King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 26: Warp Synchronous Programming; Cooperative Thread Groups; Programming Tensor Cores

Markus Hadwiger, KAUST

Reading Assignment #15++ (until Dec 14++)

Further suggested reading:

- Raihan et al., arXiv, Feb 2019, Modeling Deep Learning Accelerator Enabled GPUs
  - **https://arxiv.org/abs/1811.08309**
  - See also GPGPU-SIM: **http://www.gpgpu-sim.org/**
- CUTLASS 2.4 template library (last update Nov 2020)
  - **https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/**
  - **https://github.com/NVIDIA/cutlass**
- Register Cache: Caching for Warp-Centric CUDA Programs
  - **https://devblogs.nvidia.com/register-cache-warp-cuda/**
- cuSPARSE library description in the CUDA SDK
- CUSP library: **http://cusplibrary.github.io/**
- Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS, Maxim Naumov
  - **https://developer.download.nvidia.com/assets/cuda/files/psts_white_paper_final.pdf**

Quiz #4: Dec 9

Organization

- First 30 min of lecture
- No material (book, notes, ...) allowed

Content of questions

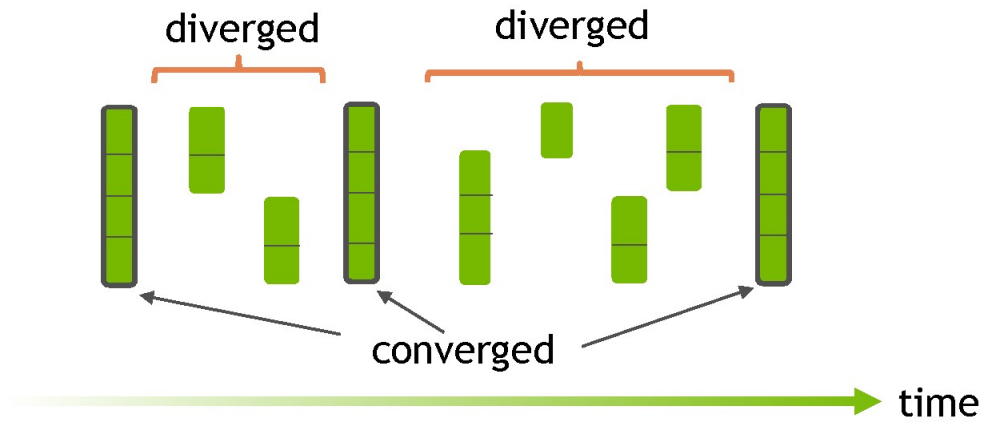
- Lectures (both actual lectures and slides)
- Reading assigments
- Programming assignments (algorithms, methods)
- Solve short practical examples

WARP SYNCHRONOUS PROGRAMMING IN CUDA 9.0

CUDA WARP THREADING MODEL

NVIDIA GPU multiprocessors create, manage, schedule and execute threads in **warps** (32 parallel threads).

Threads in a warp may diverge and re-converge during execution.



Full efficiency may be realized when all 32 threads of a warp are converged.

WARP SYNCHRONOUS PROGRAMMING

Warp synchronous programming is a CUDA programming technique that leverages warp execution for efficient inter-thread communication.

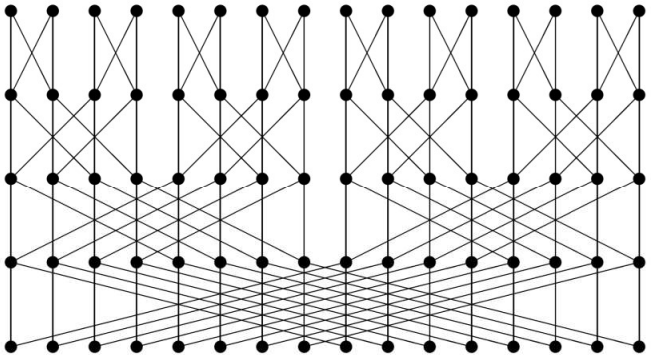
- e.g. reduction, scan, aggregated atomic operation, etc.

CUDA C++ supports warp synchronous programming by providing warp synchronous built-in functions and cooperative group collectives.

EXAMPLE: SUM ACROSS A WARP

```
val = input[lane_id];
val += __shfl_xor_sync(0xffffffff, val, 1);
val += __shfl_xor_sync(0xffffffff, val, 2);
val += __shfl_xor_sync(0xffffffff, val, 4);
val += __shfl_xor_sync(0xffffffff, val, 8);
val += __shfl_xor_sync(0xffffffff, val, 16);
```

$\text{val} = \sum_{i=0}^{32} input[i]$

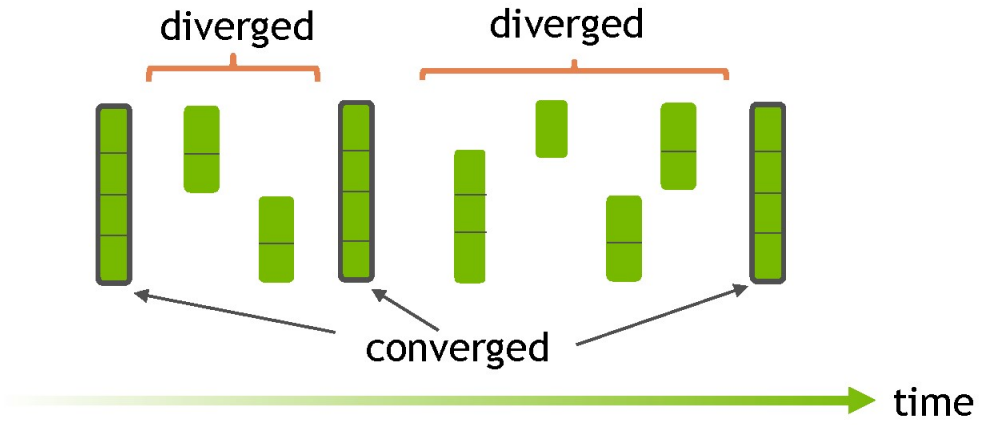


HOW TO WRITE WARP SYNCHRONOUS PROGRAMMING

Make Sync Explicit

Thread re-convergence

- Use built-in functions to converge threads explicitly
- Do not rely on implicit thread re-convergence.



HOW TO WRITE WARP SYNCHRONOUS PROGRAMMING

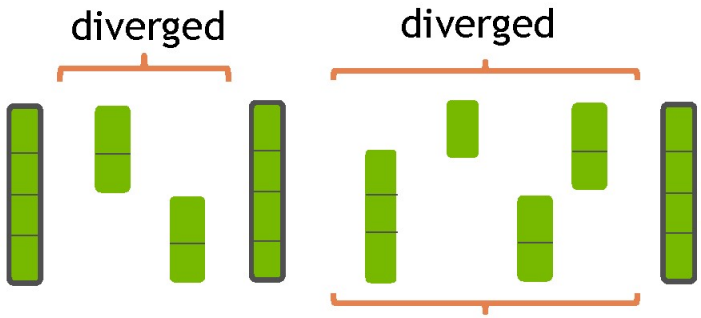
Make Sync Explicit

Thread re-convergence

- Use built-in functions to converge threads explicitly
- Do not rely on implicit thread re-convergence.

Data exchange between threads

- Use built-in functions to sync threads and exchange data in one step.
- When using shared memory, avoid data races between convergence points.



Reading and writing the same memory location by different threads may cause data races.

WARP SYNCHRONOUS BUILT-IN FUNCTIONS

Three Categories (New in CUDA 9.0)

Active-mask query: which threads in a warp are active

- __activemask

Synchronized data exchange: exchange data between threads in warp

- __all_sync, __any_sync, __uni_sync, __ballot_sync
- __shfl_sync, __shfl_up_sync, __shfl_down_sync, __shfl_xor_sync
- __match_any_sync, __match_all_sync

Threads synchronization: synchronize threads in a warp and provide a memory fence

- __syncwarp

EXAMPLE: ALIGNED MEMORY COPY

__activemask __all_sync

```
// pick the optimal memory copy based on the alignment

__device__ void memorycopy(char *tptr, char *sptr, size_t size) {

   unsigned mask = __activemask();

   if (__all_sync(mask, is_all_aligned(tptr, sptr, 16))
       return memcpy_aligned_16(tptr, sptr, size);

   if (__all_sync(mask, is_all_aligned(tptr, sptr, 8))
       return memcpy_aligned_8(tptr, sptr, size);
   …
}
```

EXAMPLE: ALIGNED MEMORY COPY

__activemask __all_sync

```
// pick the optimal memory copy based on the alignment

__device__ void memorycopy(char *tptr, char *sptr, size_t size) {

   unsigned mask = __activemask();

   if (__all_sync(mask, is_all_aligned(tptr, sptr, 16))
       return memcpy_aligned_16(tptr, sptr, size);

   if (__all_sync(mask, is_all_aligned(tptr, sptr, 8))
       return memcpy_aligned_8(tptr, sptr, size);
   …
}
```

Find the active threads

EXAMPLE: ALIGNED MEMORY COPY

__activemask __all_sync

```
// pick the optimal memory copy based on the alignment

__device__ void memorycopy(char *tptr, char *sptr, size_t size) {

   unsigned mask = __activemask();

   if (__all_sync(mask, is_all_aligned(tptr, sptr, 16))
       return memcpy_aligned_16(tptr, sptr, size);

   if (__all_sync(mask, is_all_aligned(tptr, sptr, 8))
       return memcpy_aligned_8(tptr, sptr, size);
   …
}
```

Find the active threads

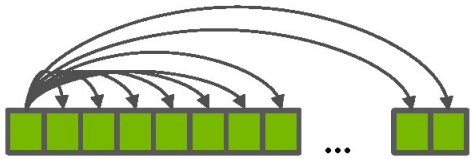
Returns true when all threads in ‘mask’ have the same predicate value

EXAMPLE: SHUFFLE

__shfl_sync, __shfl_down_sync

Broadcast: all threads get the value of ‘x’ from lane id 0

```
y = __shfl_sync(0xffffffff, x,  0);
```

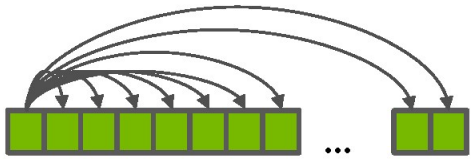


EXAMPLE: SHUFFLE

__shfl_sync, __shfl_down_sync

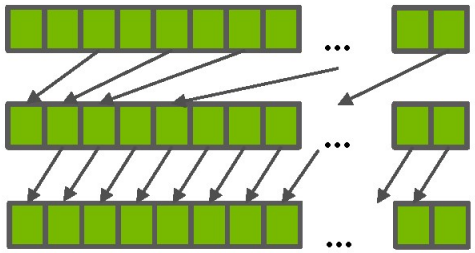
Broadcast: all threads get the value of 'x' from lane id 0

```
y = __shfl_sync(0xffffffff, x,  0);
```



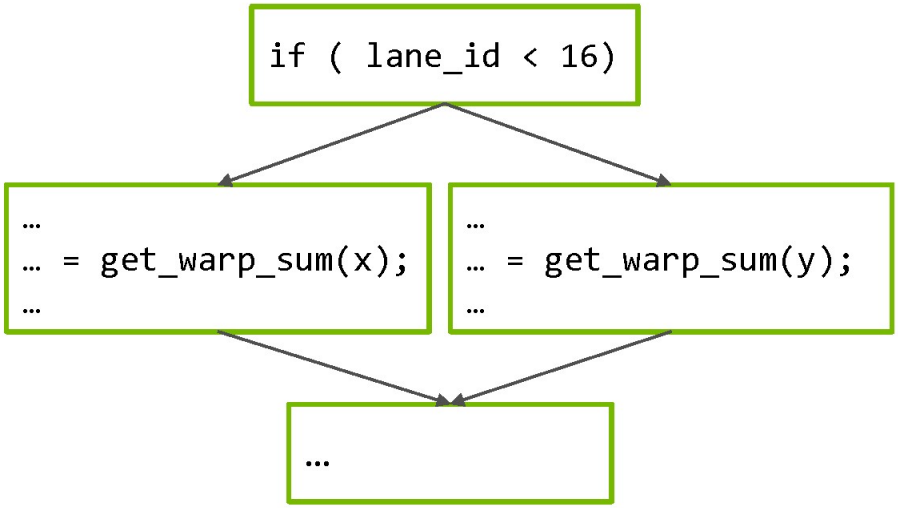
Reduction:

```
for (int offset = 16; offset > 0; offset /= 2)
    val += __shfl_down_sync(0xffffffff, val, offset);
```



