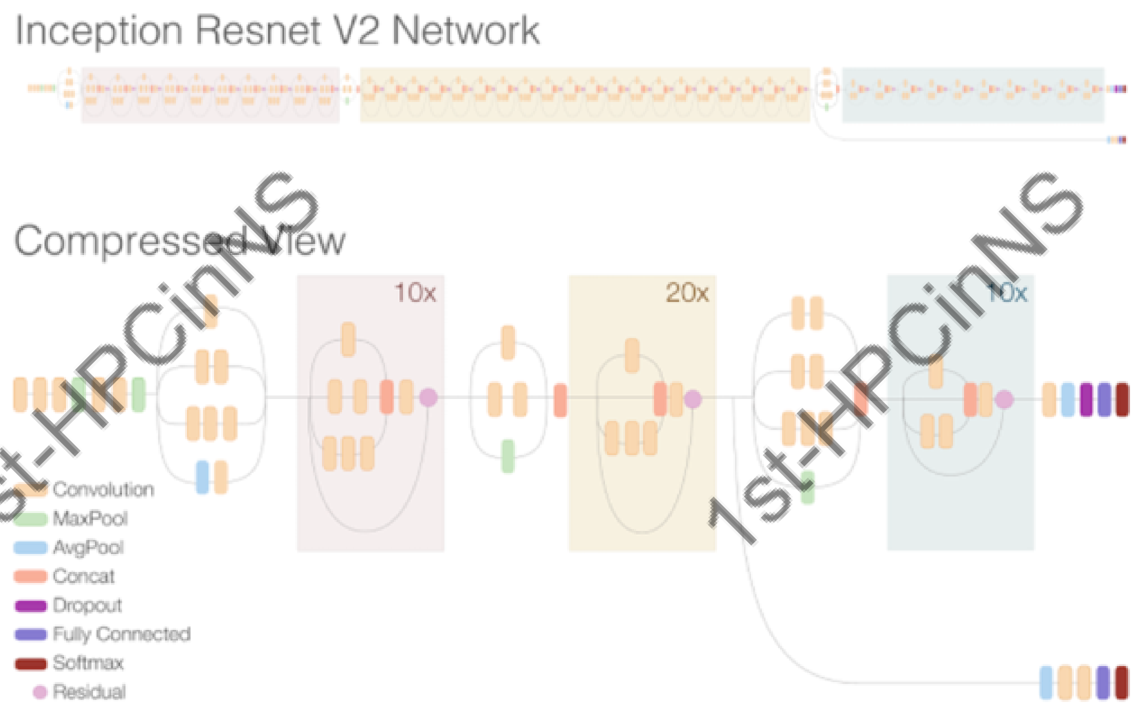
GPUs for Neural Networks

Saeed Reza Kheradpisheh

Need for More Computing Power

- Lots of Data
- Complex Architectures
- Many Models

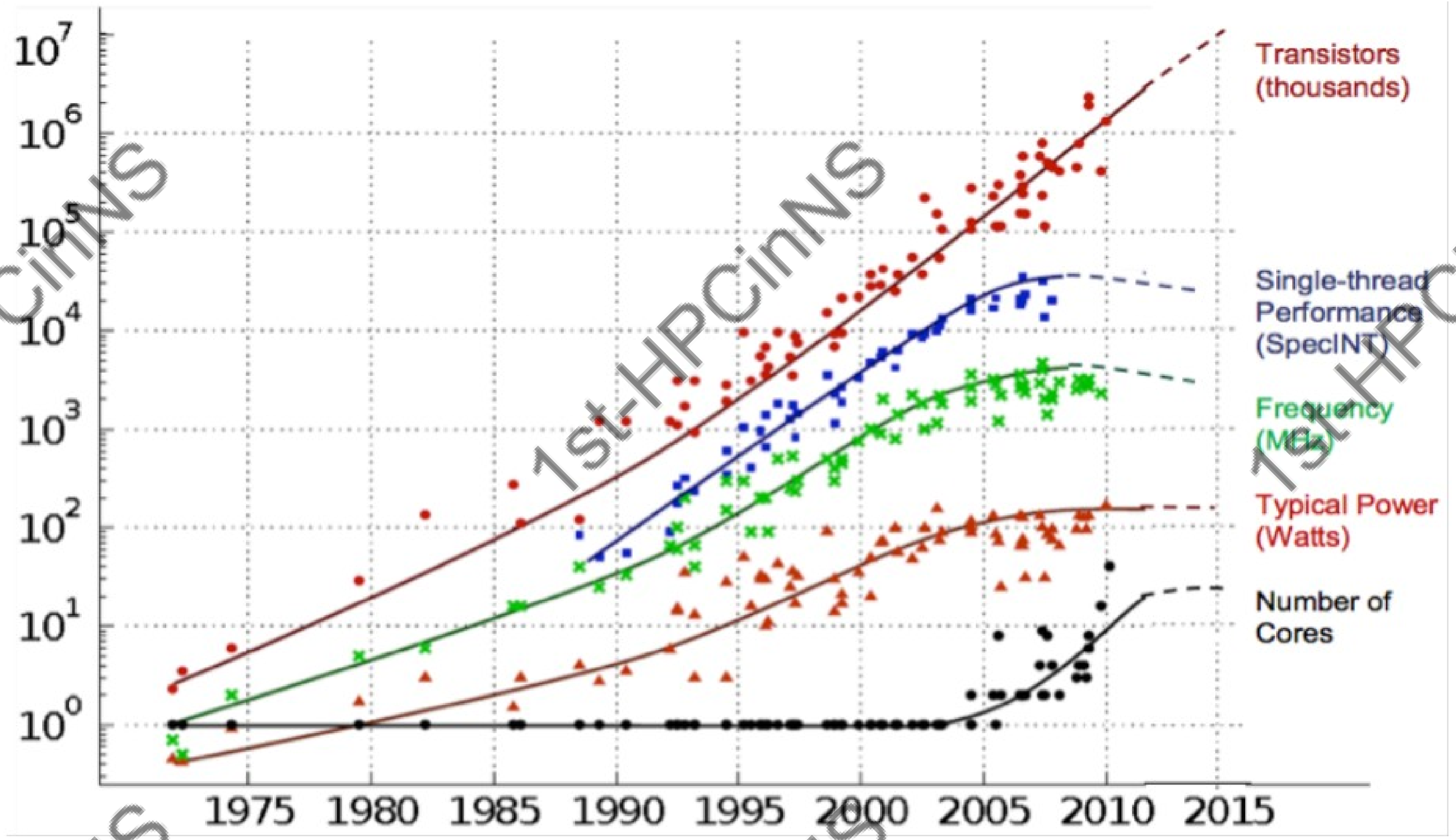


Schematic diagram of Inception-ResNet-v2

Historic Ways for More Compute

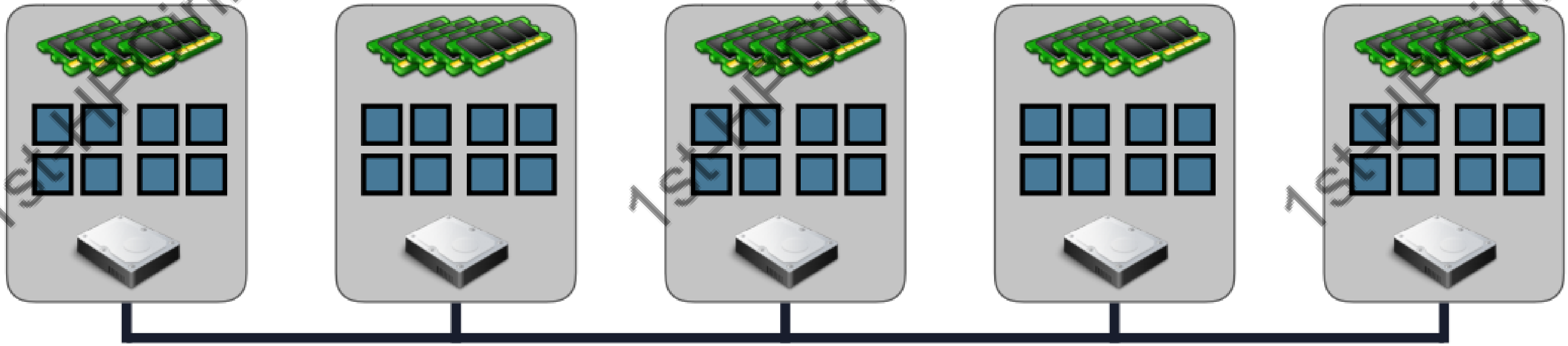
- Faster Clock Rates
- Multi-Core
- Distributed Computing

