Parallel Accelerators

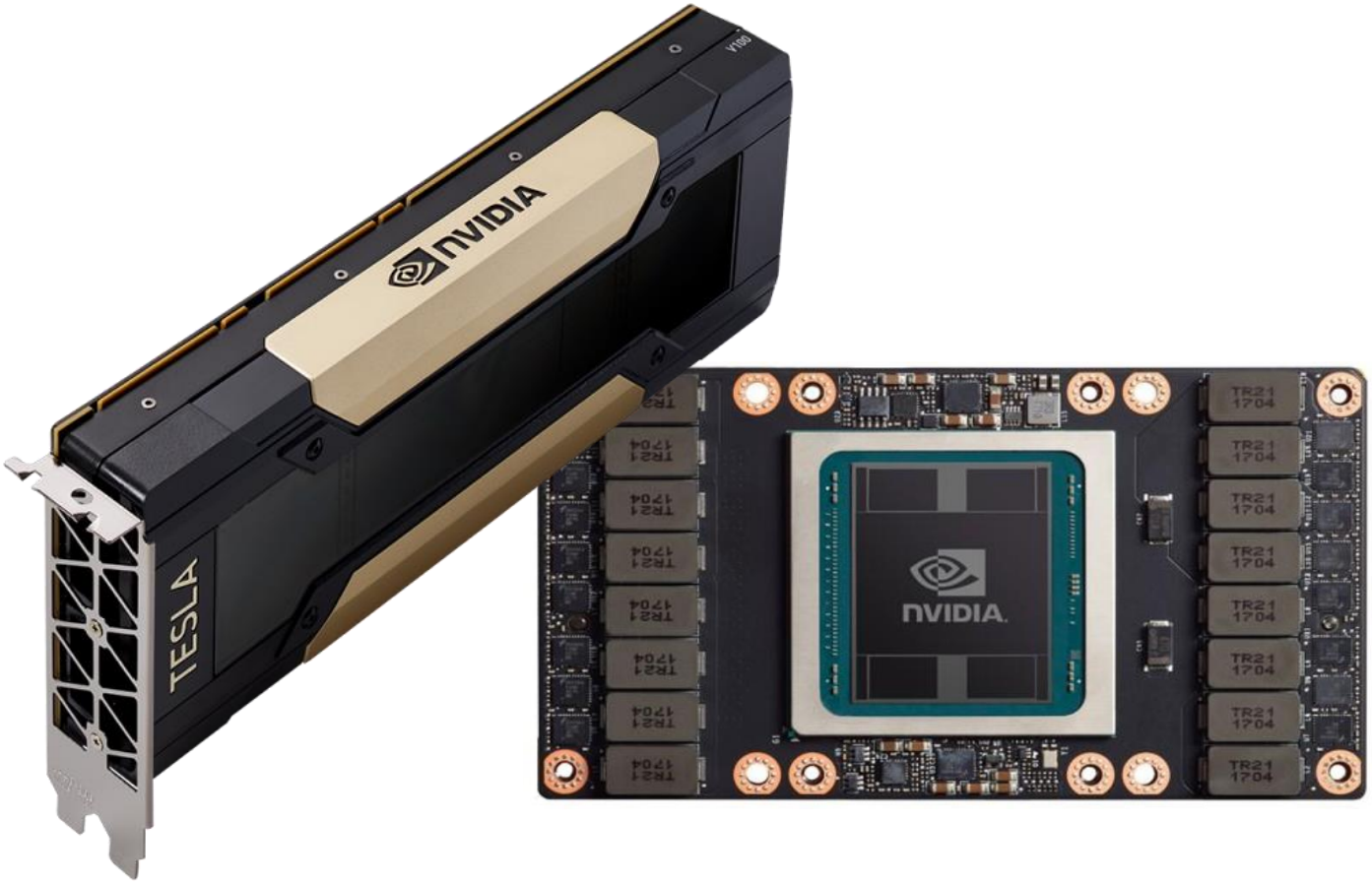
Přemysl Šůcha

``Parallel algorithms", 2017/2018
CTU/FEL

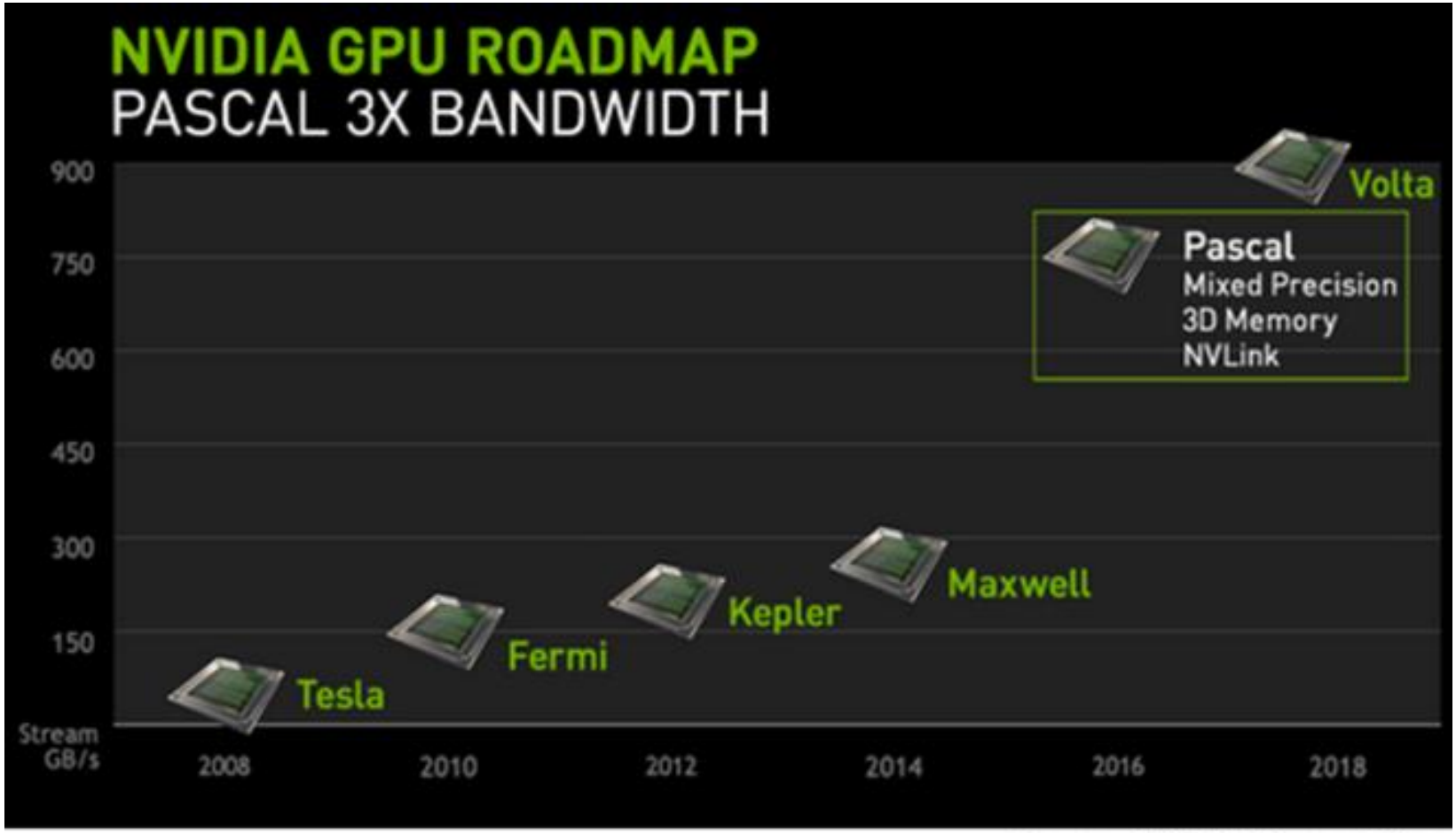
Topic Overview

- Graphical Processing Units (GPU) and CUDA
- Vector addition on CUDA
- Intel Xeon Scalable Processors

Graphical Processing Units



GPU – Nvidia - Roadmap



GPU - Use

- GPU is especially well-suited to address problems that can be expressed as **data-parallel computations**.
- The same program is executed on many data elements in parallel - with high **arithmetic intensity**.
- Applications that process **large data sets** can use a data-parallel programming model to speed up the computations (3D rendering, image processing, video encoding, …)
- Many algorithms **outside the field of image rendering and processing** are accelerated by data-parallel processing too (machine learning, general signal processing, physics simulation, finance, …).

GPU - Overview

- CPU code runs on the **host**, GPU code runs on the **device**.
- A **kernel** consists of multiple threads.
- Threads execute in 32-thread groups called **warps**.