EXAMPLE: DIVERGENT BRANCHES

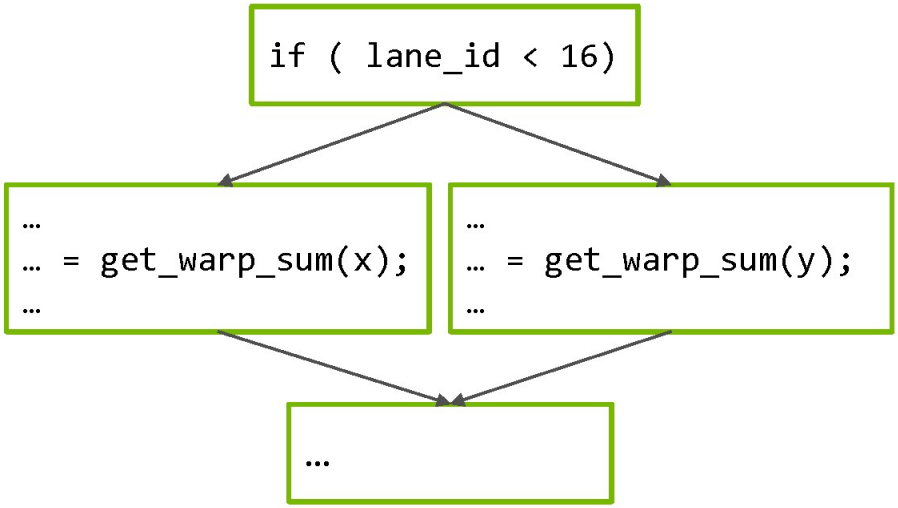
All *_sync built-in functions can be used in divergent branches on Volta




```
#define FULLMASK 0xffffffff

__device__ int get_warp_sum(int v) {
   for (int i = 1; i < 32; i = i*2)
       v += __shfl_xor_sync(FULLMASK, v, i);
   return v;
}
```

EXAMPLE: DIVERGENT BRANCHES

All *_sync built-in functions can be used in divergent branches on Volta




```
#define FULLMASK 0xffffffff

__device__ int get_warp_sum(int v) {
   for (int i = 1; i < 32; i = i*2)
       v += __shfl_xor_sync(FULLMASK, v, i);
   return v;
}
```

Possible to write a library function that performs warp synchronous programming w/o requiring it to be called convergently.

EXAMPLE: REDUCTION VIA SHARED MEMORY

__syncwarp

Re-converge threads and perform memory fence

```
v += shmem[tid+16]; __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+8];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+4];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+2];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+1];  __syncwarp();
shmem[tid] = v;
```

BUT WHAT'S WRONG WITH THIS CODE?

```
v += shmem[tid+16];
shmem[tid] = v;
v += shmem[tid+8];
shmem[tid] = v;
v += shmem[tid+4];
shmem[tid] = v;
v += shmem[tid+2];
shmem[tid] = v;
v += shmem[tid+1];
shmem[tid] = v;
```

IMPLICIT WARP SYNCHRONOUS PROGRAMMING

Unsafe and Unsupported

Implicit warp synchronous programming builds upon two unreliable assumptions,

- implicit thread re-convergence points, and
- Implicit lock-step execution of threads in a warp.

Implicit warp synchronous programming is unsafe and unsupported.

Make warp synchronous programming safe by making synchronizations explicit.

IMPLICIT THREAD RE-CONVERGENCE

Unreliable Assumption 1

Example 1:

```
if (lane_id < 16)
   A;
else
   B;
assert(__activemask() == 0xffffffff);
```

IMPLICIT THREAD RE-CONVERGENCE

Unreliable Assumption 1

Example 1:

```
if (lane_id < 16)
   A;
else
   B;
assert(__activemask() == 0xffffffff);
```

not guaranteed to be true

Solution

- Do not reply on implicit thread re-convergence
- Use warp synchronous built-in functions to ensure convergence

IMPLICIT LOCK-STEP EXECUTION

Unreliable Assumption 2

Example 2

```
if (__activemask() == 0xffffffff) {
    assert(__activemask() == 0xffffffff);
}
```

IMPLICIT LOCK-STEP EXECUTION

Unreliable Assumption 2

Example 2

```
if (__activemask() == 0xffffffff) {
    assert(__activemask() == 0xffffffff); not guaranteed to be true
}
```

Solution

- Do not reply on implicit lock-step execution
- Use warp synchronous built-in functions to ensure convergence

IMPLICIT LOCK-STEP EXECUTION

Unreliable Assumption 2

Example 3

```
shmem[tid] += shmem[tid+16];
shmem[tid] += shmem[tid+8];
shmem[tid] += shmem[tid+4];
shmem[tid] += shmem[tid+2];
shmem[tid] += shmem[tid+1];
```

IMPLICIT LOCK-STEP EXECUTION

Unreliable Assumption 2

Example 3

```
shmem[tid] += shmem[tid+16];
shmem[tid] += shmem[tid+8];
shmem[tid] += shmem[tid+4];
shmem[tid] += shmem[tid+2];
shmem[tid] += shmem[tid+1];
```

data race

Solution

- Make sync explicit

```
v += shmem[tid+16]; __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+8];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+4];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+2];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+1];  __syncwarp();
shmem[tid] = v;
```

LEGACY WARP-LEVEL BUILT-IN FUNCTIONS

Deprecated in CUDA 9.0

Legacy built-in functions

- __all(), __any(), __ballot(), __shfl(), __shfl_up(), __shfl_down(), __shfl_xor()

These legacy warp-level built-in functions can perform data exchange between the active threads in a warp.

They do not ensure which threads are active.

They are deprecated in CUDA 9.0 on all architectures.

COOPERATIVE GROUPS VS BUILT-IN FUNCTIONS

Example: warp aggregated atomic

```
// increment the value at ptr by 1 and return the old value
__device__ int atomicAggInc(int *p);
```

```
coalesced_group g = coalesced_threads();


int res;
if (g.thread_rank() == 0)
   res = atomicAdd(p, g.size());
res = g.shfl(res, 0);
return g.thread_rank() + res;
```

```
int mask = __activemask();
int rank = __popc(mask & __lanemask_lt());
int leader_lane = __ffs(mask) - 1;
int res;
if (rank == 0)
   res = atomicAdd(p, __popc(mask));
res = __shfl_sync(mask, res, leader_lane);
return rank + res;
```

WARP SYNCHRONOUS PROGRAMMING IN CUDA 9.0

New warp synchronous built-in functions ensure reliable synchronizations.

New warp synchronous built-in functions can be used divergently on Volta.

Legacy warp built-in functions are deprecated.

Cooperative groups offers

- Higher-level abstraction of thread groups
- Four levels of thread grouping
- More scalable code and better software decomposition

BETTER COMPOSITION

Barrier synchronization hidden within functions

```
__device__ int sum(int *x, int n)
{
    ...
    __syncthreads();
    ...
    return total;
}

__global__ void parallel_kernel(float *x)
{
    ...
    // Entire thread block must call sum
    sum(x, n);
}
```

All threads in thread block must arrive at this barrier.

Hidden constraint on caller due to implementation of *sum*.

BETTER COMPOSITION

Explicit cooperative interfaces

```
__device__ int sum(thread_group g, int *x, int n)
{
    ...
    g.sync()
    ...
    return total;
}

__global__ void parallel_kernel(...)
{
    ...
    // Entire thread block must call sum
    sum(this_thread_block(), x, n);
    ...
}
```

Participating thread group provided by caller.

The need to synchronize in *sum* is visible in code.

FUTURE ROADMAP

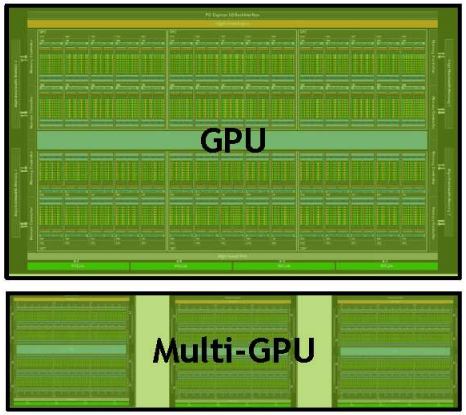
Partition by label or predicate, more complex scopes

(Volta specific)

```
thread_group cta = this_thread_block();
thread_group g = partition(cta, cta.thread_rank() & 1);
```


```
thread_group g = tiled_partition(cta, 64);
```

At all scopes!



FUTURE ROADMAP

Library of collectives (sort, reduce, etc.)

```
template <int BlockThreads>
__global__ int BlockReduce(float *d_in, ...)
{
    static_thread_block<BlockThreads> cta = this_thread_block();
    // Statically allocate shared reduction storage
    __shared__ reduce_storage<decltype(cta), float> group_reduce;

    // Compute the block-wide sum for thread-0
    float total = cooperative_groups::reduce(
                          cta, d_in[cta.rank()], group_reduce);
}
```

On a simpler note:

```
// Collective key-value sort, default allocator
cooperative_groups::sort(this_thread_block(), myValues, myKeys);
```

HONORABLE MENTION

The ones that didn't make it into their own slide

_CG_DEBUG : Define to enable various runtime safety checks. This helps debug incorrect API usage, incorrect synchronization, or similar issues (Automatically turned on with –G).

Tools help detect incorrect warp-synchronization with the racecheck tool.

Match is a new Volta instruction that is able to return who in your warp has the same 32 or 64 bit value

Developers **now have** a flexible model for synchronization and communication between groups of threads.

Shipping in CUDA 9.0

Provides safety, composability, and high performance

Flexibility to synchronize at various architecture and program defined scopes.

Deploy everywhere from Kepler to Volta

COOPERATIVE GROUPS

Kyrylo Perelygin, Yuan Lin

GTC 2017

Cooperative Groups: a flexible model for synchronization and communication within groups of threads.

