

Optimal DNN Primitive Selection with Partitioned Boolean Quadratic Programming

Andrew Anderson, David Gregg

ACM Code generation and Optimization (CGO) Conference

Presenter: Constant (Sang-Soo) Park

http://esoc.hanyang.ac.kr/people/sangsoo_park/index.html

February 02, 2020



Neural Acceleration Study

Contents of presentation

- **Introduction and Background**

- Fast acceleration, Acceleration HW, Convolutional neural network (CNN)
- Convolution shapes, Primitives (DNN convolution algorithm)

- **Key contribution: Primitive selection**

- Partitioned Boolean quadratic assignment problem (PBQP) based selection

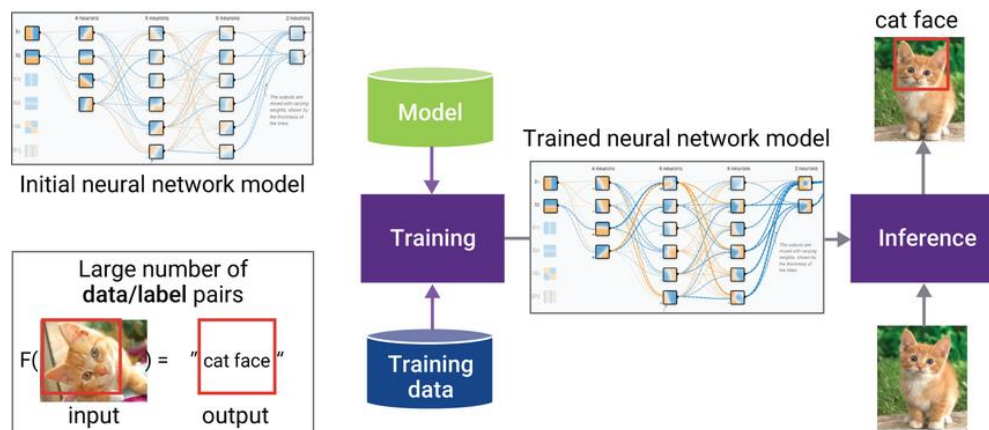
- **Performance**

- **Conclusion and Discussion**

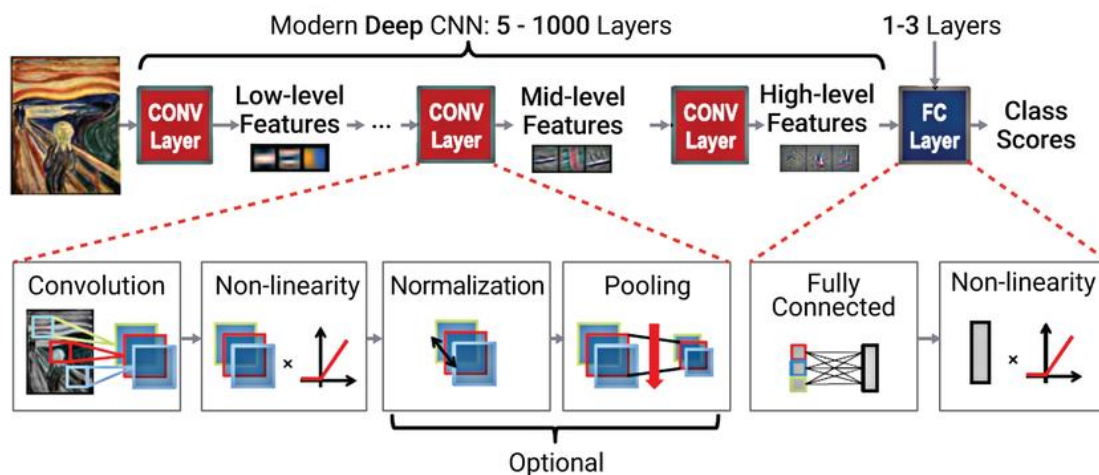
Introduction: Fast Acceleration

- **Fast inference or training**

- Inference: Image classification, Acoustic speech recognition
- Training: ImageNet, TIMIT learning, Etc.
- **The goal of fast acceleration is to classify or learn data quickly**
- **Target: Convolution or Fully-connected layer**



Neural network models in deep learning framework^[1]



Modern convolutional neural networks ^[1]

[1] https://www.synopsys.com/designware-ip/technical-bulletin/building-efficient-deep-learning-dwtb_q318.html

Introduction: Acceleration HW

- **From processor to dedicated accelerator**
 - Processor: CPU (x86, ARM, RISC-V), GPU (NVIDIA, AMD), DSP (CEVA, Qualcomm)
 - Dedicated accelerator: ASIC/FPGA based accelerator



