GPU memory

Final project abstract + hypothesis instructions posted (linked right above schedule)

GPU memory model

Local memory for th state

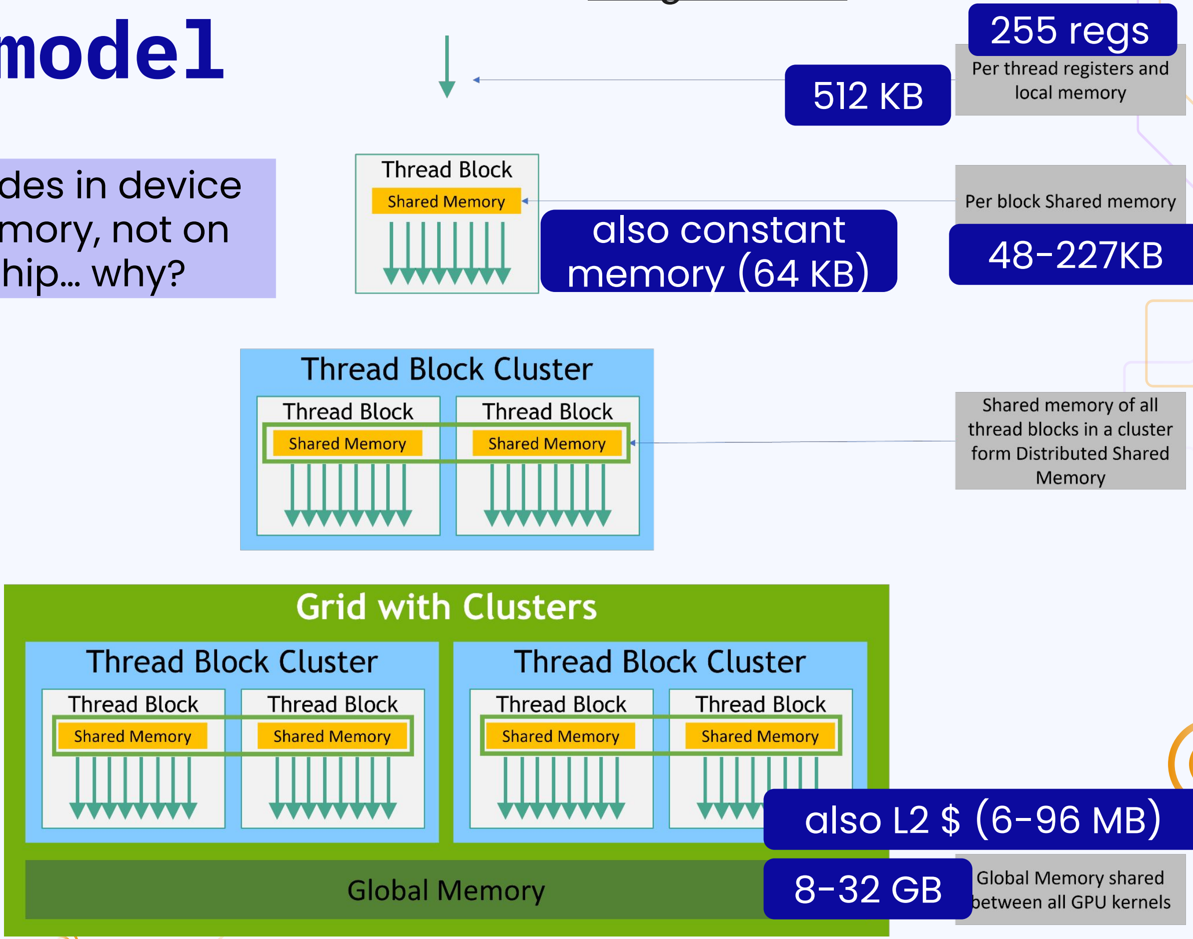
Resides in device memory, not on chip… why?

Shared memory to communicate across threads

How?

