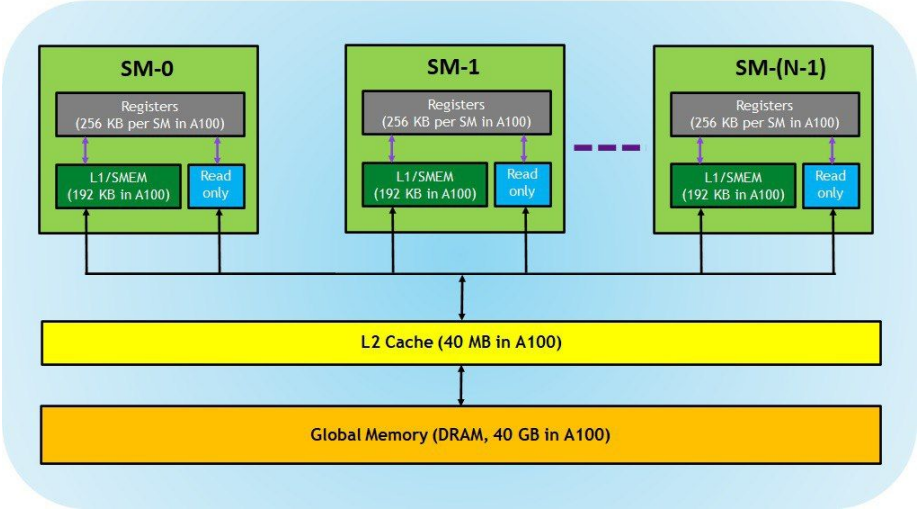
GPU Programming

Memory Hierarchy

Global Memory - This is the framebuffer size of the GPU and DRAM sitting in the GPU.

L2 cache - The L2 cache is shared across all SMs, so every thread in every CUDA block can access this memory.

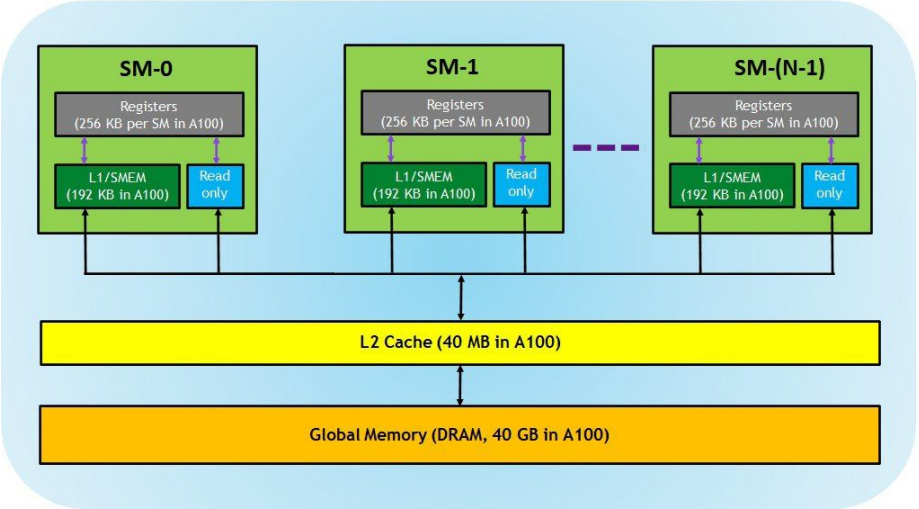
Read only memory - Each SM has an instruction cache, constant memory, texture memory and RO cache, which is read-only to kernel code.



Memory Hierarchy

L1/Shared memory (SMEM) - Every SM has a fast, on-chip scratchpad memory that can be used as **L1 cache** and **shared memory**.

Registers - These are private to each thread, which means that registers assigned to a thread are not visible to other threads. The compiler makes decisions about register utilization.



CUDA Kernel

CUDA kernel is a function that gets executed on GPU.

Any problem or application can be divided into small independent problems and solved independently among CUDA block. Each CUDA block offers to solve a sub-problem into finer pieces with parallel threads executing and cooperating with each other.

Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time.