CPU: Ryzen Threadripper



GPU: VOTLA, TURING, PASCAL

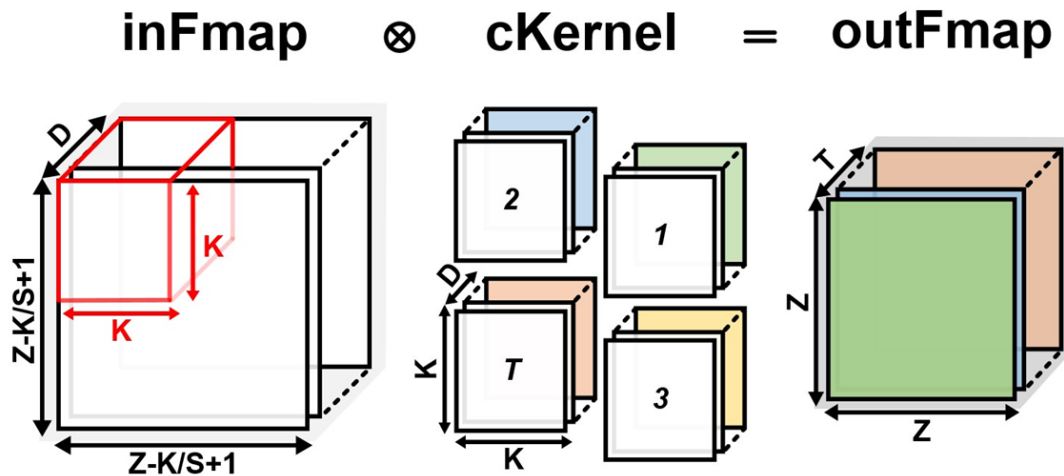


FPGA: ALVEO

Introduction: Convolutional NN (CNN)

• Convolution: Key operation in CNN

- Multiply and Accumulate (MAC) operation between **feature map** (inFmap) and **kernel** (cKernel)
- Total number of MAC operation is $O(Z \times Z \times D \times K^2 \times T)$



Graph of convolution layer^[2]

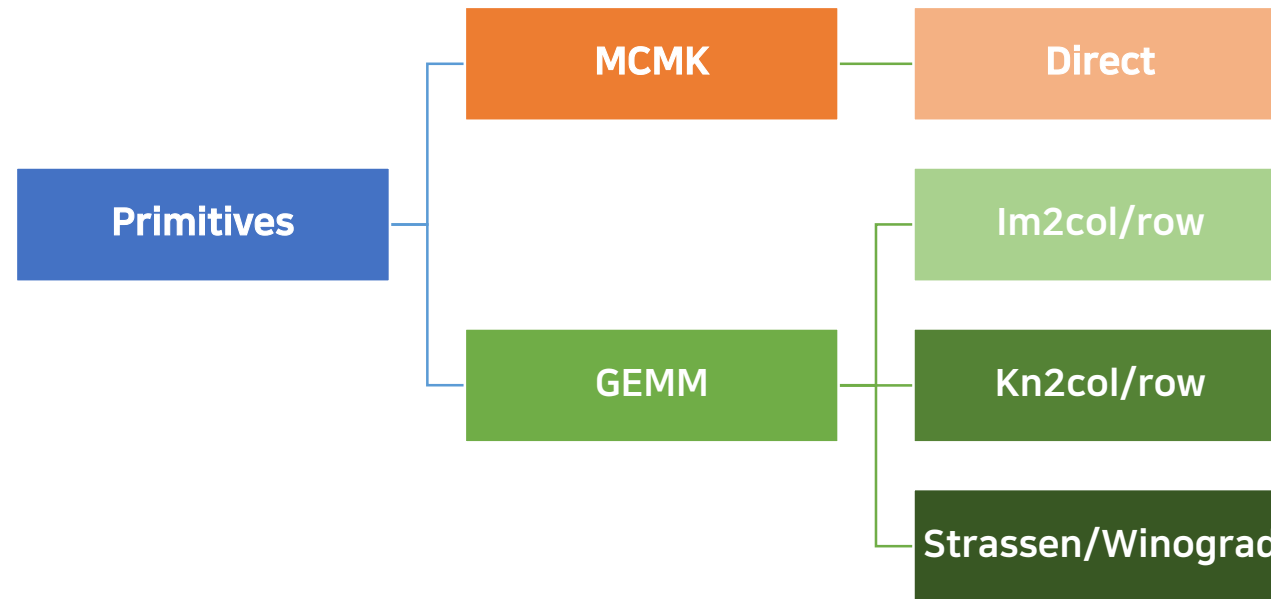
```
for (t=0; t<T; t++) {  
    for (r=0; r<Z; r=r+S) {  
        for (c=0; c<Z; c=c+S) {  
            for (d=0; d<D; d++) {  
                for (i=0; i<K; i++) {  
                    for (j=0; j<K; j++) {  
                        outFmap[t][r][c] += inFmap[d][r+i][c+j] * cKernel[t][d][i][j]  
                    }  
                }  
            }  
        }  
    }  
    outFmap[t][r][c] = activation(outFmap[t][r][c])  
}
```

Pseudo code of convolution layer^[2]

Background: Primitives

- **Definition: Belonging to a very early period (원시적인)**

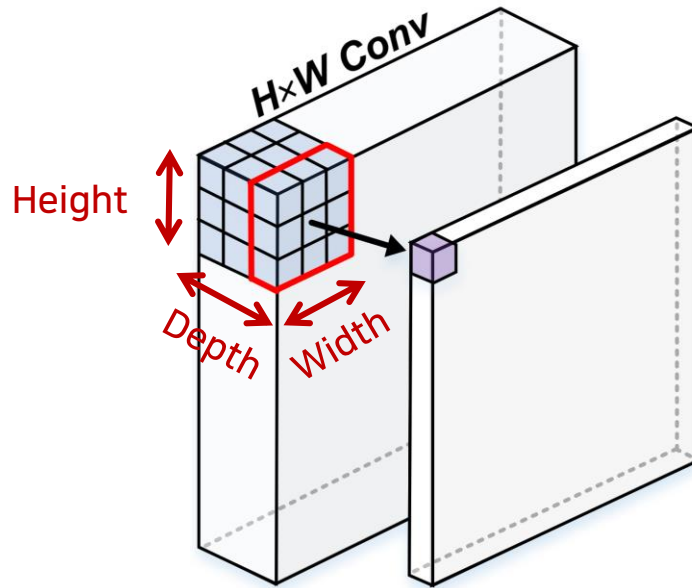
- In CNN, method to implement one of each of the types of layer in DNN
- Ex) **Convolution layer** in CNN, **Recurrent layer** in GRU/LSTM