CPU Trends

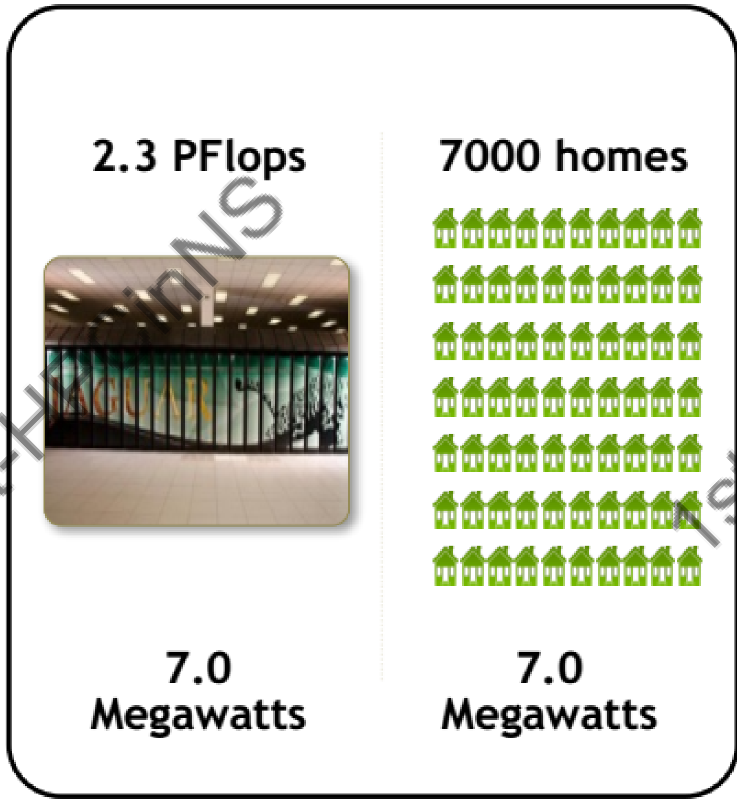


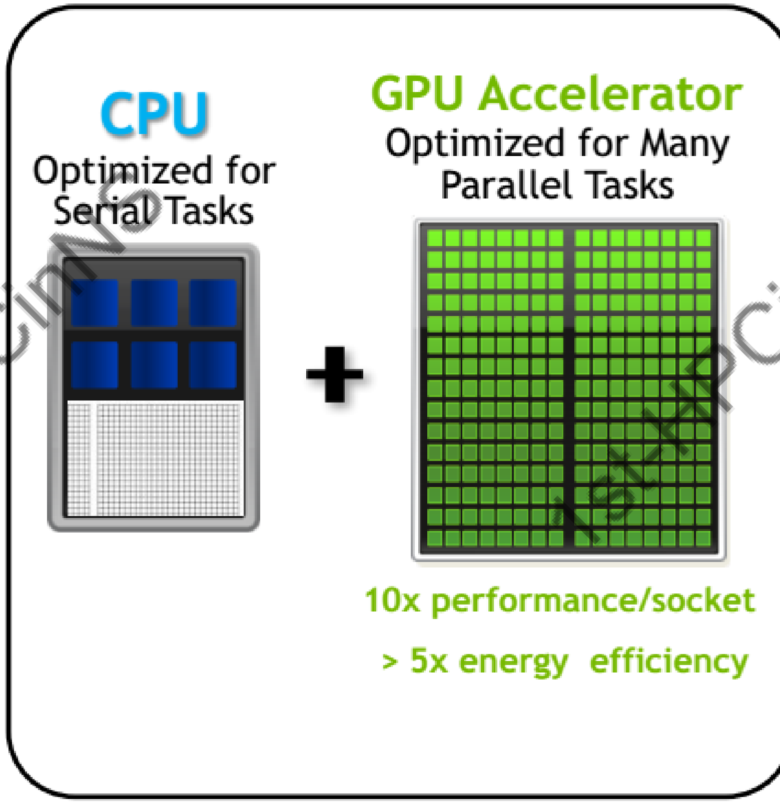
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1975
1980
1985
1990
1995
2000
2005
2010
2015
Transistors (thousands)
Single-thread Performance (SpecINT)
Frequency (MHz)
Typical Power (Watts)
Number of Cores

Distributed Computing



GPU Accelerated Computing

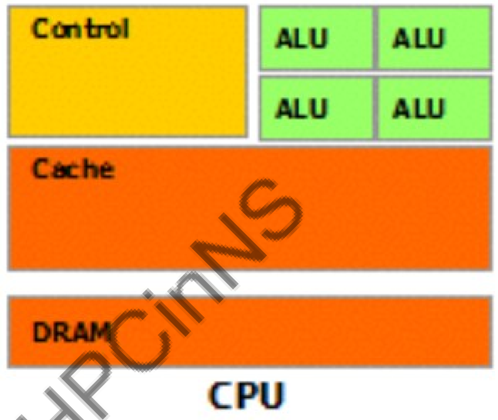




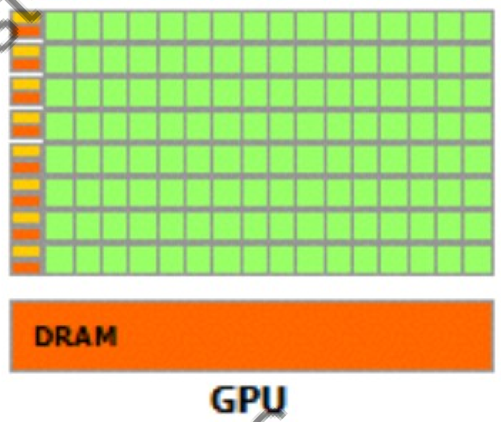
Traditional CPUs are not economically feasible

