Execution on GPUs

GPU overview

Programmer

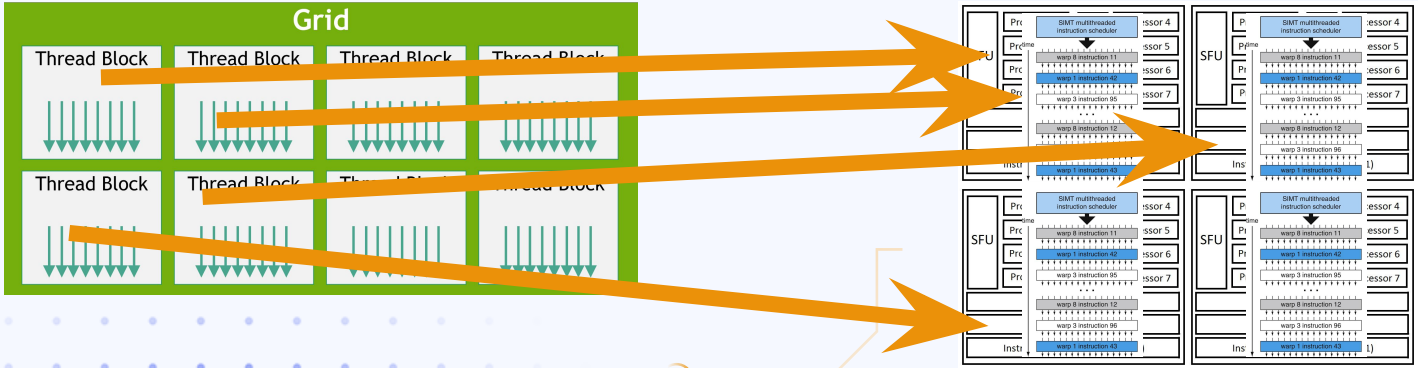
Writes one program to run on many, many threads (SPMD)

Chooses block size, # threads

Hardware

Schedules each block to a Streaming Multiprocessor

Threads inside blocks are split up into warps to enable lockstep execution (SIMT)



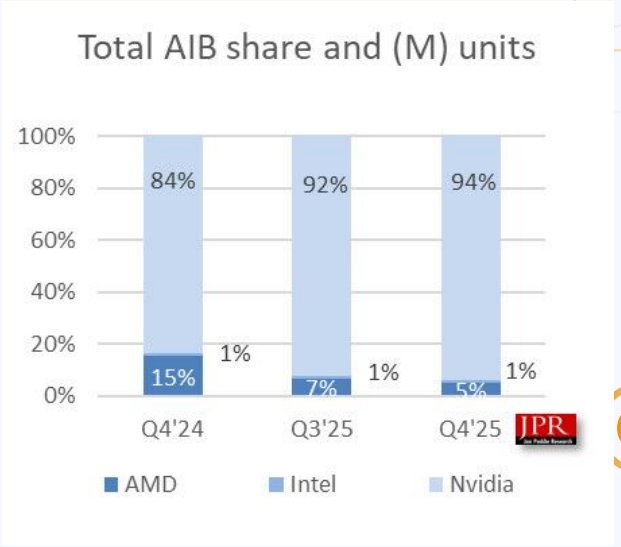
A note about nVidia

They’re not paying me

These slides use nVidia/CUDA terminology for practicality (matches dominant platform)

See H&P Fig. 4.25 for translations to their terms and OpenCL

Is this good for the consumer/society? I’m not an economist but I’m happy to hear your takes