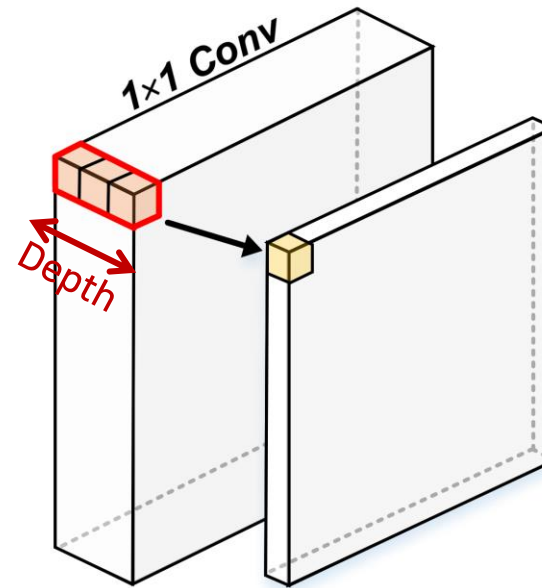
Hierarchy of DNN Primitives

Background: Convolution shapes

- **Layer used in convolutional neural network**
 - **H×W convolution (Height×Width×Depth)**, **1×1 convolution (1×1×Depth)**
 - Different optimal primitives in each layer



H×W convolution layer

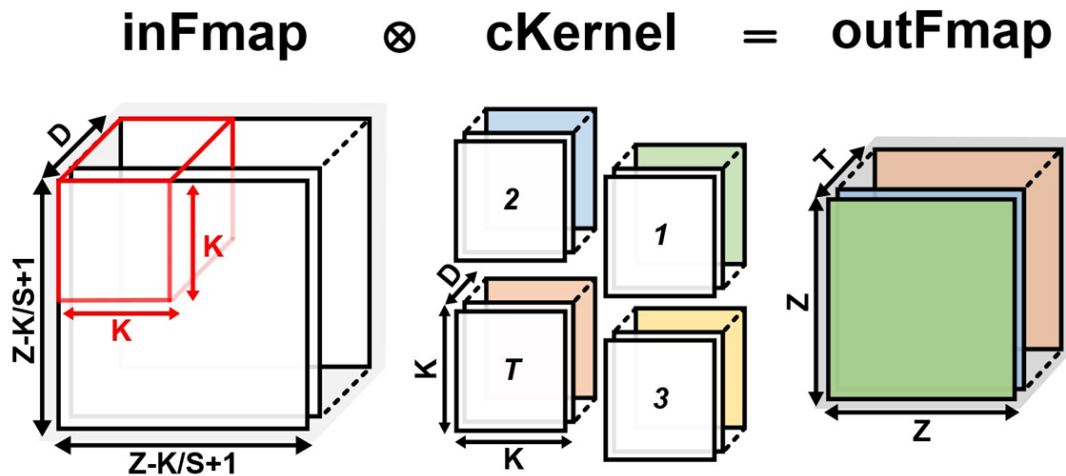


1×1 convolution layer

Background: Primitives

• Multi Channel Multi Kernel (MCMK)

- The most intuitive way to implement a convolutional layer
- Implemented using multiple For loop statement (6~7 levels)
- Most inefficient primitive in data locality



Graph of convolution layer^[2]

