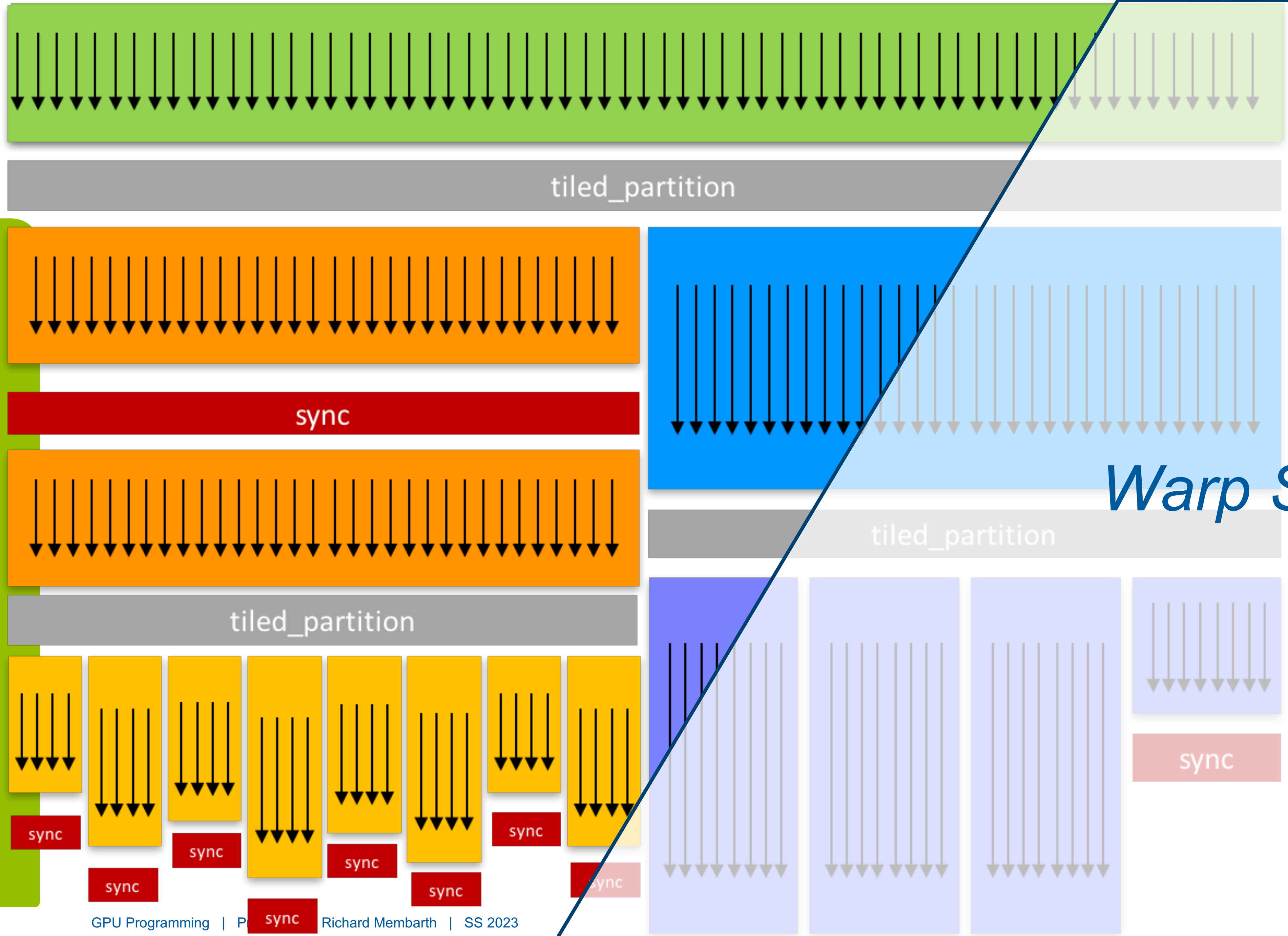


Warp Scheduling

17.05.23

Goals

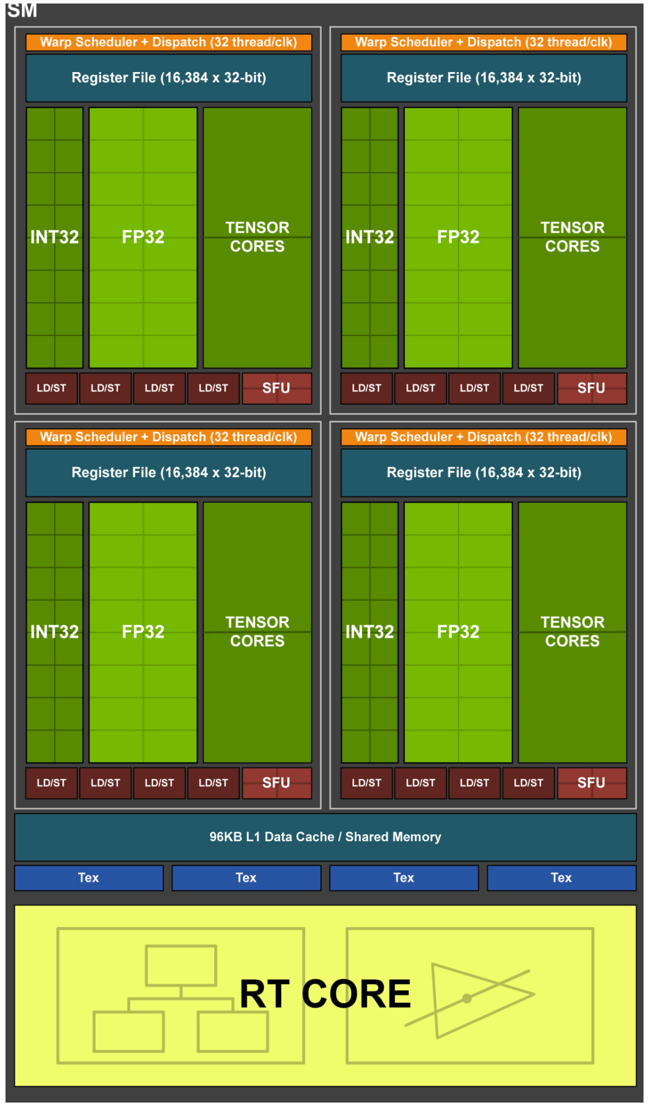
- What is **Independent Thread Scheduling**?
- Utilizing **warp-level primitives** effectively
- **Tensor Cores**