Era of GPU-accelerated computing is here

CPU v.s. GPU

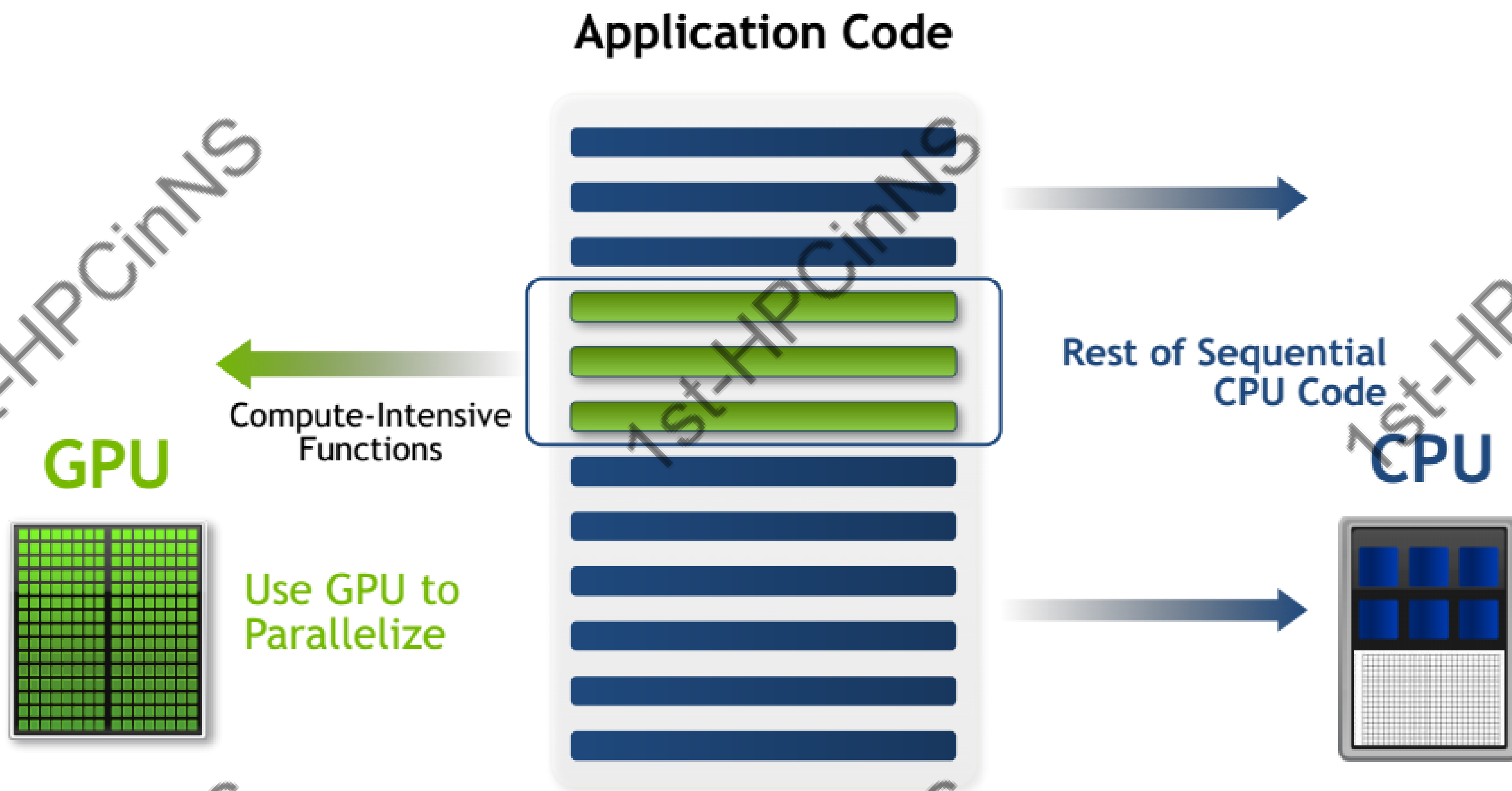

Control
ALU
ALU
ALU
ALU
Cache
DRAM
CPU


Core I7 3960X
Extreme Edition
6
Cores
316
Gigaflops
2.3
Billion
Transistors

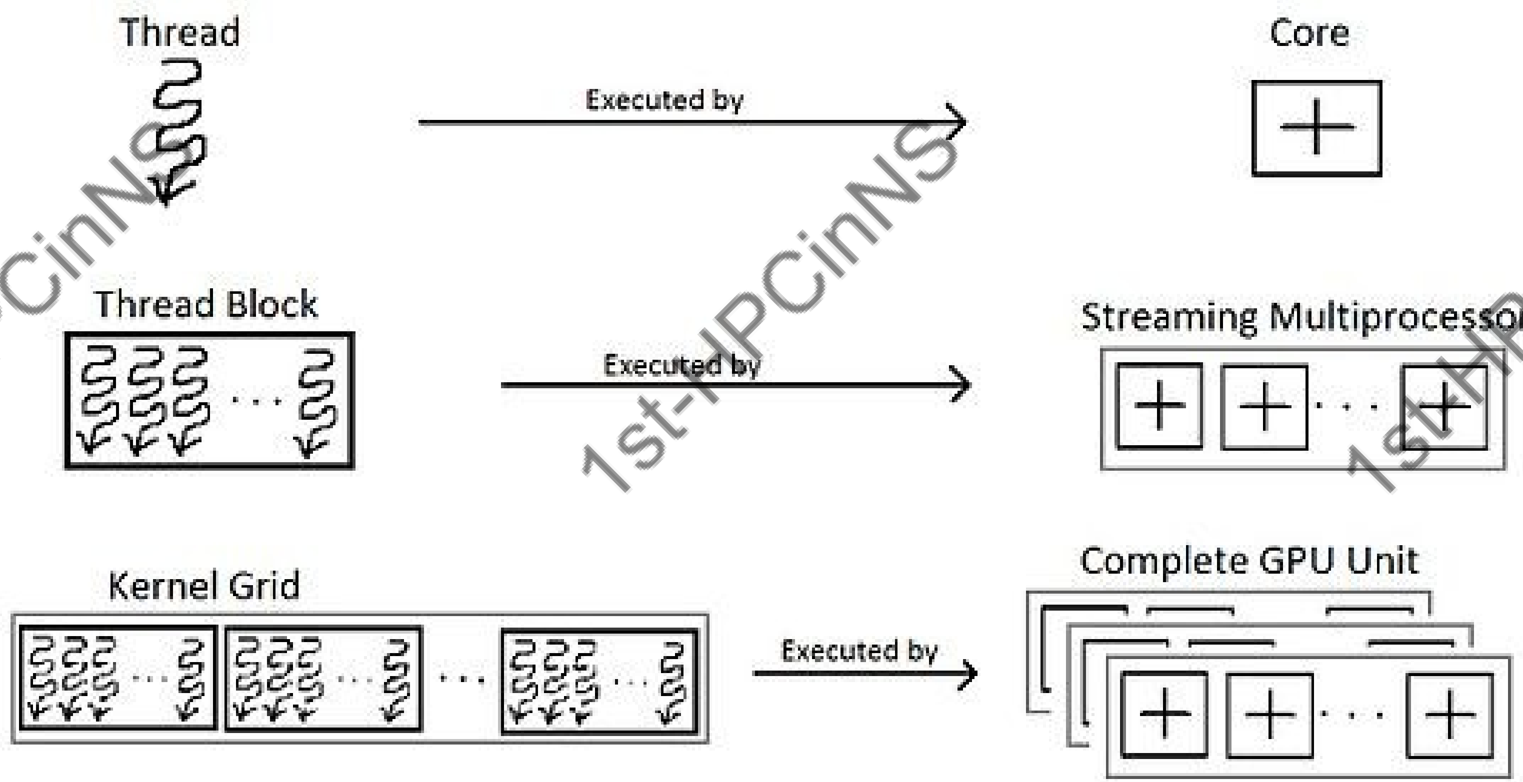

DRAM
GPU


GeForce GTX TITAN
2688
CUDA Cores
4,500
Gigaflops
7.1
Billion
Transistors

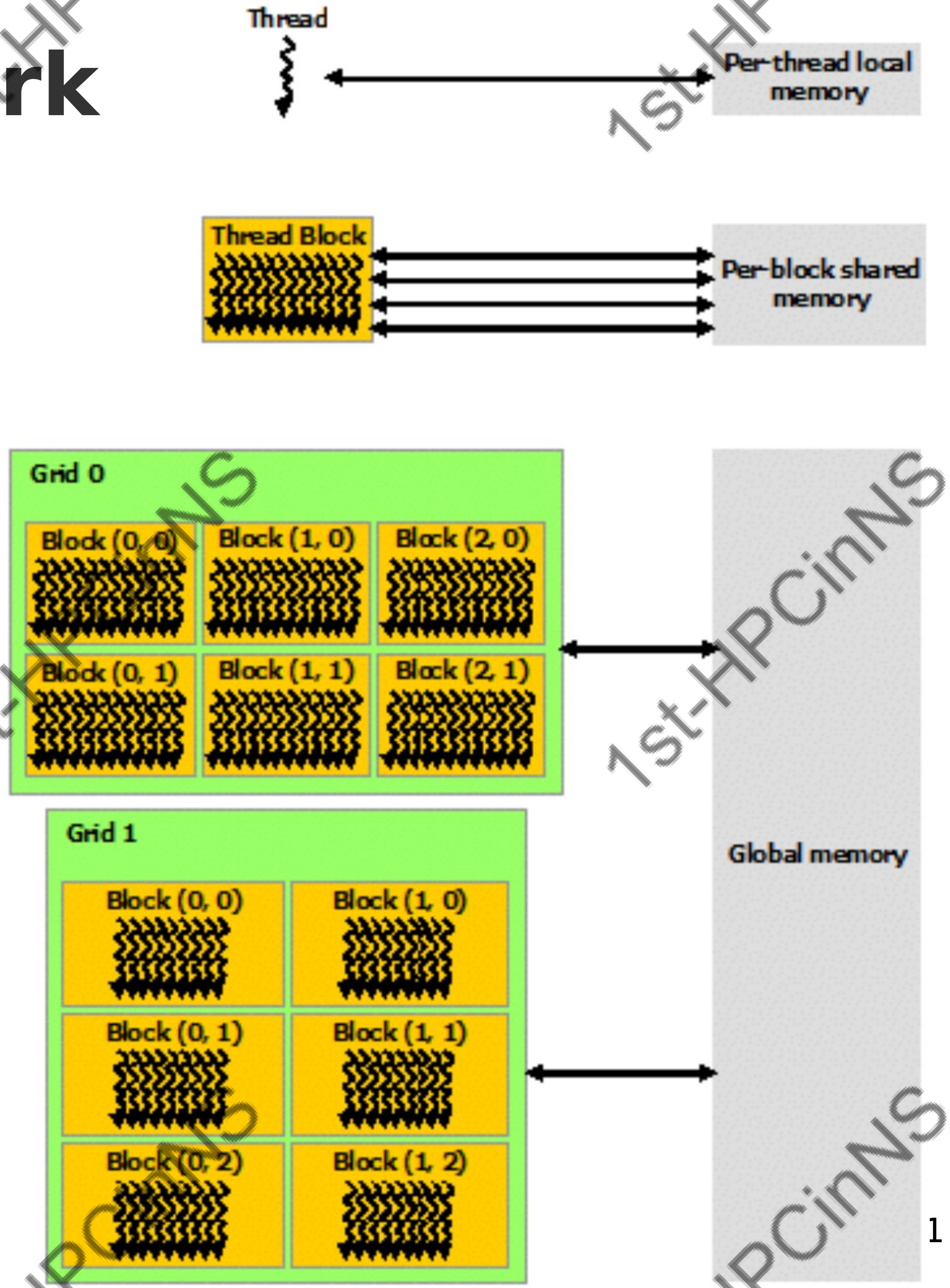
Parallelizable code



CUDA framework



CUDA framework



Matrix Multiplication

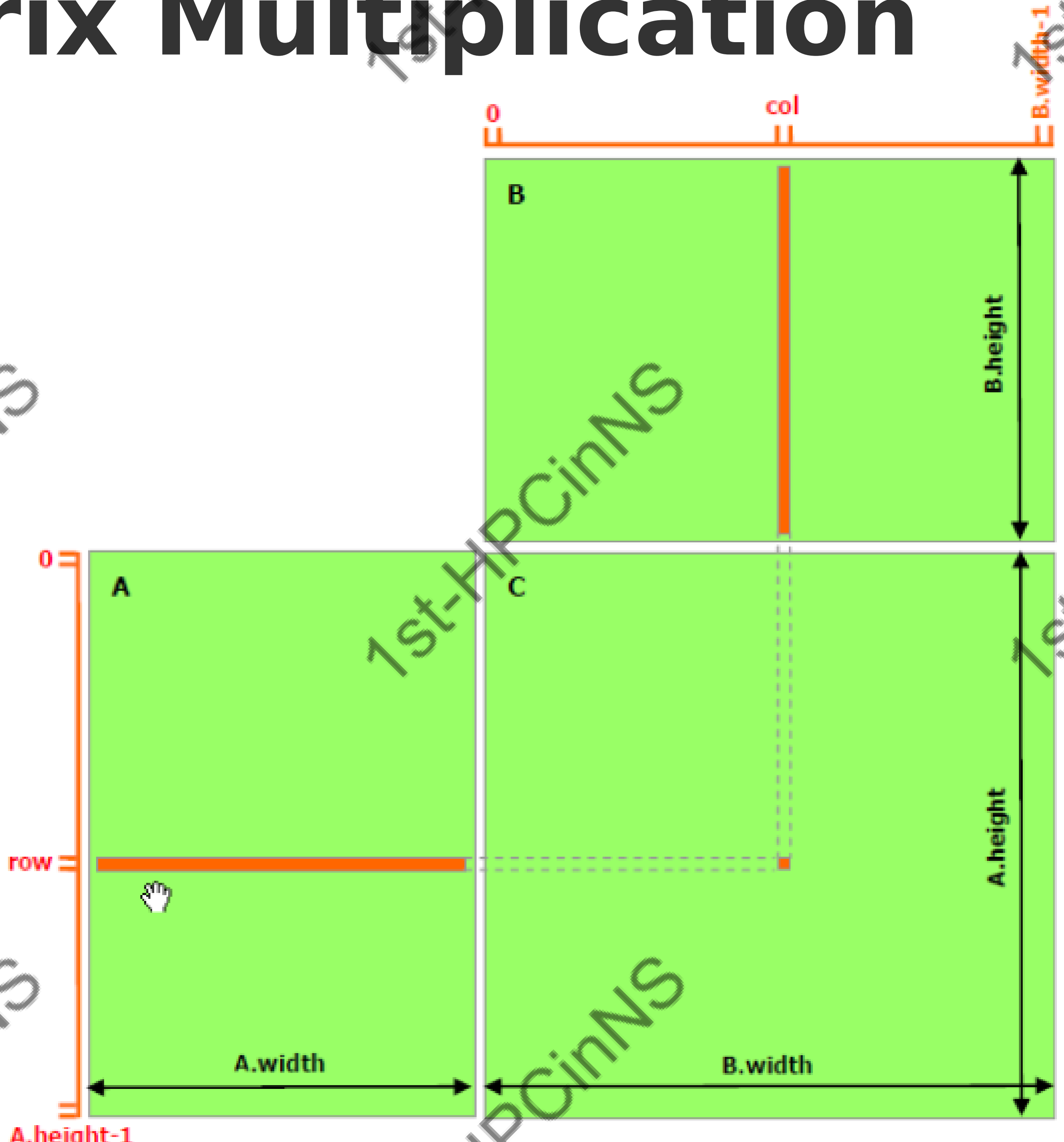
$$\underset{\mathbf{A}}{\begin{bmatrix} \boxed{a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,k}} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,k} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & a_{m,3} & \cdots & a_{m,k} \end{bmatrix}} \underset{\mathbf{B}}{\begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & b_{2,3} & \cdots & b_{2,n} \\ b_{3,1} & b_{3,2} & b_{3,3} & \cdots & b_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{k,1} & b_{k,2} & b_{k,3} & \cdots & b_{k,n} \end{bmatrix}} = \underset{\mathbf{C}}{\begin{bmatrix} \boxed{c_{1,1}} & c_{1,2} & c_{1,3} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & c_{2,3} & \cdots & c_{2,n} \\ c_{3,1} & c_{3,2} & c_{3,3} & \cdots & c_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & c_{m,3} & \cdots & c_{m,n} \end{bmatrix}}$$

