Demystifying Parallel and Distributed Deep Learning:

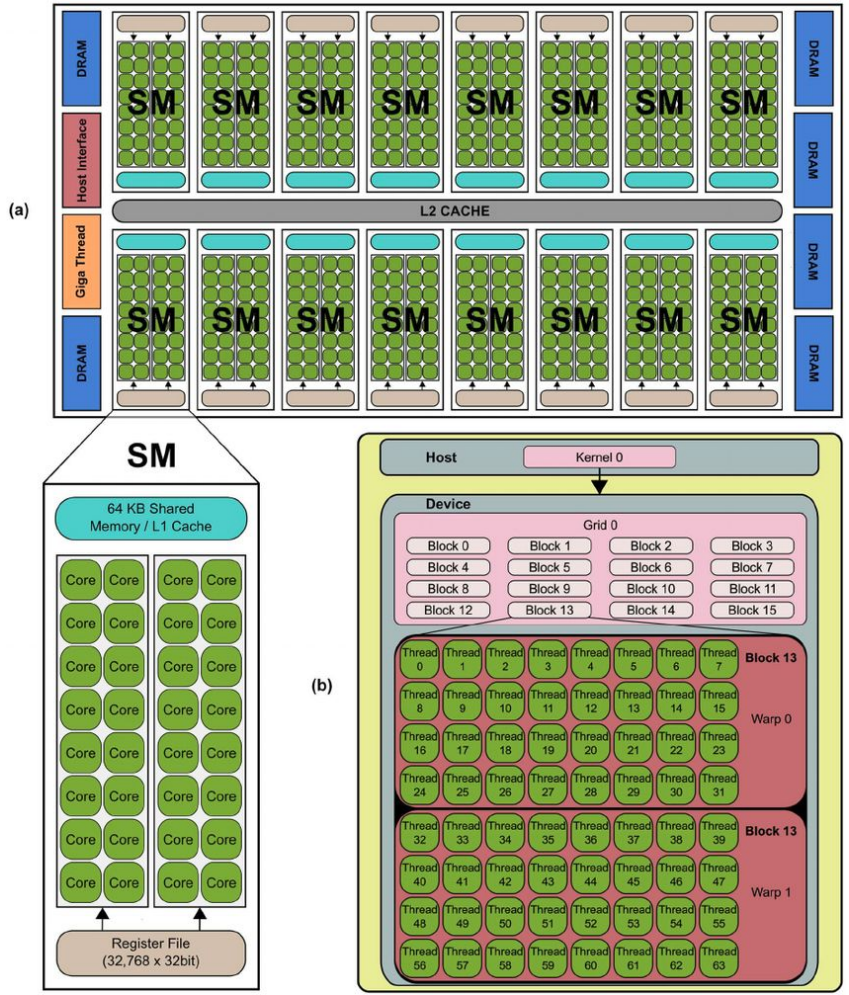
Tim Anderson, Sarah Walling, Kyle Lucke

GPU Parallelism: Basics

- 3D video == Matrix Multiplication
- Thousands of parallel cores
  - NVIDIA 1080 Ti contains 3584 parallel cores