source

source

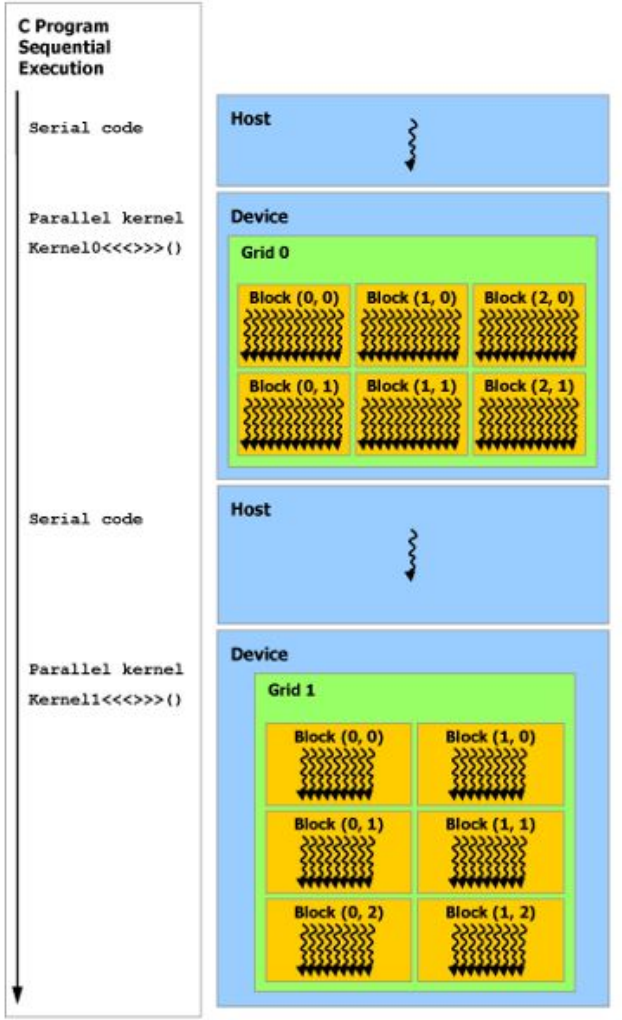
CUDA programming model

API for programming nVidia GPUs at the thread level

Allows for specifying host (CPU) code (to set up threads) + device (GPU) code (parallel **kernel**)

Host and device have separate hardware resources

Note: this is not a graphics course, and not a GPU programming course
We're studying the details of highly parallel architectures that use the SPMD programming model



nVidia PTX ~~ISA~~

PTX = parallel thread execution

Full summary in P&H fig. B.4.3, documentation here

Suffixes may define operand data type, memory space

Arithmetic: add, sub, mul, etc

Special function: sqrt, sin, cos, etc

Logical: and, or, xor, etc

Memory: ld, st, tex (texture lookup), atom

Control: branch, call, ret, sync, exit

Why isn't it a "true" ISA?

- Actual machine instructions are proprietary/may differ by device – device has to translate PTX to its own instructions
- Virtual registers

Each PTX instruction is issued once per warp* for SIMT execution

CSR multiplication in CUDA

```
__host__
// set up matrix in CSR format here: ...
// 8 blocks, 256 threads per block to do multiplication
csrMult<<<8, 256>>>(2048, Rp, C, V, x, y);

// GPU side
__device__
void csrMult(int n, int* Rp, int* C, float* V, float* x, float* y) {
    int r = blockIdx.x * blockDim.x + threadIdx.x;
    if (r < n) {
        int rBeg = Rp[r];
        int rSize = Rp[r + 1] - Rp[r];
        y[r] = multRow(rSize, C + rBeg, V + rBeg, x);
    }
}
```

Instead of loop, use these provided variables as per-thread "coordinates" for data access

What do we need to watch out for when programming using CUDA?

Hardware-aware SPMD

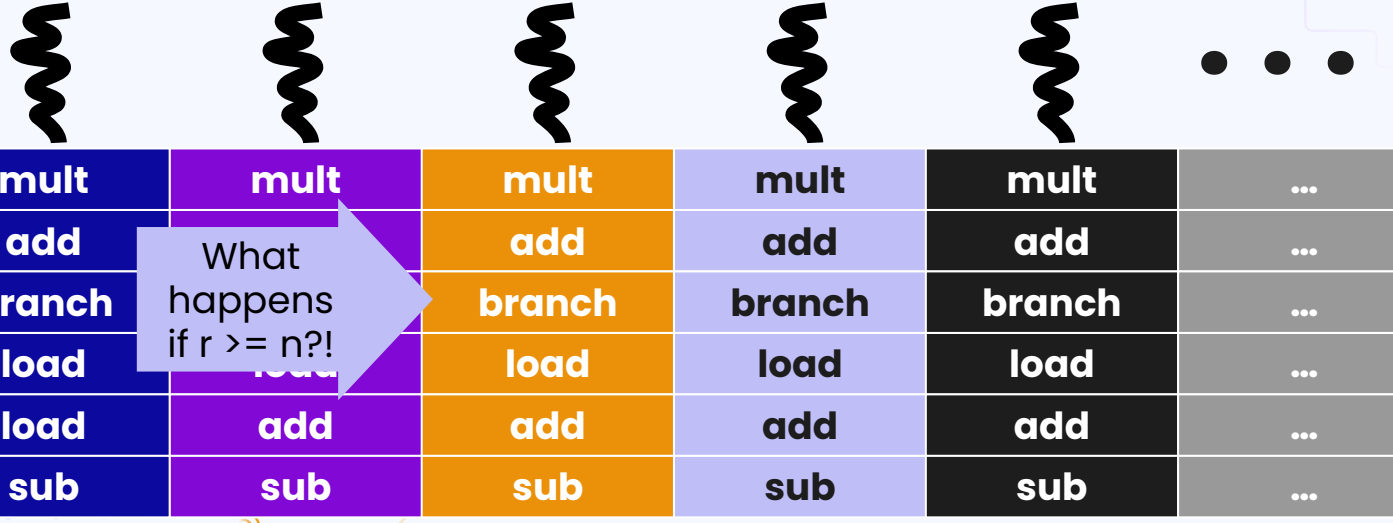
SPMD promises us the power of writing scalar programs and running them with high parallelism

