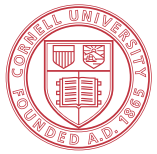
ECE 6775
High-Level Digital Design Automation
Fall 2025

Deep Learning Acceleration on FPGAs



Cornell University

CSL

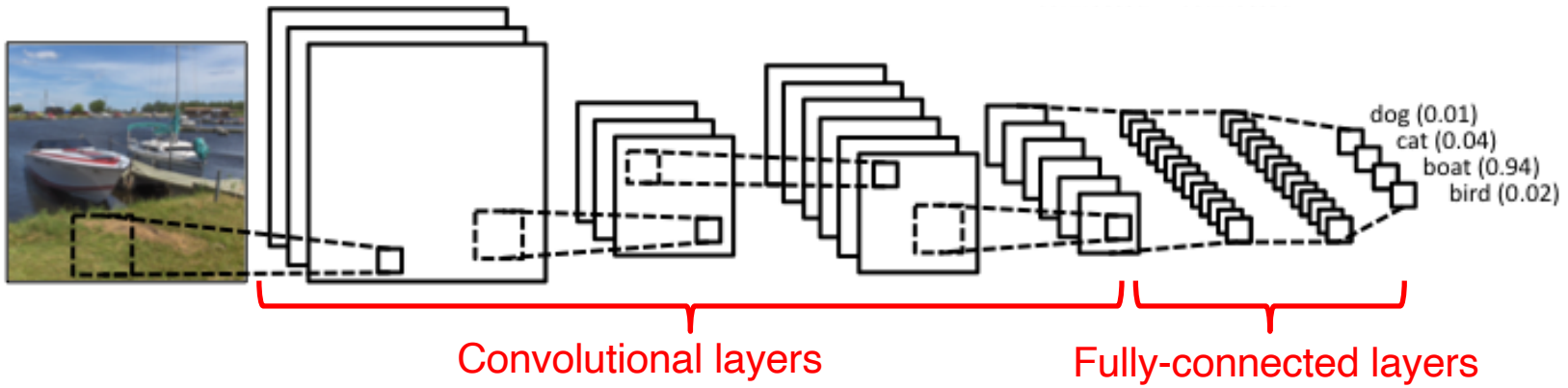
Announcements

- **HW 2 hard deadline**: tonight at 11:59pm
- **Lab 4** teams finalized on CMS: **get started early!**
- **In-class prelim** on Tuesday 10/28
  - **Coverage: Lectures 01–11, 13, 14**
    - Lec12 NN tutorial excluded
    - Lec15 includes useful discussion on URE and pipelining
    - Lec16 (today) reviews the roofline model
  - We'll do a more in-depth prelim review in the next lecture

Agenda

- CNN acceleration on FPGAs
- Potential of FPGA-based LLM inference

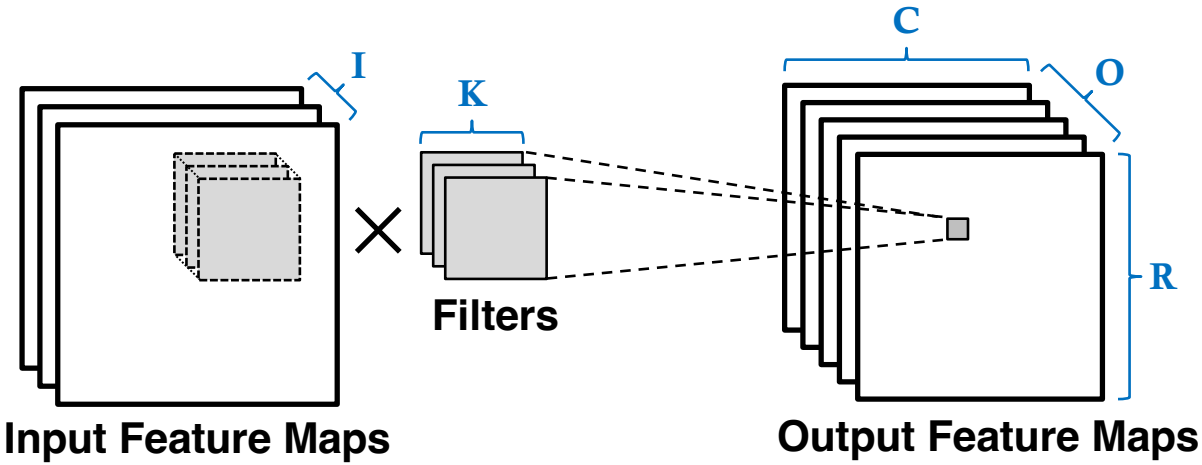
Recap: Convolutional Neural Network (CNN)