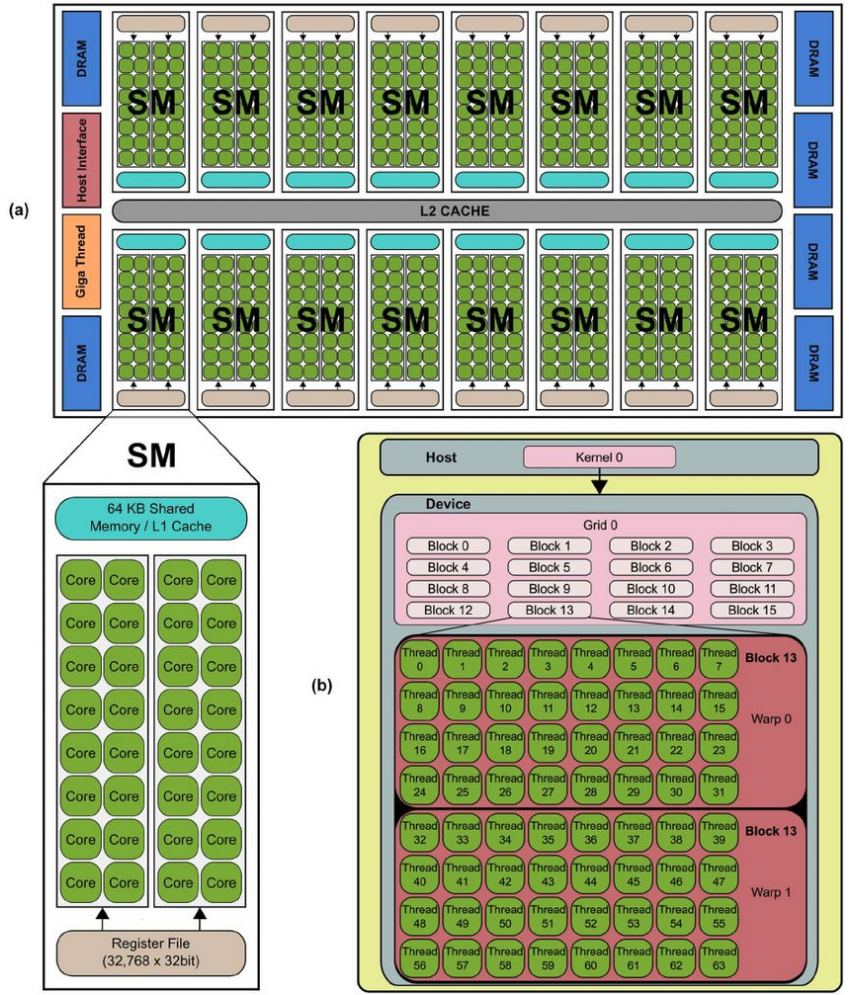
GPU Parallelism: Basics

- 3D video == Matrix Multiplication
- Thousands of parallel cores
- Contains hundreds of Streaming Multiprocessors (SMs)
  - 128 cores/SM
  - 32 cores/ thread warp



GPU Parallelism: Basics

- 3D video == Matrix Multiplication
- Thousands of parallel cores
- Contains hundreds of Streaming Multiprocessors (SMs)
  - 128 cores/SM
  - 32 cores/ thread warp
- Tasks broken into ***N*** blocks of up to 32 threads
  - If N > number of SMs, when one SM finishes its block, it will process another until all blocks are done

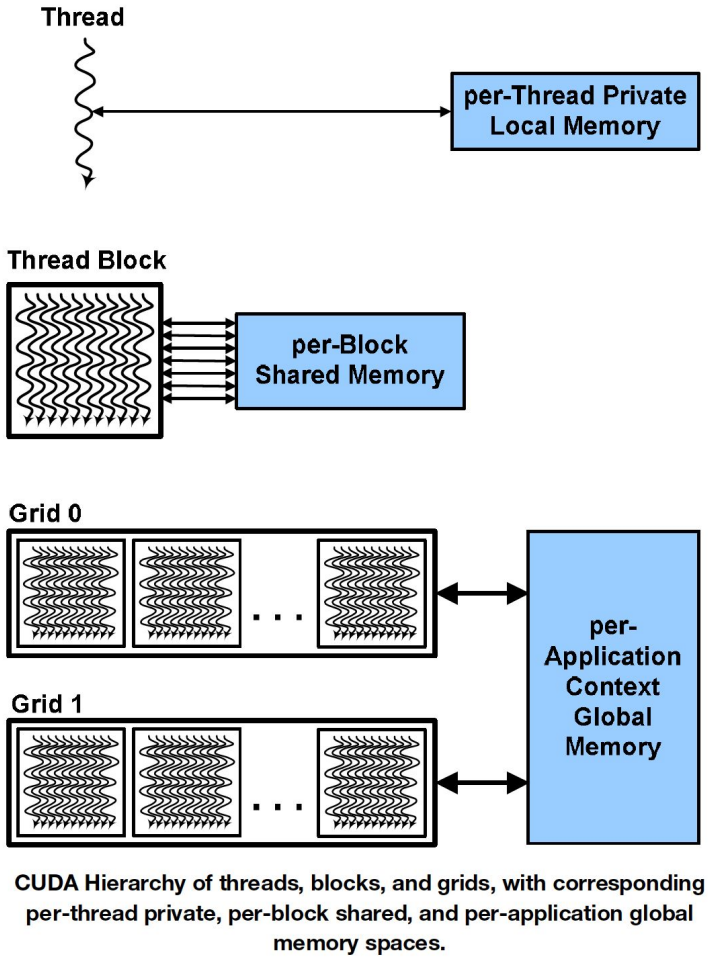


GPU memory

Low Latency

- Thread-Local memory
- Shared Memory
- Global Memory

High Capacity / Accessibility



CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

GPU Matrix Multiplication Example: **CUDA**

- Thread blocks given to SMs until all SMs are in use
- When one SM finishes, it processes the next unprocessed block until all blocks have been computed
- row and col are computed from the block and thread indices
- No data dependency on writing to c

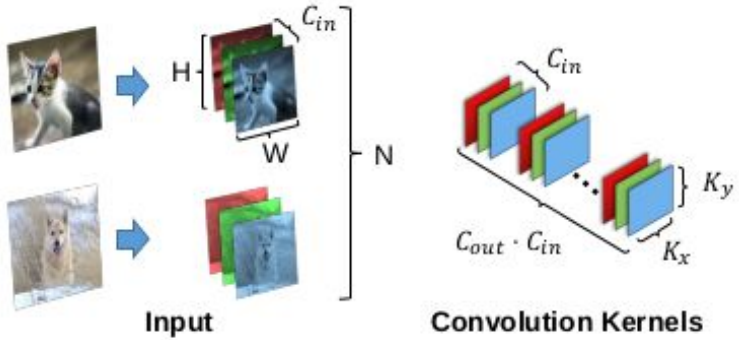
```
__global__ void gpu_matrix_mult(int *a,int *b, int *c, int m, int n, int k)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;
    if( col < k && row < m)
    {
        for(int i = 0; i < n; i++)
        {
            sum += a[row * n + i] * b[i * k + col];
        }
        c[row * k + col] = sum;
    }
}
```

5. Concurrency in Operators

"Given that neural network layers operate on 4-dimensional tensors and the high locality of the operations, there are several opportunities for parallelizing layer execution. In most cases, computations (e.g., in the case of pooling operators) can be directly parallelized. However, in order to expose parallelism in other operator types, computations have to be reshaped."