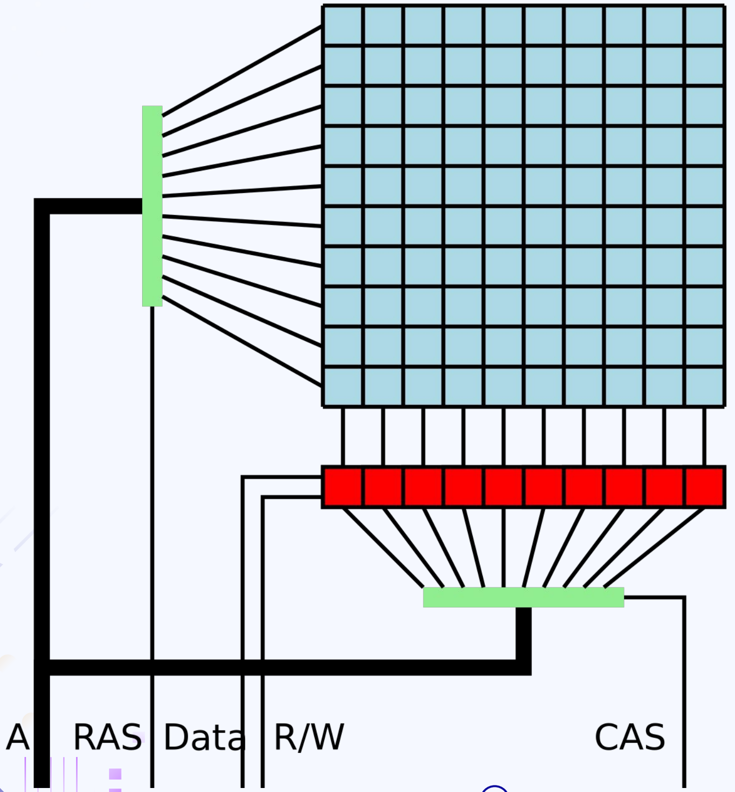
```
__shared__ bool isDone;
…
if (threadIdx.x == 0)
     if (x) isDone = true;
…
__syncthreads();
if (isDone) {
     …
}
```

image source



DRAM and GDDR

image source



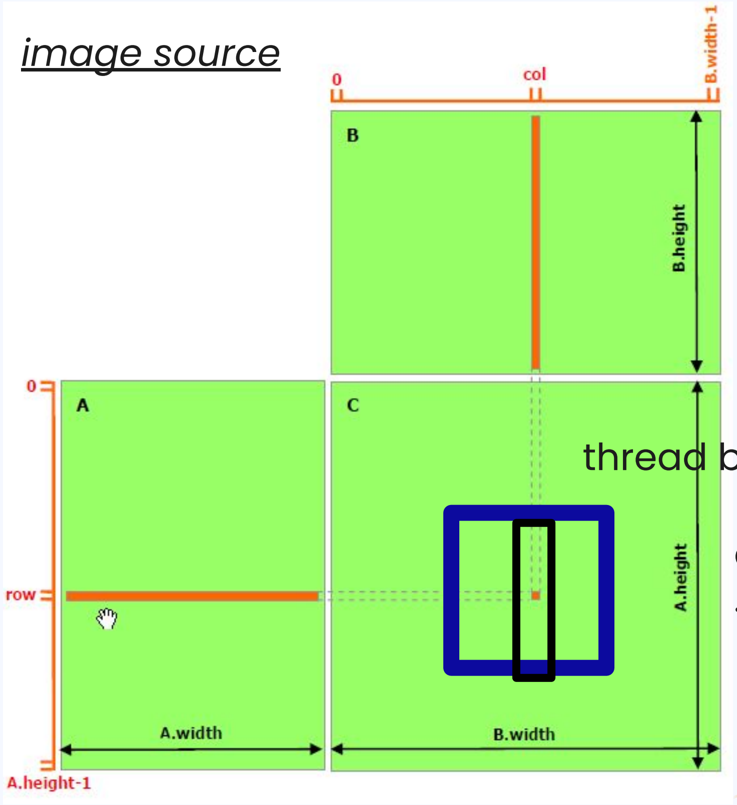
(Not pictured: further organized into banks)
Activating a row takes several cycles
After row is activated, data can be read

Latency/locality tradeoff:
Controller waits for enough requests to a single row before servicing all of them at once

Generally, GDDR (graphics DDR) variants have higher latency and higher bandwidth

GPU memory: matrix multiply

image source



What's wrong with this?

multiple columns/rows may not fit in shared memory
column doesn't play well with cache lines

thread block computes these elts

every thread here will access same column
…but all the rest will access different columns
(same for rows)

not optimally taking advantage of sharing memory within block!