At a glance

Scalable Cooperation among groups of threads

Flexible parallel decompositions

Composition across software boundaries

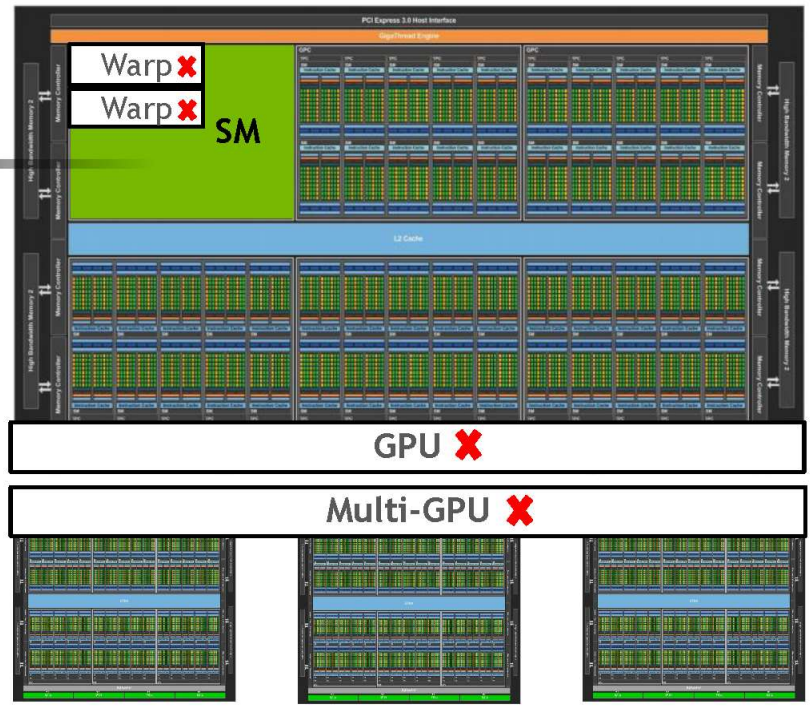
Deploy Everywhere

Benefits all applications

Examples include:
- Persistent RNNs
- Physics
- Search Algorithms
- Sorting

LEVELS OF COOPERATION: TODAY

`__syncthreads()`: block level synchronization barrier in CUDA



LEVELS OF COOPERATION: CUDA 9.0

For current coalesced set of threads:

```
auto g = coalesced_threads();
```

For warp-sized group of threads:

```
auto block = this_thread_block();
auto g = tiled_partition<32>(block)
```

For CUDA thread blocks:

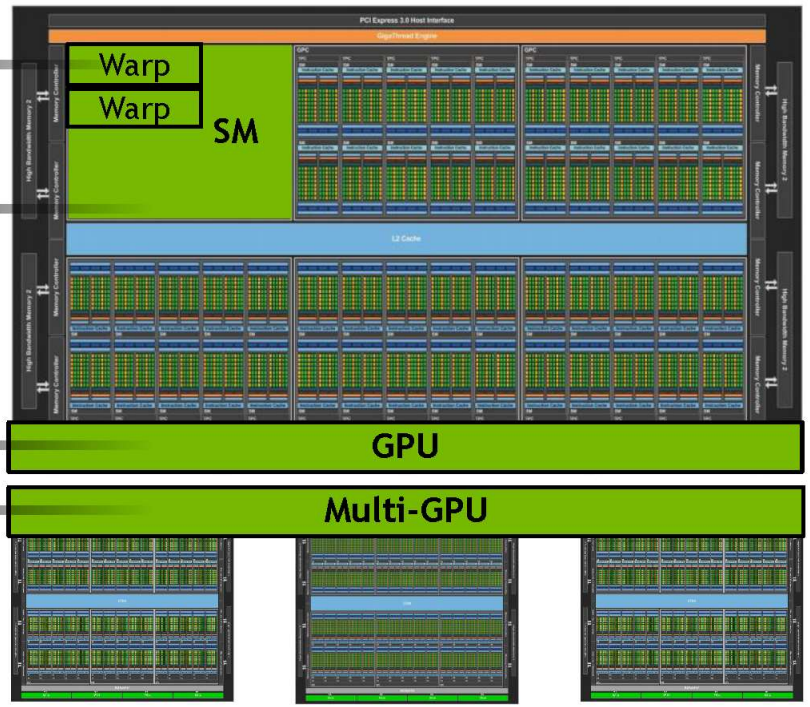
```
auto g = this_thread_block();
```

For device-spanning grid:

```
auto g = this_grid();
```

For multiple grids spanning GPUs:

```
auto g = this_multi_grid();
```



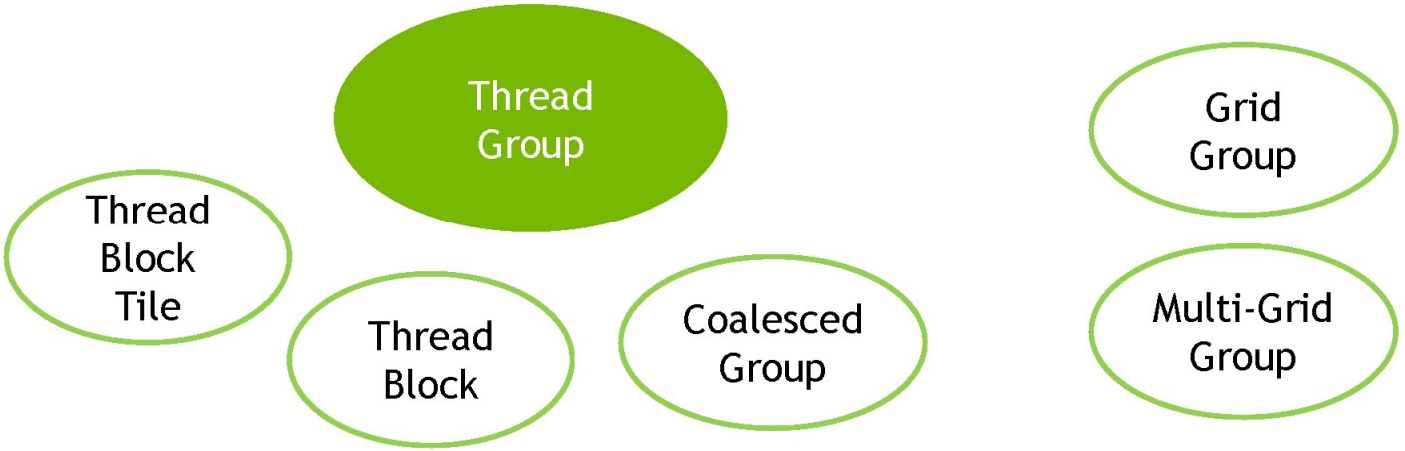
All Cooperative Groups functionality is within a **cooperative_groups::** namespace

THREAD GROUP

Base type, the implementation depends on its construction.

Unifies the various group types into one general, collective, thread group.

We need to extend the CUDA programming model with handles that can represent the groups of threads that can communicate/synchronize



THREAD BLOCK

Implicit group of all the threads in the launched thread block

Implements the same interface as thread_group:

```
void sync();                // Synchronize the threads in the group
unsigned size();            // Total number of threads in the group
unsigned thread_rank();     // Rank of the calling thread within [0, size]
bool is_valid();            // Whether the group violated any API constraints
```

And additional thread_block specific functions:

```
dim3 group_index();         // 3-dimensional block index within the grid
dim3 thread_index();        // 3-dimensional thread index within the block
```

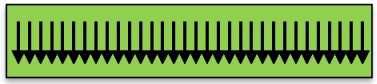
PROGRAM DEFINED DECOMPOSITION

All threads launched

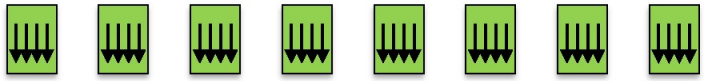
```
thread_block g = this_thread_block();
```

All threads in thread block

```
thread_group tile32 = tiled_partition(g, 32);
```



```
thread_group tile4 = tiled_partition(tile32, 4);
```



Restricted to powers of two, and <= 32 in initial release

GENERIC PARALLEL ALGORITHMS

Per-Block

```
g = this_thread_block();
reduce(g, ptr, myVal);
```

Per-Warp

```
g = tiled_partition(this_thread_block(), 32);
reduce(g, ptr, myVal);
```

```
__device__ int reduce(thread_group g, int *x, int val) {
  int lane = g.thread_rank();
  for (int i = g.size()/2; i > 0; i /= 2) {
    x[lane] = val;       g.sync();
    val += x[lane + i];  g.sync();
  }
  return val;
}
```

THREAD BLOCK TILE

A subset of threads of a thread block, divided into tiles in row-major order

```
thread_block_tile<32> tile32 = tiled_partition<32>(this_thread_block());
```

```
thread_block_tile<4> tile4 = tiled_partition<4>(this_thread_block());
```

Exposes additional functionality:

```
.shfl()
.shfl_down()
.shfl_up()
.shfl_xor()
```

```
.any()
.all()
.ballot()
.match_any()
.match_all()
```

Size known at compile time = fast!

STATIC TILE REDUCE

Per-Tile of 16 threads

```
g = tiled_partition<16>(this_thread_block());
tile_reduce(g, myVal);
```

```
template <unsigned size>
__device__ int tile_reduce(thread_block_tile<size> g, int val) {
  for (int i = g.size()/2; i > 0; i /= 2) {
    val += g.shfl_down(val, i);
  }
  return val;
}
```

GRID GROUP

A set of threads within the same grid, guaranteed to be resident on the device

New CUDA Launch API to opt-in:
`cudaLaunchCooperativeKernel(…)`

```
__global__ kernel() {
    grid_group grid = this_grid();
    // load data
    // loop - compute, share data
        grid.sync();
        // devices are now synced
}
```

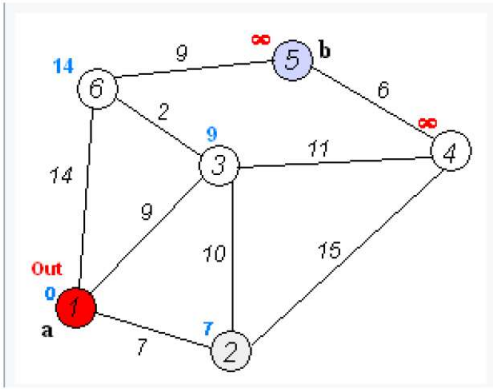
Device needs to support the `cooperativeLaunch` property.

```
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, kernel, numThreads, 0));
```

GRID GROUP

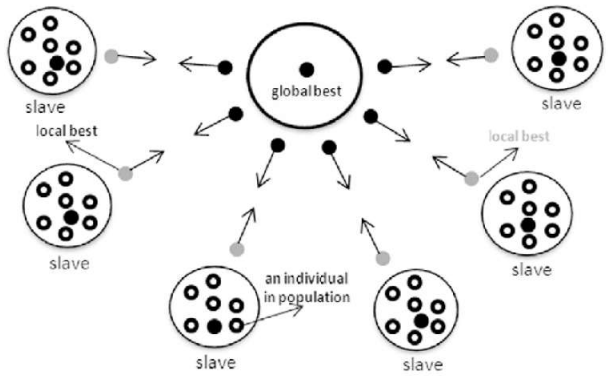
The goal: keep as much state as possible resident

Shortest Path / Search



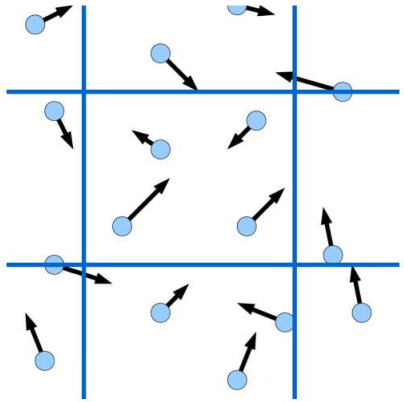
Weight array perfect for persistence
Iteration over vertices?
Fuse!