Name	Description
N	Minibatch size
C	Number of channels, features, or neurons
H	Image Height
W	Image Width
K_x	Convolution kernel width
K_y	Convolution kernel height

(a) Data Dimensions



(b) Convolution Dimensions

5.3 Convolution

Convolution is basically a dot-product between the kernel filter and the local regions selected by the moving window, that sample a patch with the same size as our kernel.

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

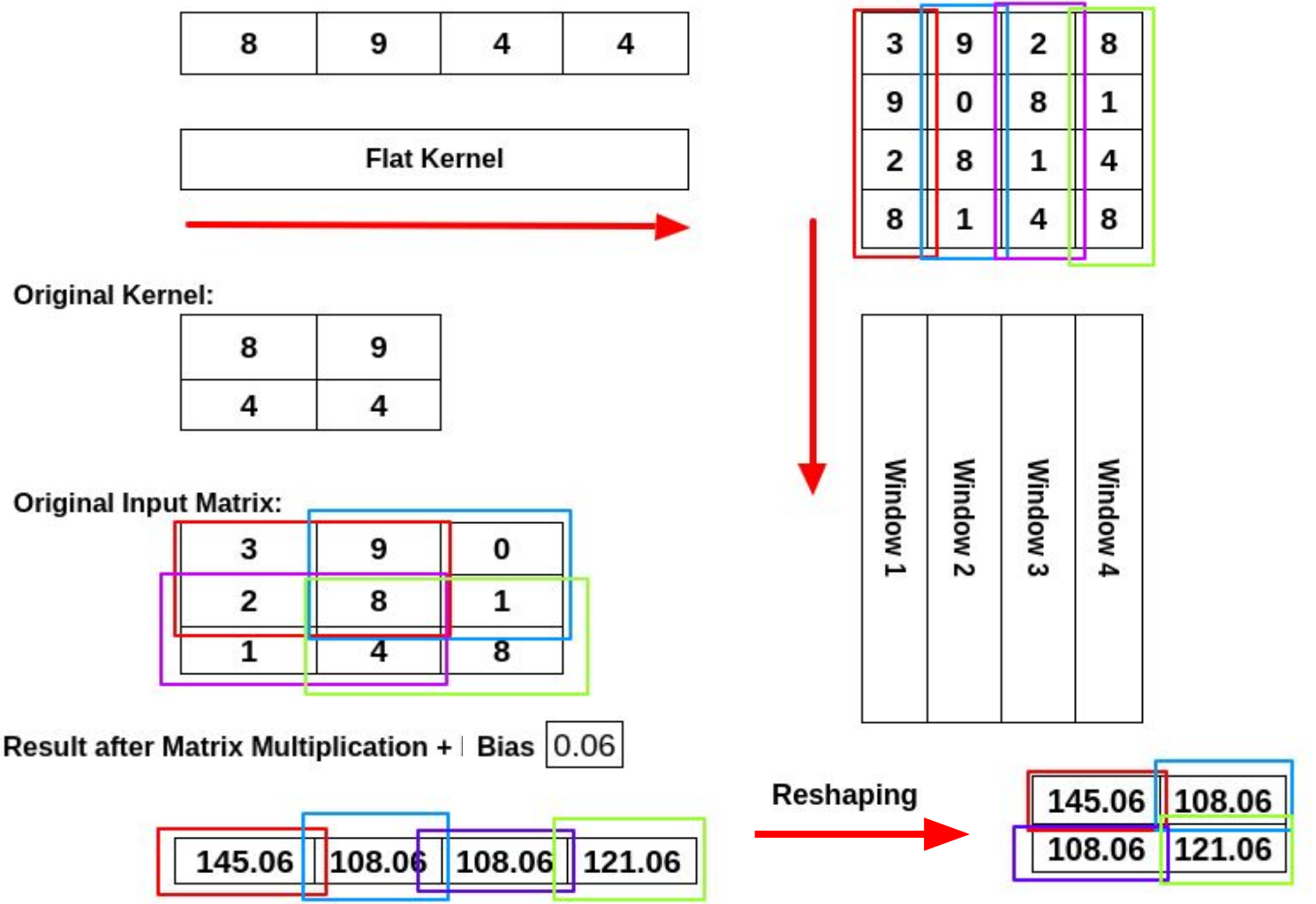
4		

Source: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

im2col

What would happen if we expand all possible windows on memory and perform the dot product as a matrix multiplication? 200x or more speedup - at the expense of more memory consumption.