- Front: convolutional layers learn visual features
- Back: fully-connected layers perform classification using the visual features

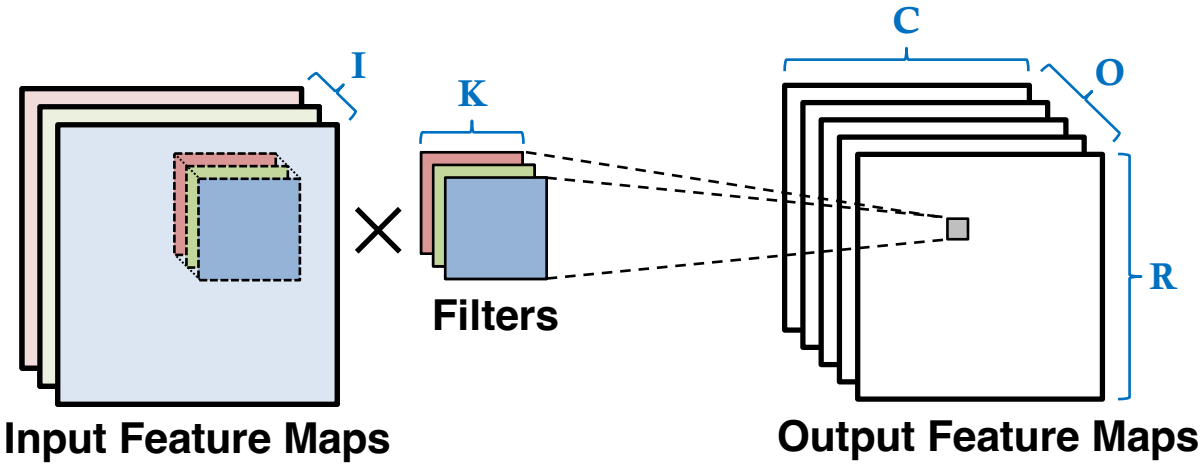
Image credit: http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

Convolutional Layer



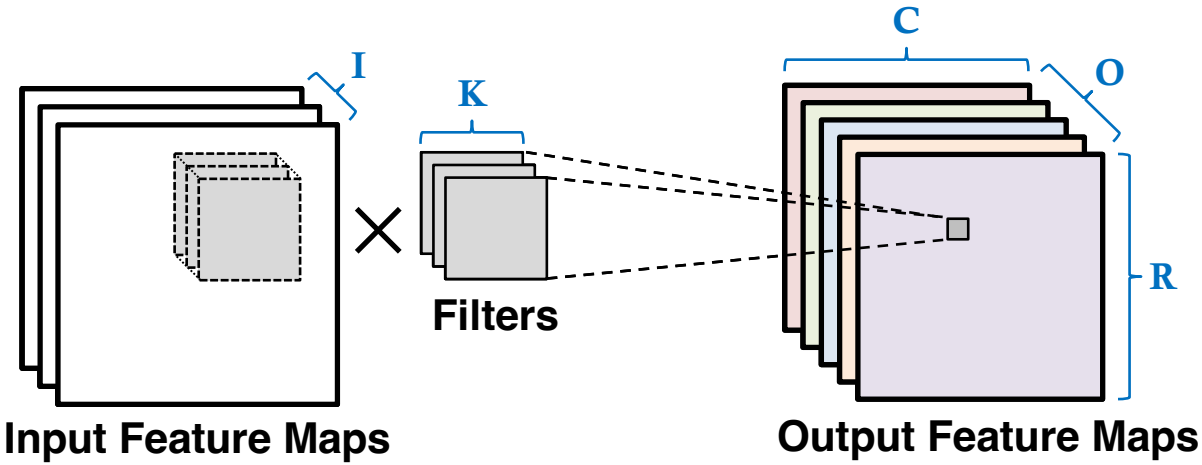
- An output pixel is connected to its neighboring region on each input feature map (fmap)
- All pixels on an output feature map use the same filter weights