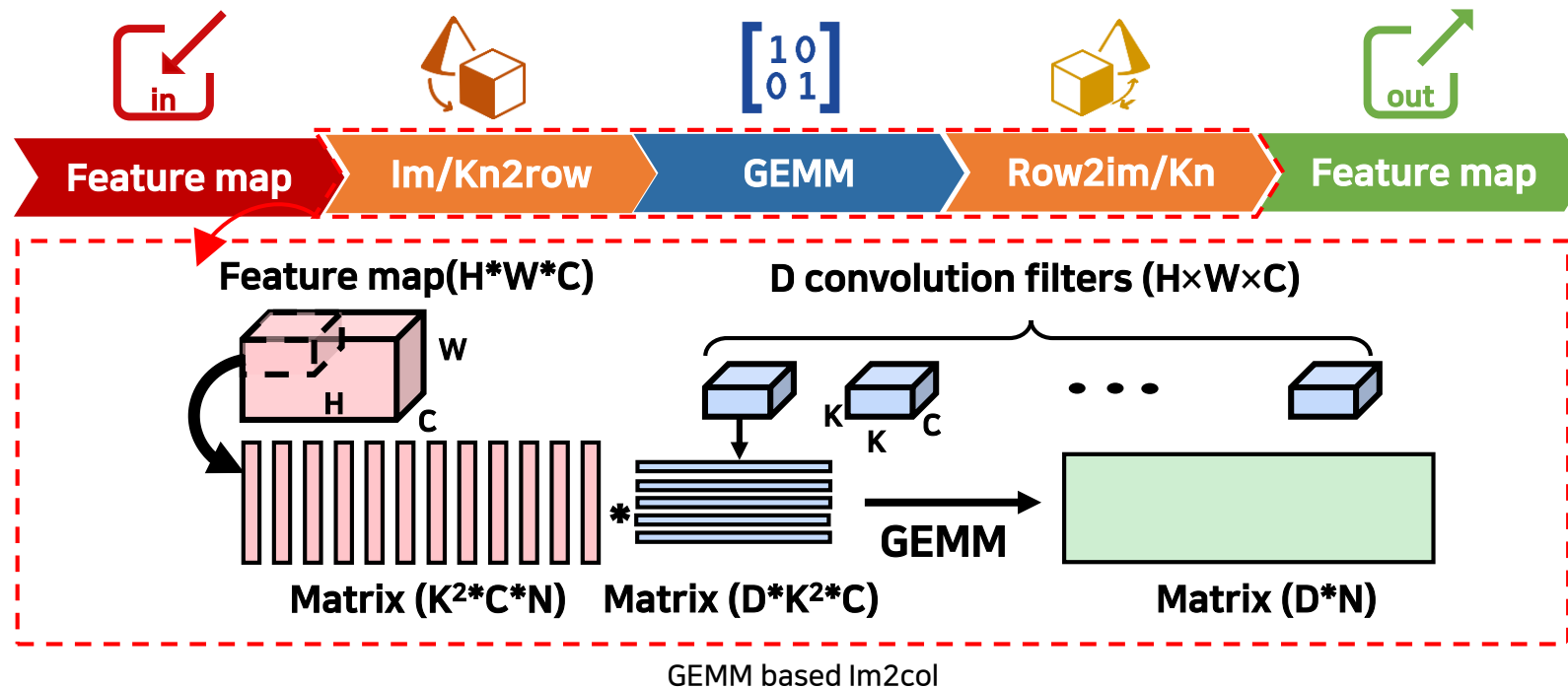
```
for (t=0; t<T; t++) { // type index of kernel
  for (r=0; r<Z; r=r+S) { // row index of feature map
    for (c=0; c<Z; c=c+S) { // col index of feature map
      for (d=0; d<D; d++) { // depth index of feature map
        for (i=0; i<K; i++) { // row index of kernel
          for (j=0; j<K; j++) { // col index of kernel
            outFmap[t][r][c] += inFmap[d][r+i][c+j] * cKernel[t][d][i][j]
          }
        }
      }
    }
  }
  outFmap[t][r][c] = activation(outFmap[t][r][c])
}
```

Pseudo code of convolution layer^[2]

Background: Primitives

- **General matrix multiplication (GEMM)**

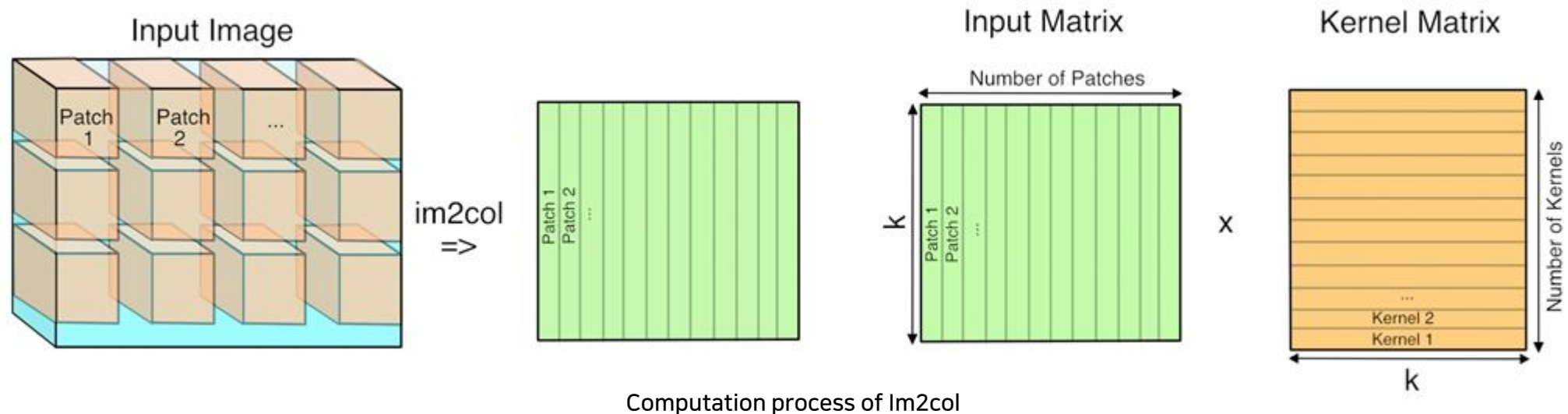
- Transforming convolution operation to matrix multiplication
- Transformation processor for matrix multiplication, results obtained through matrix multiplication
- Advantages of data locality over MCMK