It's (relatively) easy to write a SPMD program that runs, BUT:

- Hardware is going to affect performance
- Same themes we've been talking about (control hazards, data hazards, getting around memory latency) come up in the optimization guide for CUDA

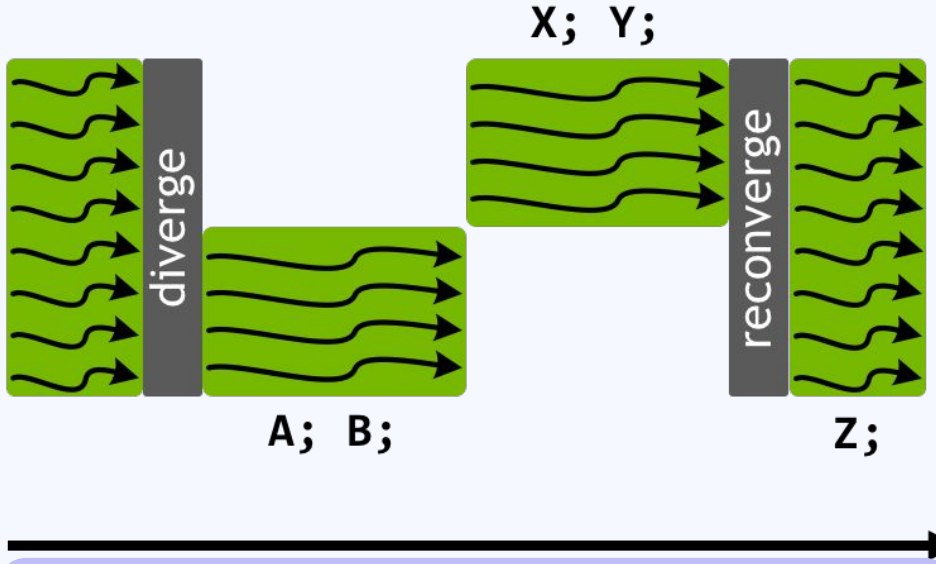
Kernel execution

```
int r = blockIdx.x * blockDim.x + threadIdx.x;
if (r < n) {
    int rBeg = Rp[r];
    int rSize = Rp[r + 1] - Rp[r];
    y[r] = multRow(rSize, C + rBeg, V + rBeg, x);
}
```



Active threads (within a warp)