Genetic Algorithms / Master driven algorithms



Synchronization between a master block and slaves

Particle Simulations

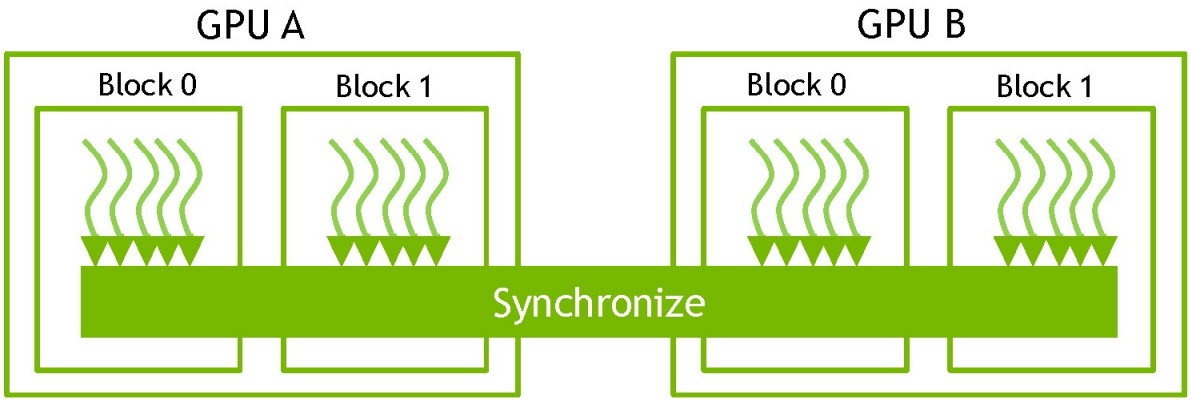


Synchronization between update and collision simulation

MULTI GRID GROUP

A set of threads guaranteed to be resident on the same system, on multiple devices

```
__global__ void kernel() {
    multi_grid_group multi_grid = this_multi_grid();
    // load data
    // loop - compute, share data
        multi_grid.sync();
        // devices are now synced, keep on computing
}
```



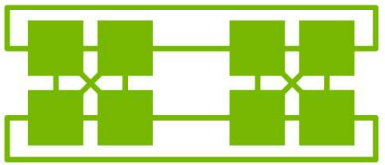
MULTI GRID GROUP

Launch on multiple devices at once

New CUDA Launch API to opt-in:
`cudaLaunchCooperativeKernelMultiDevice(…)`

Devices need to support the `cooperativeMultiDeviceLaunch` property.

```
struct cudaLaunchParams params[numDevices];
for (int i = 0; i < numDevices; i++) {
    params[i].func = (void *)kernel;
    params[i].gridDim = dim3(…); // Use occupancy calculator
    params[i].blockDim = dim3(…);
    params[i].sharedMem = …;
    params[i].stream = …; // Cannot use the NULL stream
    params[i].args = …;
}
cudaLaunchCooperativeKernelMultiDevice(params, numDevices);
```

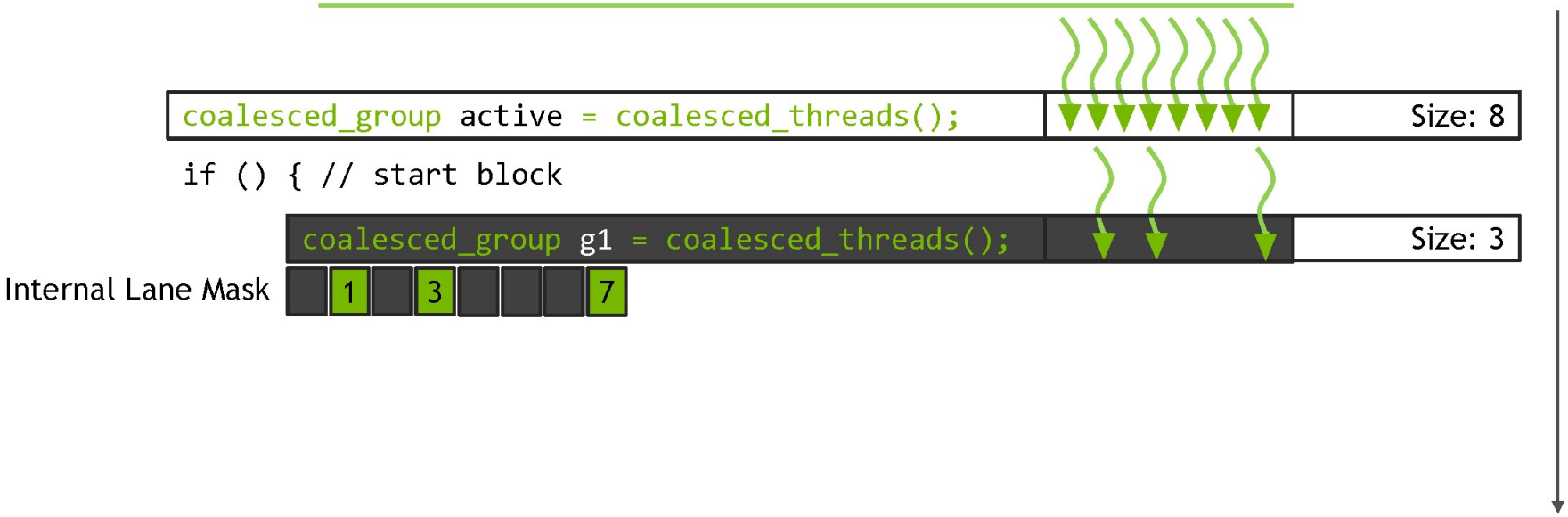


COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD

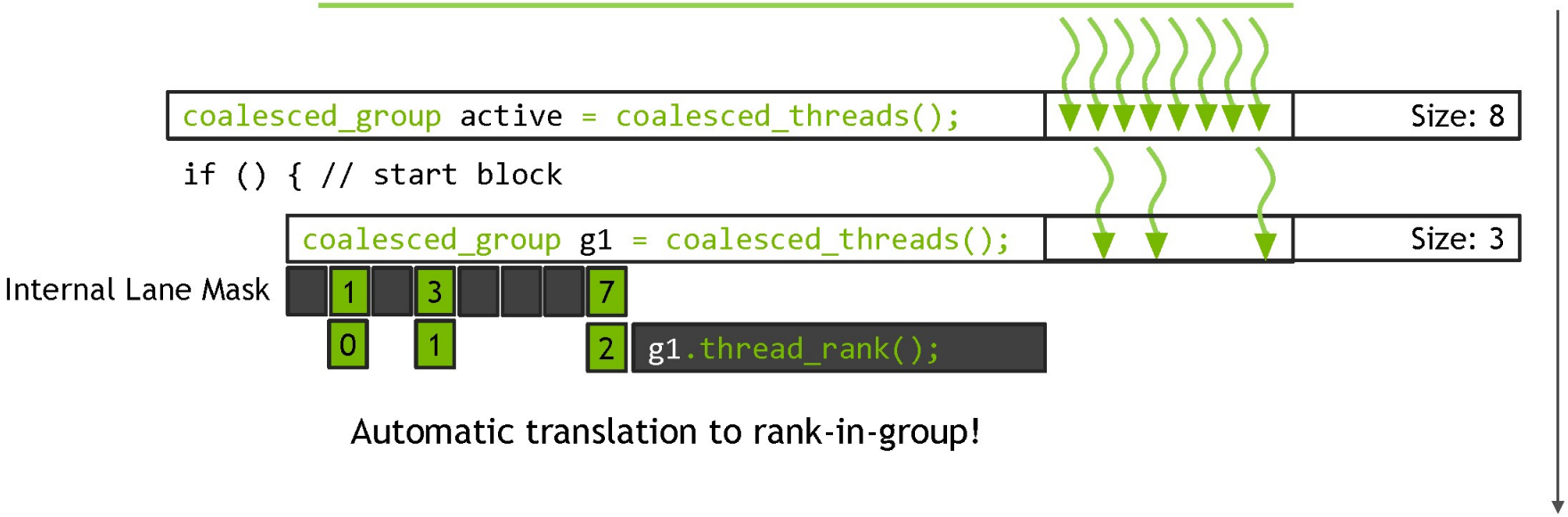
COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



COALESCED GROUP

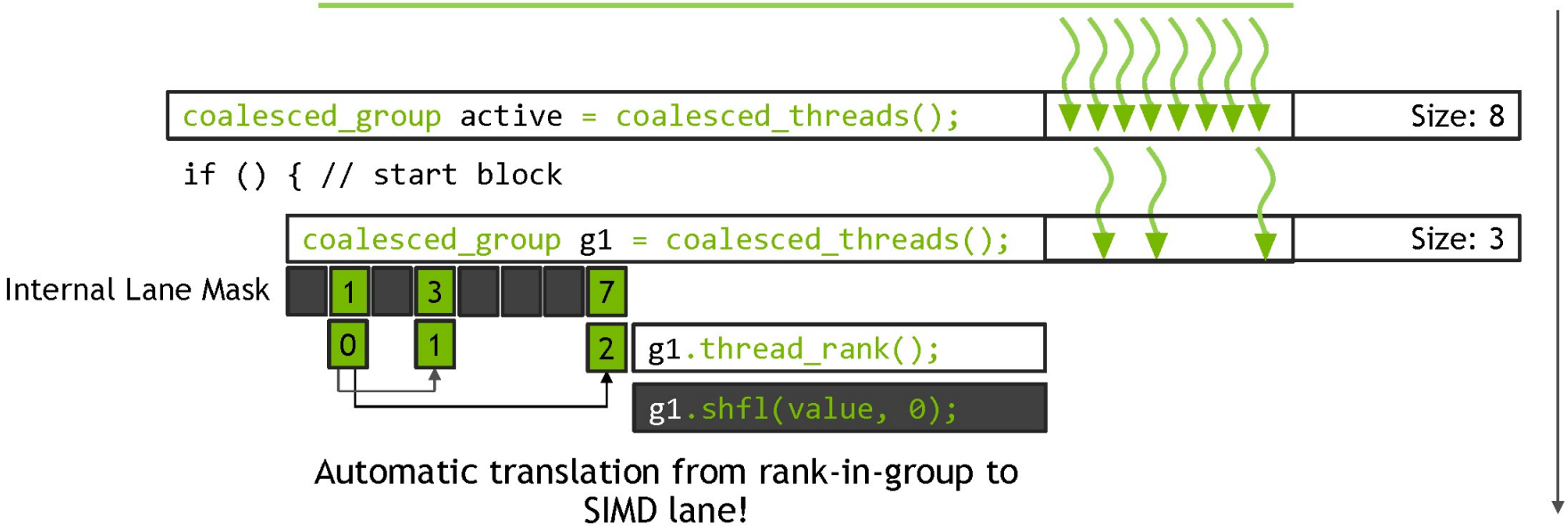
Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



Automatic translation to rank-in-group!

COALESCED GROUP

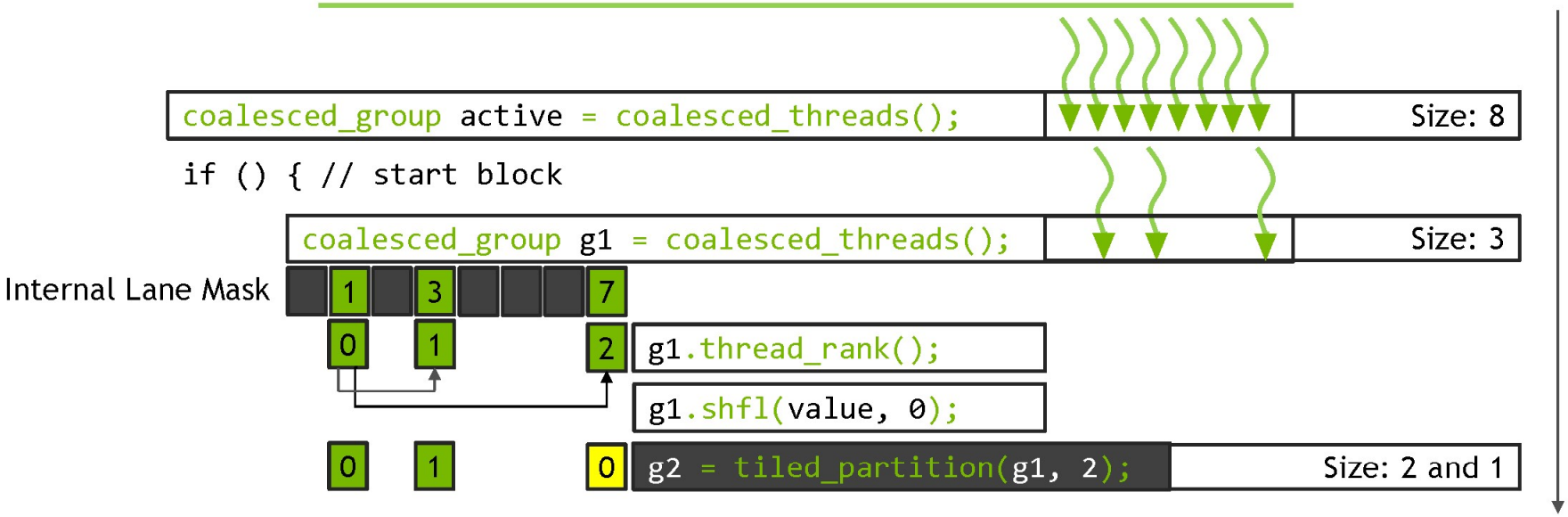
Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



Automatic translation from rank-in-group to SIMD lane!