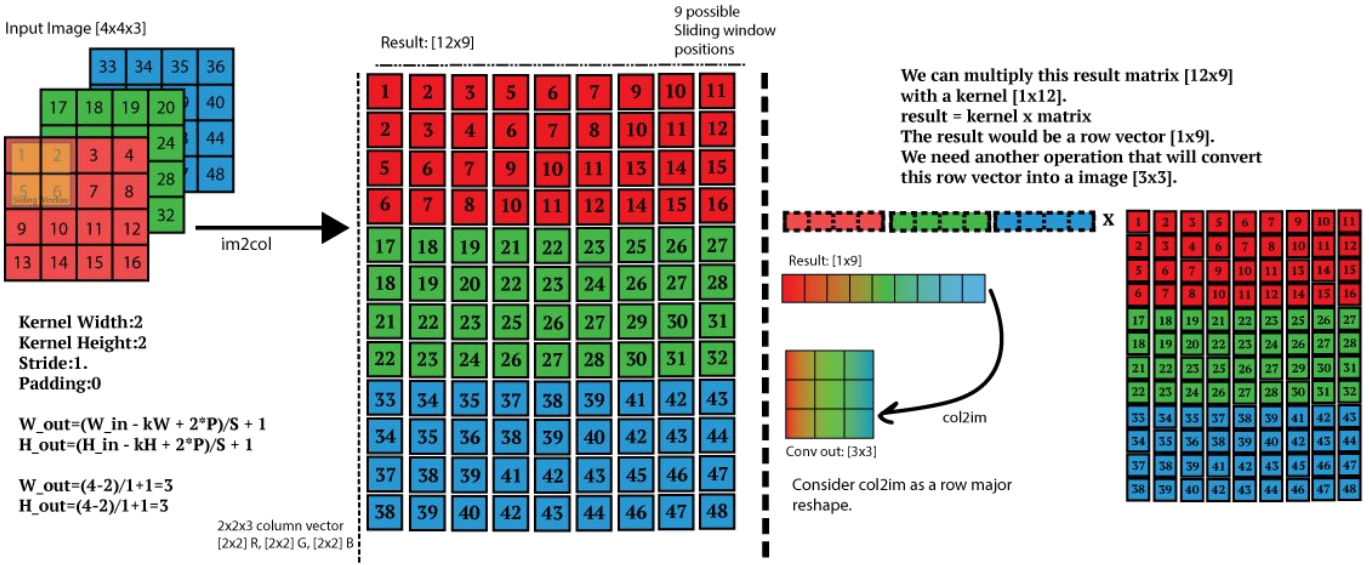
Take each window, flatten it out and stack them as columns in a matrix. Now, if we flatten out the kernel into a row vector and do matrix multiplication between the two, we should get the exact same result after reshaping the output.



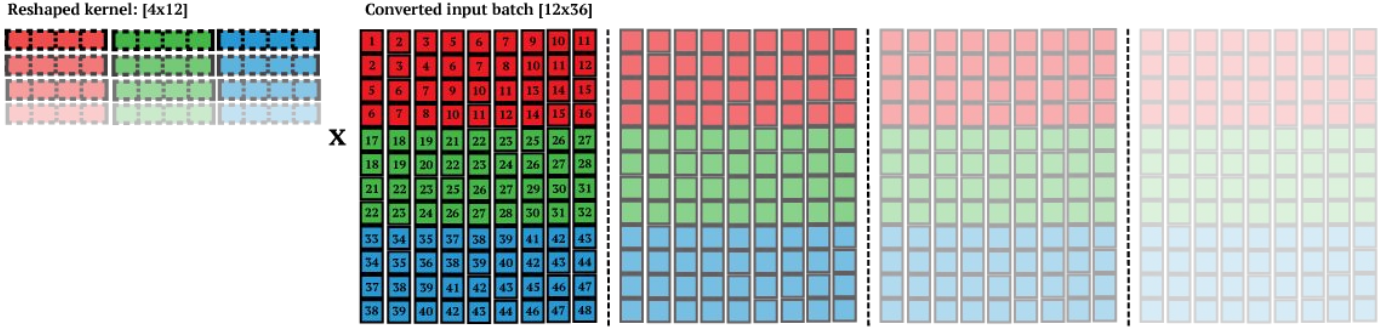
Source: https://towardsdatascience.com/how-are-convolutions-actually-performed-under-the-hood-226523ce7fbf

Source: https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/making_faster.html

Image to column operation (im2col)
Slide the input image like a convolution but each patch become a column vector.



We get true performance gain
when the kernel has a large number of filters, ie: F=4
and/or you have a batch of images (N=4). Example for the input batch [4x4x3x4], convolved with 4 filters [2x2x3x2] .
The only problem with this approach is the amount of memory



Fourier Transforms

Perform convolutions as products in the Fourier domain (frequency domain), and reuse transformed feature maps many times. The significant operations in training convolutional networks can all be viewed as convolutions between pairs of 2D matrices, which can represent input and output feature maps, gradients of the loss with respect to feature maps, or weight kernels. Typically, convolutions are performed for all pairings between two sets of 2D matrices.