Background: Primitives

- **Image to column/row (Im2col/row)**

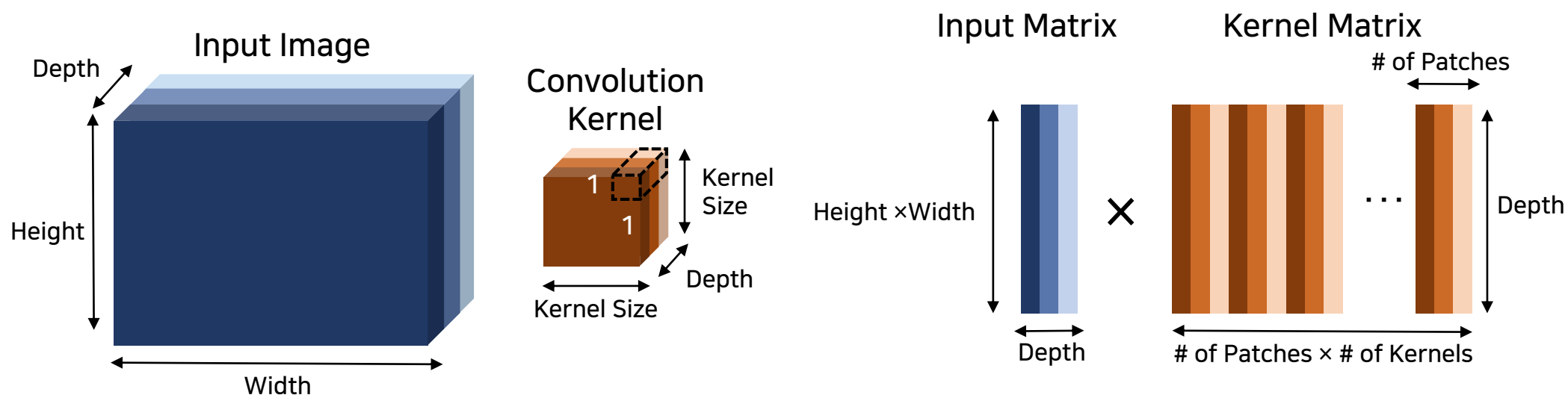
- Converting feature map into patches to match the kernel size
- Ex) Kernel width \times height \times depth \times **N** (**N is # of patches**)
- Appropriate for H \times W convolution layer (data locality for horizontal/vertical orientations)
- Im2col's transposed image to row (Im2row) is mainly used



Background: Primitives

- **Kernel to column/row (Kn2col/row)^[3]**

- Converting convolution kernel into patches to math 1×1 convolution
- Ex) Kernel width (5) \times height (5) \times depth (3) \rightarrow width (1) \times height (1) \times depth (3) \times **patches (25)**
- Appropriate for 1×1 convolution layer (data locality for channel orientations)
- Kn2col's transposed image to row (Kn2row) is mainly used



Computation process of Kn2col (Left), GEMM in Kn2col (Right)

Background: Primitives

- **Strassen, Winograd matrix multiplication^[4]**

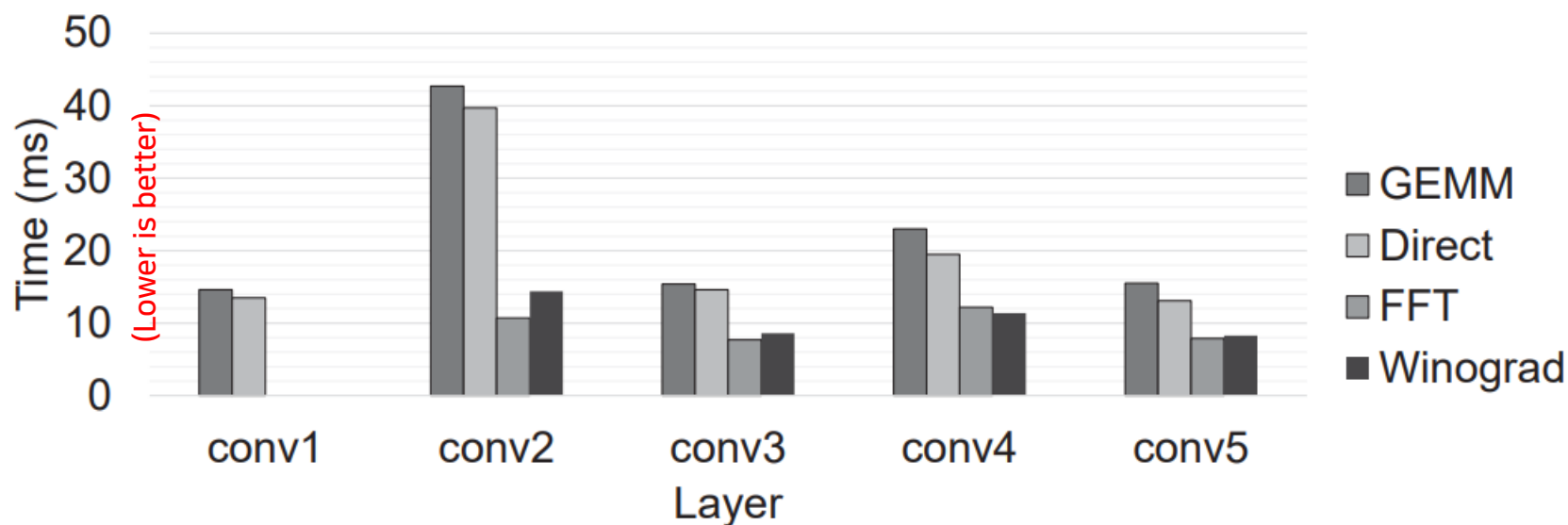
- Multiplication requires more **overhead** than adder (Latency, Power consumption)
- To compute 2×3 , 3×1 matrix multiplication
- Naïve (Mul: 6, Add: 4), Winograd (Mul: 4, Add: 8, Shift: 2)

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

Equation of Winograd matrix multiplication

Key contribution: Why selection ?