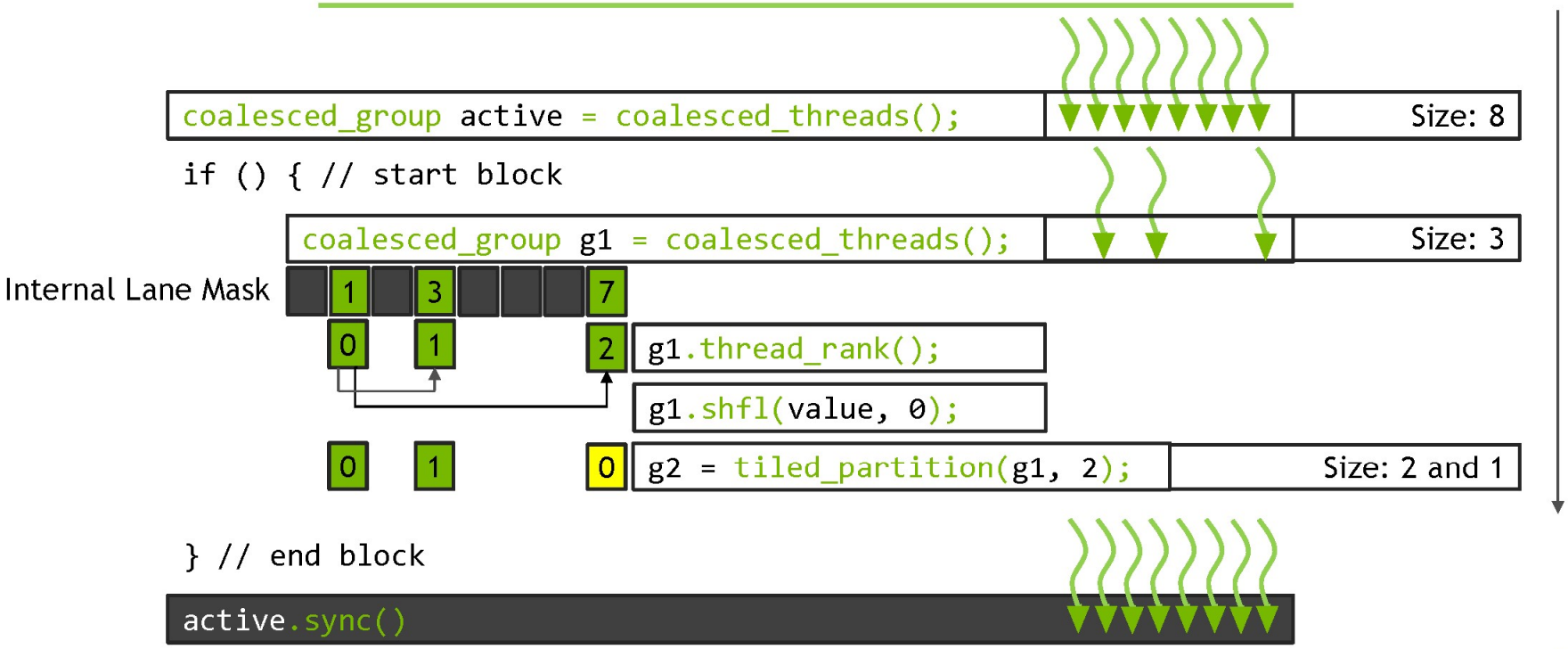
COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD

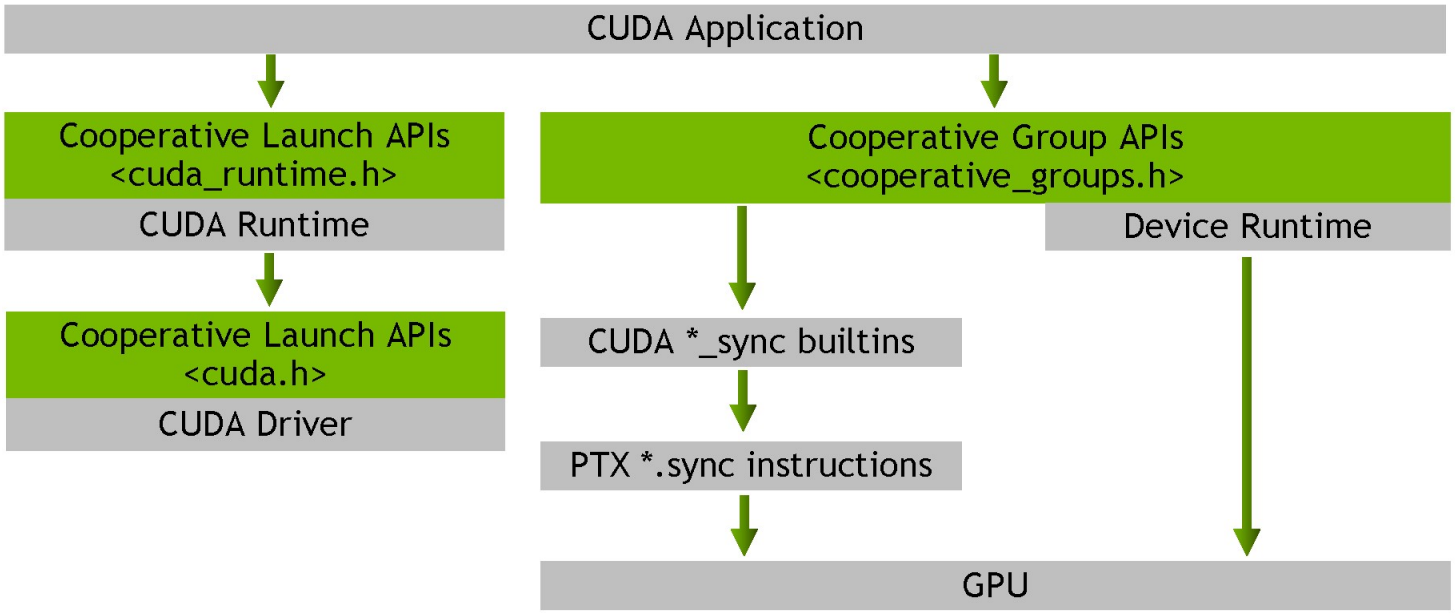


ATOMIC AGGREGATION

Opportunistic cooperation within a warp

```
inline __device__ int atomicAggInc(int *p)
{
    coalesced_group g = coalesced_threads();
    int prev;
    if (g.thread_rank() == 0) {
        prev = atomicAdd(p, g.size());
    }
    prev = g.thread_rank() + g.shfl(prev, 0);
    return prev;
}
```

ARCHITECTURE



Programming Tensor Cores

NVIDIA Volta SM

Multiprocessor: SM

- 64 FP32 + INT32 cores
- 32 FP64 cores
- 8 tensor cores (FP16/FP32 mixed-precision)

4 partitions inside SM

- 16 FP32 + INT32 cores each
- 8 FP64 cores each
- 8 LD/ST units each
- 2 tensor cores each
- Each has: warp scheduler, dispatch unit, register file

NVIDIA Turing SM

Multiprocessor: SM

- 64 FP32 + INT32 cores
- 2 (!) FP64 cores
- 8 Turing tensor cores (FP16/32, INT4/8 mixed-precision)
- 1 RT (ray tracing) core

4 partitions inside SM

- 16 FP32 + INT32 cores each
- 4 LD/ST units each
- 2 Turing tensor cores each
- Each has: warp scheduler, dispatch unit, 16K register file

NVIDIA GA100 SM

Multiprocessor: SM

- 64 FP32 + 64 INT32 cores
- 32 FP64 cores
- 4 3rd gen tensor cores
- 1 2nd gen RT (ray tracing) core

4 partitions inside SM

- 16 FP32 + 16 INT32 cores
- 8 FP64 cores
- 8 LD/ST units each
- 1 3rd gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file

NVIDIA GA102 SM

Multiprocessor: SM

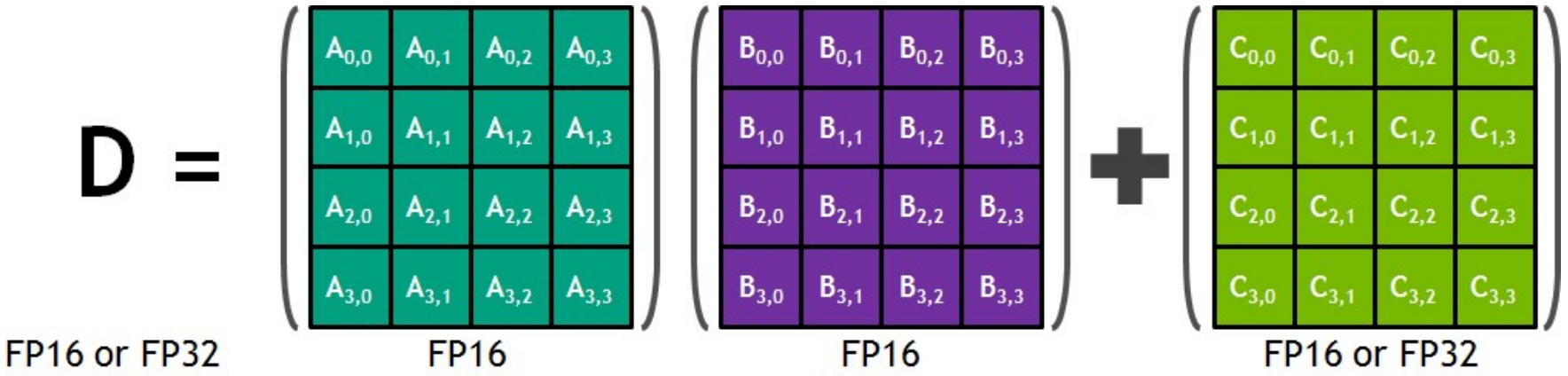
- 128 (64+64) FP32 + 64 INT32 cores
- 2 (!) FP64 cores
- 4 3rd gen tensor cores
- 1 2nd gen RT (ray tracing) core

4 partitions inside SM

- 16+16 FP32 + 16 INT32 cores
- 4 LD/ST units each
- 1 3rd gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file

Tensor Cores

Mixed-precision, fast matrix-matrix multiply and accumulate (mma)



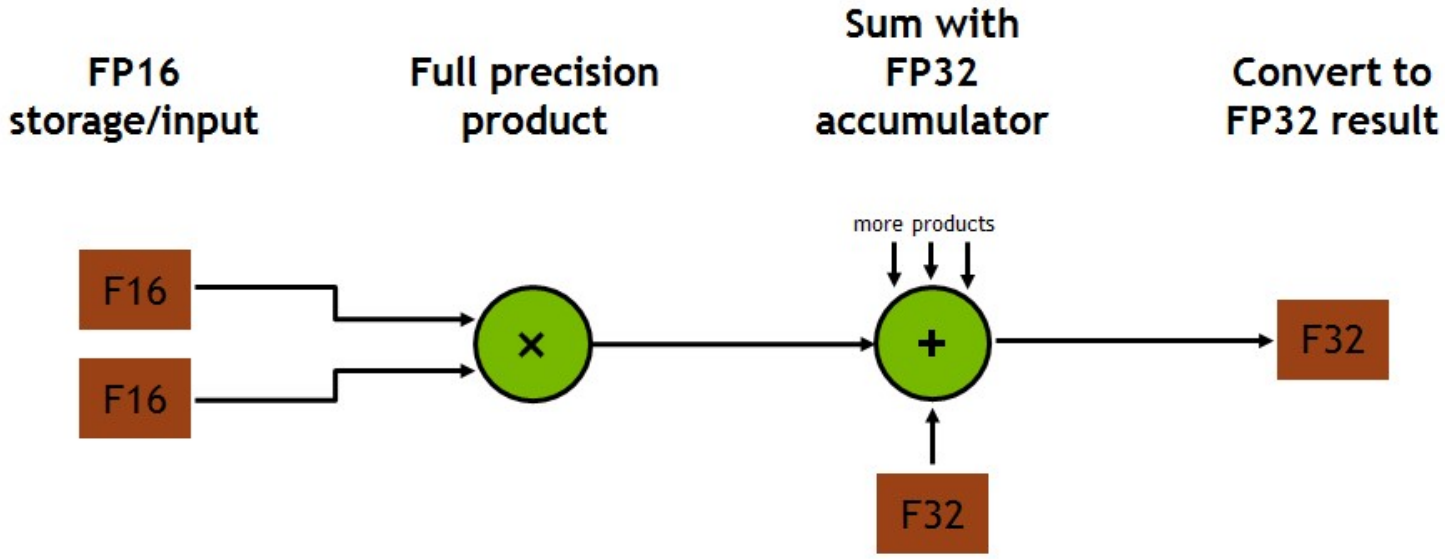
From this, build larger sizes, higher dimensionalities, ...