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



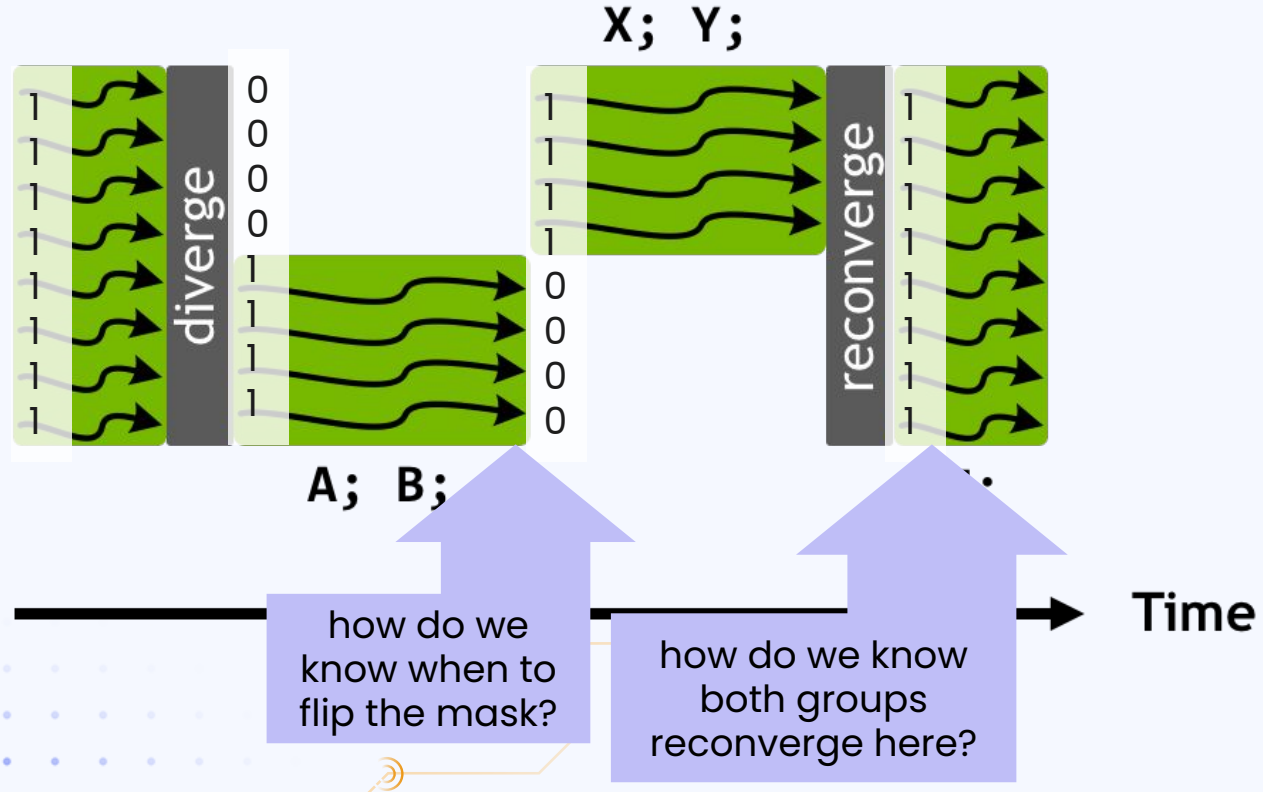
How to keep track of which threads are active?
How to keep track of when to reconverge?