Tiling

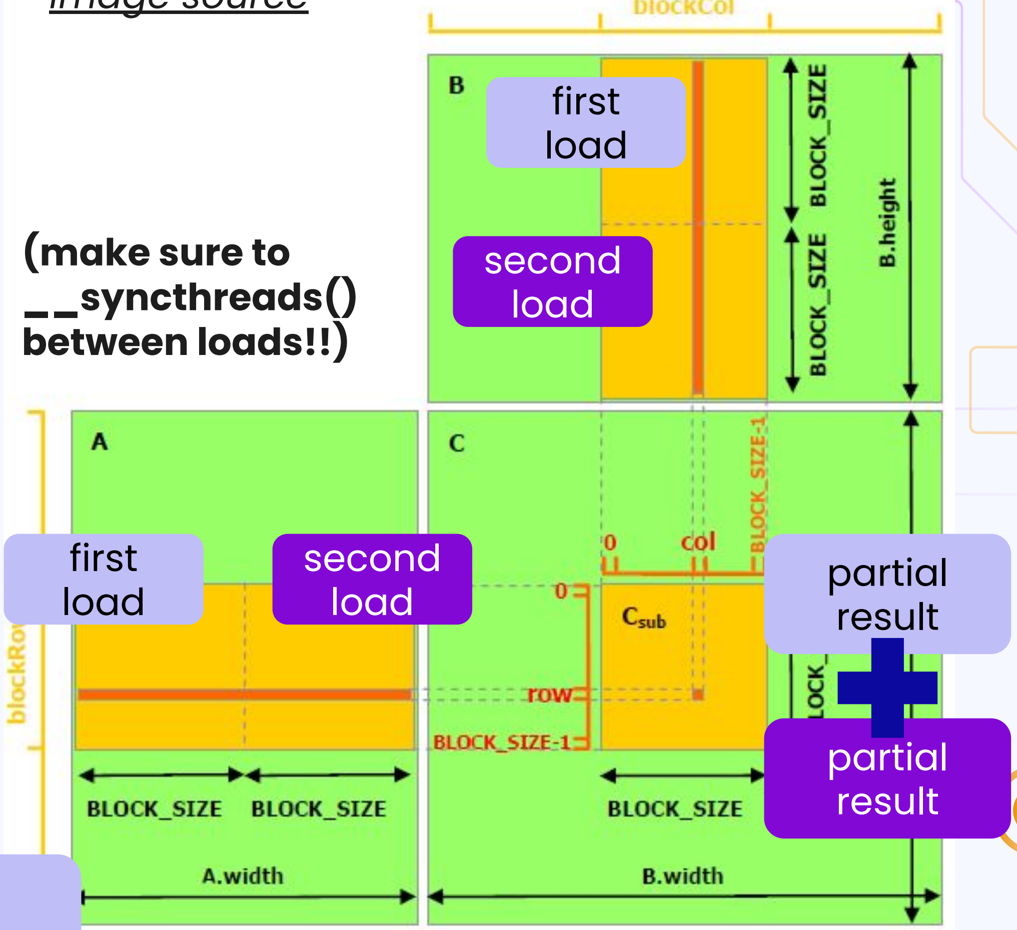
(full code, including loading from CPU memory to device memory, in image source link)

```
int blockRow = blockIdx.y;
int blockCol = blockIdx.x;
int row = threadIdx.y;
int col = threadIdx.x;
...
// for-loop on m (number of tiles to load):
Matrix Asub = GetSubMatrix(A, blockRow, m);
Matrix Bsub = GetSubMatrix(B, m, blockCol);
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
As[row][col] = GetElement(Asub, row, col);
Bs[row][col] = GetElement(Bsub, row, col);
```

Each thread does two loads/stores here
Can hardware design make this more efficient?

image source

(make sure to __synchthreads() between loads!!)



Shared memory as cache

```
// For CSR row multiplication example, adapted from P&H fig. B.8.5

__shared__ float cache [blocksize];
unsigned int block_begin = blockIdx.x * blockDim.x;
unsigned int block_end = block_begin + blockDim.x;
unsigned int row = block_begin + threadIdx.x;
if(row<num_rows) cache [threadIdx.x] = x[row];
__syncthreads();
...
// when reading each x_j
if (j >= block_begin j < block_end)
    x_j = cache[j-block_begin];
else
    x_j = x[j];
```

Not guaranteed locality (as we were in matrix multiplication), but increases performance as long as multiplying row *i* accesses values of x near *x[i]*

Coalesced memory access

Coalescing unit detects if accesses from same warp are in adjacent addresses and performs single, wide access (reduces uses of DRAM line)

Works for both global memory and local memory!