The larger the convolution kernels are, the more beneficial FFT becomes, yielding up to 16× performance over the GEMM (im2col) method

Fourier Transforms

By computing the Fourier transforms of the matrices in each set once, we can efficiently perform all convolutions as pairwise products. Although it has long been known that convolutions can be computed as products in the Fourier domain, until recently the number of feature maps used in convolutional networks has been too small to make this method effective.

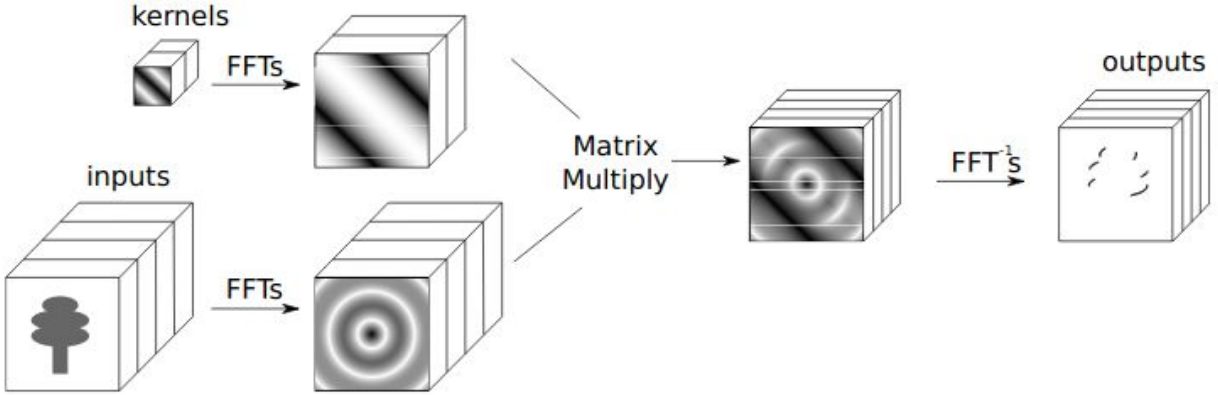


Figure 1: Illustration of the algorithm. Note that the matrix-multiplication involves multiplying all input feature maps by all corresponding kernels.

Source: https://blog.usejournal.com/understanding-winograd-fast-convolution-a75458744ff

Winograd

Conventional FFT based convolution is fast for large filters, but state of the art convolutional neural networks use small, 3 × 3 filters.

Compare to im2col:

Say we have input image *f* of size (4) and filter *g* of size *(3)*:

Using the im2col technique we convert into matrix multiplication such that:

$$f = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}, g = \begin{bmatrix} -1 \\ -2 \\ -3 \end{bmatrix}$$

= 6 multiplications

Winograd

Instead of doing the dot product, calculate the result with this formula:

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

where:

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$

$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

Kernel doesn't change - only needs to be computed once beforehand!

= 4 multiplications + 4 adds

Winograd

The minimal filtering algorithm for computing *m* outputs with a kernel size *r*, which we call *F(m, r)*, requires:

$$\mu(F(m,r)) = m + r - 1$$

Here we see $\mu(F(2,3)) = 2 + 3 - 1 = 4$, which we know is the minimum numbers of multiplications.

Winograd

We can nest minimal 1D algorithms *F(m, r)* and *F(n, s)* to form minimal 2D algorithms for computing *m*×*n* outputs with an *r*×*s* filter,which we call *F(m*×*n, r*×*s)*

These require: $$\mu(F(m \times n, r \times s)) = \mu(F(m, r))\mu(F(n, s)) = (m + r - 1)(n + s - 1)$$

F(2×2,3×3) uses 4×4 = 16 multiplications, whereas the standard algorithm uses 2×2×3×3 = 36

This is an arithmetic complexity reduction of 36/16= **2.25**

Winograd

How do you write this in matrix form? $Y = A^T\left[(Gg) \odot (B^T d)\right]$

(where ⊙ indicates element-wise multiplication)

For *F(2,3)*, the matrices are:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$$

$$d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$$

Winograd

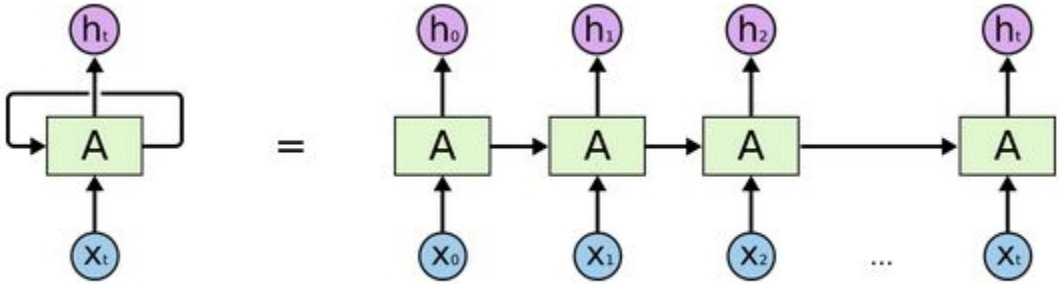
The 1D algorithm *F(m, r)* is nested with itself to obtain a minimal 2D algorithm, *F(m×m, r×r),* giving us:

$$Y = A^T \left[[GgG^T] \odot [B^T dB] \right] A$$

where now *g* is an *r×r* filter and *d* is an (*m+r−1*)×(*m+r−1*) image tile

Each image channel is divided into tiles of size (m+r−1)×(m+r−1), with r−1 elements of overlap between neighboring tiles

5.4 Recurrent Units



An unrolled recurrent neural network.