Parallelism in the Convolutional Layer



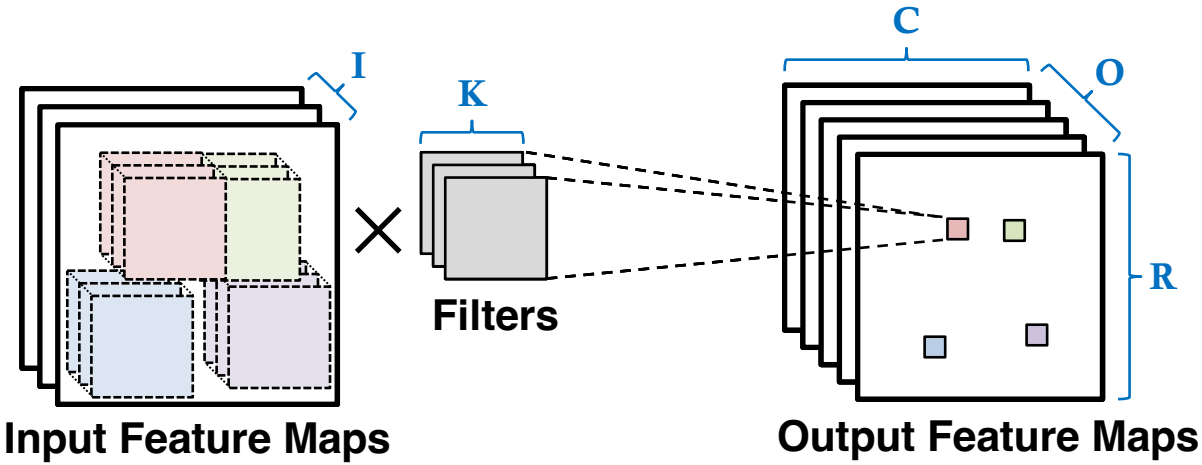
- Four main sources of parallelism
  1. **Across input feature maps (i.e., input channels)**

Parallelism in the Convolutional Layer



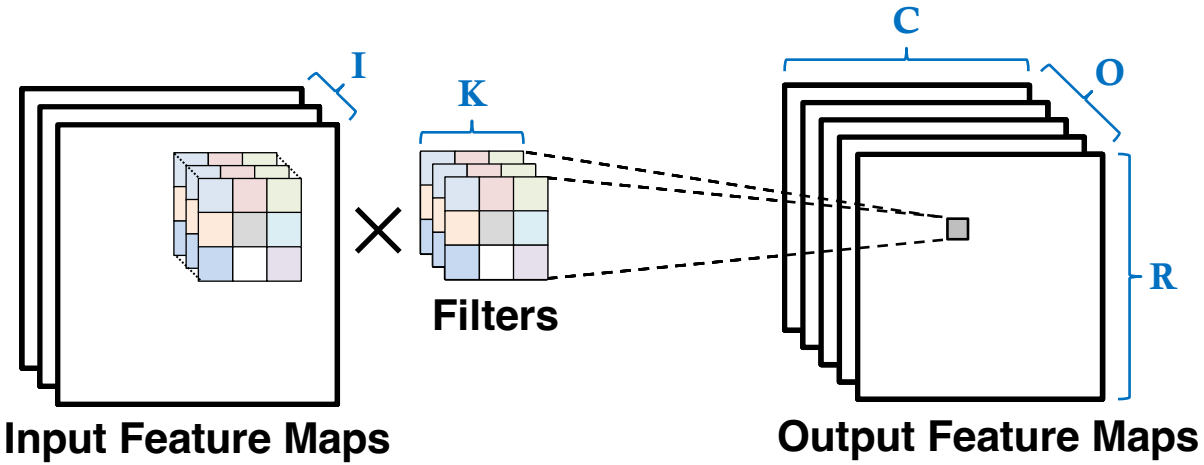
- Four main sources of parallelism
  1. Across input feature maps (i.e., input channels)
  2. **Across output feature maps (i.e., output channels)**