Independent Warp Scheduling

Warp Scheduling

- Each warp scheduler can schedule to a number of
  - ALUs
    - FP32, INT32, FP64
  - Tensor cores
  - Special function units
    - sqrt, exp, …
  - Load/store units
- State of multiple warps is kept on chip
  - Latency hiding
- Every clock cycle
  - Warp scheduler elects eligible warp
  - Schedules instruction from that warp

TU 102 SM

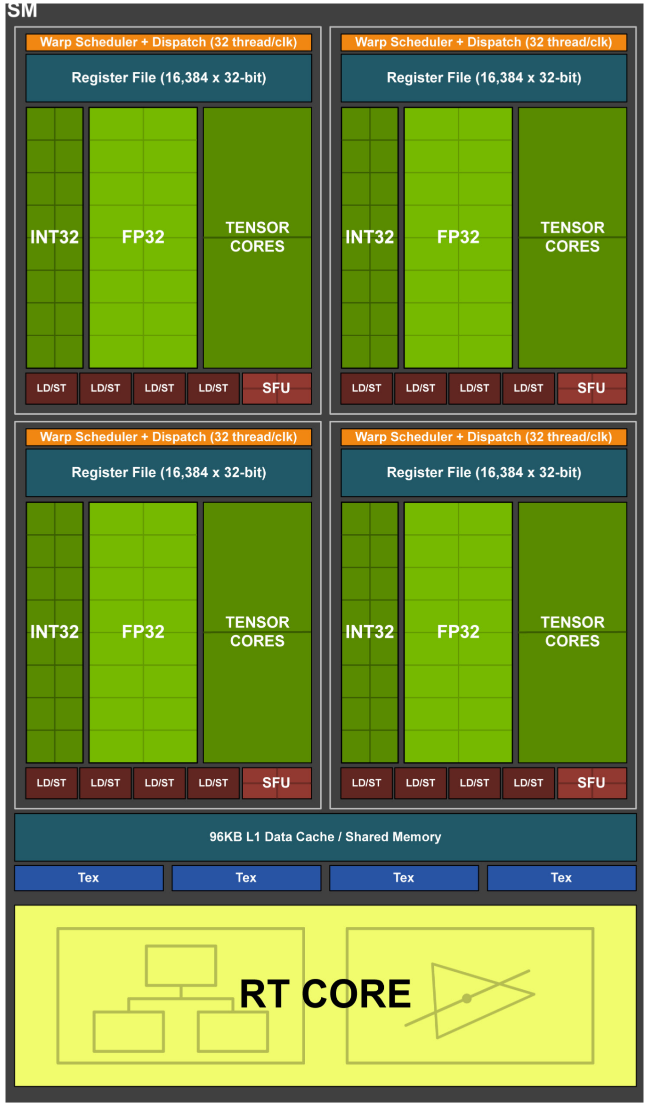

SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

Warp Scheduling

- SIMT/SIMD execution
  - One control flow, multiple data paths
  - Spend more silicon on raw computation

- What about control flow?

TU 102 SM


SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

Control Flow

- Unconditional branches
  - No problem

```
f();

goto x;
```

- Uniform branches
  - No problem

```
if (blockIdx.x < 16) {
    ...
}
```

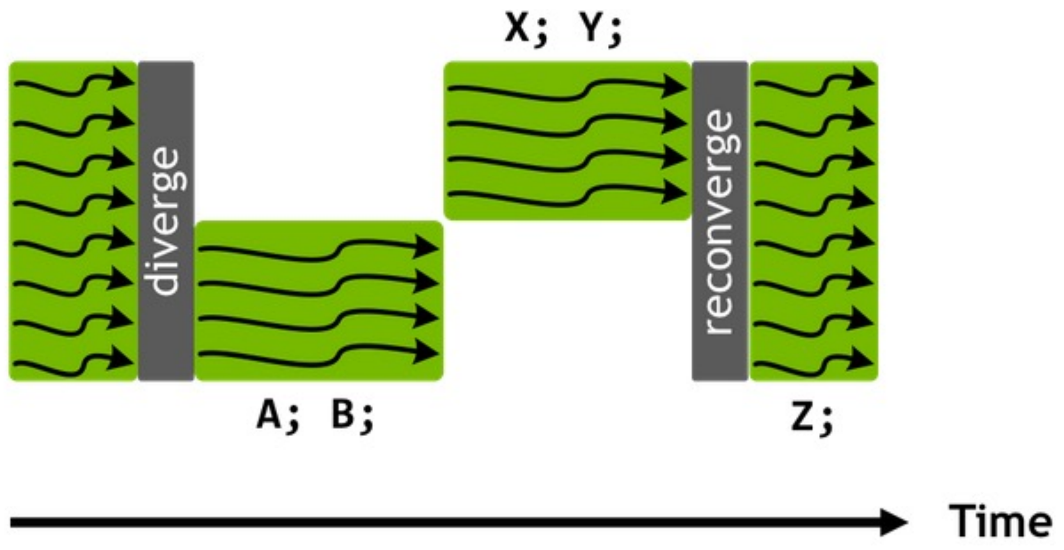
- Non-uniform branches
  - Problem

```
if (threadIdx.x < 16) {
    ...
}
```

Traditional SIMT Execution

- 32 threads (1 warp) execute in SIMT fashion
  - 1 program counter shared per warp
  - Divergent paths leave threads inactive
    - Whole warp must execute each branch sequentially
  - Reconverge after divergent section

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Control Flow

- Implement via predication
- What about branches in branches?