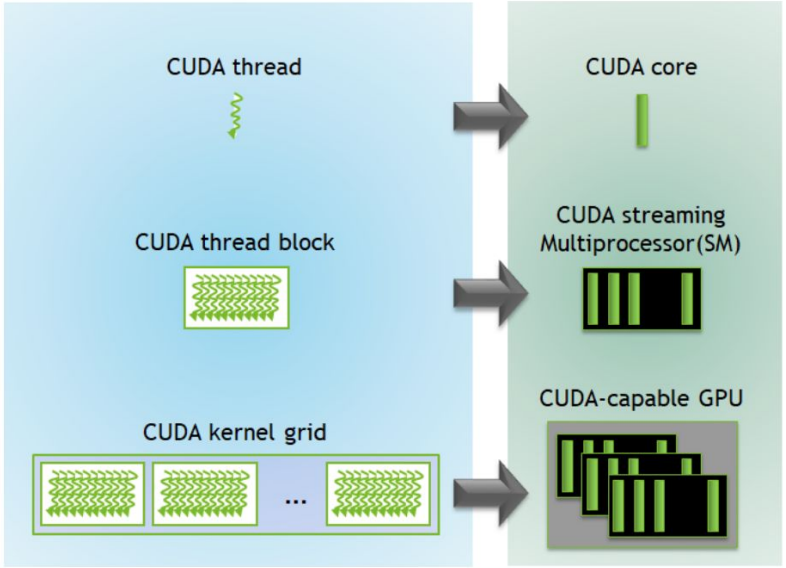


Figure 3. Kernel execution on GPU.

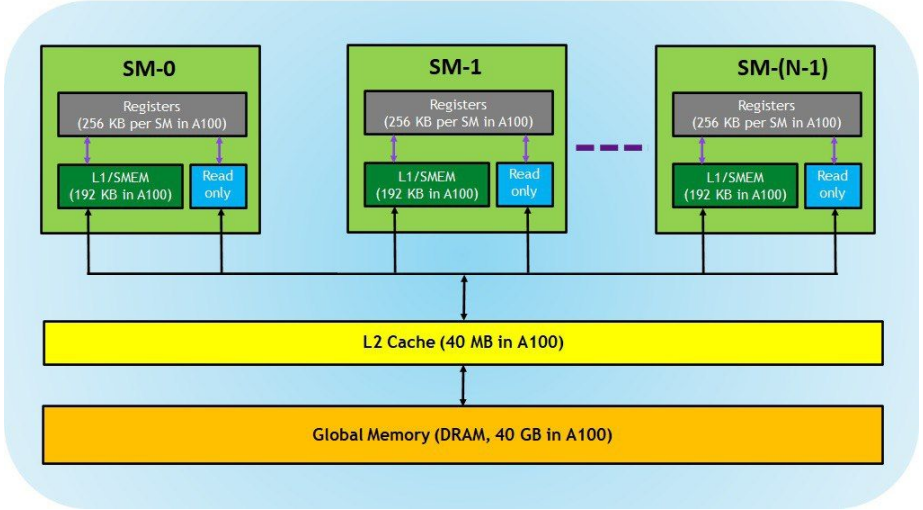
Thus a kernel is executed as a **grid** of **blocks** of **threads**.

CUDA Block

The CUDA runtime can choose how to allocate blocks to multiprocessors in a GPU (streaming multiprocessors **SM**).

Each **CUDA block** is executed by one **SM** and cannot not be migrated to other SMs in GPU (except during preemption, CUDA dynamic parallelism).

One SM can run several concurrent CUDA blocks depending on resources needed by CUDA blocks. At that each block can execute in any order relative to others.



CUDA blocks - A collection or group of **threads**. Each block can contain up to 1,024 threads.

L2 cache is used to store data that is frequently accessed by multiple thread blocks.