Parallelism in the Convolutional Layer



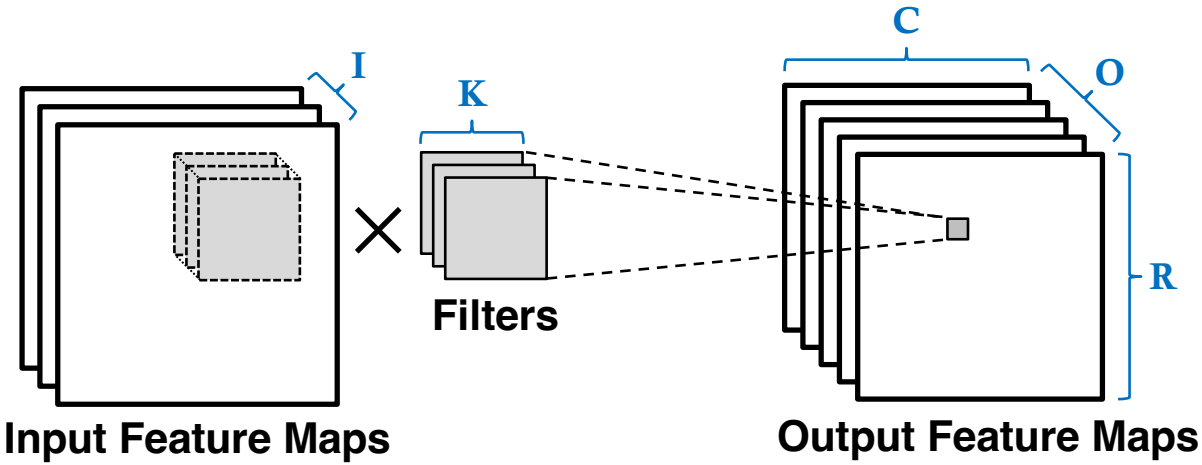
- Four main sources of parallelism
  1. Across input feature maps (i.e., input channels)
  2. Across output feature maps (i.e., output channels)
  3. **Across different output pixels (i.e., filter positions)**

Parallelism in the Convolutional Layer



- Four main sources of parallelism
  1. Across input feature maps (i.e., input channels)
  2. Across output feature maps (i.e., output channels)
  3. Across different output pixels (i.e., filter positions)
  4. **Across filter pixels**

Parallelism in the Code




```
1 for (row=0; row<R; row++) {
2    for (col=0; col<C; col++) {
3       for (to=0; to<O; to++) {
4          for (ti=0; ti<I; ti++) {
5             for (ki=0; ki<K; ki++) {
6                for (kj=0; kj<K; kj++) {
                    output_fm[to][row][col] +=
                    weights[to][ti][ki][kj]*input_fm[ti][S*row+ki][S*col+kj];
}}}}}}
```

Loop over output pixels

Loop over output channels

Loop over input channels

Loop over filter pixels

S is the stride

Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks

Cheng Zhang[1], Peng Li[2], Guangyu Sun[1,3], Yijin Guan[1], Bingjun Xiao[2], Jason Cong[2,3,1]

[1] Peking University
[2] UCLA
[3] PKU/UCLA Joint Research Institute in Science and Engineering