- Threads are grouped into **blocks**.
- A collection of blocks is called a **grid**.

GPU - Hardware Organization Overview

- GPU chip consists of one or more streaming **multiprocessors** (**SM**).
- A multiprocessor consists of 1 (CC 1.x), 2 (CC 2.x), or 4 (CC 3.x, 5.x, 6.x, 7.x) **warp schedulers**. (CC = CUDA Capability)
- Each warp scheduler can issue to 2 (CC 5 and 6) or 1 (CC 7) **dispatch units**.
- A multiprocessor consists of **functional units** of several types.

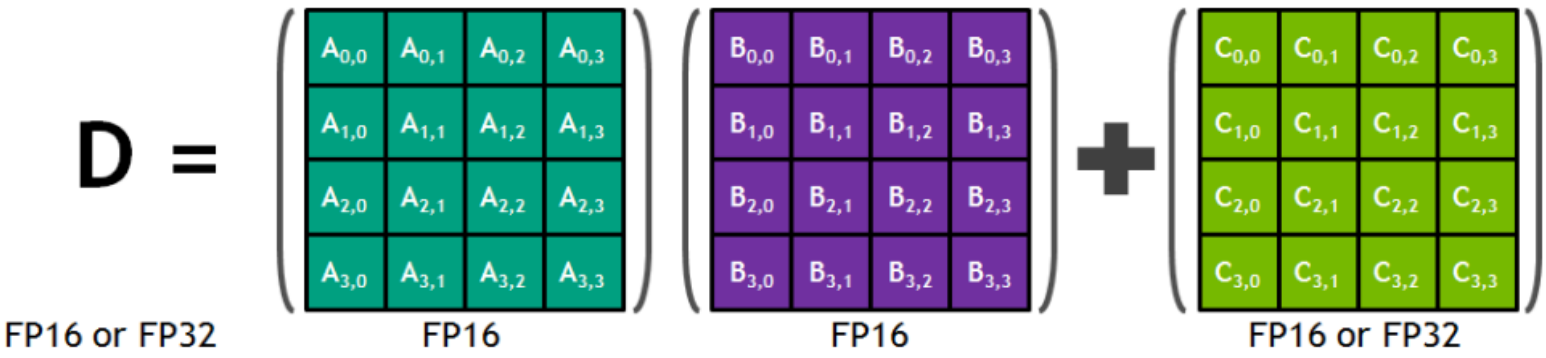
Streaming Multiprocessor (SM) - Volta

GPU - Functional Units

- **INT, FP32 (CUDA Core)** - functional units that executes most types of instructions, including most integer and single precision floating point instructions.
- **FP64 (Double Precision)** - executes double-precision floating point instructions.
- **SFU (Special Functional Unit)** - executes reciprocal and transcendental instructions such as sine, cosine, and reciprocal square root.
- **LD/ST (Load/Store Unit)** – handles load and store instructions.
- **TENSOR CORE** – for deep learning matrix arithmetic.