- Branch stack
  - Push mask of active threads
  - Run first half
  - Pop mask of active threads
  - Run other half

```
if (A) {
    if (B) {
        if (C) {
            ...
        } else {
            ...
        }
    } else {
        ...
    }
}
```

Example 1

```
__global__ void test(int* out, int N) {
    unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

    if (threadIdx.x < N)
        out[idx] = N;
}
```

https://godbolt.org/z/TocT1fqEW

Example 1: Predication

```
test(int*, int):
MOV R1, c[0x0][0x44]
S2R R0, SR_CTAID.X
S2R R3, SR_TID.X
IMAD R0, R0, c[0x0][0x28], R3
ISETP.GE.U32.AND P0, PT, R0, c[0x0][0x148], PT
@P0 EXIT
ISCADD R2.CC, R0, c[0x0][0x140], 0x2
MOV32I R3, 0x4
MOV R4, c[0x0][0x148]
IMAD.U32.U32.HI.X R3, R0, R3, c[0x0][0x144]
ST.E [R2], R4
EXIT
.L_1:
BRA `(.L_1)
.L_20:
```

Example 2

```
__global__ void test(int* out, int N) {
    unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

    if (threadIdx.x < N)
        if (idx % 2 == 0)
            out[idx] = out[idx] + 2;
        else
            out[idx] = out[idx] - 8;
}
```

https://godbolt.org/z/fs11YEvTq

Example 2: Predication

```
test(int*, int):
MOV R1, c[0x0][0x44]
S2R R0, SR_CTAID.X
S2R R3, SR_TID.X
IMAD R0, R0, c[0x0][0x28], R3
ISETP.GE.U32.AND P0, PT, R0, c[0x0][0x148], PT
@P0 EXIT
ISCADD R2.CC, R0, c[0x0][0x140], 0x2
MOV32I R3, 0x4
IMAD.U32.U32.HI.X R3, R0, R3, c[0x0][0x144]
LOP32I.AND R0, R0, 0x1
LD.E R4, [R2]
ISETP.NE.U32.AND P0, PT, R0, 0x1, PT
@!P0 IADD32I R5, R4, -0x8
@!P0 ST.E [R2], R5
@!P0 EXIT
IADD32I R4, R4, 0x2
ST.E [R2], R4
EXIT
.L_1:
BRA `(.L_1)
.L_20:
```

Example 3

```
__device__ void A();
__device__ void B();
__device__ void C();

__global__ void test() {
    unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

    if (idx < 16)
        A();
    else
        B();

    C();
}
```

https://godbolt.org/z/edWKMaEr6

Example 3: Branch Stack

```
test():
MOV R1, c[0x0][0x44]
S2R R0, SR_CTAID.X
SSY `(.L_2)
S2R R3, SR_TID.X
IMAD R0, R0, c[0x0][0x28], R3
ISETP.GE.U32.AND P0, PT, R0, 0x10, PT
@!P0 BRA `(.L_3)
JCAL `(_Z1Bv)
NOP.S (*"TARGET= .L_2 "*)
.L_3:
JCAL `(_Z1Av)
NOP.S (*"TARGET= .L_2 "*)
.L_2:
JCAL `(_Z1Cv)
MOV RZ, RZ
EXIT
.L_4:
BRA `(.L_4)
.L_19:
```

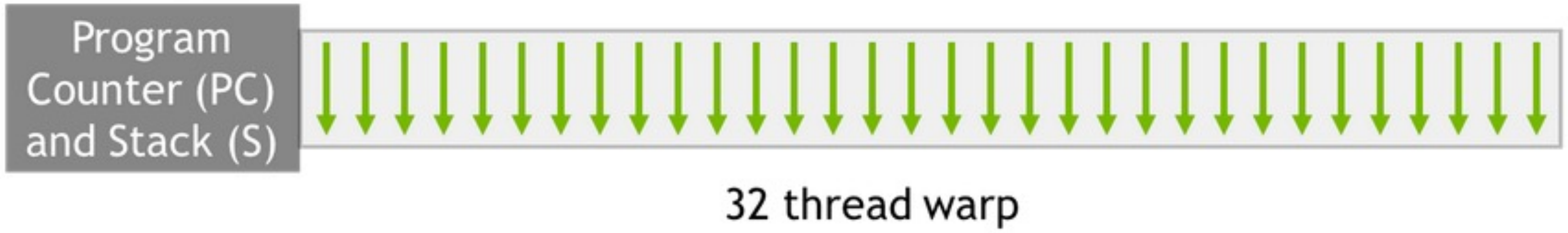
Example 3: $CC \geq 7.0$

```
test():
IMAD.MOV.U32 R1, RZ, RZ, c[0x0][0x28]
S2R R0, SR_CTAID.X
BMOV.32.CLEAR RZ, B6
BSSY B6, `(.L_5)
S2R R3, SR_TID.X
IMAD R0, R0, c[0x0][0x0], R3
ISETP.GE.U32.AND P0, PT, R0, 0x10, PT
@!P0 BRA `(.L_6)
MOV R20, 32@lo((test() + .L_2@srel))
MOV R21, 32@hi((test() + .L_2@srel))
CALL.ABS.NOINC `(_Z1Bv)
.L_2:
BRA `(.L_3)
.L_6:
MOV R20, 32@lo((test() + .L_3@srel))
MOV R21, 32@hi((test() + .L_3@srel))
CALL.ABS.NOINC `(_Z1Av)
.L_3:
BSYNC B6
.L_5:
MOV R20, 32@lo((test() + .L_4@srel))
MOV R21, 32@hi((test() + .L_4@srel))
CALL.ABS.NOINC `(_Z1Cv)
.L_4:
EXIT
.L_7:
BRA `(.L_7)
.L_20:
```