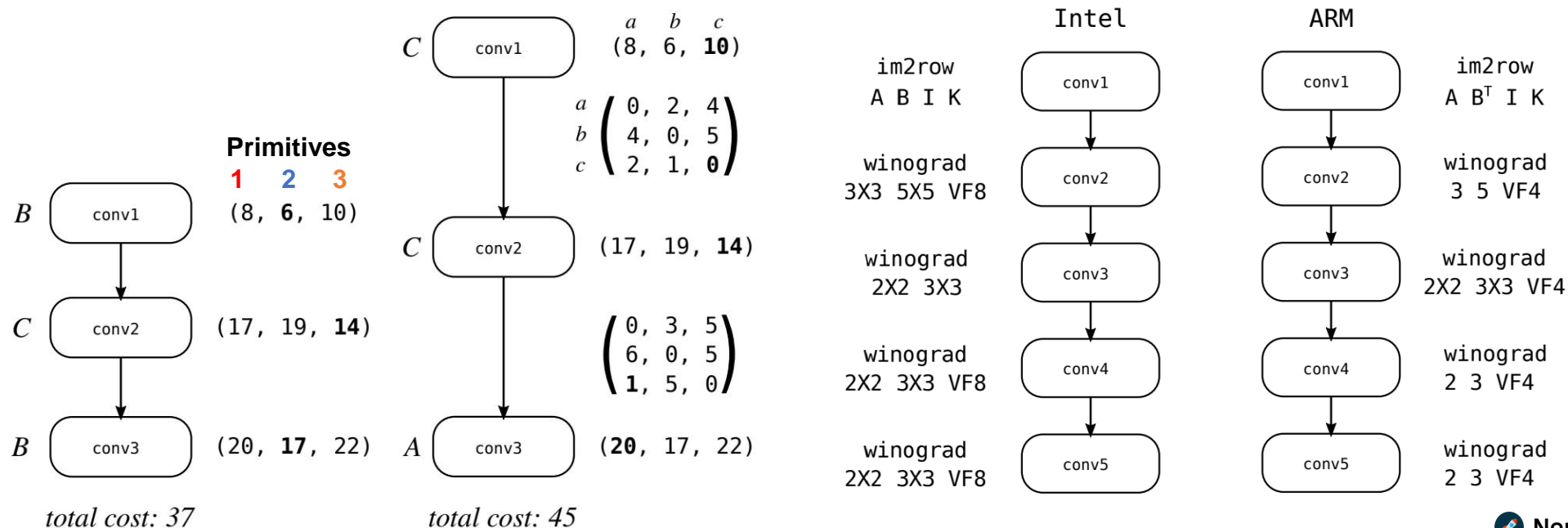
- **Different optimal primitives in each layer^[5]**
 - Optimal primitives depending on device, layer type
 - Heuristic primitive selection running neural network in cuDNN



Optimal primitives in AlexNet

Key contribution: Primitive selection

- **Select the primitive with the lowest cost (Latency)**
 - Set **the target device** and choose **the lowest cost primitive**
 - Pre-computed cost affected by C, H, W, δ , K, M
 - Cost incurred between primitives using Partitioned Boolean Quadratic Programming (PBQP)
 - Ex) data format conversion between primitives



Performance: Intel Haswell

- **PBQP-solver on x86 device**

- Primitive selection on Caffe v1.0
- BLAS: OpenBLAS, MKL-DNN
- Single thread (MKL-DNN is Slightly better in AlexNet, GoogLeNet, not in VGG)
- Multi thread (PBQP is always best)

Network	SUM2D	L . OPT	PBQP	CAFFE
(S) AlexNet	711.75	231.75	100	419.565
(S) GoogLeNet	1401	465.25	249	1267.07
(M) AlexNet	712.25	186	44.25	286.518
(M) GoogLeNet	1400.25	261.5	123.5	919.196

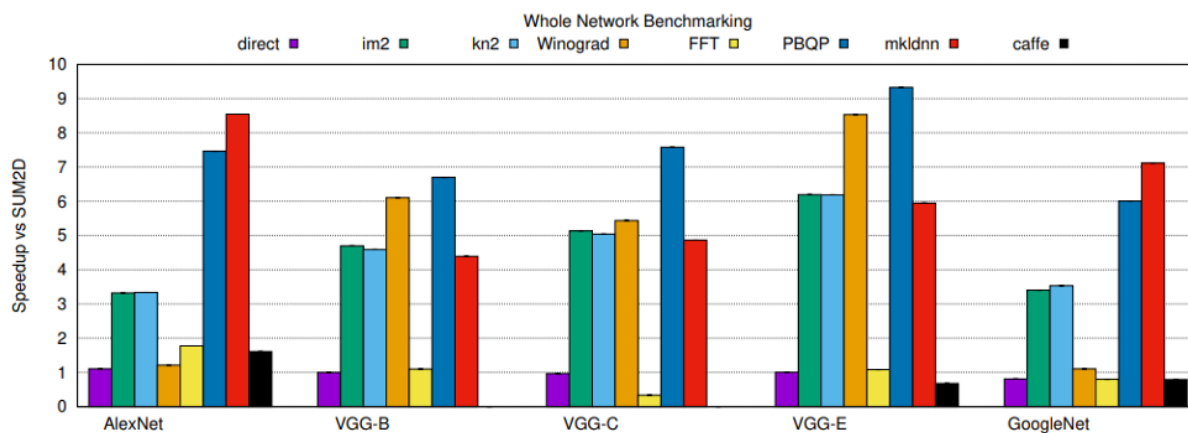


Figure 5. Single-Threaded: Comparison of approaches with PBQP selection on Intel Haswell