Algorithm 1 Matrix Multiplication

```
for row in rows of matrix A do
    for column in columns of matrix B do
        for element in vectors do
            Multiply element-wise
            Add to cumulative sum
        end for
    end for
end for
```

$$c_{i,j} = \sum_{h=1}^{k} a_{i,h} b_{h,j}$$

Matrix Multiplication

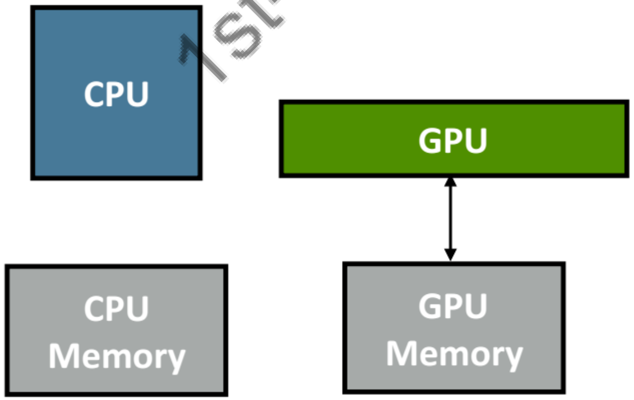


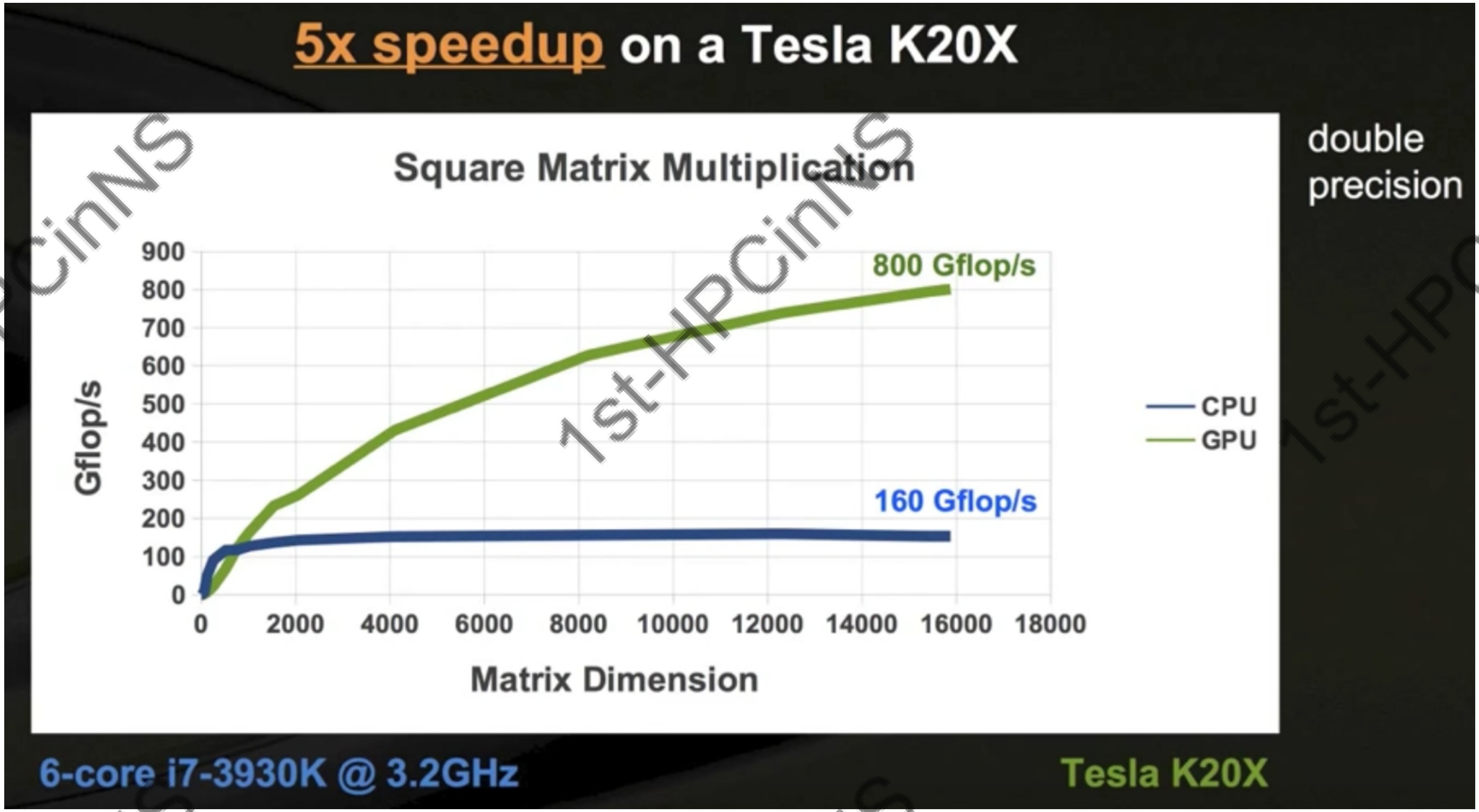
CUDA kernel

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

CPU code

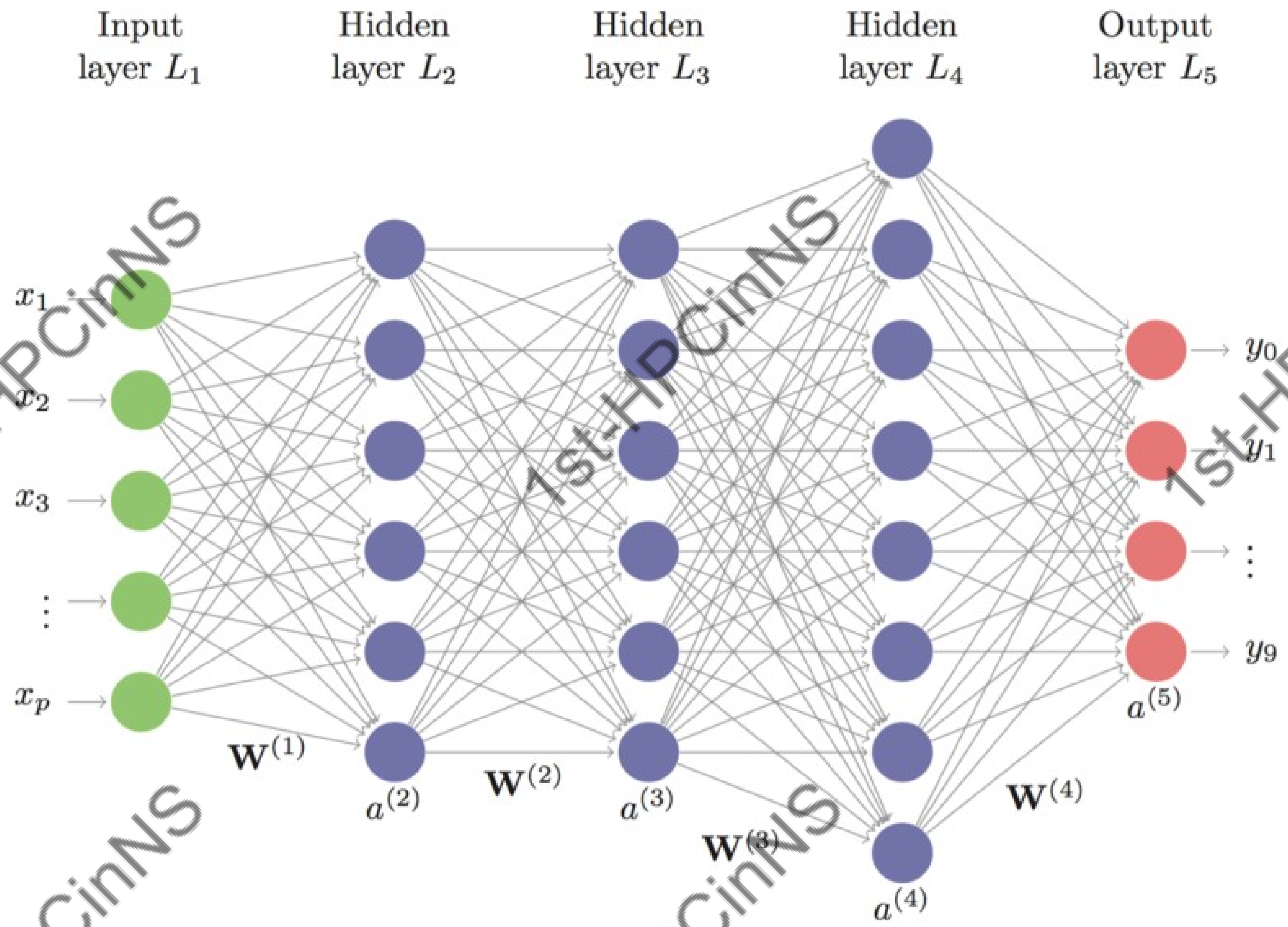
```
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

}
```