no sync needed to switch to other branch

Traditional SIMT Execution

- Per warp state
  - Program counter
  - Active mask
  - Branch stack



- Only one branch active per warp at any moment

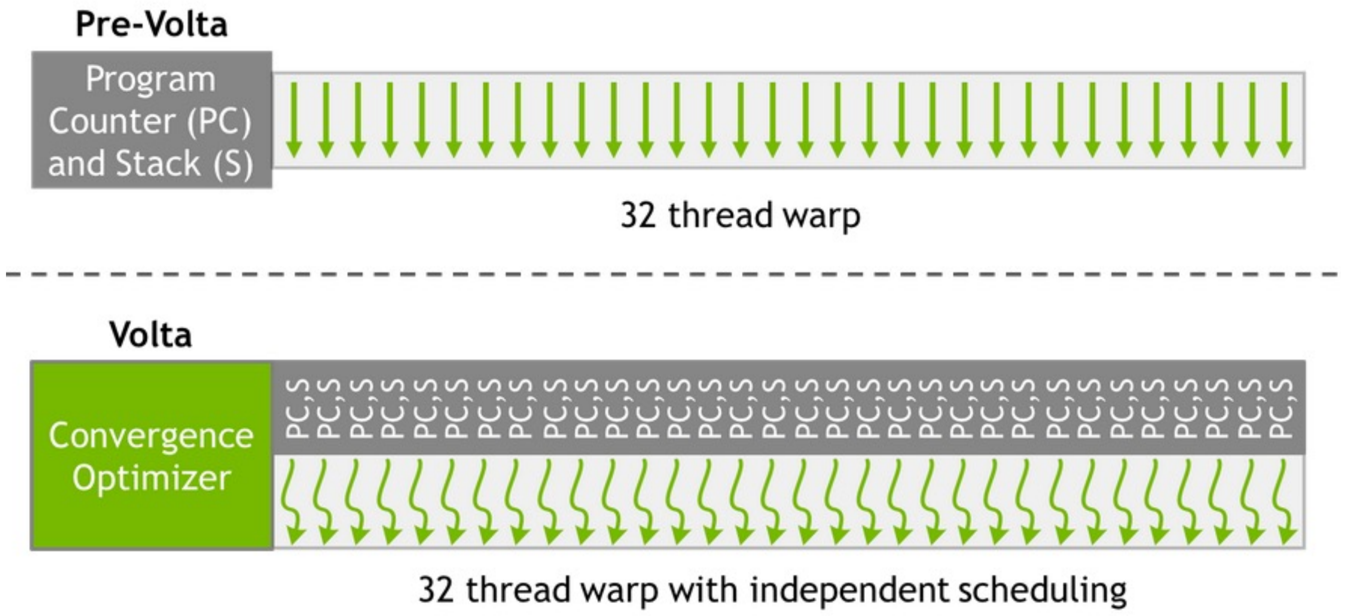
Traditional SIMT Execution

```
__device__ unsigned int mutex = 0;

__device__ void acquire() {
    while (atomicCAS(&mutex, 0, 1) == 1);
        /* mutex acquired */
}

__device__ void release() {
    atomicCAS(&mutex, 1, 0);
}
```

Volta SIMT Model

$CC \geq 7.0$

- Execution state **per thread**
  - Requires 2 additional registers
- Makes it possible to
  - Yield execution of any thread
  - Make better use of execution resources
  - Allow thread to wait for data produced by another
- Forward progress guarantee for all branches
- Convergence optimizer tries to maintain as much convergence

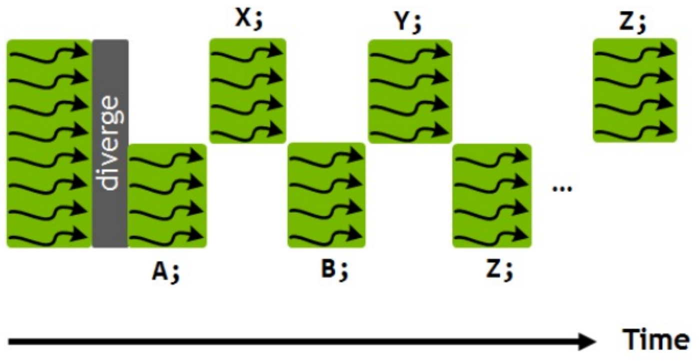

Pre-Volta
Program Counter (PC) and Stack (S)
32 thread warp
Volta
Convergence Optimizer
PC,S
32 thread warp with independent scheduling