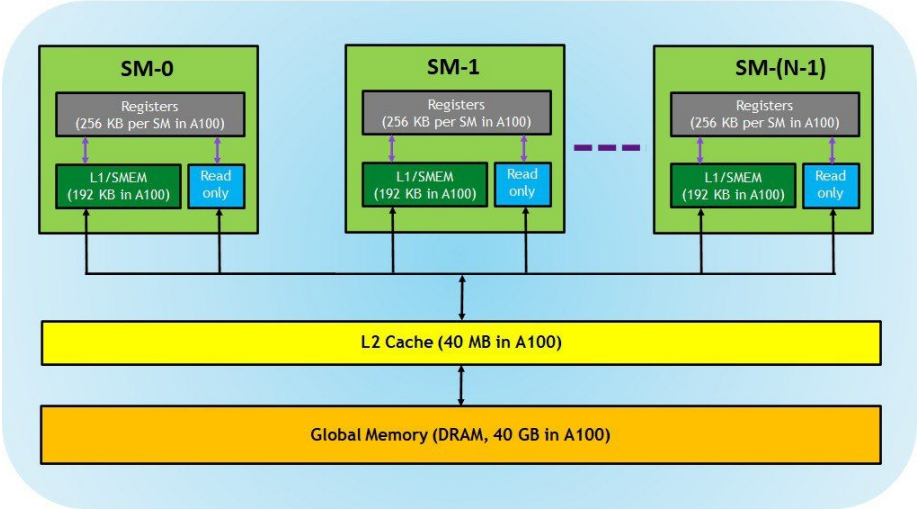
Threads

Threads are the smallest unit of execution.

Threads are executed in **registers** (cuda core, tensor core)**.**

Registers are private to each thread, which means that registers assigned to a thread are not visible to other threads. The compiler makes decisions about register utilization.

Synchronization barriers - Enable multiple threads to wait until all threads have reached a particular point of execution before any thread continue.

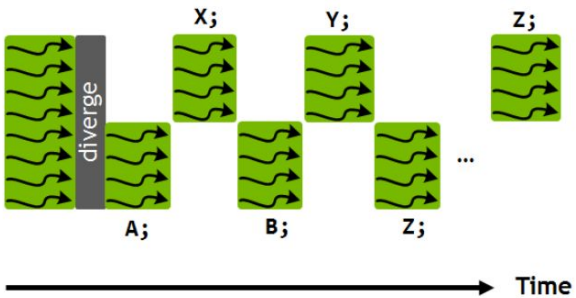


Shared Memory / L1 cache - Threads within the same block can communicate and share data through shared memory, which is faster than global memory but has a smaller capacity.

SIMT

SIMT (Single Instruction, Multiple Threads): GPUs use the SIMT execution model, where multiple threads execute the same instruction but on different data. This allows for massive parallelism and efficient use of GPU resources.

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



SIMT can accommodate branching.
Given an *if-else* construct beginning with *if (condition)*, the threads for which *condition==true* will be active when running statements in the *if* clause, and the threads for which *condition==false* will be active when running statements in the *else* clause.
The results should be correct, but the inactive threads will do no useful work while they are waiting for statements in the active clause to complete.

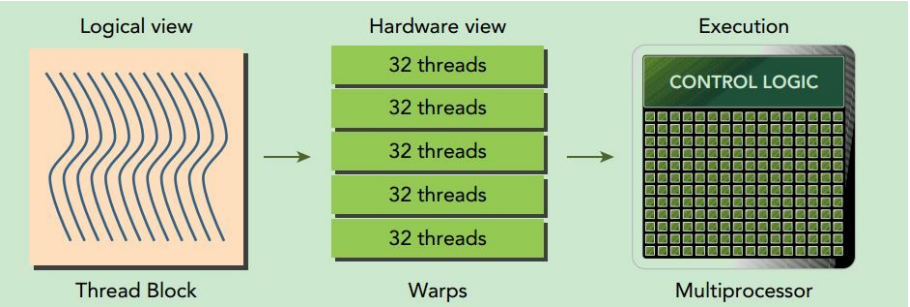
Selected threads can be activated or deactivated, so that instructions and data are processed only on the active threads, while the local data remain unchanged on inactive threads.

https://cvw.cac.cornell.edu/gpu-architecture/gpu-characteristics/simt_warp

Warp

Warp is a group of threads that execute the same instruction at the same time.

At runtime, a block of threads is divided into warps for SIMT execution. The threads in a warp are then processed together by a set of 32 CUDA cores.