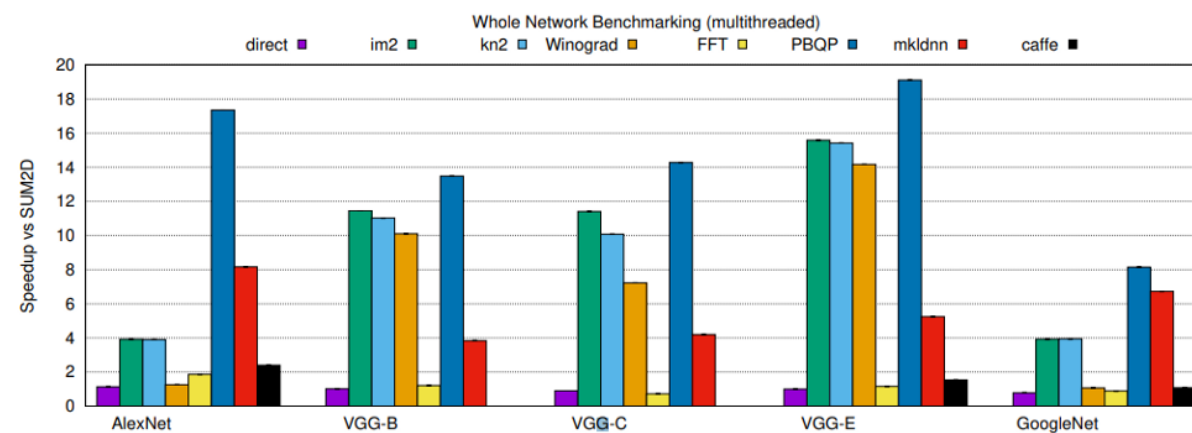


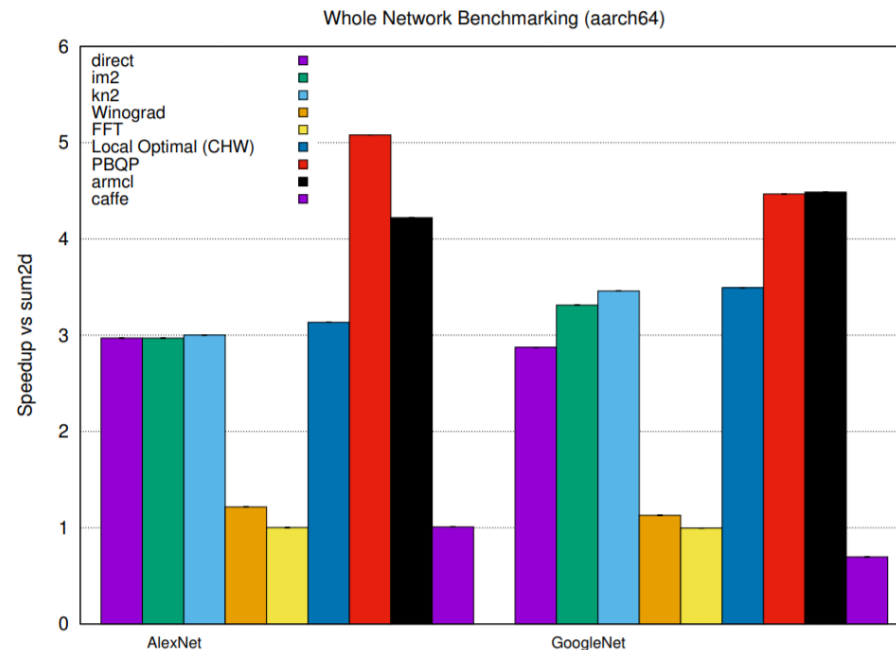
Figure 6. Multi-Threaded: Comparison of approaches with PBQP selection on Intel Haswell

Performance: ARM Cortex-A57

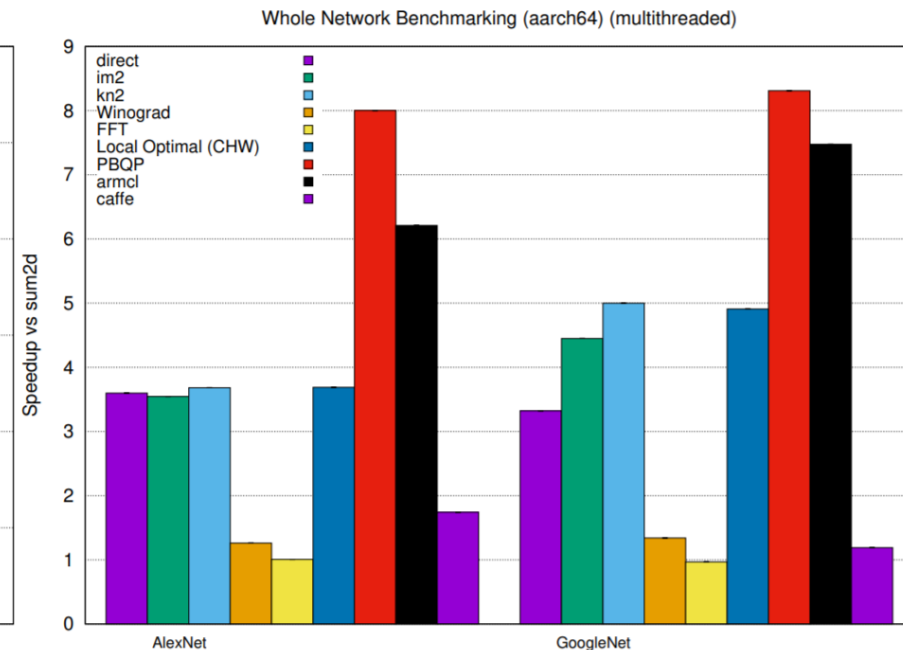
- **PBQP-solver on ARM device**

- Primitive selection on Caffe v1.0
- BLAS: OpenBLAS, ARM's ComputeLibrary

Network	SUM2D	L.OPT	PBQP	CAFFE
(S) AlexNet	2369.5	744.25	461	2341.09
(S) GoogleNet	4544.75	1695.25	1025	5782.4
(M) AlexNet	2432.5	639.25	294	1342.62
(M) GoogleNet	4509.75	919.25	547.5	3707.91



