async communication
  - Reduce CPU-GPU data copy.
  - Use streams to overlap CPU-GPU memcpies and kernel launchs.

# Summary

❑ cuDNN is a highly-optimized DNN routine library provided by NVIDIA.

❑ NCCL is a topology-aware and highly-optimized library for multi-GPU communication provided by NVIDIA.

NVIDIA