image source

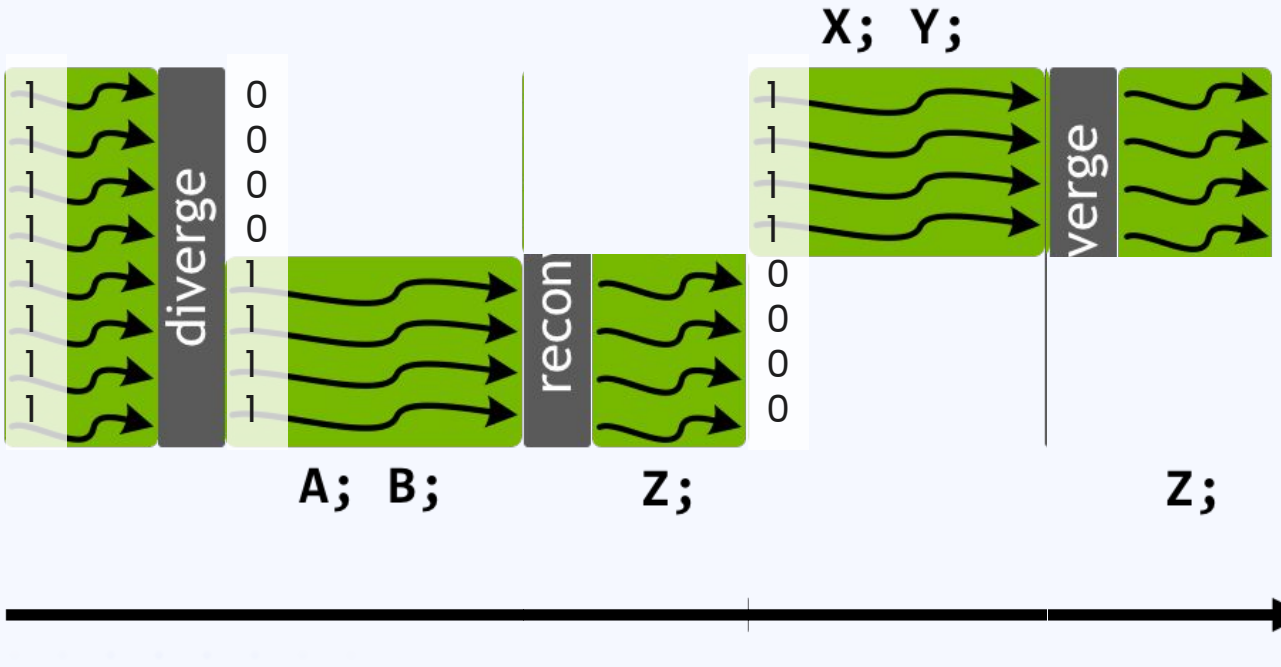
Execution mask

Use single PC and keep track of which threads are using that PC

```
@p bra L
A;
B;
bra END
L: X;
Y;
END: Z;
```



i.e. avoid this



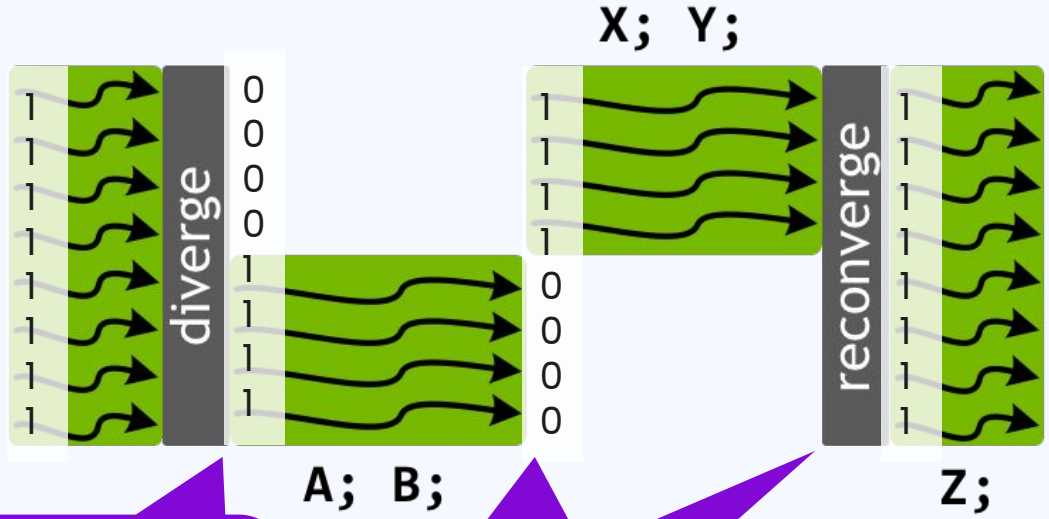

```
@p bra L
A;
B;
bra END
L: X;
Y;
END: Z;
```

source: SIMD reconvergence paper by Fung et al; supported by nVidia patents

Stack-based reconvergence

```
 0 @p bra L
 8 A;
10 B;
18 bra END
20 L: X;
28 Y;
30 END: Z;
```

(somehow) identified as reconvergence point



diverge on branch: push divergent outcomes on stack

reconvergence PC reached: pop

Reconvergence stack
recon point/next PC/mask

30 (END)/8/00001111
30 (END)/20 (L)/11110000
-/30 (END)/11111111

Determining convergence point

A CTA with divergent threads may have lower performance than a CTA with uniformly executing threads, so it is important to have divergent threads re-converge as soon as possible. All control constructs are assumed to be divergent points unless the control-flow instruction is marked as uniform, using the `.uni` suffix. For divergent control flow, the optimizing code generator automatically determines points of re-convergence. Therefore, a compiler or code author targeting PTX can ignore the issue of divergent threads, but has the opportunity to improve performance by marking branch points as uniform when the compiler or author can guarantee that the branch point is non-divergent.