Volta SIMT Model

$CC \geq 7.0$

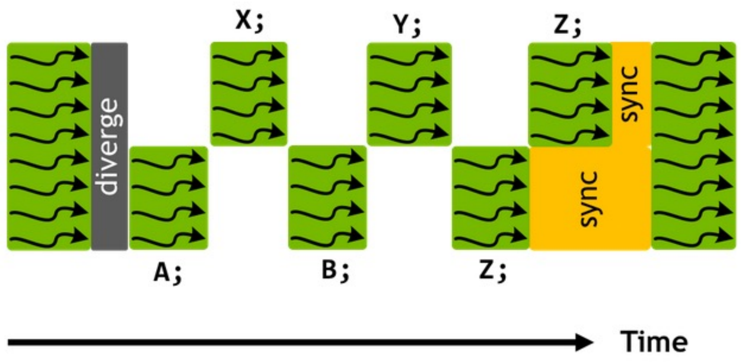
- Keep multiple branches in flight
- Still execute same instruction for active threads

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



- Use **__syncwarp()** to enforce convergence

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



__nanosleep

CC ≥ 7.0

- Warp can now yield execution

```
__device__ unsigned int mutex = 0;

__device__ void acquire() {
    unsigned int ns = 8;
    while (atomicCAS(&mutex, 0, 1) == 1) {
        __nanosleep(ns);
        if (ns < 256)
            ns *= 2;
    }
    /* mutex acquired */
}

__device__ void release() {
    atomicCAS(&mutex, 1, 0);
}
```

Pitfalls

- Applications assuming reads/writes are implicitly visible to other threads in warp require **`__syncwarp()`**
  - Whenever data is exchanged via global/shared within warp
  - Non-exited threads must execute a **`__syncwarp()`** with the same mask
  - Guarantees **memory ordering** among threads participating
- **`__syncthreads()`** must be reached by all non-exited threads
- Use **racecheck** and **synccheck** to check for violations

Effects

- If kernel consumes < 30 registers per thread
  - No change to theoretical occupancy
  - Most cases will see no difference
  - But depends on workload

- To disable **ITS** compile with

```
-arch=compute_60 -code=sm_7x
```

Warp-Level Primitives

Data Exchange

- To exchange data between threads in a warp
  - `__shfl_sync`, `__ballot_sync`, …
  - Mask is the set of threads participating in operation
  - Can be used in thread-divergent branches $CC \geq 7.0$

```
if (threadIdx.x % 2) {
    val += __shfl_sync(FULL_MASK, val, 0);
    // ...
} else {
    val += __shfl_sync(FULL_MASK, val, 0);
    // ...
}
```

Active Mask Query

- `__activemask()` returns 32-bit mask of currently active threads
- How to use `__activemask()` correctly?
  - Implicit lock step execution is not guaranteed

```
if (threadIdx.x < NUM_ELEMENTS) {
    unsigned int mask = __activemask();
    val = input[threadIdx.x];
    for (int offset = 16; offset > 0; offset /= 2)
        val += __shfl_down_sync(mask, val, offset);
    // ...
}
```

Not guaranteed to execute `__activemask()` together
→ results in partial sum

Active Mask Query

- Use a synchronization primitive to query correct mask for branch
  - Before branch

```
unsigned int mask = __ballot_sync(FULL_MASK, threadIdx.x < NUM_ELEMENTS);
if (threadIdx.x < NUM_ELEMENTS) {
    val = input[threadIdx.x];
    for (int offset = 16; offset > 0; offset /= 2)
        val += __shfl_down_sync(mask, val, offset);
    // ...
}
```

Warp Synchronization

- Can be used to explicitly force synchronization

```
void __syncwarp(unsigned int mask=FULL_MASK);
```

- Forces executing thread to wait until all threads in mask hit a **`__syncwarp()`**
  - With the same **`mask`**

Warp Synchronization

```
__shared__ shmem[blockDim.x];
unsigned int tid = threadIdx.x;

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

✗

```
__shared__ shmem[blockDim.x];
unsigned int tid = threadIdx.x;
int v = shmem[tid];

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;
```

✓

Opportunistic Warp-Level Programming

- Some computations don't depend on specific threads
  - Use whatever threads currently available
- **Warp aggregation**
  - Example: Combine atomic operations of multiple threads into a single atomic

```
__device__ int atomicInc(int *ptr) {
    return atomicAdd(ptr, 1);
}
```

Opportunistic Warp-Level Programming

```
__device__ int atomicAggInc(int *ptr) {
    int mask = __match_any_sync(__activemask(), ptr);
    // select leader
    int leader = __ffs(mask) - 1;
    // compute total increment
    int change = __popc(mask);
    // perform atomicAdd (only leader)
    int res;
    if (lane_id() == leader)
        res = atomicAdd(ptr, change);
    // broadcast result
    res = __shfl_sync(mask, res, leader);
    // compute result per lane
    return res + __popc(mask & ((1 << lane_id()) - 1));
}
```

Implicit Warp-Sync Programming

- Implicit warp-synchronous programming is unsafe
  - Legacy warp-level primitives already **deprecated** since CUDA 10