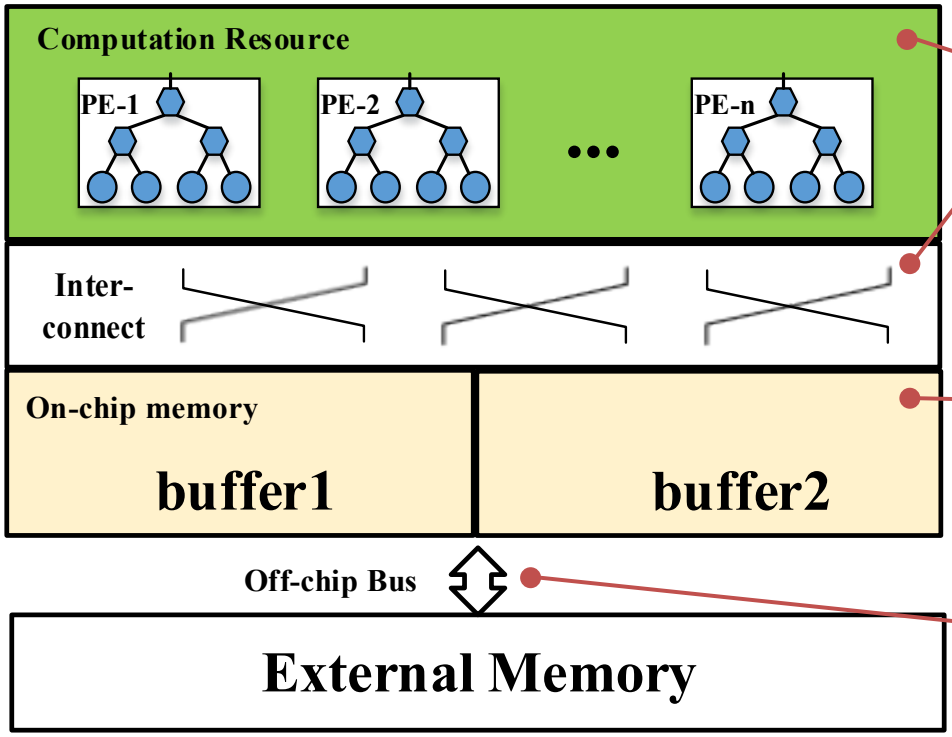
FPGA 2015

Main Contributions

1. Analysis of the different sources of parallelism in the convolution kernel of a CNN
2. Quantitative performance modeling of the hardware design space using the Roofline method
3. Design and implementation of a CNN accelerator for FPGA using Vivado HLS, evaluated on AlexNet

Challenges to FPGA Acceleration

- We can't just unroll all the loops due to limited FPGA resources
- Must choose the right code transformations to exploit the parallelism in a resource efficient way