GPU - Tensor Core

- V100 GPU contains 640 Tensor Cores: eight (8) per SM.
- Tensor Core performs 64 floating point FMA (fused multiply–add) operations per clock.
- Matrix-Matrix multiplication (GEMM) operations are at the core of neural network training and inferencing.
- Each Tensor Core operates on a 4x4 matrices.



Streaming Multiprocessor (SM)

- Each SM has a set of temporary **registers** split amongst threads.
- Instructions can access high-speed **shared memory**.
- Instructions can access a cache-backed **constant space**.
- Instructions can access **local memory**.
- Instructions can access **global space**. (very slow in general)

GPU Architecture - Volta

GPU Architectures

Tesla Product	**Tesla K40**	**Tesla M40**	**Tesla P100**	**Tesla V100**
Manufacturing Process	28nm	28nm	16nm	12nm
Transistors	7.1 Billion	8.0 Billion	15.3 Billion	21.1 Billion
SMs / GPU	15	24	28	40
F32 Cores / SM	192	128	64	64
F32 Cores / GPU	2880	3072	3584	5120
Peak FP32 TFLOPS	5	6.8	10.6	15.7
Peak FP64 TFLOPS	1.7	0,21	5.3	7.8
GPU Boost Clock	810/870 MHz	1114 MHz	1480 MHz	1530 MHz
TDP	235 W	250 W	300 W	300 W
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Maximum TDP	244W	250W	250W	300W