```
assert(__activemask() == FULL_MASK); // assume this is true
__syncwarp();
assert(__activemask() == FULL_MASK); // this may fail
```

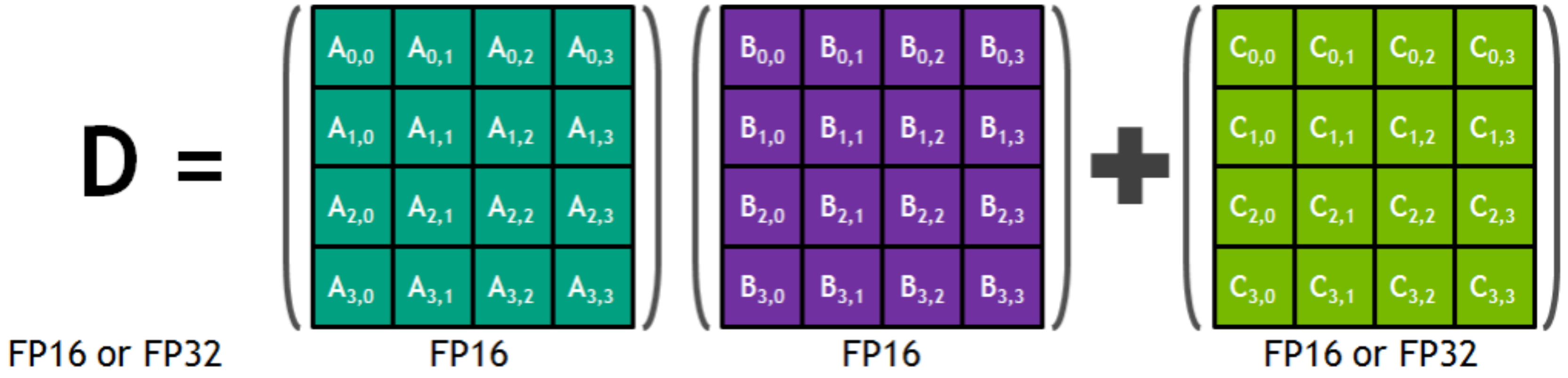
- To force legacy behavior / disable ITS compile with `-arch=compute_60 -code=sm_7x`

Tensor Cores

Tensor Cores

$CC \geq 7.0$

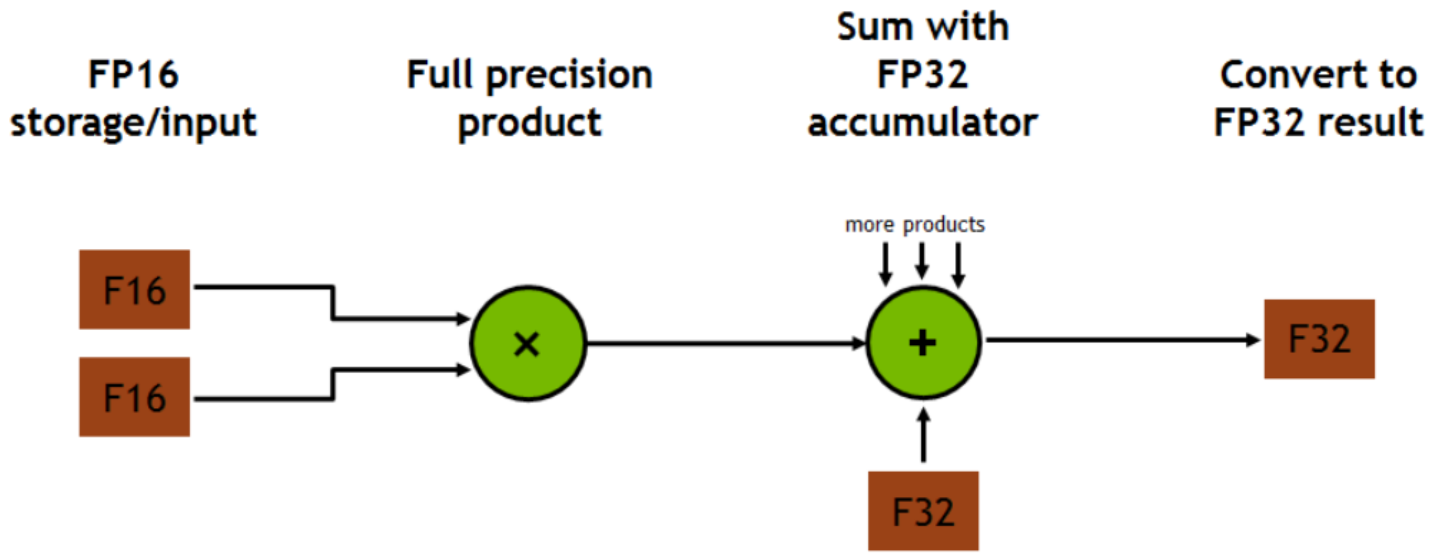
- Volta+ architectures introduce **Tensor Cores**
  - Programmable Matrix-Multiply-And-Accumulate $D = A \times B + C$
    - Titan V / Tesla V100 → 8 Tensor Cores per SM
- 1 Tensor Core
  - 4x4x4 matrix processing array


D =
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
+
$C_{0,0}$ $C_{0,1}$ $C_{0,2}$ $C_{0,3}$
$C_{1,0}$ $C_{1,1}$ $C_{1,2}$ $C_{1,3}$
$C_{2,0}$ $C_{2,1}$ $C_{2,2}$ $C_{2,3}$
$C_{3,0}$ $C_{3,1}$ $C_{3,2}$ $C_{3,3}$
FP16 or FP32
FP16
FP16
FP16 or FP32

Tensor Cores

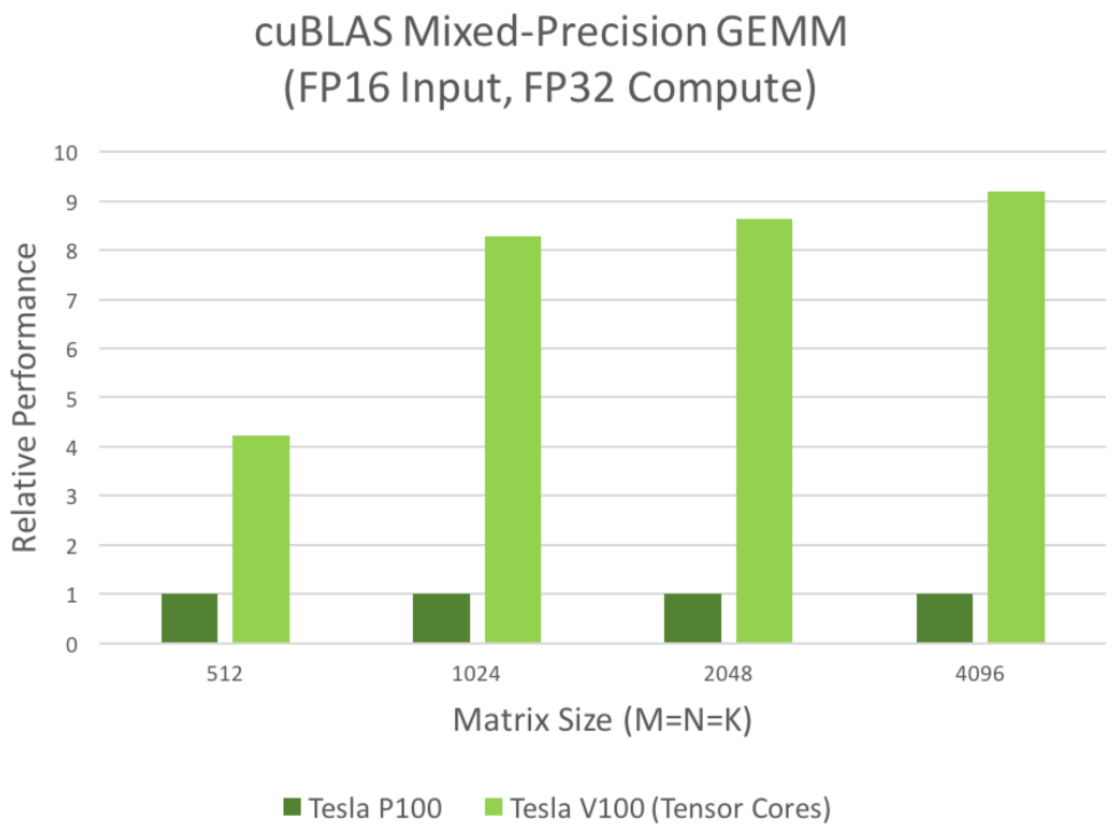
$CC \geq 7.0$

- Each core does 64 floating point FMA (2 ops) operations per clock
  - 8 Tensor Cores → 64 * 2 * 8 Ops/cycle → 1024 Ops/cycle
- Used concurrently by a full warp
  - 16x16x16 matrix operation



Tensor Cores in Libraries

- Currently available in **cuDNN** and **cuBLAS**
  - To speed up GEMM and convolutions
  - Set math type to
    - **CUBLAS_TENSOR_OP_MATH**
    - **CUDNN_TENSOR_OP_MATH**
- Restrictions on input size and layout
  - Fall back to non tensor core implementation if not possible



How to use Tensor Cores