Since RNN units are usually chained together (forming consecutive recurrent layers), two types of concurrency can be considered: within the same layer, and between consecutive layers.

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Intralayer Concurrency

Fuse all computations into one function (kernel), saving intermediate results in scratch-pad memory.

Reduces kernel scheduling overhead and conserves round-trips to the global memory, using the multi-level memory hierarchy of the massively parallel GPU

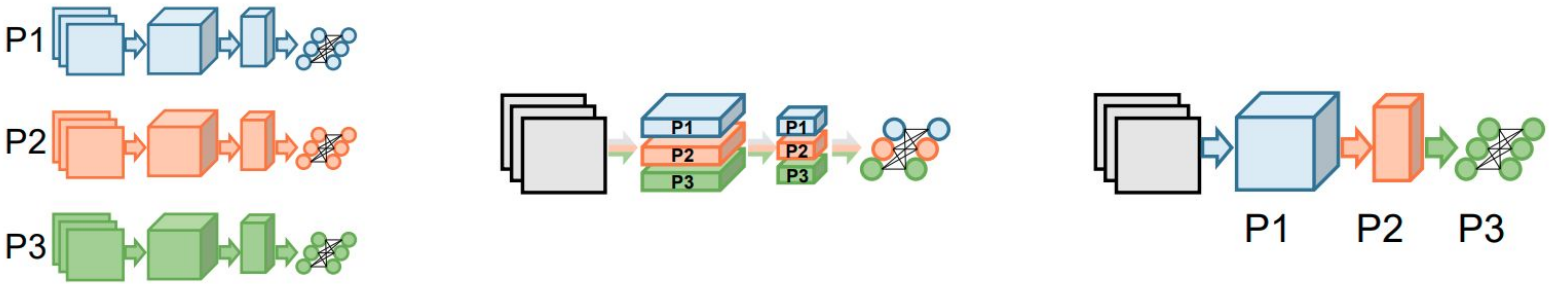
Interlayer Concurrency

i.e. **pipeline parallelism** (section 6) - stack RNN unit computations, let them immediately propagate through to next layer once its data dependencies have been met

6. Concurrency in Networks

3 Main types of parallelism

- Data Parallelism
- Model Parallelism
- Pipeline Parallelism



(a) Data Parallelism (b) Model Parallelism (c) Layer Pipelining

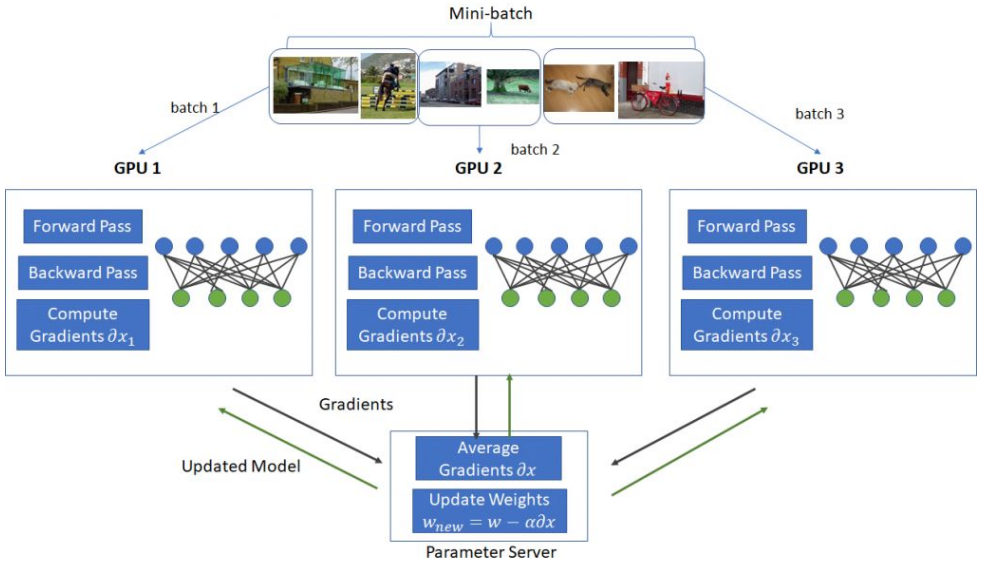
Fig. 14. Neural Network Parallelism Schemes

Data Parallelism

Most operators are independent w.r.t. sample size

Only necessary to synchronize when operator depends on multiple batches

- Batch Normalization
- Gradient Descent



Model Parallelism

Same minibatch given to each processor, different parts of DNN computed on each processor

- Reduced memory requirements
- More difficult to implement than data parallelism
  - Communication between many processors limits performance