SM

```
C = matmul(A, B)
```

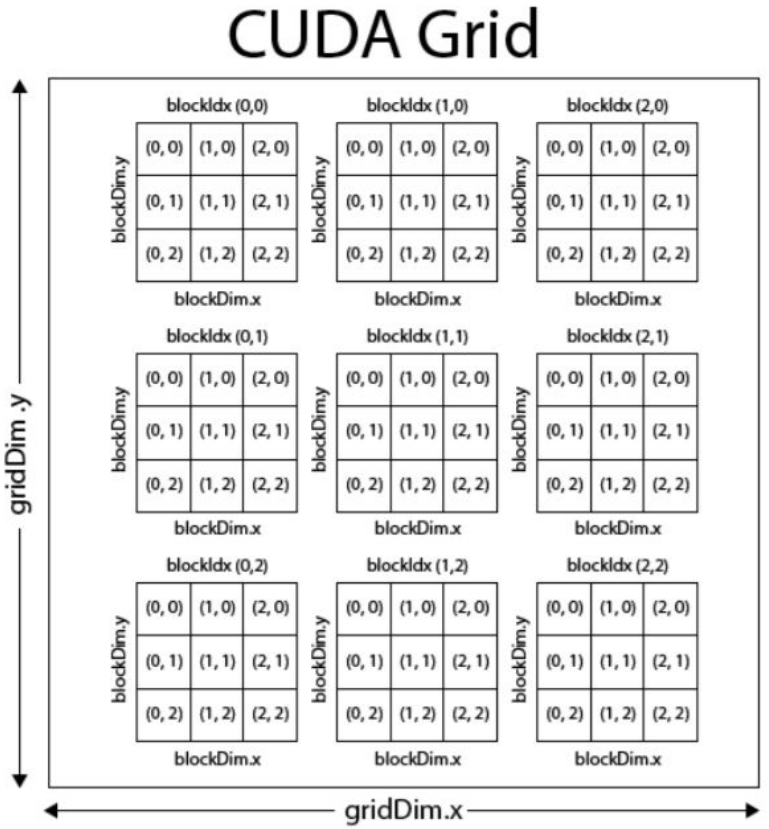
	CUDA Cores				Tensor Cores					
NVIDIA Architecture	FP64	FP32	FP16	INT8	FP64	TF32	FP16	INT8	INT4	INT1
Volta	32	64	128	256			512			
Turing	2	64	128	256			512	1024	2048	8192
Ampere (A100)	32	64	256	256	64	512	1024	2048	4096	16384
Ampere, sparse						1024	2048	4096	8192	

The table shows a single SM's multiply-add operations per clock for various data types on NVIDIA's recent GPU architectures. Each multiply-add comprises two operations, thus one would multiply the throughput in the table by 2 to get FLOP counts per clock.

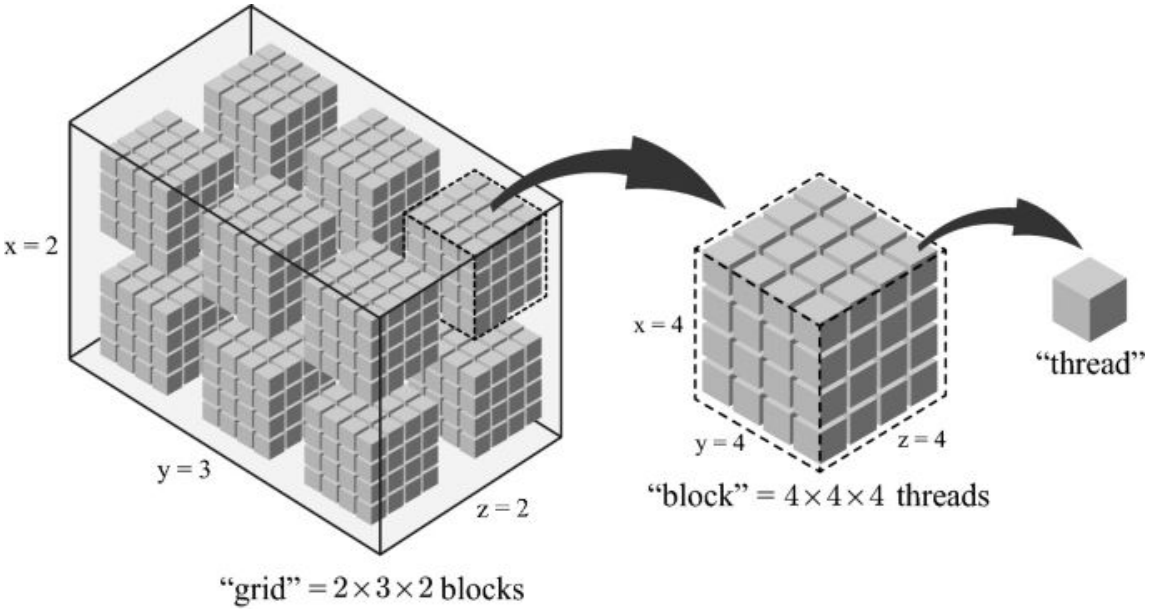
https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf

CUDA Programming: Indexing

- `blockIdx`: The block index within the grid
- `gridDim`: The dimensions of the grid
- `blockDim`: The dimensions of the block
- `threadIdx`: The thread index within the block.

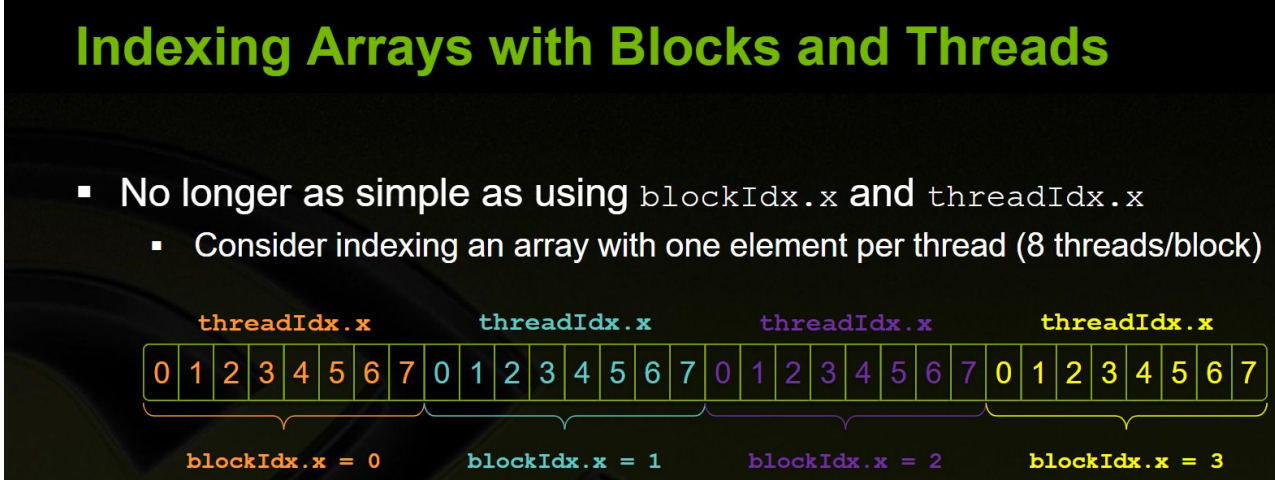


CUDA Programming: Indexing



Vector Addition

Indexing Arrays with Blocks and Threads



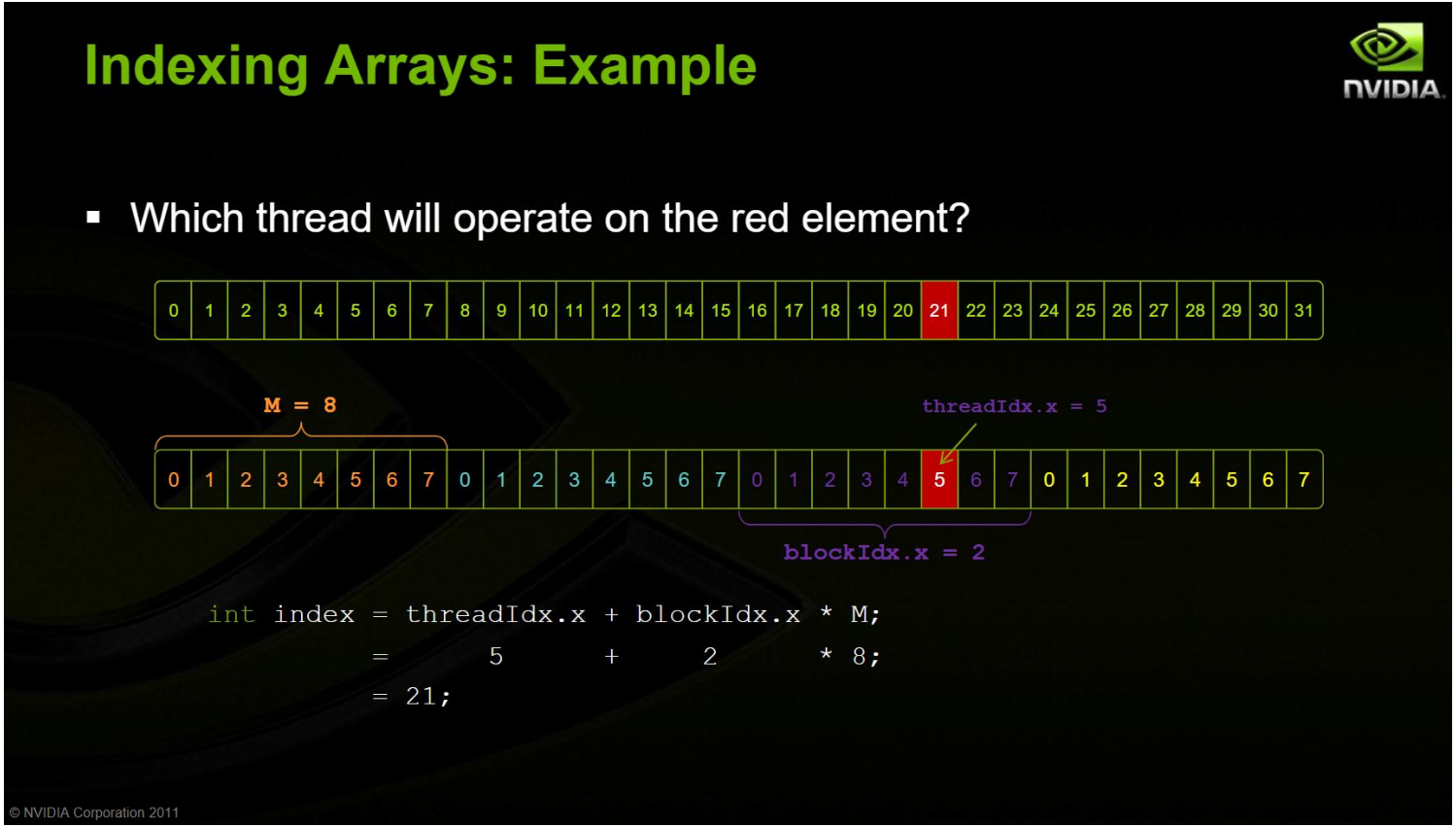
- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

- With M threads per block, a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Vector Addition

Indexing Arrays: Example



- Which thread will operate on the red element?


```
int index = threadIdx.x + blockIdx.x * M;
          =       5       +      2       * 8;
          = 21;
```

Examples of CUDA kernels for Vector Addition

Matrix Multiplication

GEMM - **Ge**neral **M**atrix **M**ultiplication

C = α**AB** + □

cuBLAS is a high-performance linear algebra library provided by NVIDIA for use with CUDA-enabled GPUs.

CUTLASS is an open-source linear algebra library developed by NVIDIA for high-performance computing on GPUs. CUTLASS is primarily written in C++ and leverages CUDA for GPU acceleration.

```
# Row Major
A = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

# how its stored in memory
A = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Column Major
A = [[1, 4, 7],
     [2, 5, 8],
     [3, 6, 9]]

# how its stored in memory
A = [1, 4, 7, 2, 5, 8, 3, 6, 9]
```

Examples of CUDA kernels for Matrix Multiplication

Thank you for your attention!