```
#include <mma.h>
using namespace nvcuda::wmma;
```

```
// contains section of a matrix distributed across all threads in warp
template<typename Use, int m, int n, int k, typename T, typename Layout=void> class fragment;

// waits until all warps are at load matrix and then loads matrix
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);

// waits until all warps are at store matrix and then stores matrix
void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t layout);

// fill fragment with constant value v
void fill_fragment(fragment<...> &a, const T& v);

// perform warp-synchronous matrix multiply-accumulate d = a*b + c
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b,
const fragment<...> &c, bool satf=false);
```

Example

```
__global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K,
                             float alpha, float beta) {
    // leading dimensions; matrices packed in memory, no transpose
    int lda = M; // multiples of 16
    int ldb = K; // multiples of 16
    int ldc = M; // multiples of 16

    // tile using a 2D grid
    int warp_m = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
    int warp_n = (blockIdx.y * blockDim.y + threadIdx.y);

    // 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::col_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> acc_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

    wmma::fill_fragment(acc_frag, 0.0f);
```

Example

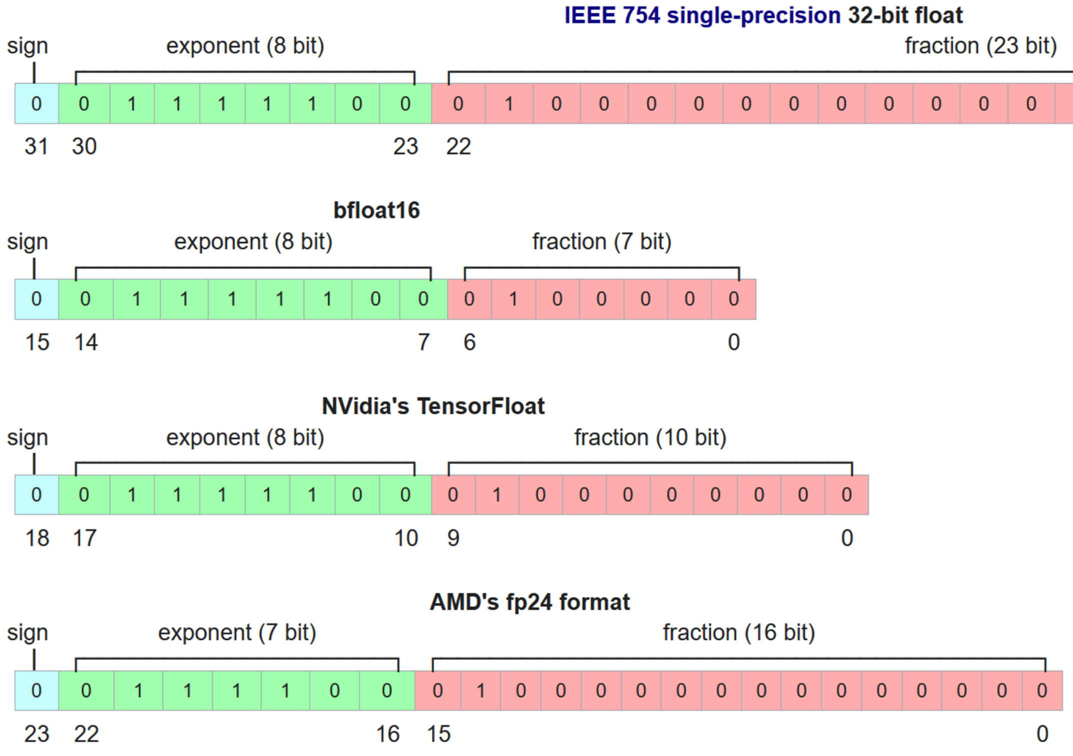
```
// loop over k
for (int i = 0; i < K; i += 16) {
    int a_row = warp_m * 16;
    int a_col = i;