Limited compute and routing resources

Technique: loop unrolling, pipelining, interchange

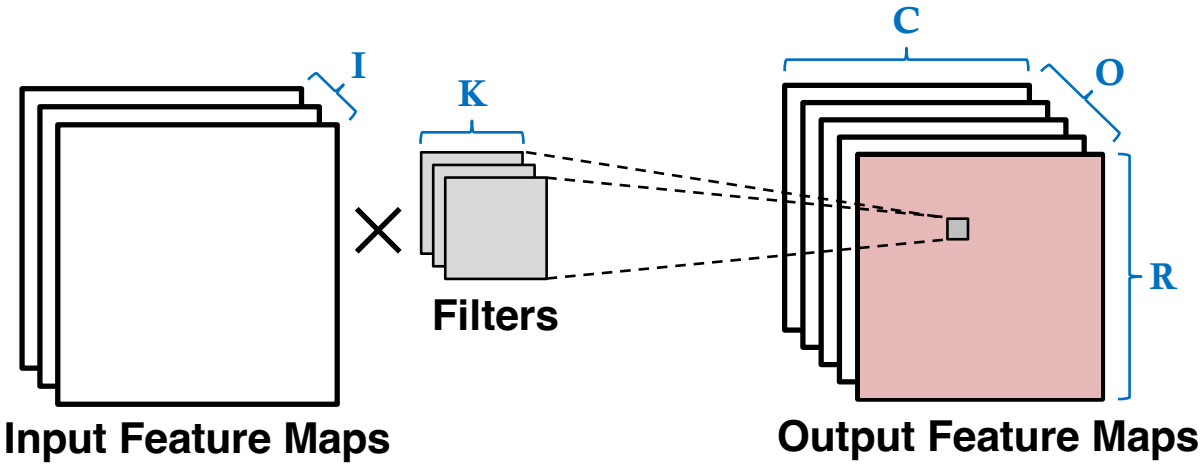
Limited on-chip storage

Technique: loop tiling

Limited off-chip bandwidth

Technique: data reuse, compute-memory balance

Motivation for Loop Tiling



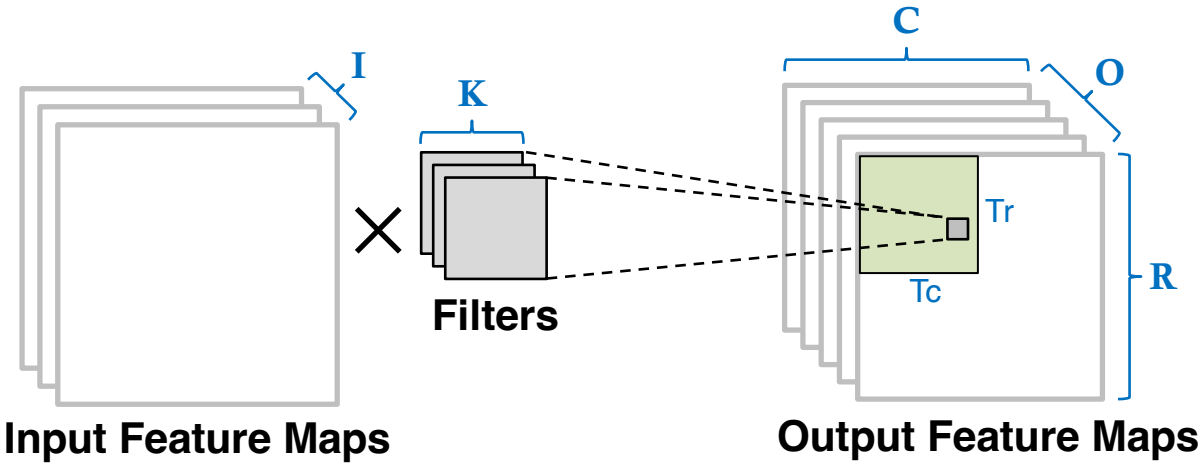

```
1 for (row=0; row<R; row++) {
2   for (col=0; col<C; col++) {
3     for (to=0; to<O; to++) {
4       for (ti=0; ti<I; ti++) {
5         for (ki=0; ki<K; ki++) {
6           for (kj=0; kj<K; kj++) {
              output_fm[to][row][col] +=
              weights[to][ti][ki][kj]*input_fm[ti][S*row+ki][S*col+kj];
}}}}}}
```

Loop over **pixels** in an output fmap

If we offload these loops, we must store entire output feature maps on-chip

Loop Tiling

Offloading just the inner loops requires only a small portion of the data to be stored on FPGA chip




```
1 for (row=0; row<R; row+=Tr) {
2   for (col=0; col<C; col+=Tc) {
3     for (to=0; to<O; to++) {
4       for (ti=0; ti<I; ti++) {
5         for (trr=row; trr<min(row+Tr, R); trr++) {
6           for (tcc=col; tcc<min(col+Tc, C); tcc++) {
7             for (ki=0; ki<K; ki++) {
8               for (kj=0; kj<K; kj++) {
                  output_fm[to][trr][tcc] +=
                  weights[to][ti][ki][kj]*input_fm[ti][S*trr+ki][S*tcc+kj];
          }}}}}}
}}}}
```

Loop over different **tiles**

Loops over **pixels** in each tile

Code with Loop Tiling

Tile sizes: rows (Tr), columns (Tc), input channels (Ti), and output channels (To)

```
1 for (row=0; row<R; row+=Tr) {
2    for (col=0; col<C; col+=Tc) {
3       for (to=0; to<O; to+=To) {
           // software: write output feature map
4          for (ti=0; ti<I; ti+=Ti) {
              // software: write input feature map + filters

5             for (trr=row; trr<min(row+Tr, R); trr++) {
6                for (tcc=col; tcc<min(col+Tc, C); tcc++) {
7                   for (too=to; too<min(to+To, O); too++) {
8                      for (tii=ti; tii<min(ti+Ti, I); tii++) {
9                         for (ki=0; ki<K; ki++) {
10                           for (kj=0; kj<K; kj++) {
                                output_fm[too][trr][tcc] +=
                                weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
              }}}}}}
           }
           // software: read output feature map
}}}
```

CPU Portion

FPGA Portion (Accelerator)

Optimizing for Accelerator Performance

```
5           for (trr=row; trr<min(row+Tr, R); trr++) {
6              for (tcc=col; tcc<min(col+Tc, C); tcc++) {
7                 for (too=to; too<min(to+To, O); too++) {
8                    for (tii=ti; tii<min(ti+Ti, I); tii++) {
9                       for (ki=0; ki<K; ki++) {
10                         for (kj=0; kj<K; kj++) {
                              output_fm[too][trr][tcc] +=
                              weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
            }}}}}}
```

FPGA Portion (Accelerator)

Optimizing for Accelerator Performance

```
9           for (ki=0; ki<K; ki++) {
10            for (kj=0; kj<K; kj++) {
5               for (trr=row; trr<min(row+Tr, R); trr++) {
6                 for (tcc=col; tcc<min(col+Tc, C); tcc++) {
7                   for (too=to; too<min(to+To, O); too++) {
8                     for (tii=ti; tii<min(ti+Ti, I); tii++) {
                        output_fm[too][trr][tcc] +=
                        weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
          }}}}}}
```