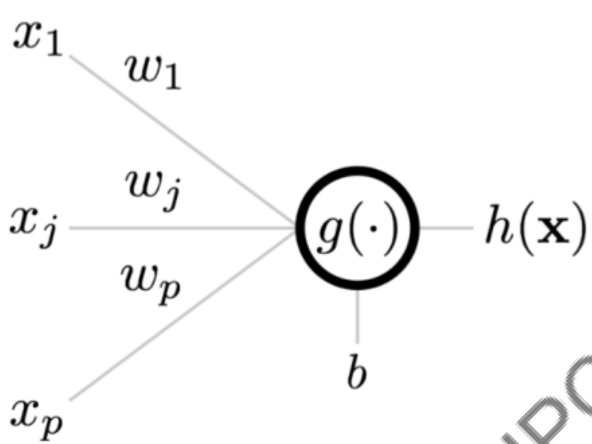
5x speedup on a Tesla K20X
Square Matrix Multiplication
double precision
800 Gflop/s
160 Gflop/s
Gflop/s
900
800
700
600
500
400
300
200
100
0
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
Matrix Dimension
CPU
GPU
6-core i7-3930K @ 3.2GHz
Tesla K20X

Deep learning

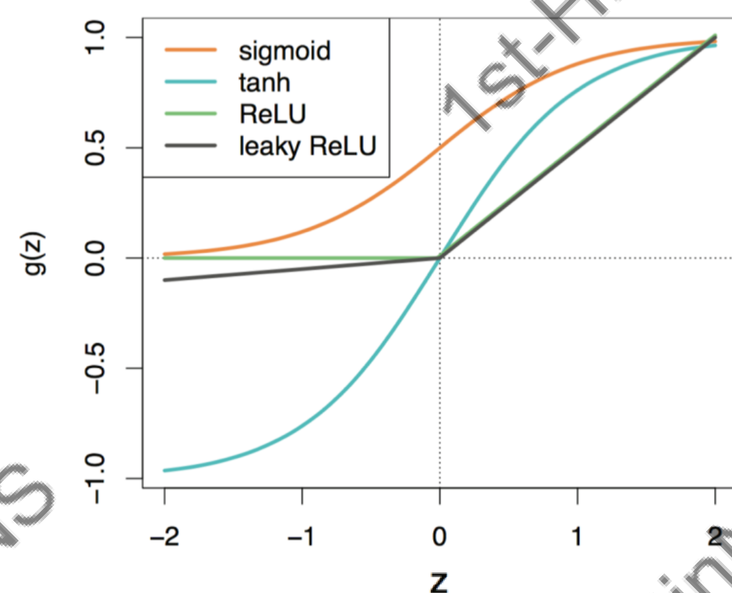


Neuron

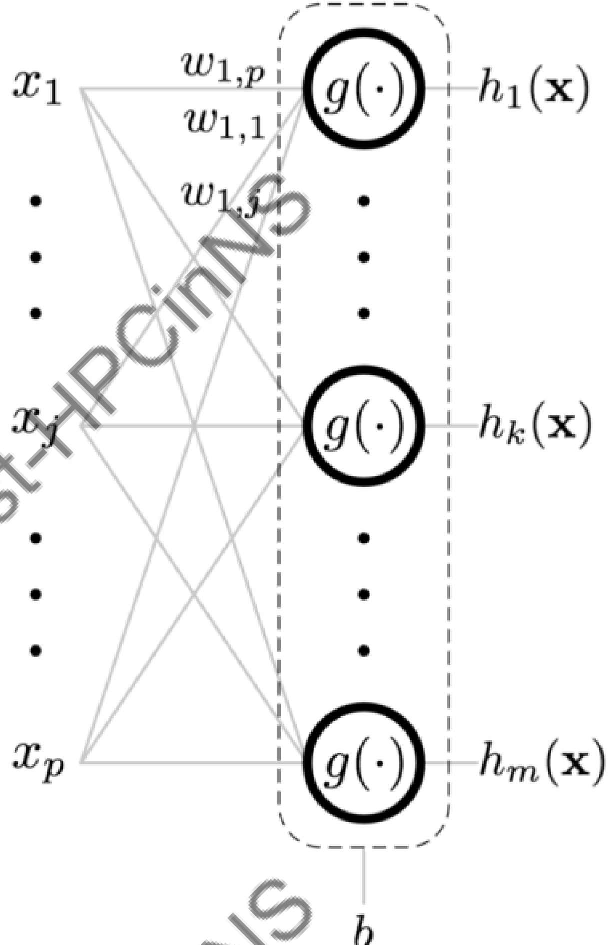
$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_p \end{bmatrix}$$



$$h(\mathbf{x}) = g\left(\sum_{i=1}^{p} w_i x_i + b\right)$$
$$= g(\mathbf{w}^T\mathbf{x} + b)$$