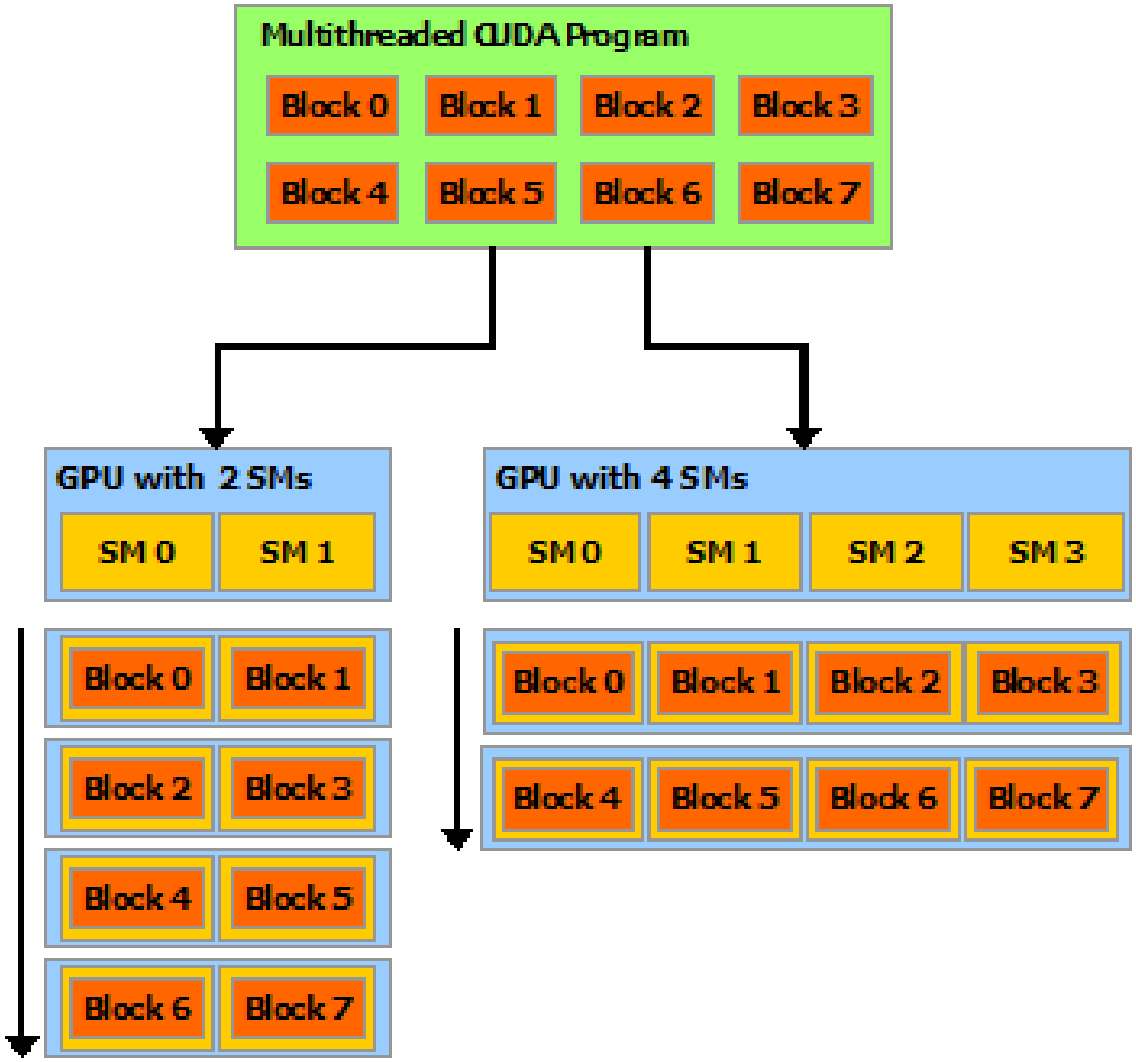
Single-Instruction, Multiple-Thread

- SIMT is an execution model where single instruction, multiple data (**SIMD**) **is combined with multithreading**.
- The SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called **warps**.
- A warp start together at the same program address, but they have their **own instruction address counter** and **register state** and are therefore **free to branch** and **execute independently**.

CUDA

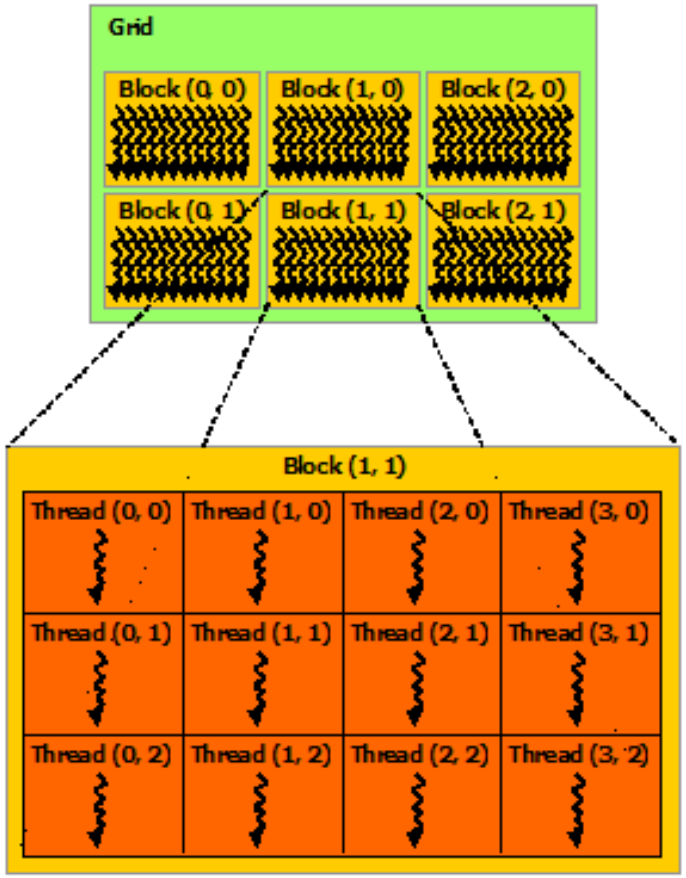
- The NVIDIA GPU architecture is built around a **scalable** array of multithreaded Streaming Multiprocessors (SMs).
- CUDA (Compute Unified Device Architecture) provides a way how a CUDA **program can be executed on any number of SMs**.
- A multithreaded program is partitioned into **blocks** of threads that execute independently from each other.
- A GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

CUDA


Multithreaded CUDA Program
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 2 SMs
SM 0
SM 1
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 4 SMs
SM 0
SM 1
SM 2
SM 3
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7

Grid/Block/Thread

- threads can be identified using a 1-D, 2-D, or 3-D thread index, forming a 1-D, 2-D, or 3-D block of threads, called a **thread block**.
- Blocks are organized into a 1-D, 2-D, or 3-D **grid** of **thread blocks**.



2-D grid with 2-D thread blocks

Kernel

- CUDA C extends C by allowing the programmer to define C functions, called **kernels**.

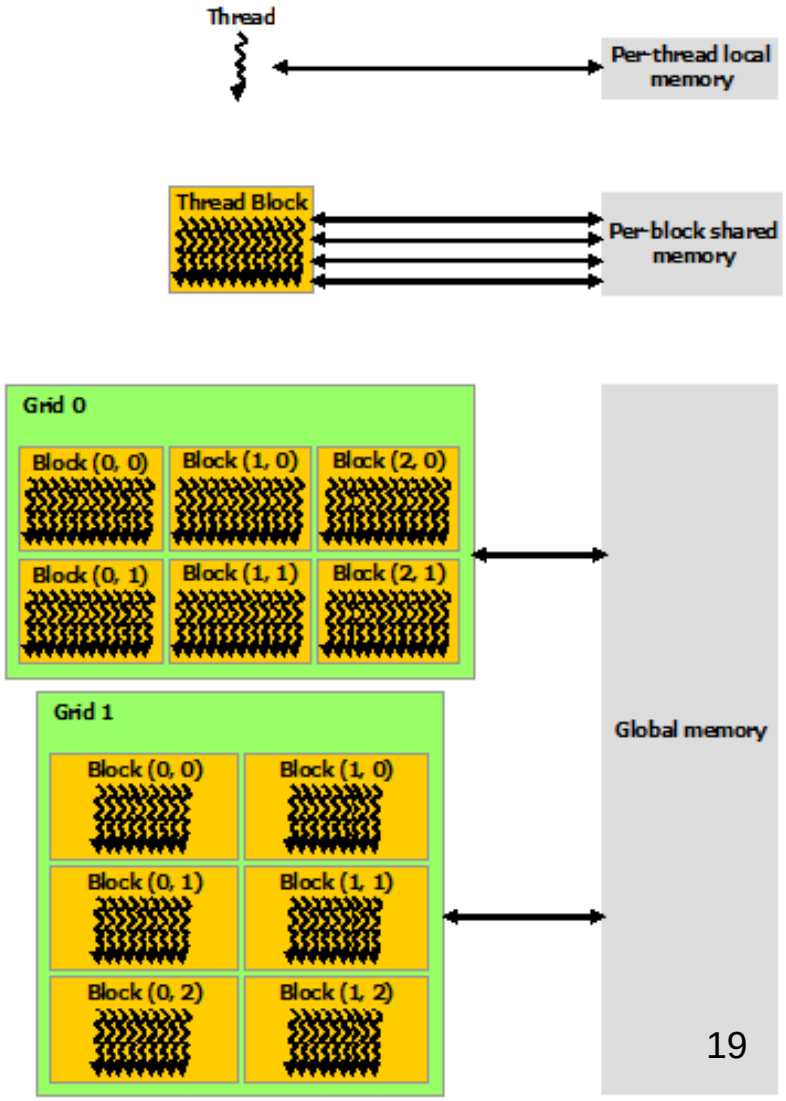
```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}

int main()
{ ...
        // Kernel invocation with N threads inside 1 thread block
        VecAdd<<<1, N>>>(A, B, C);
}
```