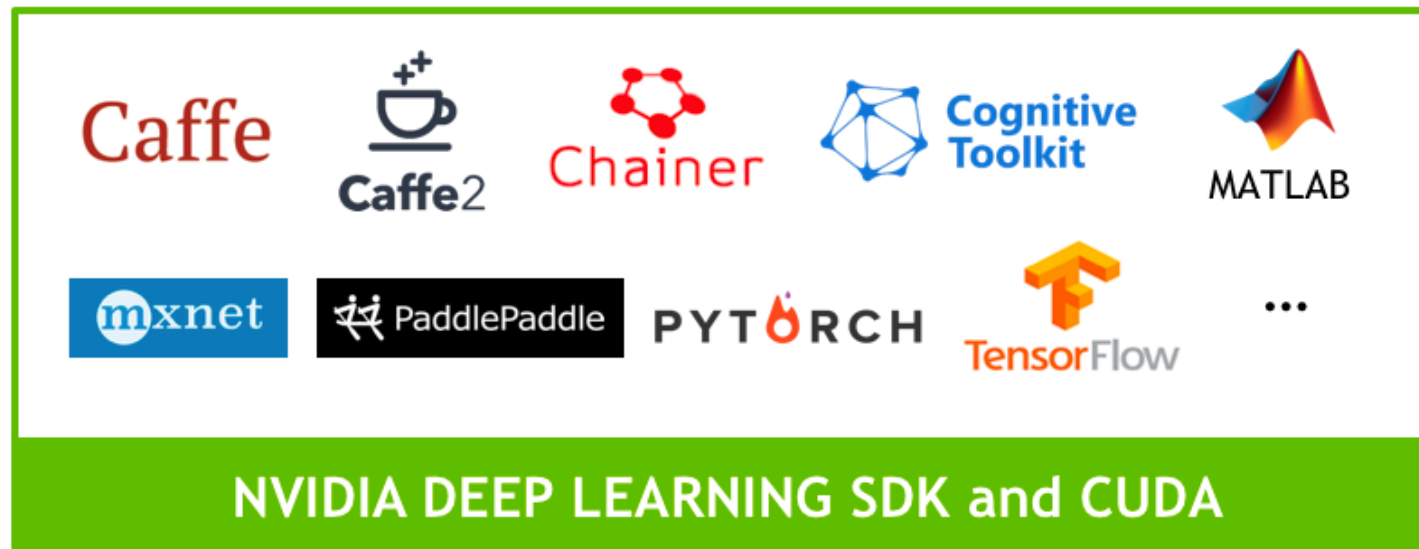
(a) **Single-Threaded** Comparison of approaches with PBQP selection on **ARM Cortex-A57**



(b) **Multi-Threaded** Comparison of approaches with PBQP selection on **ARM Cortex-A57**

Conclusion and Discussion

- **Extremely effective to select the optimal primitives**
 - However, it is effective only pre-computed device
 - Similar concept (cuDNN, Tensorflow XLA)



Conclusion and Discussion: cuDNN

- **Don't trust the primitive selection in cuDNN**
 - cuDNN finds the most optimal primitive while the convolution is calculated
 - If the latency is odd, check the primitive !

4.78. cudnnGetConvolutionBackwardDataAlgorithm

```
cudaStatus_t cudnnGetConvolutionBackwardDataAlgorithm(  
    cudnnHandle_t          handle,  
    const cudnnFilterDescriptor_t    wDesc,  
    const cudnnTensorDescriptor_t    dyDesc,  
    const cudnnConvolutionDescriptor_t convDesc,  
    const cudnnTensorDescriptor_t    dxDesc,  
    cudnnConvolutionBwdDataPreference_t preference,  
    size_t                  memoryLimitInBytes,  
    cudnnConvolutionBwdDataAlgo_t    *algo)
```

This function serves as a heuristic for obtaining the best suited algorithm for a given layer specifications. Based on the input preference, this function will return the fastest algorithm within a given memory limit. For an exhaustive search for the best algorithm, use the function `cudnnFindConvolutionBackwardDataAlgorithm`.

For the following terms, the short-form versions shown in the paranthesis are used in the table below, for brevity:

- CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM (**_IMPLICIT_GEMM**)
- CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM (**_IMPLICIT_PRECOMP_GEMM**)
- CUDNN_CONVOLUTION_FWD_ALGO_GEMM (**_GEMM**)
- CUDNN_CONVOLUTION_FWD_ALGO_DIRECT (**_DIRECT**)
- CUDNN_CONVOLUTION_FWD_ALGO_FFT (**_FFT**)
- CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING (**_FFT_TILING**)
- CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD (**_WINOGRAD**)
- CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED (**_WINOGRAD_NONFUSED**)
- CUDNN_TENSOR_NCHW (**_NCHW**)
- CUDNN_TENSOR_NHWC (**_NHWC**)
- CUDNN_TENSOR_NCHW_VECT_C (**_NCHW_VECT_C**)

Thank you