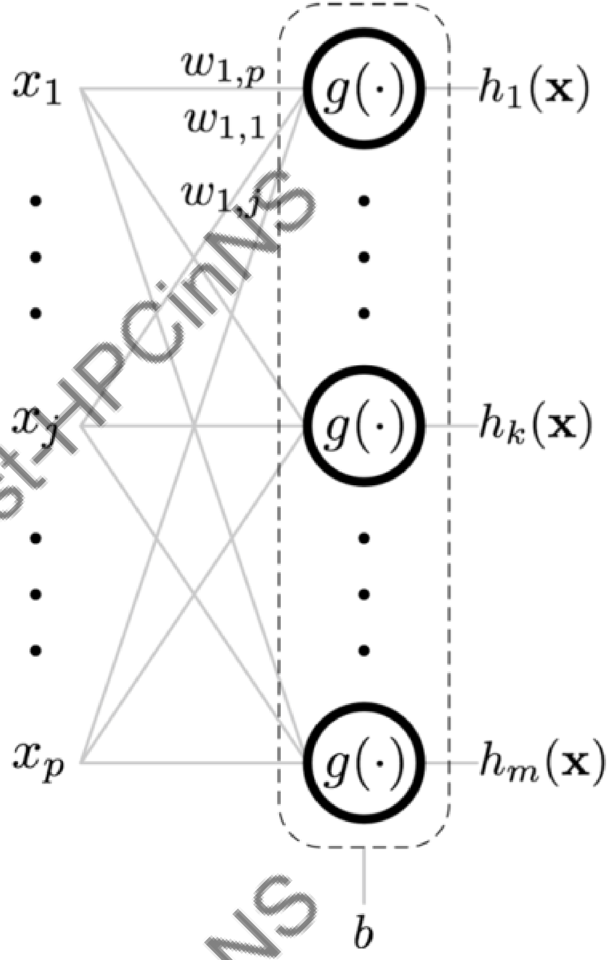


Layer of Neurons



$$h_m(\mathbf{x}) = g\left(\sum_{i=1}^{p} w_{m,i} x_i + b\right)$$

Layer of Neurons



$$h_m(\mathbf{x}) = g\left(\sum_{i=1}^{p} w_{m,i}x_i + b\right)$$

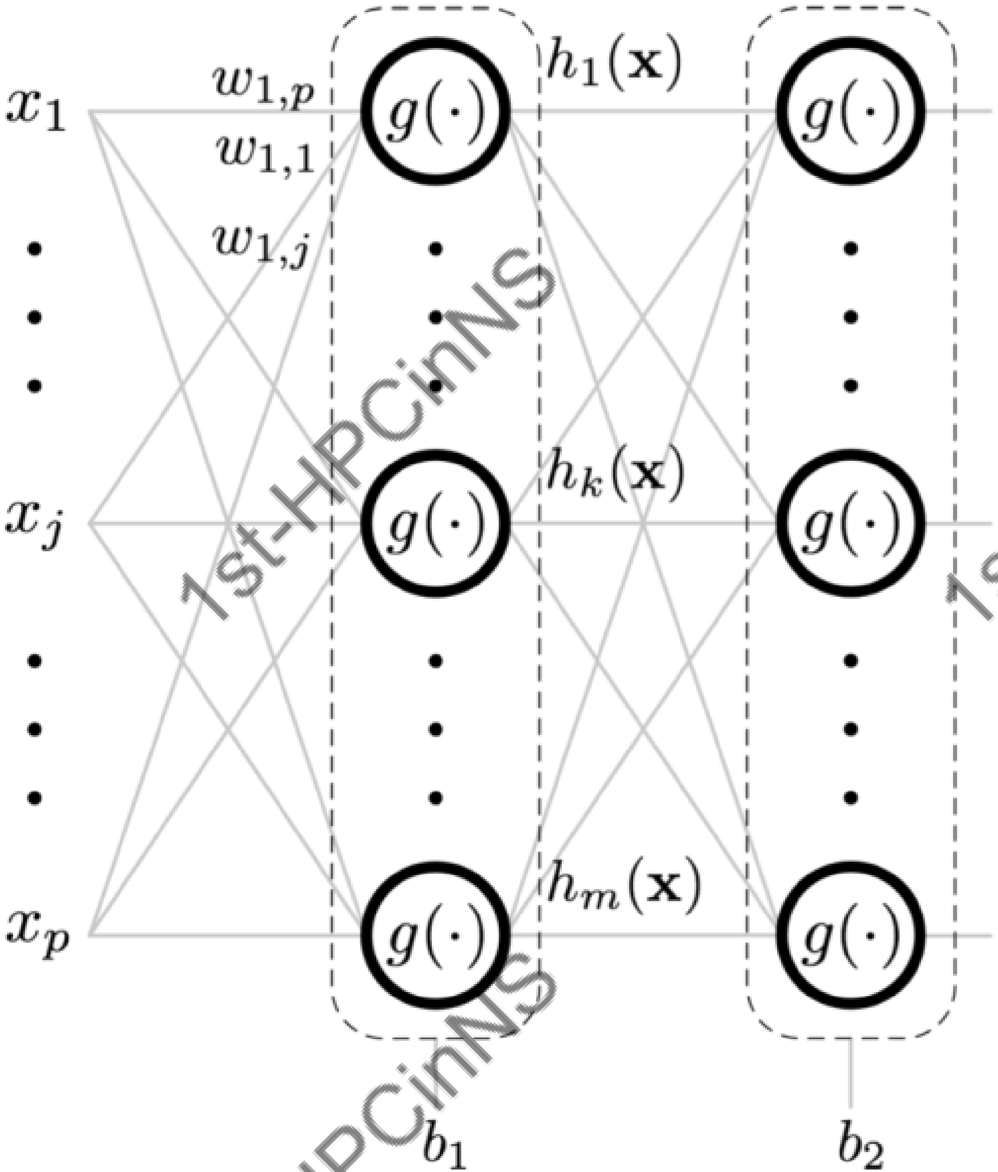
$$\mathbf{x} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,784} \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,10} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,10} \\ \vdots & \vdots & \ddots & \vdots \\ w_{784,1} & w_{784,2} & \cdots & w_{784,10} \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{10} \end{bmatrix}$$

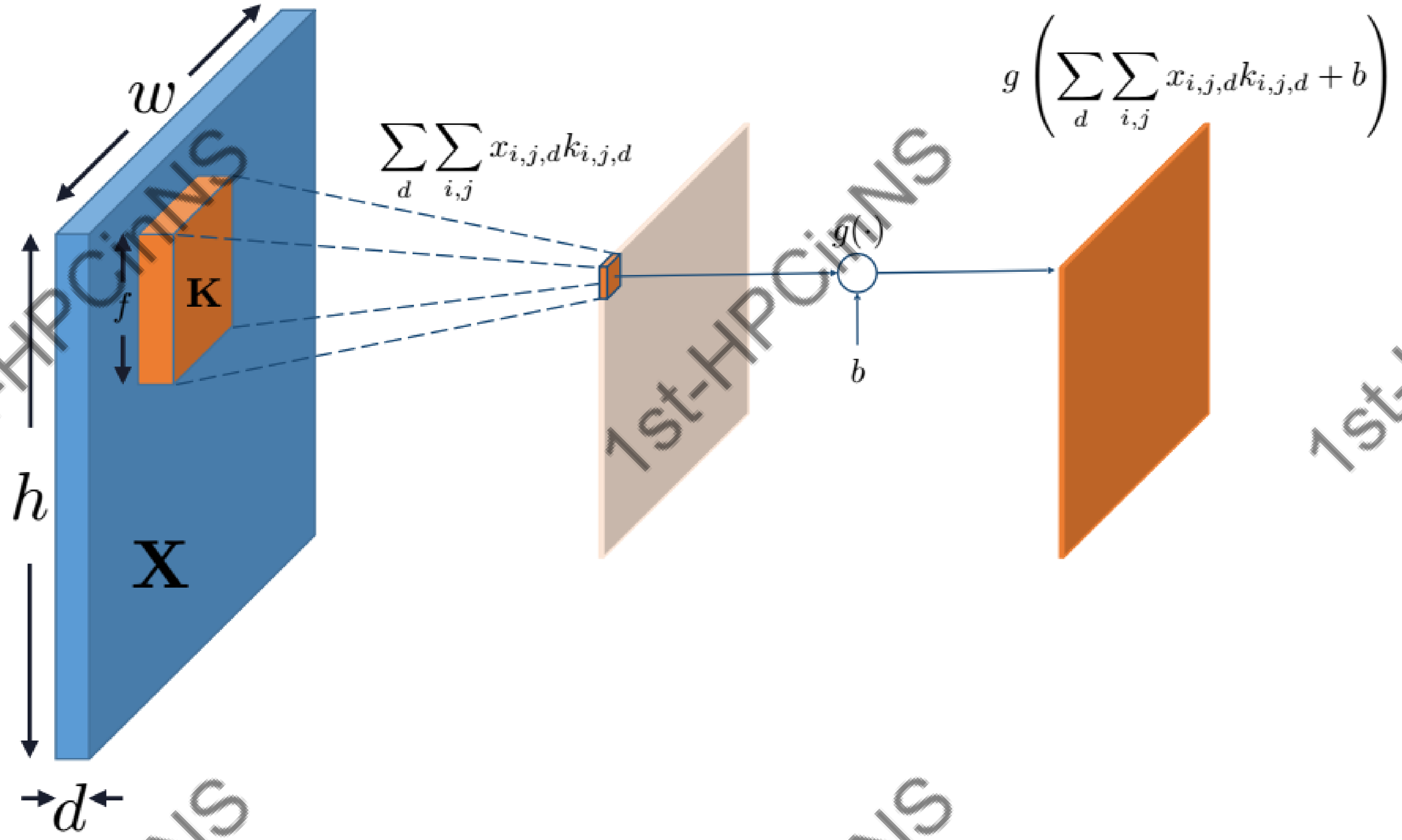
Matrix multiplication

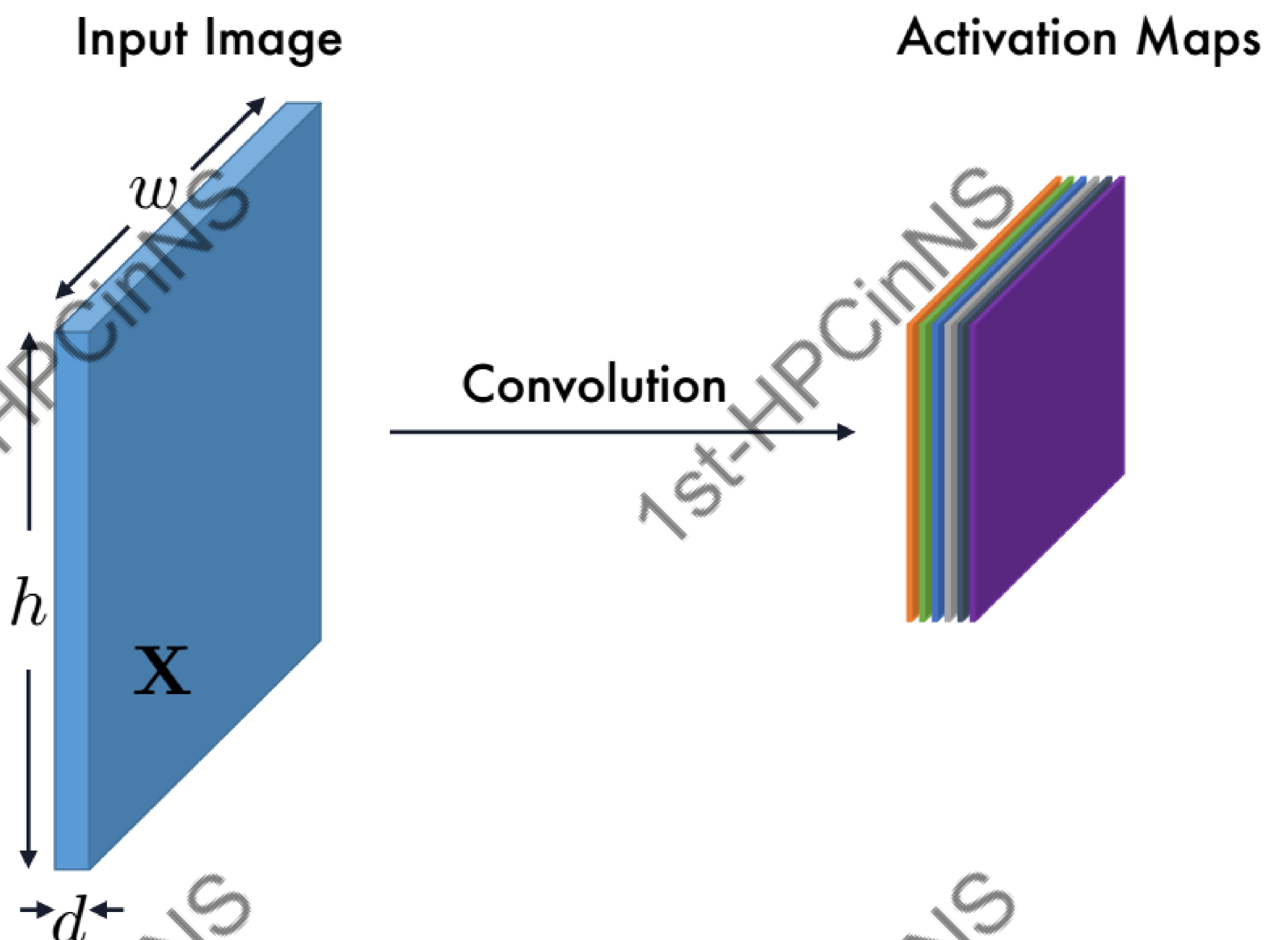
$$\mathbf{xW} + \mathbf{b} = \begin{bmatrix} \sum_{i=1}^{784} x_{1,i} w_{i,1} + b_1 & \sum_{i=1}^{784} x_{1,i} w_{i,2} + b_2 & \cdots & \sum_{i=1}^{784} x_{1,i} w_{i,10} + b_{10} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{784} x_{N,i} w_{i,1} + b_1 & \sum_{i=1}^{784} x_{N,i} w_{i,2} + b_2 & \cdots & \sum_{i=1}^{784} x_{N,i} w_{i,10} + b_{10} \end{bmatrix}$$