Important for programmer to be mindful of memory indexing

Example: avoid having each thread do its own malloc (source)

```
__shared__ int* data;
if (threadIdx.x == 0) {
    size_t size = blockDim.x * 64;
    data = (int*)malloc(size);
}
__syncthreads();
```

now adjacent threads can do adjacent accesses into `data`, eg `data[threadIdx.x]`!

Shared memory banks

Shared memory is banked (32 banks for 32 threads/warp; successive words in successive banks)

really fast as long as no bank conflicts (have to do an extra round of accesses for every conflict – can significantly slow down warp)

Which code is better for working with data of length `n = 2 * blocksize` when `i = threadIdx.x`?

```
A[i * 2] = A[i * 2] + B[i * 2] // 0, 2, 4, 6… 2n - 2
A[i * 2 + 1] = A[i * 2 + 1] + B[i * 2 + 1] // 1, 3, 5, 7… 2n - 1
```

vs

```
A[i] = A[i] + B[i] // 0, 1, 2, 3, … n - 1
A[n + i] = A[n + i] + B[n + 1] // n, n + 1, n + 2, … 2n - 1
```

What's wrong with our CSR mult?

```
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];
        float sum = 0
        for (int i = 0; i < rSize; i++) {
            sum += V[rBeg + i] * x[C[rBeg + i]];
        }
        y[r] = sum;
    }
}
```

Solution: pad and transpose

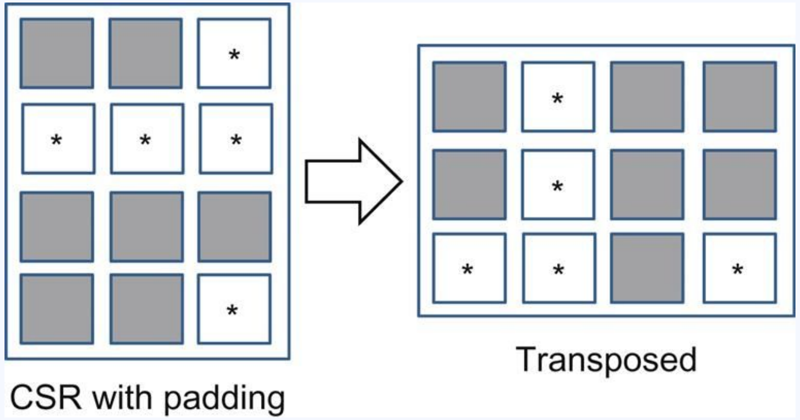
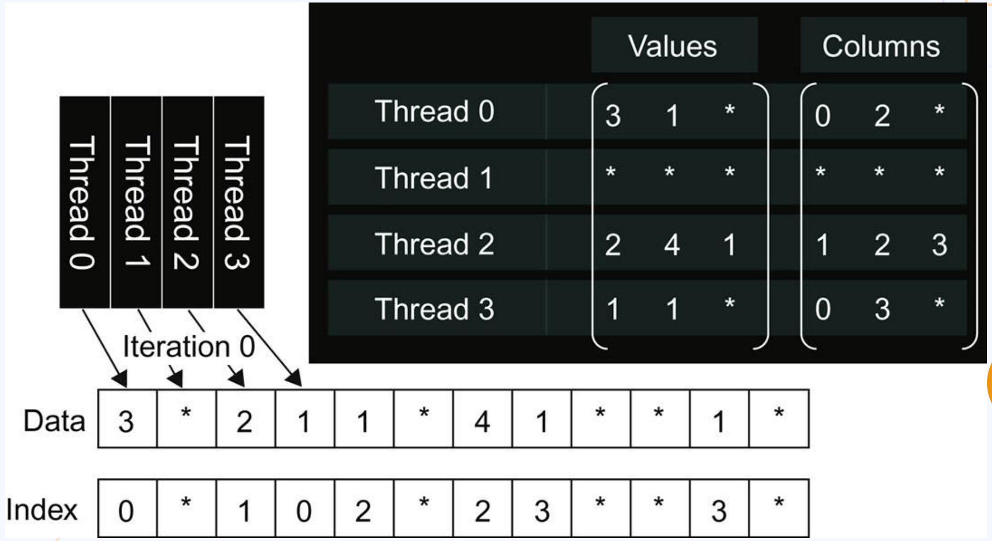