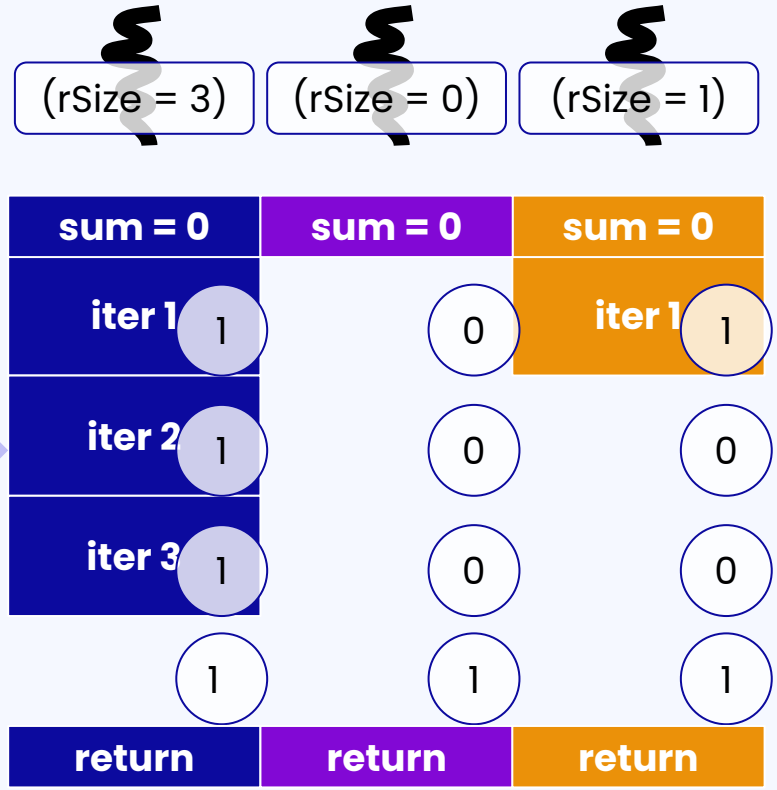
source

Also works on loops

```
float multRow(int rSize, int* Cr,
              float* Vr, float* x) {
    float sum = 0;
    for (int i = 0; i < rSize; i++) {
        sum += Vr[i] * x[Cr[i]];
    }
    return sum;
}
```

each iteration computes branch

reconverge!

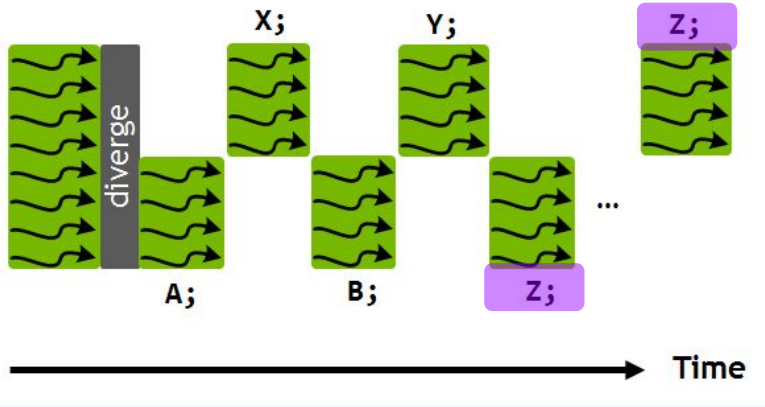


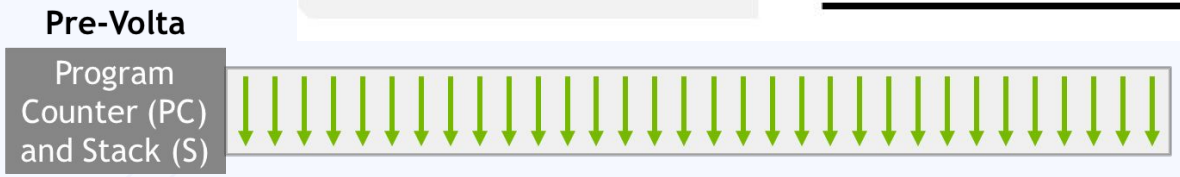
Where might execution mask usage become complicated?

nVidia Volta

Allows switching between execution paths
...how?