Model Parallelism

Same minibatch given to each processor, different parts of DNN computed on each processor

- Reduced memory requirements
- More difficult to implement than data parallelism
  - Communication between many processors limits performance
- Fully-connected layer: all-to-all connection
- Convolution: often requires inputs from multiple channels

Model Parallelism

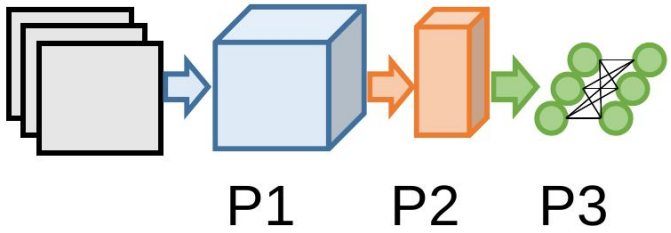
Same minibatch given to each processor, different parts of DNN computed on each processor

- Reduced memory requirements
- More difficult to implement than data parallelism
  - Communication between many processors limits performance
- Fully-connected layer: all-to-all connection
- Convolution: often requires inputs from multiple channels
- **Locally-connected layer**: parallel optimization, multiple local filters
  - Allows partitioning on C, H, W dimensions w/o all-to-all communication

Pipeline parallelism

Splits the network into sections of consecutive operations, and gives each section to a single processor

- A given processor waits until it receives data from the previous processor, computes its outputs, and sends the outputs to the next processor in the pipeline

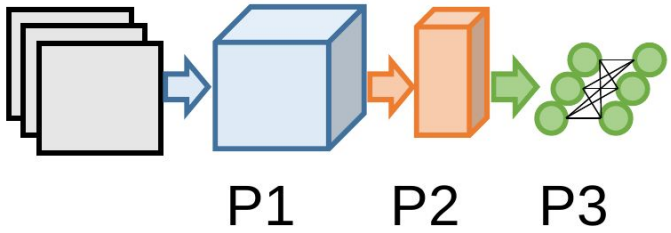


(c) Layer Pipelining

Pipeline parallelism

Splits the network into sections of consecutive operations, and gives each section to a single processor

- A given processor waits until it receives data from the previous processor, computes its outputs, and sends the outputs to the next processor in the pipeline
- Disadvantages
  - Pipeline saturation
  - latency

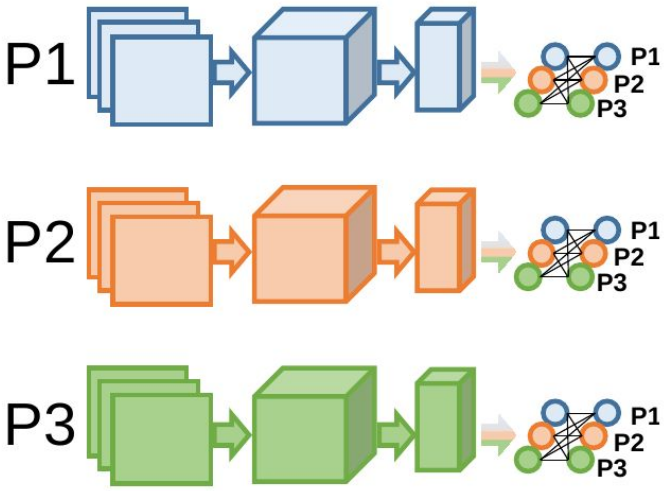


(c) Layer Pipelining

Hybrid parallelism

Parallelism modes are not mutually exclusive

- **AlexNet**:
  - Data parallelism for Conv. layer, Model for FC layer
- **DistBelief**:
  - Data parallelism (multiple simultaneous models)
  - Model parallelism + pipelining in each model replica



(b) Hybrid Parallelism

7. Concurrency in Training

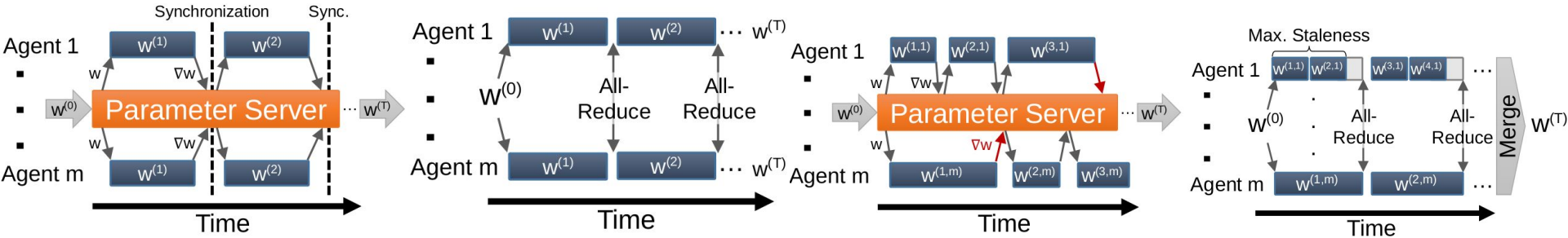
- So far, we have only discussed algorithms where there is only one copy of w and it's up-to-date value is observable by all processors
- In distributed environments, may have many training agents running independently