- *threadIdx* is a 3-component vector, so that threads can be identified using a 1-D, 2-D, or 3-D **thread index**.

Memory Hierarchy

- Each thread has private set of **registers** and **local memory**.
- Each thread block has **shared memory** visible to all threads of the block.
- All threads have access to the same **global memory**.
- There are also two additional read-only memory spaces accessible by all threads (**constant** and **texture memory**).



GPU Programming - Example

- Element by element vector addition

[1] NVIDIA Corporation, *CUDA Toolkit Documentation v9.0.176*, 2017.

Element by element vector addition

```
/* Host main routine */
int main(void)
{
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    // Allocate the host input vectors A and B and output vector C
    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }
```

Element by element vector addition

```
// Allocate the device input vectors A and B and output vector C
float *d_A = NULL;
cudaMalloc((void **)&d_A, size);
float *d_B = NULL;
cudaMalloc((void **)&d_B, size);
float *d_C = NULL;
cudaMalloc((void **)&d_C, size);


// Copy the host input vectors A and B in host memory to the device
// input vectors in device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Element by element vector addition

```
// Launch the Vector Add CUDA Kernel
int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

// Copy the device result vector in device memory to the host result vector
// in host memory.
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

Element by element vector addition

```
    // Free device global memory
    err = cudaFree(d_A);
    err = cudaFree(d_B);
    err = cudaFree(d_C);

    // Free host memory
    free(h_A);
    free(h_B);
    free(h_C);

return 0;
}
```