API currently only allows using larger sizes (16x16, ...) in warps (wmma)

Tensor Cores

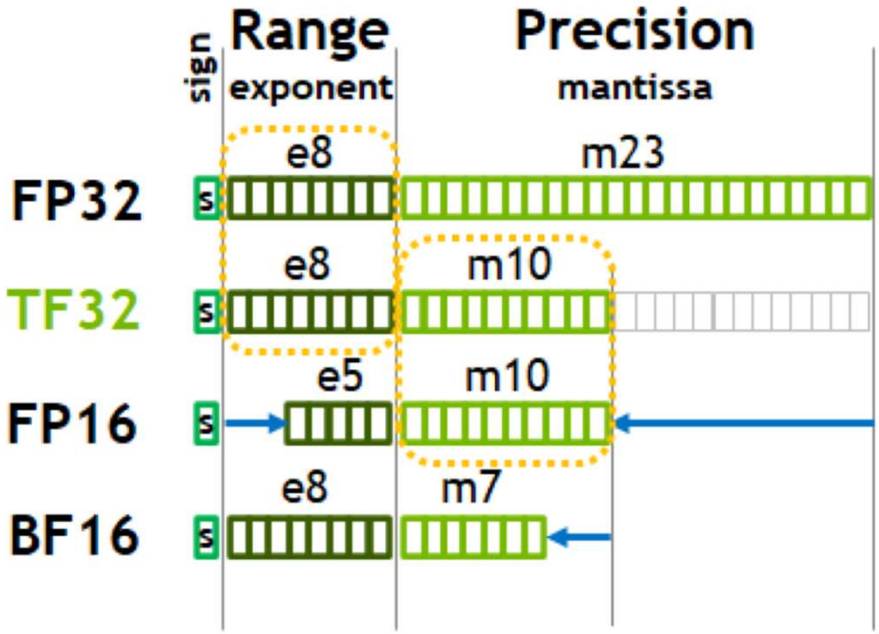
Fused matrix multiply and accumulate

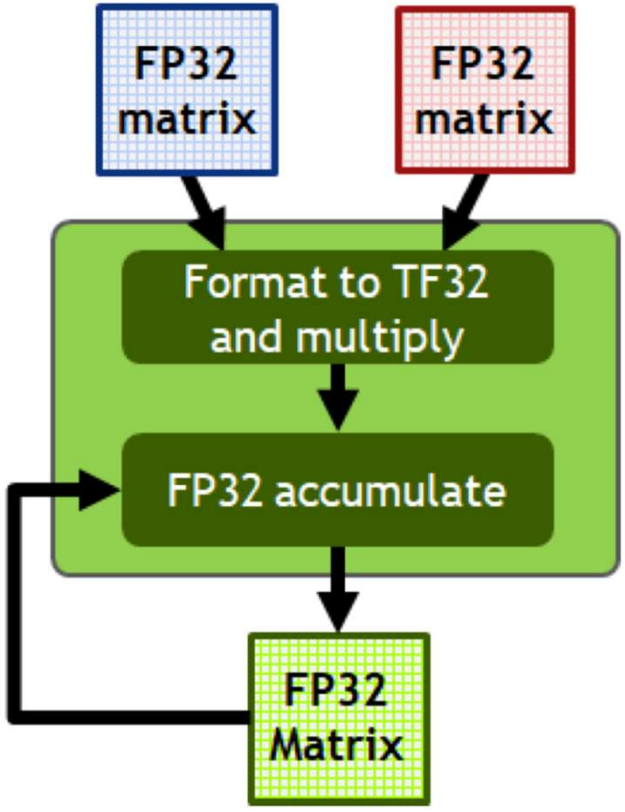
- Input matrices can be (at most) half-precision (FP16); (Ampere has more!)
- Accumulate can be FP16 or FP32; (Ampere has more!)



Tensor Cores: Many Mixed Precision Options

New in Ampere: TF32, BF16, FP64





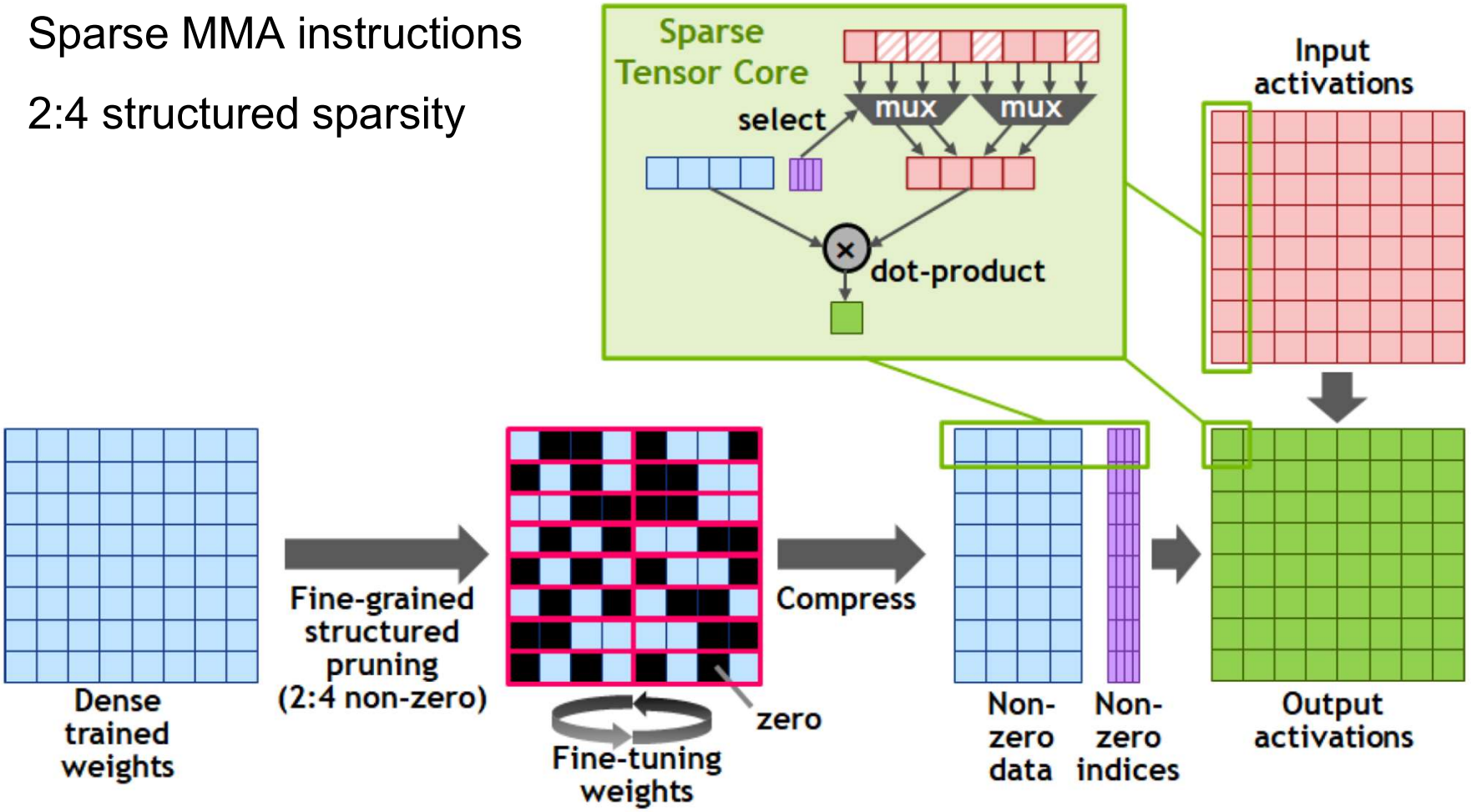
plus FP64 (new in Ampere; GA100 only)

plus INT4/INT8/binary data types (experimental; introduced in Turing)

Tensor Cores: Sparsity Support

Sparse MMA instructions

2:4 structured sparsity



Tensor Core APIs

Low-level options

- CUDA C WMMA (warp-level matrix multiply and accumulate)
- PTX wmma and mma instructions
- SASS hmma instructions (not documented)

High-level options

- NVIDIA CUTLASS (template abstractions for hi-perf matrix-multiplies)
- NVIDIA cuBLAS
- NVIDIA cuDNN
- Integration into TensorFlow, ...

CUDA C Warp Matrix Functions (WMMA)

Warp Level Matrix Multiply Accumulate (WMMA)

CUDA C Programming Guide (11.1), Appendix B.23

namespace **`nvcuda::wmma`** (and **`nvcuda::wmma::experimental`**)

```
template<typename Use, int m, int n, int k, typename T, typename Layout=void>
 class fragment;

void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm);
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t
 layout);
void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t
 layout);
void fill_fragment(fragment<...> &a, const T& v);
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...>
 &b, const fragment<...> &c, bool satf=false);
```

Concept of a matrix *fragment* (section of a matrix split across threads in a warp)

Dimensions **`m,n,k`**: **`m x k matrix_a`**; **`k x n matrix_b`**; **`m x n accumulator`**

CUDA C Warp Matrix Functions (WMMA)

Data types (`T`)

Volta, Turing, and Ampere:

wmma API splits this into fragments

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__half	__half	float	16x16x16
__half	__half	float	32x8x16
__half	__half	float	8x32x16
__half	__half	__half	16x16x16
__half	__half	__half	32x8x16
__half	__half	__half	8x32x16
unsigned char	unsigned char	int	16x16x16
unsigned char	unsigned char	int	32x8x16
unsigned char	unsigned char	int	8x32x16
signed char	signed char	int	16x16x16
signed char	signed char	int	32x8x16
signed char	signed char	int	8x32x16

CUDA C Warp Matrix Functions (WMMA)

Data types (`T`)

wmma API splits this into fragments

Ampere only:

Alternate Floating Point support:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__nv_bfloat16	__nv_bfloat16	float	16x16x16
__nv_bfloat16	__nv_bfloat16	float	32x8x16
__nv_bfloat16	__nv_bfloat16	float	8x32x16
precision::tf32	precision::tf32	float	16x16x8

Double Precision Support:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
double	double	double	8x8x4

Turing and Ampere:

Experimental support for sub-byte operations:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
precision::u4	precision::u4	int	8x8x32
precision::s4	precision::s4	int	8x8x32
precision::b1	precision::b1	int	8x8x128

CUDA C Warp Matrix Functions (WMMA)

Warp Level Matrix Multiply Accumulate (WMMA)