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





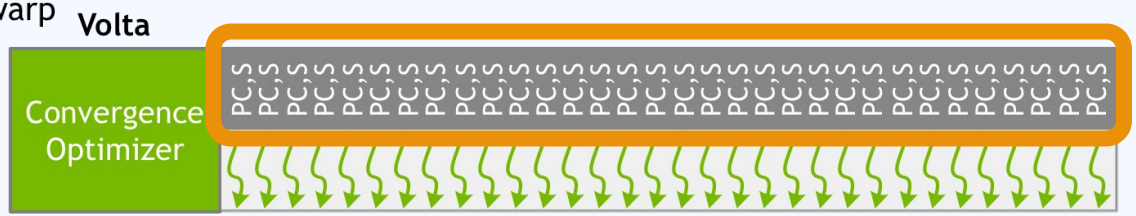
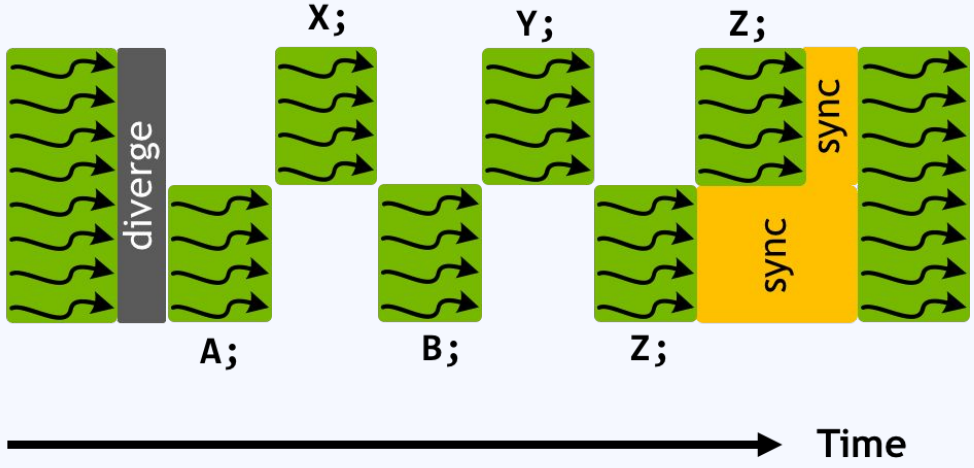


image source

It is interesting to note that Figure 12 does not show execution of statement `Z` by all threads in the warp at the same time. This is because the scheduler must conservatively assume that `Z` may produce data required by other divergent branches of execution in which case it would be unsafe to automatically enforce reconvergence. In the common case where `A`, `B`, `X`, and `Y` do not consist of synchronizing operations, the scheduler can identify that it is safe for the warp to naturally reconverge on `Z`, as on prior architectures.

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



This?

```
if (tid % 2 == 0) {
    …
}
```

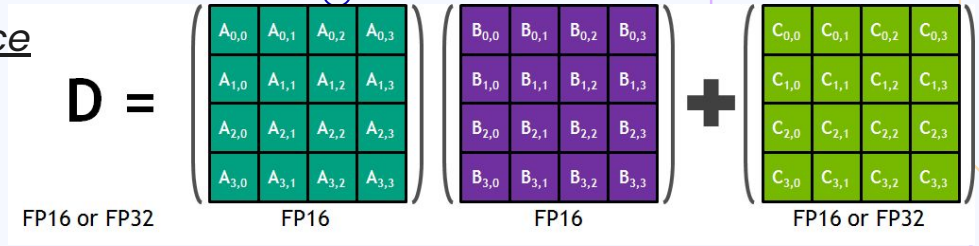
That?

```
if (tid < numThreads / 2 == 0) {
    …
}
```

What else can the programmer do?

image source

Tensor cores



An accelerator on the accelerator (see also Ray Tracing cores)

Circuits on GPUs optimized for AI (matrix operation D = A * B + C)