    int b_row = i;
    int b_col = warp_n * 16;

    // bounds checking
    if (a_row < M && a_col < K && b_row < K && b_col < N) {
        // load the inputs
        wmma::load_matrix_sync(a_frag, a + a_row + a_col * lda, lda);
        wmma::load_matrix_sync(b_frag, b + b_row + b_col * ldb, ldb);

        // perform the matrix multiplication
        wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
    }
}
```

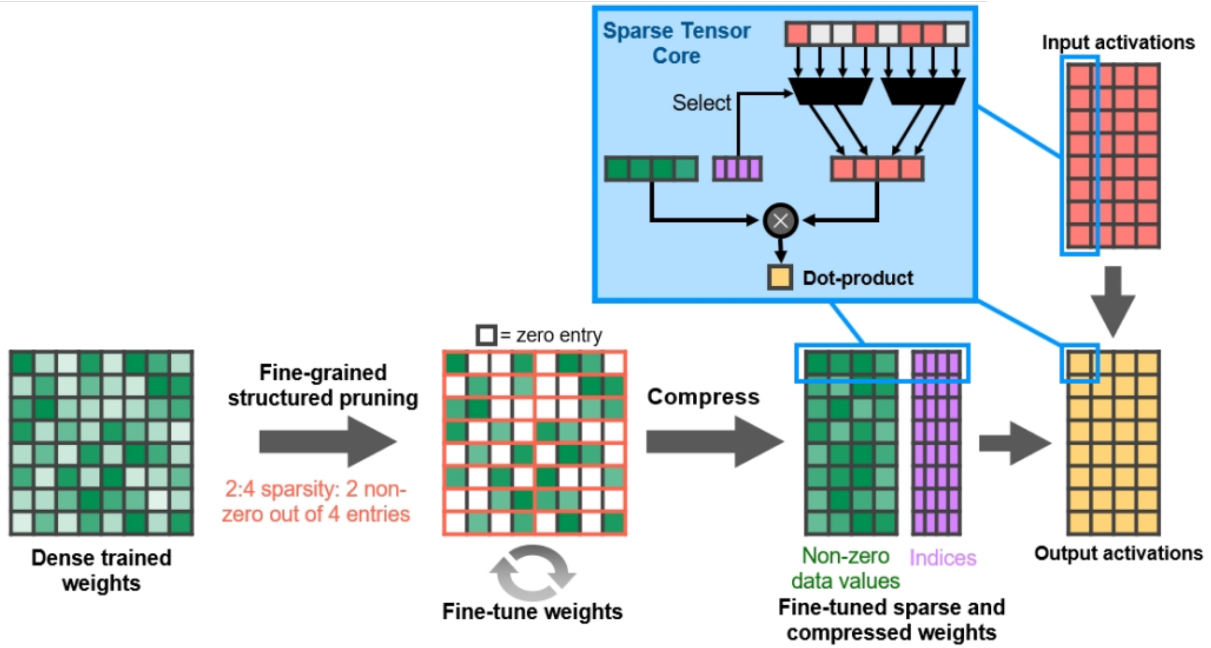
Example

```
// load in the current value of c, scale it by beta, and add this our result scaled by alpha
int c_row = warp_m * 16;
int c_col = warp_n * 16;

if (c_row < M && c_col < N) {
    wmma::load_matrix_sync(c_frag, c + c_row + c_col * ldc, ldc, wmma::mem_col_major);

    for (int I = 0; i < c_frag.num_elements; ++i)
        c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];

    // store the output
    wmma::store_matrix_sync(c + c_row + c_col * ldc, c_frag, ldc, wmma::mem_col_major);
}
```

Tensor Cores 3.0

- More number formats
  - INT8, INT4, INT1
  - BF16, TF32
- Structured sparsity
  - Can mask off some elements as being zero


IEEE 754 single-precision 32-bit float
sign
exponent (8 bit)
fraction (23 bit)
0 | 0 1 1 1 1 1 0 0 | 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
31 30 23 22 0
bfloat16
sign
exponent (8 bit)
fraction (7 bit)
0 | 0 1 1 1 1 1 0 0 | 0 1 0 0 0 0 0
15 14 7 6 0
NVidia's TensorFloat
sign
exponent (8 bit)
fraction (10 bit)
0 | 0 1 1 1 1 1 0 0 | 0 1 0 0 0 0 0 0 0 0
18 17 10 9 0
AMD's fp24 format
sign
exponent (7 bit)
fraction (16 bit)
0 | 0 1 1 1 1 0 0 | 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
23 22 16 15 0


Sparse Tensor Core
Select
Dot-product
Input activations
= zero entry
Dense trained weights
Fine-grained structured pruning
2:4 sparsity: 2 non-zero out of 4 entries
Fine-tune weights
Compress
Non-zero data values
Indices
Fine-tuned sparse and compressed weights
Output activations