Hidden Layers



Convolutional Layer




Input Image
Activation Maps
Convolution
w
h
$\mathbf{X}$
d

Example: LeNet

Gradient-Based Learning Applied to Document Recognition

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

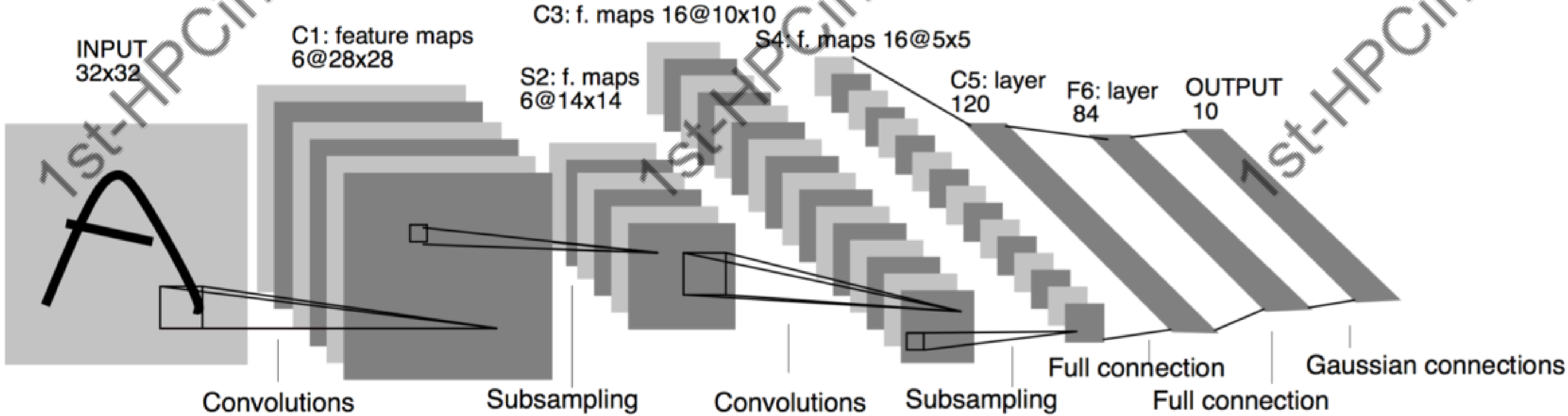


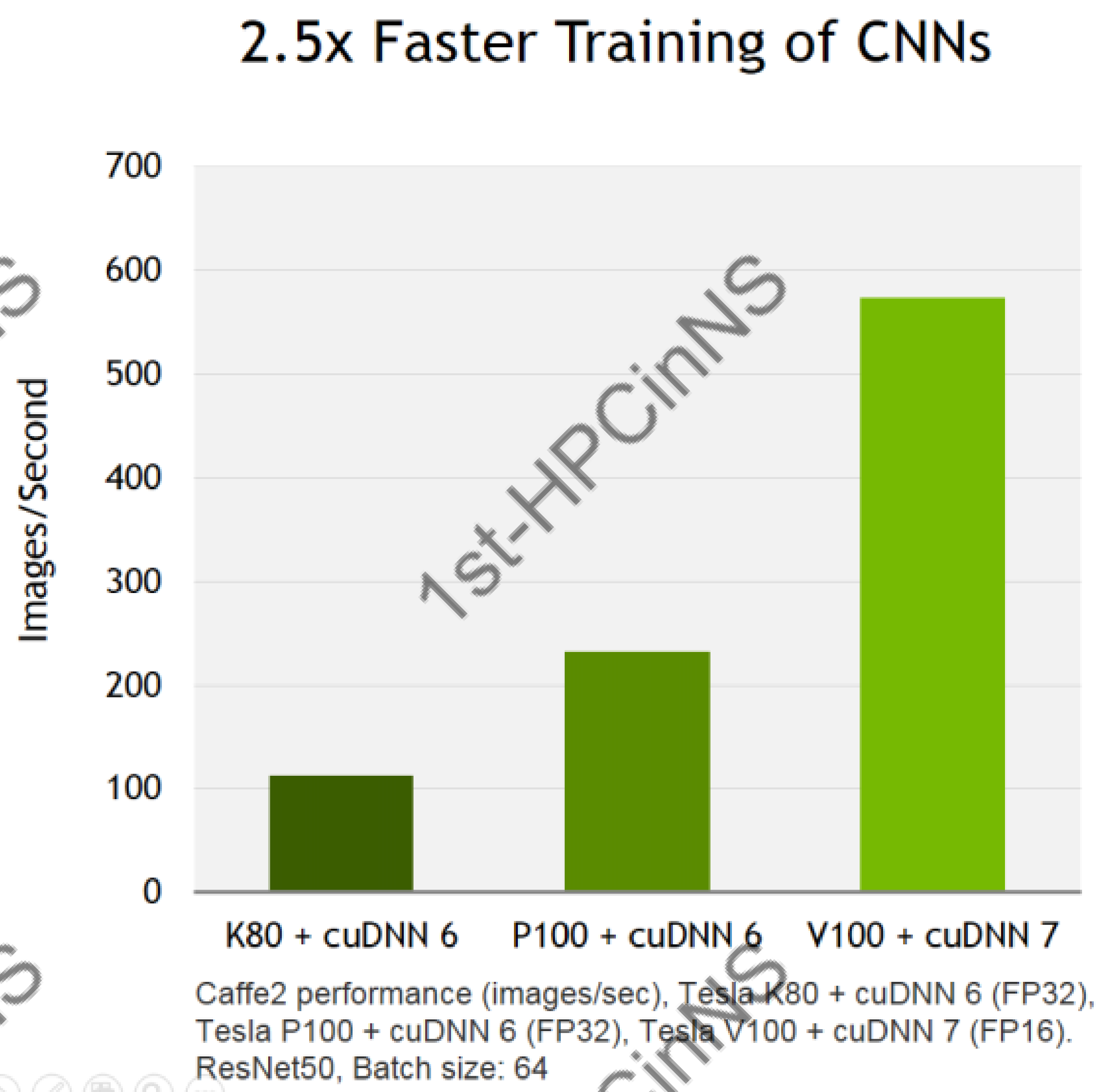
Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

cuDNN

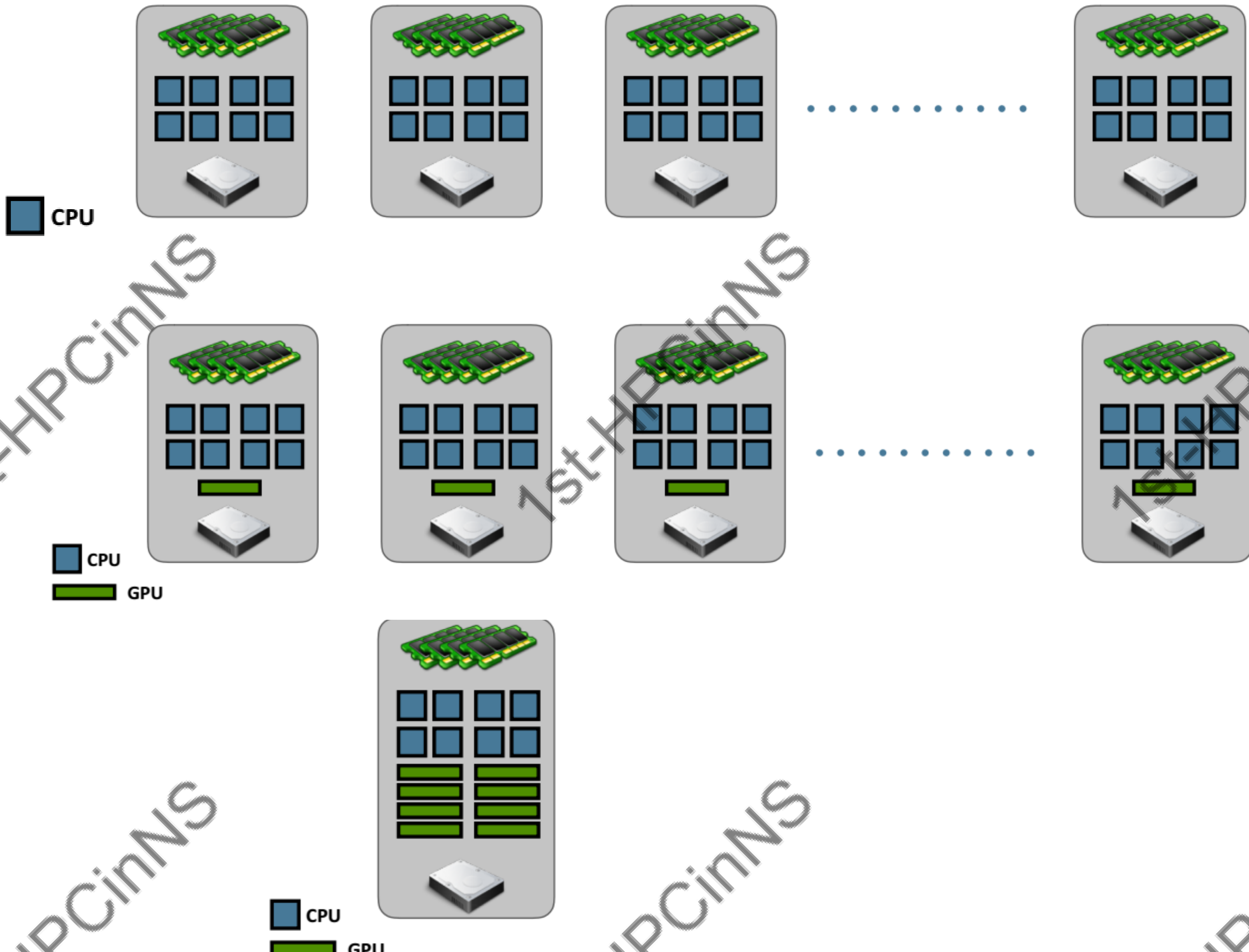
NVIDIA® cuDNN is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations of routines arising frequently in DNN applications:

- Convolution forward and backward, including cross-correlation
- Pooling forward and backward
- Softmax forward and backward
- Neuron activations forward and backward:
  - Rectified linear (ReLU)
  - Sigmoid
  - Hyperbolic tangent (TANH)
- Tensor transformation functions
- LRN, LCN and batch normalization forward and backward

CuDNN speed up

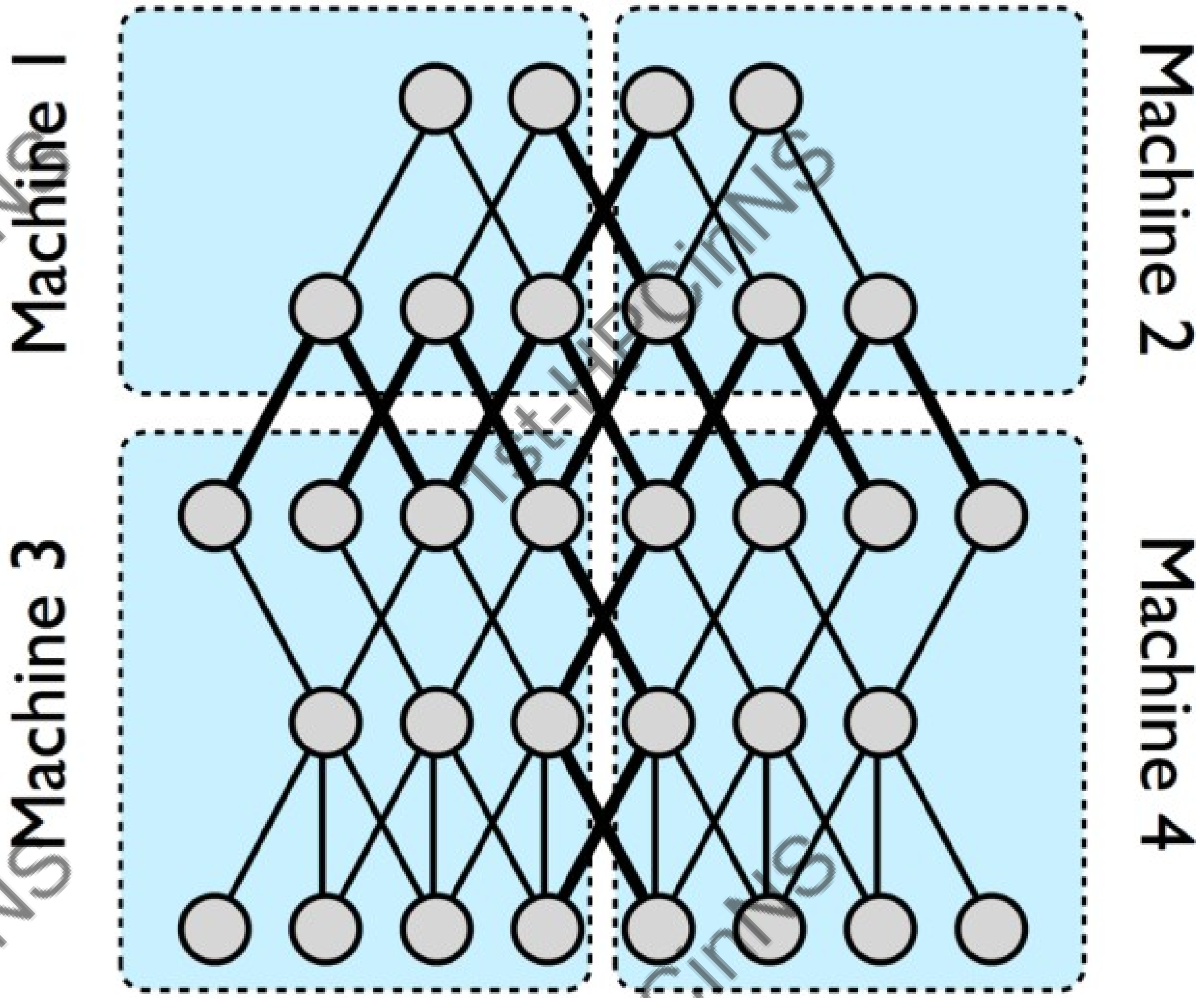


Clustering



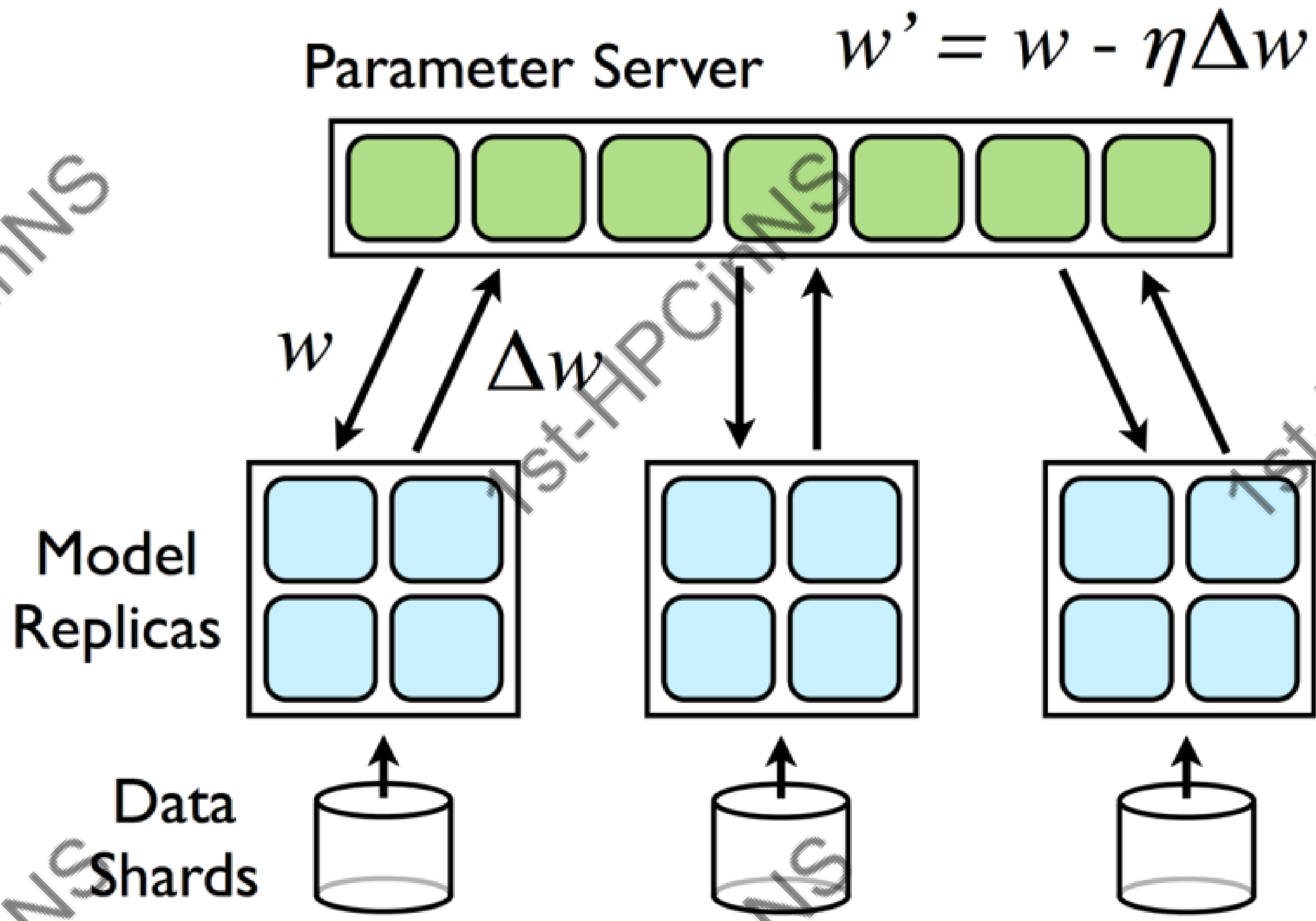
Parallelism strategies

- Model Parallelism: Split up a single model



Parallelism strategies

- Data Parallelism: Split up data to train a single model



Parallelism strategies

- Training Parallelism: Split up different parts of the training process
  - Ensemble Base Learners
  - Cross-Validation
  - Hyperparameters

Thanks!