Image source: Kirk, David B., and W. Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016., figs 10.8 and 10.9
Brown library access



	Values			Columns		
Thread 0	3	1	*	0	2	*
Thread 1	*	*	*	*	*	*
Thread 2	2	4	1	1	2	3
Thread 3	1	1	*	0	3	*

Data	3	*	2	1	1	*	4	1	*	*	1	*
Index	0	*	1	0	2	*	2	3	*	*	3	*

Padding: allows for avoiding control flow divergence
Transpose: allows for coalescing
In general will run faster, despite extraneous multiplies by 0

Inclusive scan

Also called cumulative sum, prefix sum

Used for load-balancing algorithms, polynomial evaluation, etc

turns $[x_0, x_1, x_2, \dots, x_k]$ into $[x_0, (x_0 \oplus x_1), (x_0 \oplus x_1 \oplus x_2), \dots, (x_0 \oplus x_1 \oplus x_2 \oplus \dots \oplus x_k)]$

e.g. [1, 3, 0, 7] turns into [1, 4, 4, 11]

Sequentially:

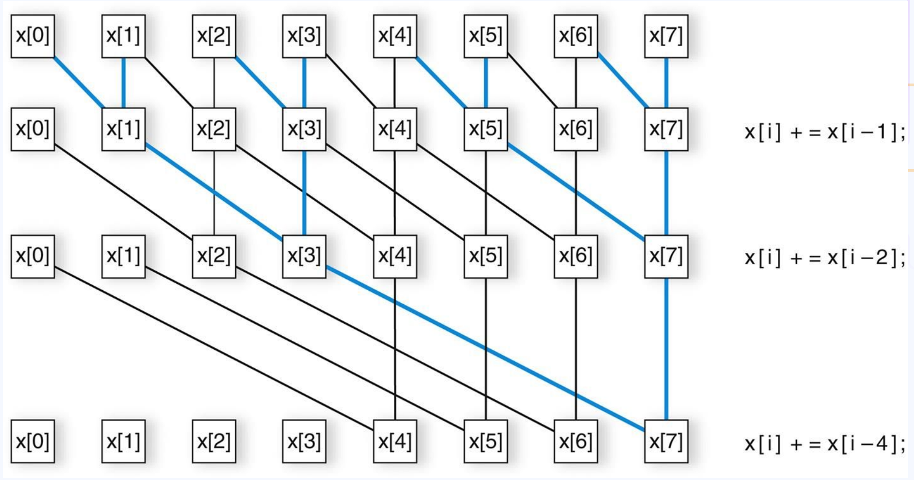
```
y[0] = x[0]
for (int i = 1; i <= k; i++) {
    y[i] = y[i - 1] + x[i];
}
```

Parallel scan

(Other, efficient algorithms exist – see Kirk and Hwu Chapter 8)

```
float plusScan(float* x) {
    int i = threadIdx.x;
    int n = blockDim.x;
    for (int o = 1; o < n; o *= 2) {
        float t;
        if (i >= o) t = x[i - o];
        __syncthreads();

        if (i >= o) x[i] = t + x[i];
        __syncthreads();
  }
  return x[i];
}
```



P&H fig. B.8.8

Does parallel scan result in bank conflicts?

? ? ?

__syncthreads() synchronizes all threads within a block.
How can threads in different blocks safely communicate with each other?

(Answer: atomic global memory accesses)

CPU/GPU communication

Must copy between CPU and GPU memory before/after launching kernel

Full code

```
int main() // on host
{
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    // Initialize input vectors
    ...
    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
```

Unified memory

GPU can access CPU's memory

Traditionally used with integrated graphics (GPU on same SoC as CPU)

CUDA example

Pros/cons?