CUDA C Programming Guide (11.1), Appendix B.23

```
#include <mma.h>

using namespace nvcuda;

__global__ void wmma_ker(half *a, half *b, float *c) {
   // Declare the fragments
   wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
   wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
   wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

   // Initialize the output to zero
   wmma::fill_fragment(c_frag, 0.0f);

   // Load the inputs
   wmma::load_matrix_sync(a_frag, a, 16);
   wmma::load_matrix_sync(b_frag, b, 16);

   // Perform the matrix multiplication
   wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

   // Store the output
   wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

PTX Warp Matrix Functions (WMMA, MMA)

Warp Level Matrix Multiply Accumulate (WMMA, MMA)

PTX ISA (6.5), Section 9.7.13

Instruction	Shape	Data-type	PTX ISA version
`wmma`	`.m16n16k16`	integer and floating-point	PTX ISA version 6.0 and later (integer support added in PTX ISA version 6.3)
`wmma`	`.m8n32k16` and `.m32n8k16`	integer and floating-point	PTX ISA version 6.1 and later (integer support added in PTX ISA version 6.3)
`wmma`	`.m8n8k32`	sub-byte integer	PTX ISA 6.3 (preview feature)
`wmma`	`.m8n8k128`	single-bit	PTX ISA 6.3 (preview feature)
`mma`	`.m8n8k4`	floating-point	PTX ISA 6.4
`mma`	`.m16n8k8`	floating-point	PTX ISA 6.5
`mma`	`.m8n8k16`	integer	PTX ISA 6.5
`mma`	`.m8n8k32`	sub-byte integer	PTX ISA 6.5

PTX Warp Matrix Functions (WMMA, MMA)

Warp Level Matrix Multiply Accumulate (WMMA, MMA)

PTX ISA (6.5), Section 9.7.13

Data-type	Multiplicands (A or B)	Accumulators (C or D)
Integer	both `.u8` or both `.s8`	`.s32`
Floating Point	`.f16`	`.f16`, `.f32`
Sub-byte integer	both `.u4` or both `.s4`	`.s32`
Single-bit integer	`.b1`	`.s32`

PTX Warp Matrix Functions (WMMA, MMA)

Warp Level Matrix Multiply Accumulate (WMMA, MMA)

PTX ISA (7.1), Section 9.7.13

Instruction	Sparsity	Multiplicand Data-type	Shape	PTX ISA version
`wmma`	Dense	Floating-point - `.f16`	`.m16n16k16`, `.m8n32k16`, and `.m32n8k16`	PTX ISA version 6.0
`wmma`	Dense	Alternate floating-point format - `.bf16`	`.m16n16k16`, `.m8n32k16`, and `.m32n8k16`	PTX ISA version 7.0
`wmma`	Dense	Alternate floating-point format - `.tf32`	`.m16n16k8`	PTX ISA version 7.0
`wmma`	Dense	Integer - `.u8`/`.s8`	`.m16n16k16`, `.m8n32k16`, and `.m32n8k16`	PTX ISA version 6.3
`wmma`	Dense	Sub-byte integer - `.u4`/`.s4`	`.m8n8k32`	PTX ISA version 6.3 (preview feature)
`wmma`	Dense	Single-bit - `.b1`	`.m8n8k128`	PTX ISA version 6.3 (preview feature)

PTX Warp Matrix Functions (WMMA, MMA)

Warp Level Matrix Multiply Accumulate (WMMA, MMA)

PTX ISA (7.1), Section 9.7.13

Instruction	Sparsity	Multiplicand Data-type	Shape	PTX ISA version
mma	Dense	Floating-point - .f64	.m8n8k4	PTX ISA version 7.0
mma	Dense	Floating-point - .f16	.m8n8k4	PTX ISA version 6.4
			.m16n8k8	PTX ISA version 6.5
			.m16n8k16	PTX ISA version 7.0
mma	Dense	Alternate floating-point format - .bf16	.m16n8k8 and .m16n8k16	PTX ISA version 7.0
mma	Dense	Alternate floating-point format - .tf32	.m16n8k4 and .m16n8k8	PTX ISA version 7.0
mma	Dense	Integer - .u8/.s8	.m8n8k16	PTX ISA version 6.5
			.m16n8k16 and .m16n8k32	PTX ISA version 7.0
mma	Dense	Sub-byte integer - .u4/.s4	.m8n8k32	PTX ISA version 6.5
			.m16n8k32 and .m16n8k64	PTX ISA version 7.0
mma	Dense	Single-bit - .b1	.m8n8k128, .m16n8k128, and .m16n8k256	PTX ISA version 7.0

PTX Warp Matrix Functions (WMMA, MMA)

Warp Level Matrix Multiply Accumulate (WMMA, MMA)

PTX ISA (7.1), Section 9.7.13

Instruction	Sparsity	Multiplicand Data-type	Shape	PTX ISA version
`mma`	Sparse	Floating-point - `.f16`	`.m16n8k16` and `.m16n8k32`	PTX ISA version 7.1
`mma`	Sparse	Alternate floating-point format - `.bf16`	`.m16n8k16` and `.m16n8k32`	PTX ISA version 7.1
`mma`	Sparse	Alternate floating-point format - `.tf32`	`.m16n8k8` and `.m16n8k16`	PTX ISA version 7.1
`mma`	Sparse	Integer - `.u8`/`.s8`	`.m16n8k32` and `.m16n8k64`	PTX ISA version 7.1
`mma`	Sparse	Sub-byte integer - `.u4`/`.s4`	`.m16n8k64` and `.m16n8k128`	PTX ISA version 7.1

PTX Warp Matrix Functions (WMMA, MMA)

Warp Level Matrix Multiply Accumulate (WMMA, MMA) load

```
wmma.load.a.sync.aligned.layout.shape{.ss}.atype r, [p] {, stride};
wmma.load.b.sync.aligned.layout.shape{.ss}.btype r, [p] {, stride};
wmma.load.c.sync.aligned.layout.shape{.ss}.ctype r, [p] {, stride};

.layout = {.row, .col};
.shape  = {.m16n16k16, .m8n32k16, .m32n8k16};
.ss     = {.global, .shared};
.atype  = {.f16, .s8, .u8};
.btype  = {.f16, .s8, .u8};
.ctype  = {.f16, .f32, .s32};

// sub-byte loads
wmma.load.a.sync.aligned.row.shape{.ss}.atype r, [p] {, stride}
wmma.load.b.sync.aligned.col.shape{.ss}.btype r, [p] {, stride}
wmma.load.c.sync.aligned.layout.shape{.ss}.ctype r, [p] {, stride}
.layout = {.row, .col};
.shape  = {.m8n8k32};
.ss     = {.global, .shared};
.atype  = {.s4, .u4};
.btype  = {.s4, .u4};
.ctype  = {.s32};

// single-bit loads
wmma.load.a.sync.aligned.row.shape{.ss}.atype r, [p] {, stride}
wmma.load.b.sync.aligned.col.shape{.ss}.btype r, [p] {, stride}
wmma.load.c.sync.aligned.layout.shape{.ss}.ctype r, [p] {, stride}
.layout = {.row, .col};
.shape  = {.m8n8k128};
.ss     = {.global, .shared};
.atype  = {.b1};
.btype  = {.b1};
.ctype  = {.s32};
```

PTX Warp Matrix Functions (WMMA, MMA)

Warp Level Matrix Multiply Accumulate (WMMA, MMA) load

```
// Load elements from f16 row-major matrix B
.reg .b32 x<8>;

wmma.load.b.sync.aligned.m16n16k16.row.f16 {x0,x1,x2,x3,x4,x5,x,x7}, [ptr];
// Now use {x0, ..., x7} for the actual wmma.mma

// Load elements from f32 column-major matrix C and scale the values:
.reg .b32 x<8>;

wmma.load.c.sync.aligned.m16n16k16.col.f32
                {x0,x1,x2,x3,x4,x5,x6,x7}, [ptr];

mul.f32 x0, x0, 0.1;
// repeat for all registers x<8>;
...
mul.f32 x7, x7, 0.1;
// Now use {x0, ..., x7} for the actual wmma.mma

// Load elements from integer matrix A:
.reg .b32 x<4>
// destination registers x<4> contain four packed .u8 values each
wmma.load.a.sync.aligned.m32n8k16.row.u8 {x0,x1,x2,x3}, [ptr];

// Load elements from sub-byte integer matrix A:
.reg .b32 x0;
// destination register x0 contains eight packed .s4 values
wmma.load.a.sync.aligned.m8n8k32.row.s4 {x0}, [ptr];
```

PTX Warp Matrix Functions (WMMA, MMA)

Warp Level Matrix Multiply Accumulate (WMMA, MMA) mma

```
wmma.mma.sync.aligned.alayout.blayout.shape.dtype.ctype d, a, b, c;

wmma.mma.sync.aligned.alayout.blayout.shape.s32.atype.btype.s32{.satfinite} d,
 a, b, c;

.alayout = {.row, .col};
.blayout = {.row, .col};
.shape  =  {.m16n16k16, .m8n32k16, .m32n8k16};
.dtype   = {.f16, .f32};
.atype   = {.s8, .u8};
.btype   = {.s8, .u8};
.ctype   = {.f16, .f32};

wmma.mma.sync.aligned.row.col.shape.s32.atype.btype.s32{.satfinite} d, a, b, c;
.shape  = {.m8n8k32};
.atype  = {.s4, .u4};
.btype  = {.s4, .u4};

wmma.mma.xor.popc.sync.aligned.row.col.shape.s32.atype.btype.s32 d, a, b, c;
.shape  = {.m8n8k128};
.atype  = {.b1};
.btype  = {.b1};
```

PTX Warp Matrix Functions (WMMA, MMA)

WMMA mma

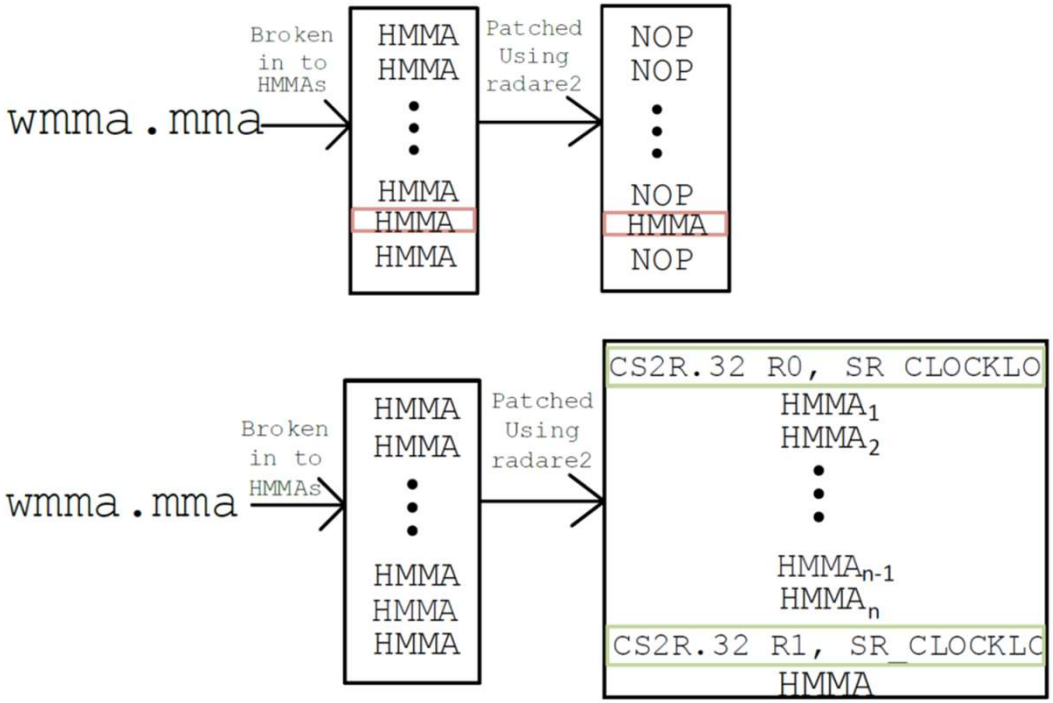
```
.global .align 32 .f16 A[256], B[256];
.global .align 32 .f32 C[256], D[256];
.reg .b32 a<8> b<8> c<8> d<8>;

wmma.load.a.sync.aligned.m16n16k16.global.row.f16
        {a0, a1, a2, a3, a4, a5, a6, a7}, [A];
wmma.load.b.sync.aligned.m16n16k16.global.col.f16
        {b0, b1, b2, b3, b4, b5, b6, b7}, [B];

wmma.load.c.sync.aligned.m16n16k16.global.row.f32
        {c0, c1, c2, c3, c4, c5, c6, c7}, [C];

wmma.mma.sync.aligned.m16n16k16.row.col.f32.f32
        {d0, d1, d2, d3, d4, d5, d6, d7},
        {a0, a1, a2, a3, a4, a5, a6, a7},
        {b0, b1, b2, b3, b4, b5, b6, b7},
        {c0, c1, c2, c3, c4, c5, c6, c7};

wmma.store.d.sync.aligned.m16n16k16.global.col.f32
        [D], {d0, d1, d2, d3, d4, d5, d6, d7};