Element by element vector addition

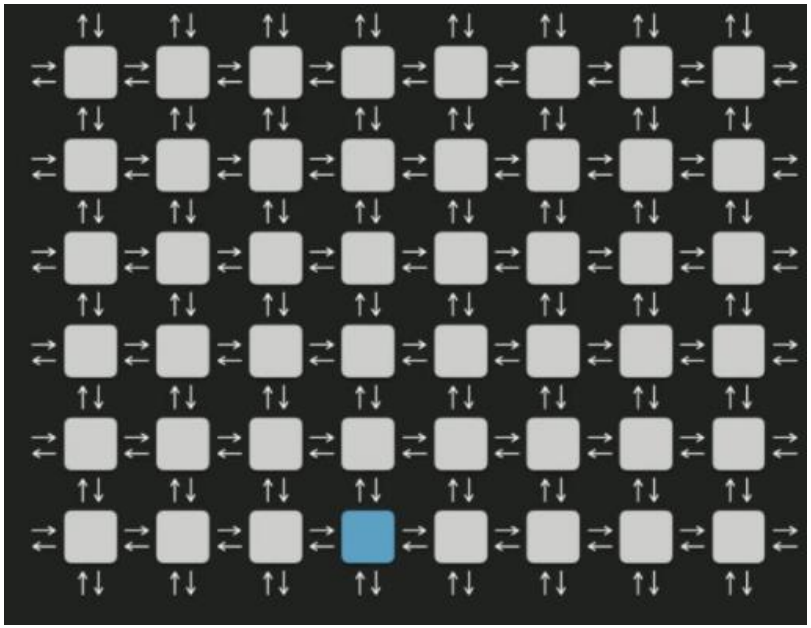
```
/**
 * CUDA Kernel Device code
 *
 * Computes the vector addition of A and B into C. The 3 vectors have the same
 * number of elements numElements.
 */
__global__ void vectorAdd(float *A, float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```

Tesla's AI chip

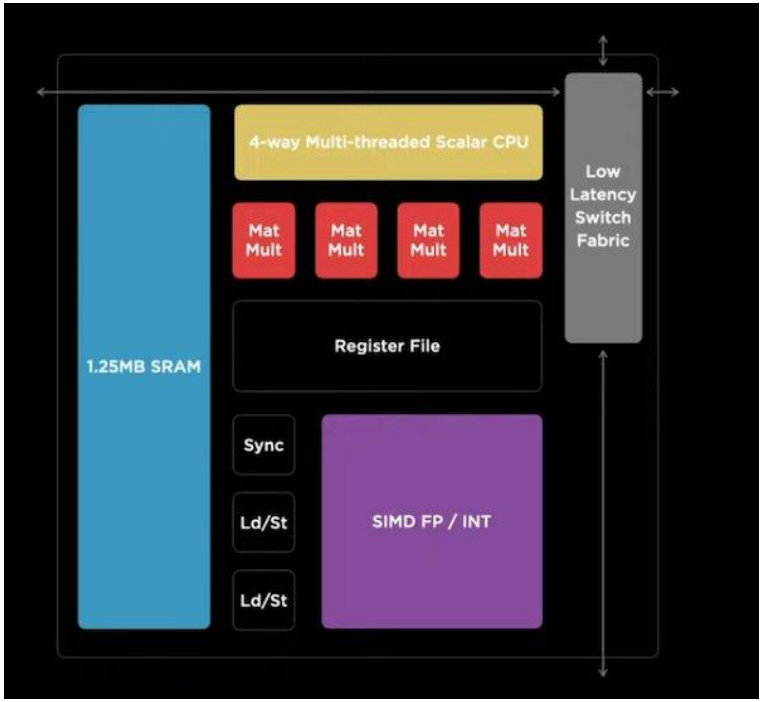
Dojo architecture

- The achitecture is focused on ML
- Dojo is based on a custom computing chip, the D1 chip, which is the building block of a large multi-chip module (MCM)-based compute plane
- Training nodes are interconned via a 2D mesh

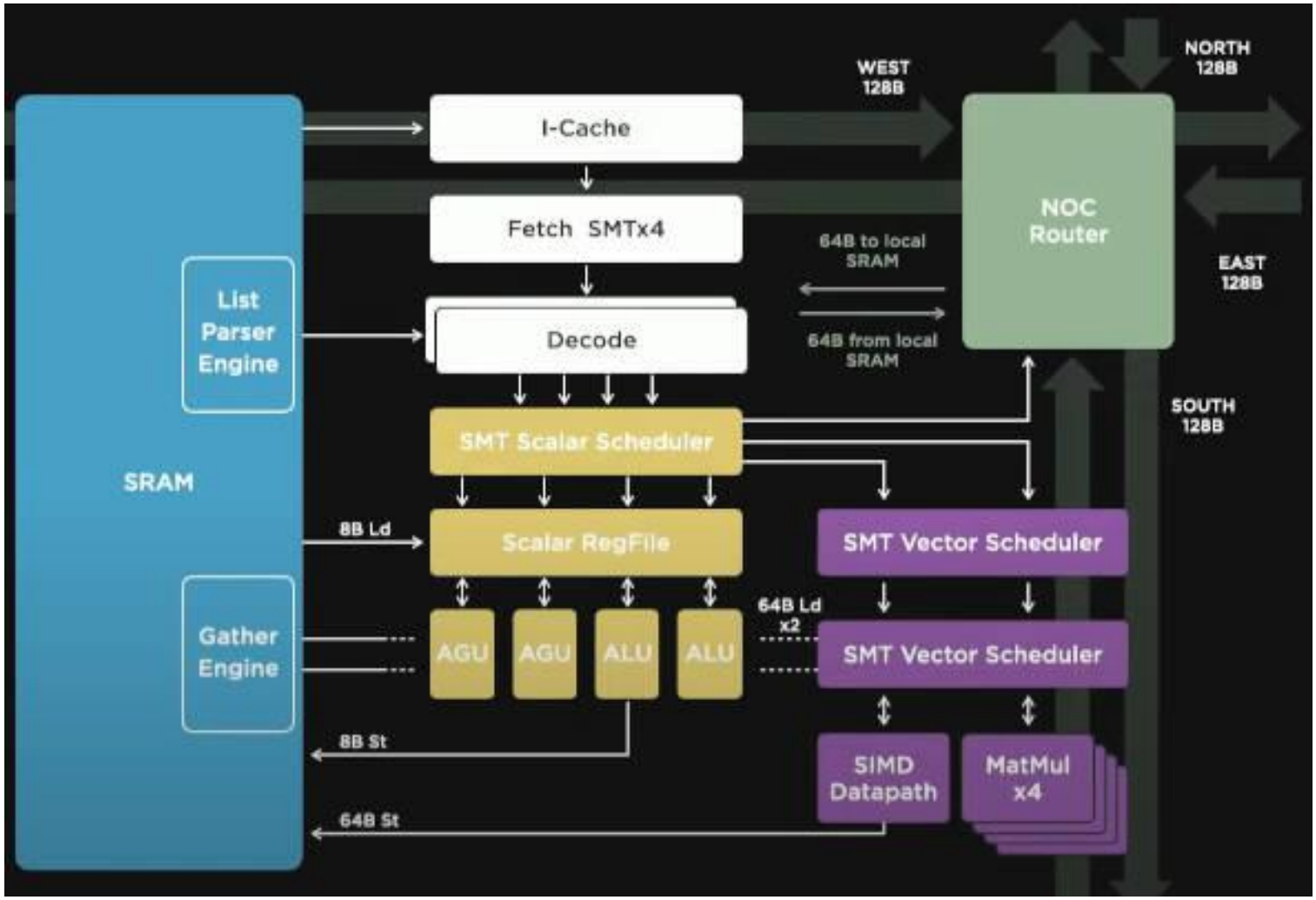