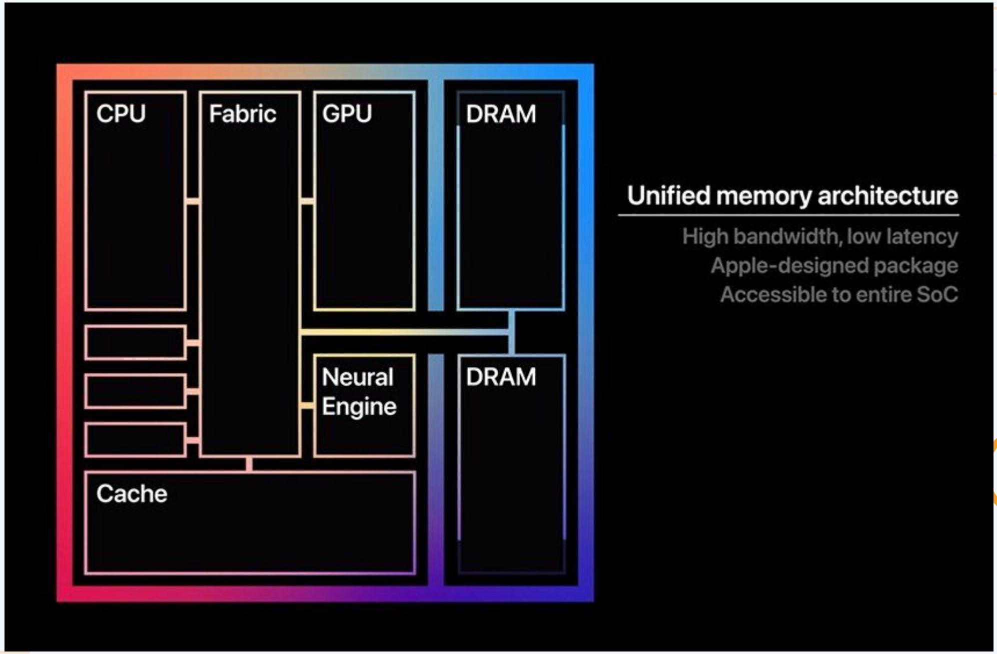


image source: Apple, inc

HW/SW interface

Vector processors and SIMD multimedia modifies the ISA to support DLP. BUT:

- Supports only modest levels of parallelism (for SIMD extensions)
- Requires changes to ISA
- Requires compiler that can effectively vectorize code (or a skilled programmer)

SPMD model/GPUs: Allows programmer to write a kernel, which the hardware schedules on many threads. BUT:

- Proper performance requires proper understanding of architecture (branching/control divergence, memory access, synchronization)
- Requires interaction of CPU/GPU (might be a pro or a con)

Bonus: tensor cores

Circuits on GPUs optimized for AI

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

Claim ~8x performance gains over SM matrix multiply

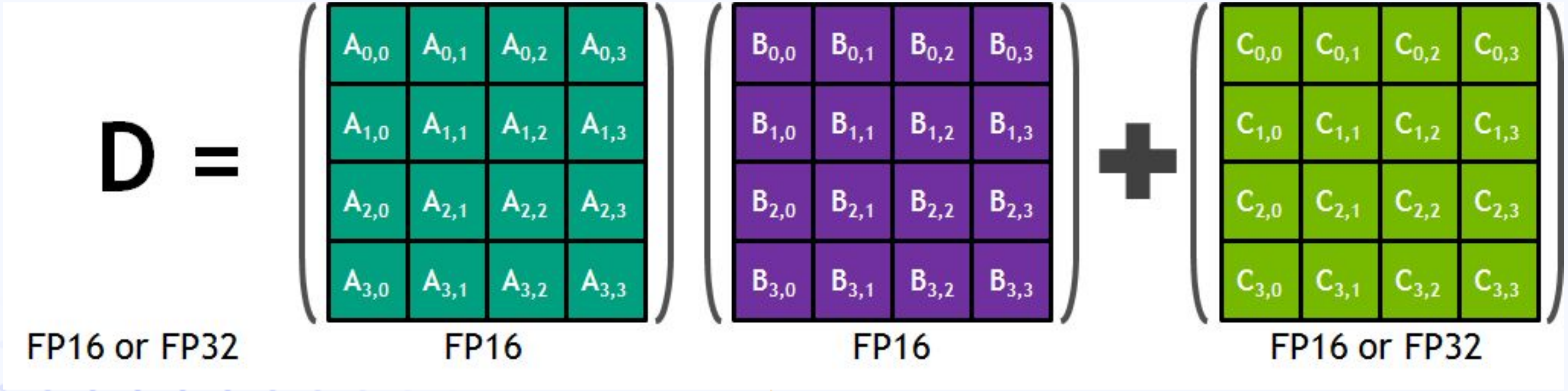


image source