Uses mixed-precision to speed up math, reduce memory demands

Claim huge (8x-32x per generation) performance gains over SM matrix multiply

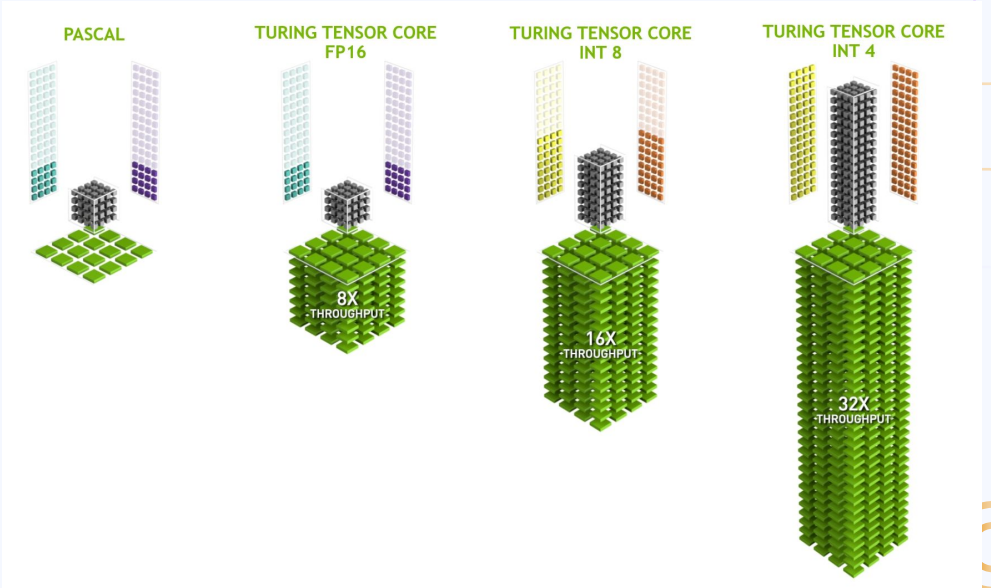
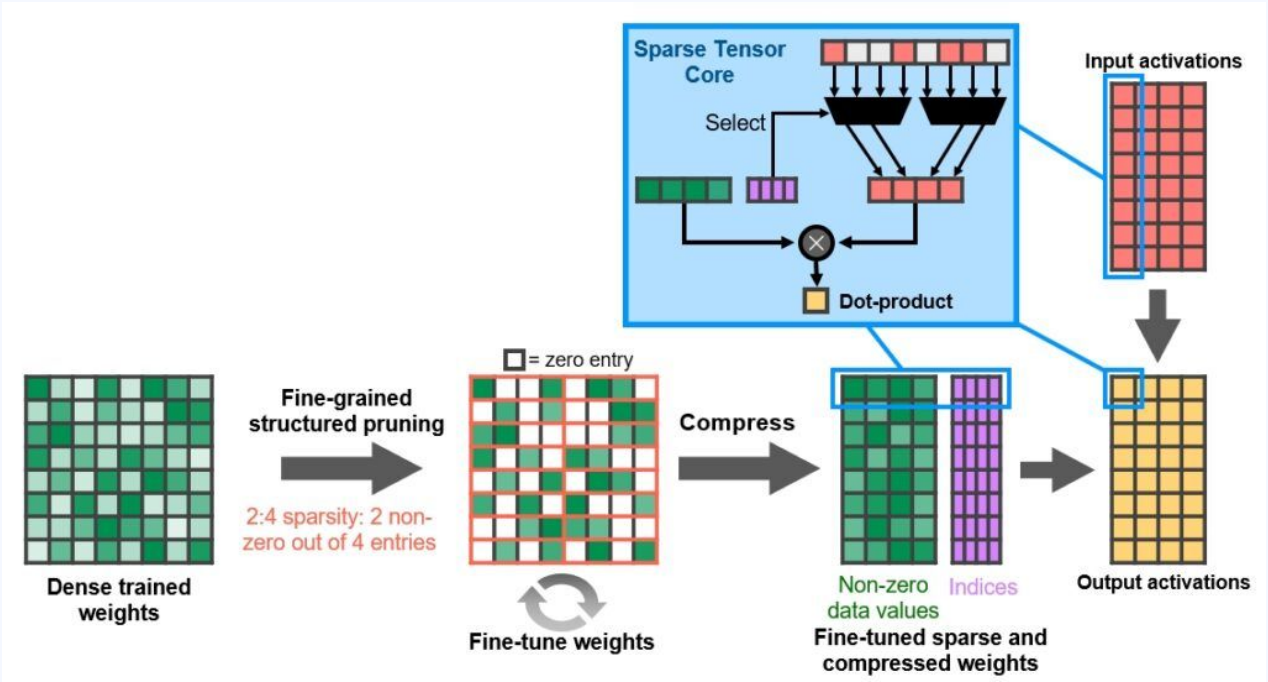


image source

Sparse acceleration

source



source

GA100 SM in Ampere