Training node

- The smallest entity is a training node
- 64-bit CPU (2GHz) fully optimized for machine learning workloads
- 1.25 MB of fast ECC-protected SRAM
- optimized matrix multiply units and SIMD instructions (FP32, BFP16, CFP8, Int32, Int16, Int8)
- 1 teraflop of ML compute
- The training node has a modular design

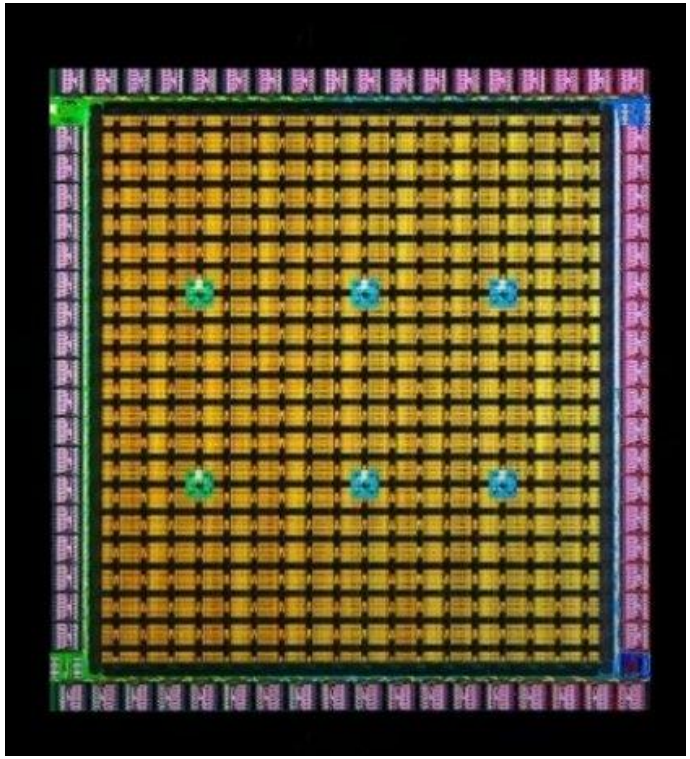


Training node



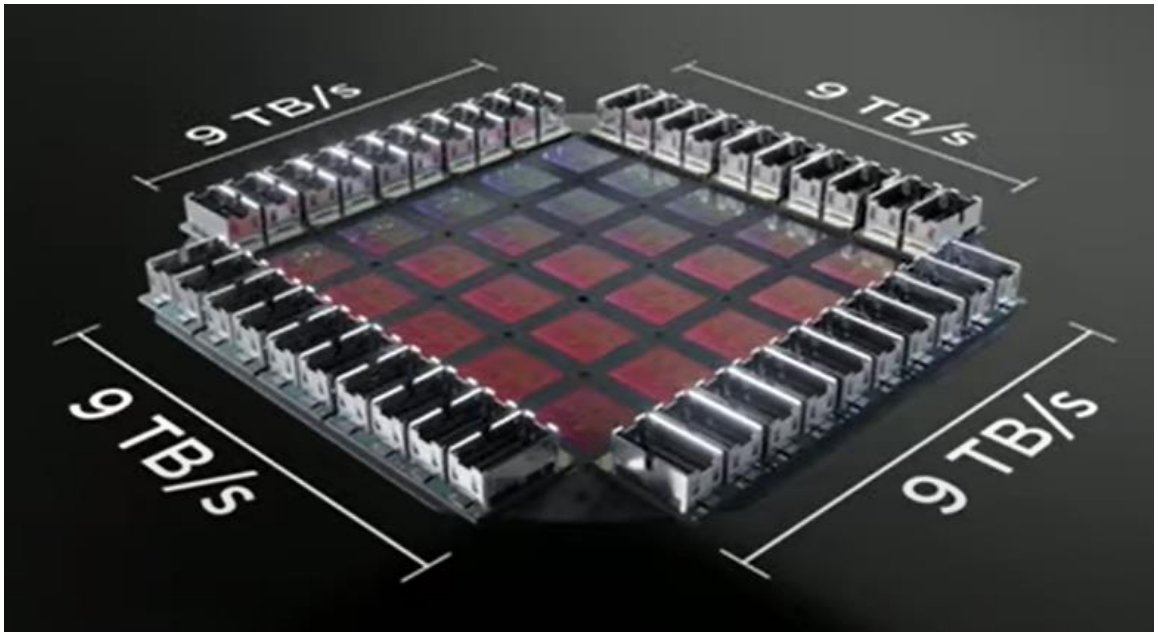
The D1 Chip

- created by an array of 354 training nodes
- 362 teraflops of machine learning compute
- the bandwidth for the communications between the training nodes (on-chip bandwidt) is 10 TBps (4 TBps off-chip).
- D1 is manufactured in 7 nm technology