// Compute an integer WMMA:
.reg .b32  a, b<4>;
.reg .b32 c<8>, d<8>;
wmma.mma.sync.aligned.m8n32k16.row.col.s32.s8.s8.s32
        {d0, d1, d2, d3, d4, d5, d6, d7},
        {a}, {b0, b1, b2,  b3},
        {c0, c1, c2, c3, c4, c5, c6, c7};

// Compute sub-byte WMMA:
.reg .b32 a, b, c<2> d<2>
wmma.mma.sync.aligned.m8n8k32.row.col.s32.s4.s4.s32
        {d0, d1}, {a}, {b}, {c0, c1};

// Compute single-bit type WMMA:
.reg .b32 a, b, c<2> d<2>
wmma.mma.xor.popc.sync.aligned.m8n8k128.row.col.s32.b1.b1.s32
        {d0, d1}, {a}, {b}, {c0, c1};
```

PTX WMMA to SASS

Raihan et al., 2019

Get SASS code from cuobjdump disassembly

Micro-benchmarking



PTX WMMA to SASS

Raihan et al., 2019

Get SASS code from cuobjdump disassembly

		Cumulative Clock Cycles
SET1	HMMA.884.F32.F32.STEP0 R8, R24.reuse.COL, R22.reuse.ROW, R8;	10
	HMMA.884.F32.F32.STEP1 R10, R24.reuse.COL, R22.reuse.ROW, R10;	12
	HMMA.884.F32.F32.STEP2 R4, R24.reuse.COL, R22.reuse.ROW, R4;	14
	HMMA.884.F32.F32.STEP3 R6, R24.COL, R22.ROW, R6;	18
SET2	HMMA.884.F32.F32.STEP0 R8, R20.reuse.COL, R18.reuse.ROW, R8;	20
	HMMA.884.F32.F32.STEP1 R10, R20.reuse.COL, R18.reuse.ROW, R10;	22
	HMMA.884.F32.F32.STEP2 R4, R20.reuse.COL, R18.reuse.ROW, R4;	24
	HMMA.884.F32.F32.STEP3 R6, R20.COL, R18.ROW, R6;	28
SET3	HMMA.884.F32.F32.STEP0 R8, R14.reuse.COL, R12.reuse.ROW, R8;	30
	HMMA.884.F32.F32.STEP1 R10, R14.reuse.COL, R12.reuse.ROW, R10;	32
	HMMA.884.F32.F32.STEP2 R4, R14.reuse.COL, R12.reuse.ROW, R4;	34
	HMMA.884.F32.F32.STEP3 R6, R14.COL, R12.ROW, R6;	38
SET4	HMMA.884.F32.F32.STEP0 R8, R16.reuse.COL, R2.reuse.ROW, R8;	40
	HMMA.884.F32.F32.STEP1 R10, R16.reuse.COL, R2.reuse.ROW, R10;	42
	HMMA.884.F32.F32.STEP2 R4, R16.reuse.COL, R2.reuse.ROW, R4;	44
	HMMA.884.F32.F32.STEP3 R6, R16.COL, R2.ROW, R6;	54

(a) Disassembled SASS instructions for Mixed precision mode

PTX WMMA to SASS

Raihan et al., 2019

Get SASS code from cuobjdump disassembly

		Cumulative Clock Cycles
SET1	HMMA.884.F16.F16.STEP0 R4, R22.reuse.T, R12.reuse.T, R4;	12
	HMMA.884.F16.F16.STEP1 R6, R22.T, R12.T, R6;	21
SET2	HMMA.884.F16.F16.STEP0 R4, R16.reuse.T, R14.reuse.T, R4;	25
	HMMA.884.F16.F16.STEP1 R6, R16.T, R14.T, R6;	34
SET3	HMMA.884.F16.F16.STEP0 R4, R18.reuse.T, R8.reuse.T, R4;	38
	HMMA.884.F16.F16.STEP1 R6, R18.T, R8.T, R6;	47
SET4	HMMA.884.F16.F16.STEP0 R4, R2.reuse.T, R10.reuse.T, R4;	51
	HMMA.884.F16.F16.STEP1 R6, R2.T, R10.T, R6;	64

(b) Disassembled SASS instructions for FP16 mode

PTX WMMA to SASS

Raihan et al., 2019, reverse-engineered matrix fragment assignment

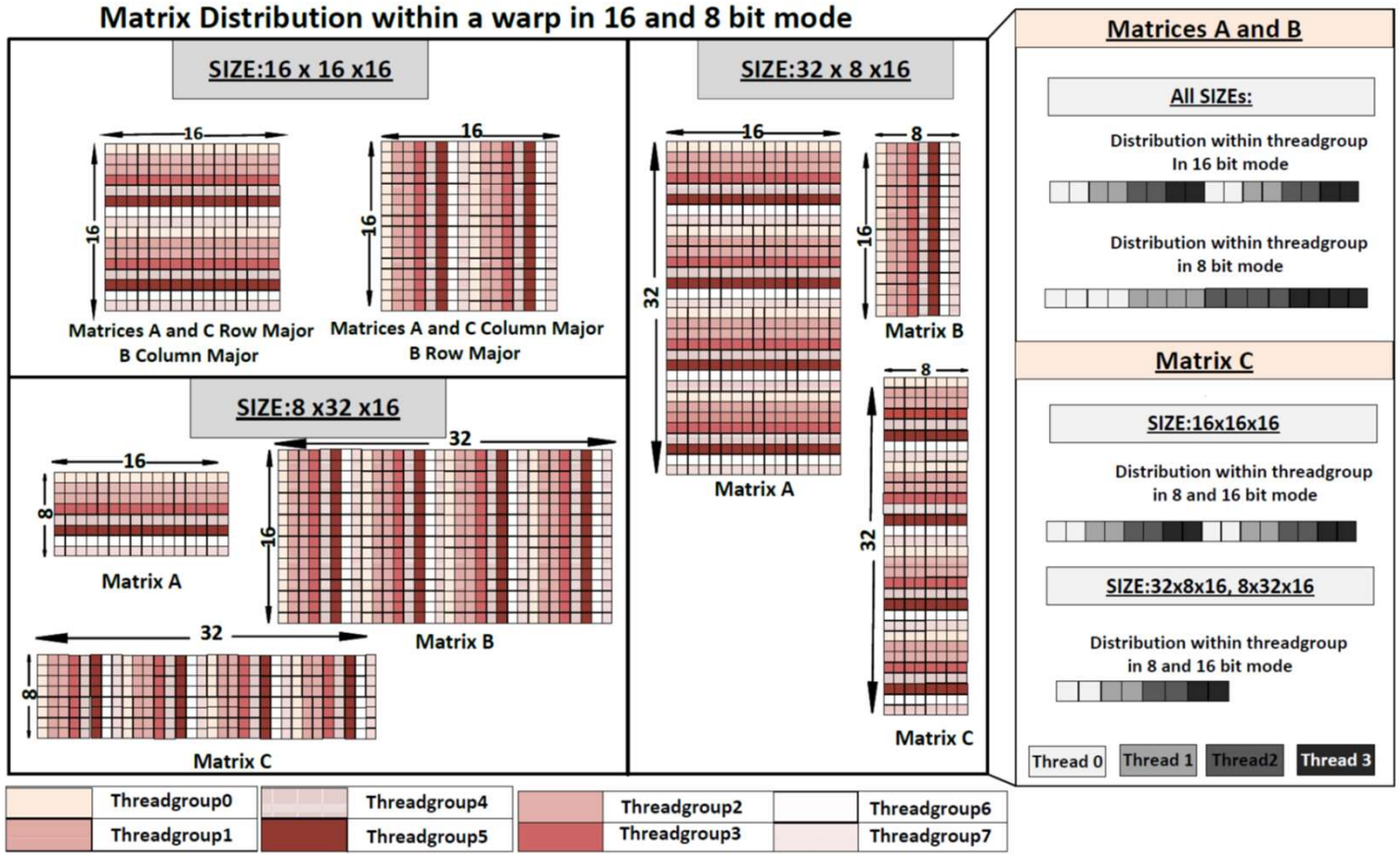


Figure 8: Distribution of operand matrix elements to threads for tensor cores in the RTX 2080 (Turing).

PTX WMMA to SASS

Raihan et al., 2019, reverse-engineered Tensor core microarchitecture

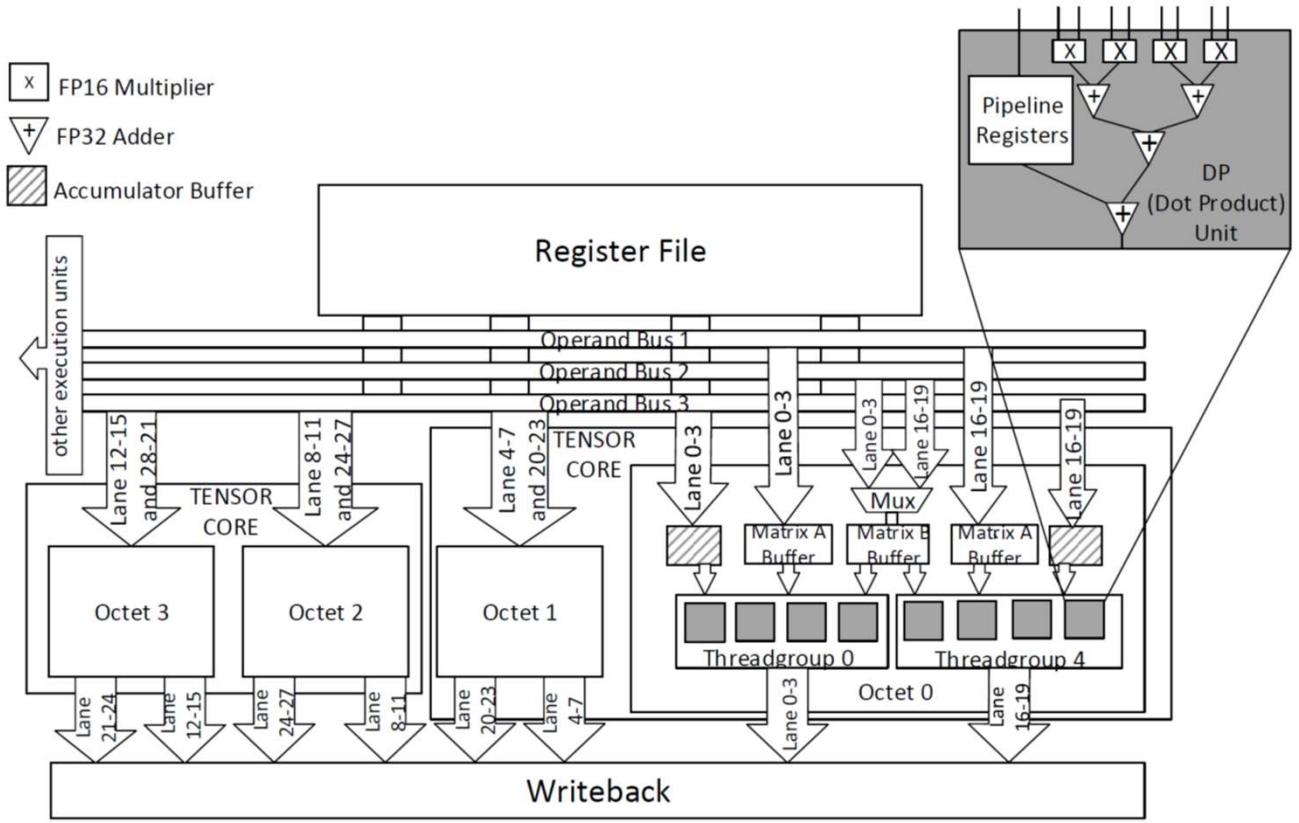


Figure 13: Proposed Tensor Core Microarchitecture

Thank you.

- NVIDIA
- Md Aamir Raihan, Negar Goli, Tor Aamodt