7. Concurrency in Training

- Model Consistency
- Centralization
- Parameter and gradient compression
- Model consolidation
- Optimization Algorithms
- Architecture Search

Model Consistency

- In distributed systems need to keep weights synchronized
- Can instead create an inconsistent model:
  - Each training agent i has a copy of weights at time t, denoted $w^{(tau, i)}$ for tau <= t
  - t - tau is referred to as the lag or staleness
  - Example: HOGWILD
    - Allows agents to read parameters and update weight at will
    - Converges under special conditions
  - HOGWILD and variants admit convergence rate of O(1/sqrt(mT)) for m participating nodes



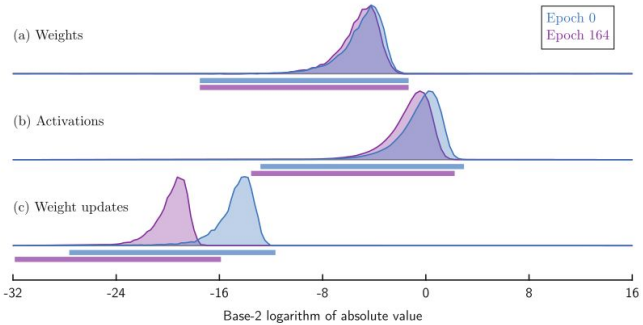
(a) Synchronous, Parameter Server (b) Synchronous, Decentralized (c) Asynchronous, Parameter Server (d) Stale-Synchronous, Decentralized

Centralization

- Can have a centralized or decentralized network
  - Centralized:
    - Typically utilize Parameter Server(PS) node(s)
    - Then, PS performs parameter update
    - Communication is less efficient, but communicate less info
    - More fault tolerant(checkpoint/restart, dynamically add and remove nodes)
  - Decentralized:
    - Utilize *allreduce* to communicate parameter updates
    - Update computed by each node separately(create own optimizers)

Parameter and Gradient Compression

- Distributed learning requires communication to converge
- More efficient communication by reducing *size* of each message
  - Quantization:
    - Parameter and weight distributions narrowly dispersed,
    - therefore, need few bits to represent entire range
    - Reduce dynamic range of FP number(e.g. utilize 16 bit float)
    - Must keep track of quantization error (i.e. local gradient accumulation)
  - Sparsification:
    - DNNs exhibit sparse gradients during param updates
    - Static pruning(thresholding)
    - relative threshold(top 1% of information)
    - adaptive threshold



(a) Parameter and Gradient Value Distribution (ResNet on CIFAR-10, adapted from [132])

Model Consolidation

- Ensemble learning:
  - Multiple instances of *w* trained on same dataset
  - Overall prediction is average of all model predictions
  - Completely parallel

$$f(x) = \frac{1}{m} \sum_{i=0}^{m} f_{w^{(\tau,i)}}(x)$$

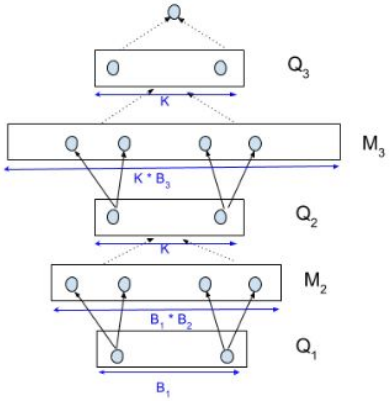
- Knowledge distillation:
  - Large network or ensemble is trained, then single NN is trained to mimic output of large net
  - Results show it is easier to train on ensemble rather than labeled data
  - Able to achieve same word error rate as ensemble of 10 DNNs
- Model Averaging
  - Run *m* SGD instances on different machines
  - Aggregate parameters once post-training or every few iterations
  - Stale-synchronous SGD tends to lead to higher overall accuracy

Optimization Algorithms

- Hyperparameter search:
  - Values significantly effect final result
  - Until recently, best technique was grid search
  - Educated guess (Predictive analysis of learning curve for dropping undesirable configurations, Bayesian optimization, etc.)
  - Tuning hyperparameters during training.
  - Metaheuristic optimization algorithms such as Particle Swarm Optimization (PSO)

Architecture Search

- Sequential Model Based Optimization(SMBO):
  - Optimize architecture candidate:
    - Define and traverse finite set of states to explore
- Reinforcement learning(RL)
  - Accuracy of resulting network as a reward function
  - Modification of DNN or hyperparameters is an action
- Evolutionary Algorithms(EA):
  - Encode DNN connections as binary genes
  - Train population with every time step, using final accuracy as fitness function
  - Highly amenable to parallelism, have been used for large scale training to achieve state of the art accuracy



(a) PNASNet Search-Space Traversal [159]

Discussion Questions

1. What will make training high-performance models available to someone on their desktop?

2. What key advances will be necessary to use larger data sets?

3. In what scenarios would you use data vs model vs pipeline parallelism?

4. What are the pros and cons of centralized vs decentralized network?