- The thermal design power (TDP) of the chip is 400 W.

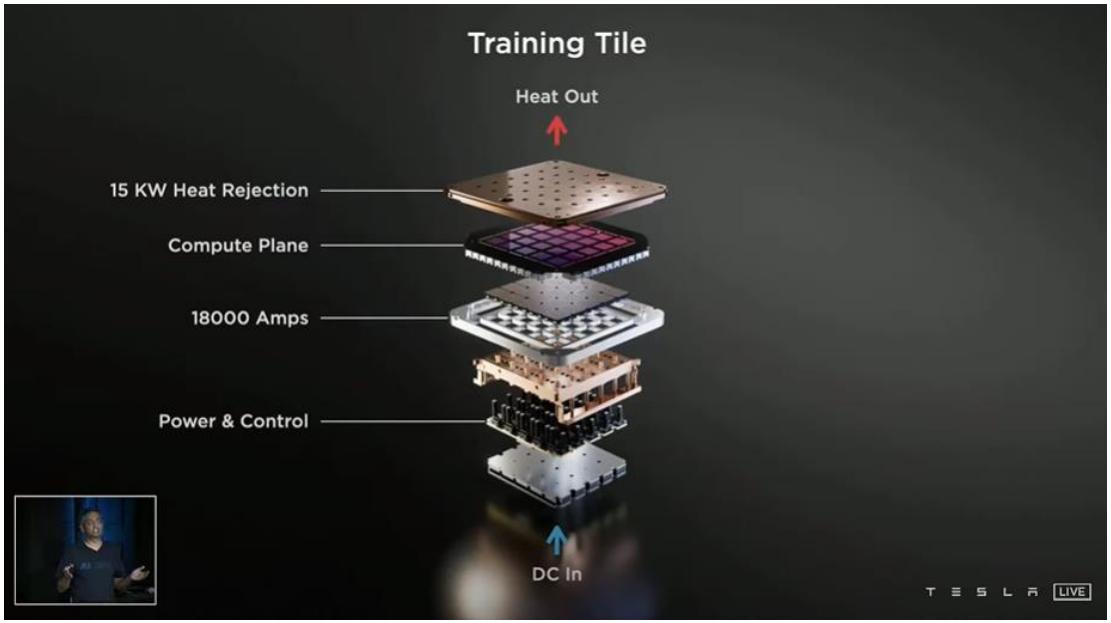


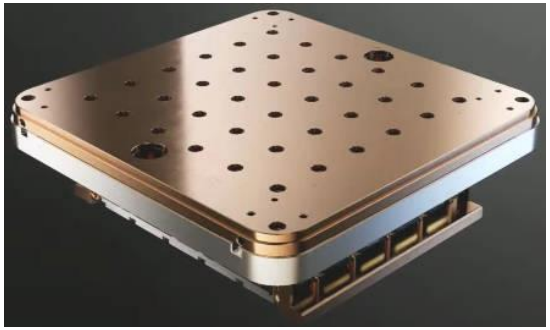
Dojo's Training Tiles

- consisting of 25 D1 chips (5 x 5)
- bandwidth between the dies is preserved
- D1 chips can consume 10 kW
- 9 PFLOPs



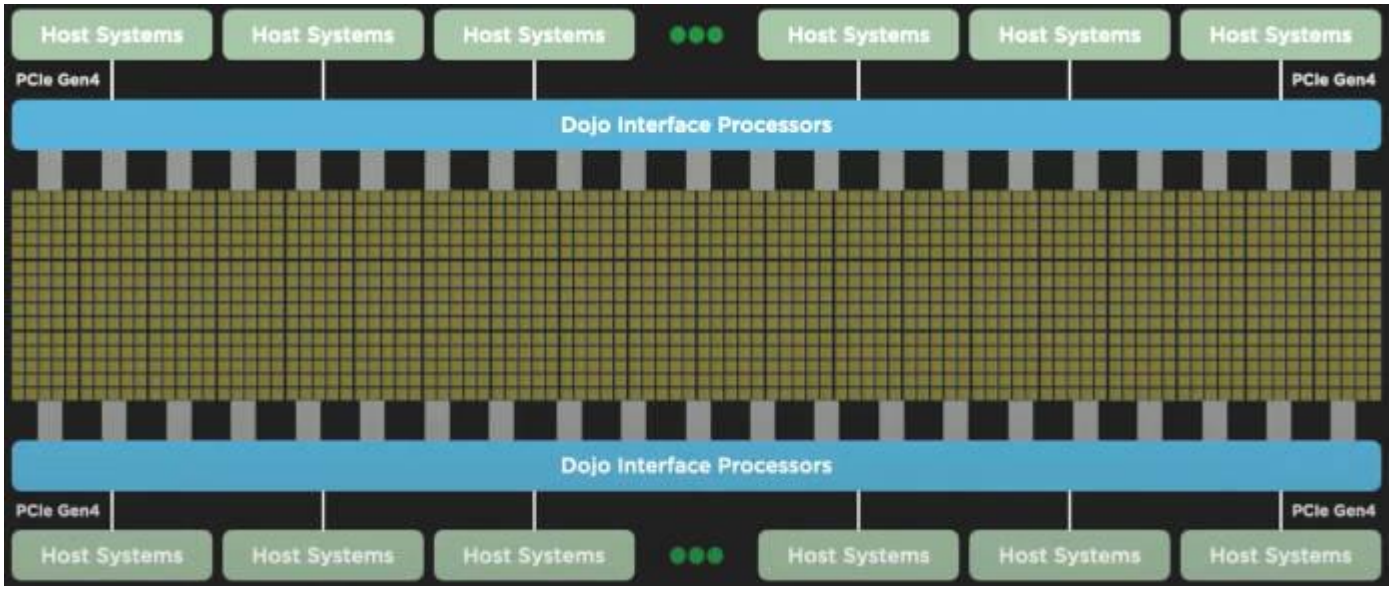
Dojo's Training Tiles





AI supercomputer

- Aim is to achieve 1.1EFLOPs in ML



Logical view

- DPU – Dojo Processing Unit
- A virtual device that can be sized according to the application needs

References

- Tesla's AI chip REVEALED! (Project Dojo), presented at Tesla AI Day (https://www.youtube.com/watch?v=DSw3IwsgNnc)