Reorder these two loops to the top level

Optimizing for Accelerator Performance

```
9           for (ki=0; ki<K; ki++) {
10             for (kj=0; kj<K; kj++) {
5                 for (trr=row; trr<min(row+Tr, R); trr++) {
6                     for (tcc=col; tcc<min(col+Tc, C); tcc++) {
                          #pragma HLS pipeline
7                         for (too=to; too<min(to+To, O); too++) {
8                             for (tii=ti; tii<min(ti+Ti, I); tii++) {
                                  output_fm[too][trr][tcc] +=
                                  weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
            }}}}}}
```

Optimizing for Accelerator Performance

```
9           for (ki=0; ki<K; ki++) {
10             for (kj=0; kj<K; kj++) {
5                 for (trr=row; trr<min(row+Tr, R); trr++) {
6                     for (tcc=col; tcc<min(col+Tc, C); tcc++) {
                          #pragma HLS pipeline
7                         for (too=to; too<min(to+To, O); too++) {
                             #pragma HLS unroll
8                            for (tii=ti; tii<min(ti+Ti, I); tii++) {
                                output_fm[too][trr][tcc] +=
                                weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
            }}}}}}
```

Optimizing for Accelerator Performance

```
9           for (ki=0; ki<K; ki++) {
10            for (kj=0; kj<K; kj++) {
5               for (trr=row; trr<min(row+Tr, R); trr++) {
6                 for (tcc=col; tcc<min(col+Tc, C); tcc++) {
                    #pragma HLS pipeline
7                   for (too=to; too<min(to+To, O); too++) {
                      #pragma HLS unroll
8                     for (tii=ti; tii<min(ti+Ti, I); tii++) {
                        #pragma HLS unroll
                        output_fm[too][trr][tcc] +=
                        weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
            }}}}}}
```

Parallelize across input (Ti) and output (To) channels

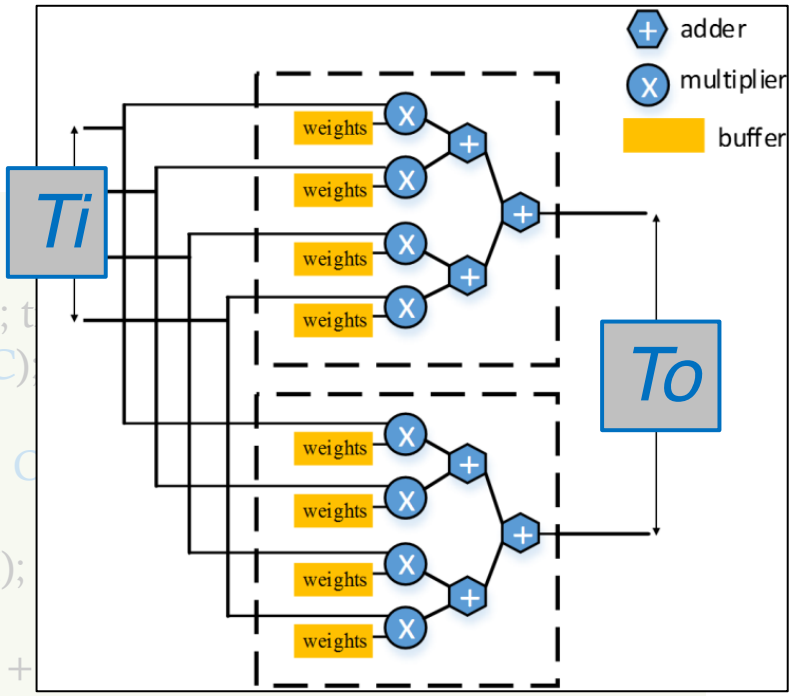
Number of cycles to execute the above loop nest

$$\approx K \times K \times Tr \times Tc + L \approx Tr \times Tc \times K^2$$

L is the pipeline depth (i.e., # of pipeline stages), assume II=1

Optimizing for Accelerator Performance

```
9           for (ki=0; ki<K; ki++) {
10            for (kj=0; kj<K; kj++) {
5               for (trr=row; trr<min(row+Tr, R); t
6                 for (tcc=col; tcc<min(col+Tc, C)
                    #pragma HLS pipeline
7                   for (too=to; too<min(to+To, C
                      #pragma HLS unroll
8                     for (tii=ti; tii<min(ti+Ti, I);
                        #pragma HLS unroll
                        output_fm[too][trr][tcc] +
                        weights[too][tii][ki][kj]*input_fm[tii][S*trr+ki][S*tcc+kj];
          }}}}}}
```

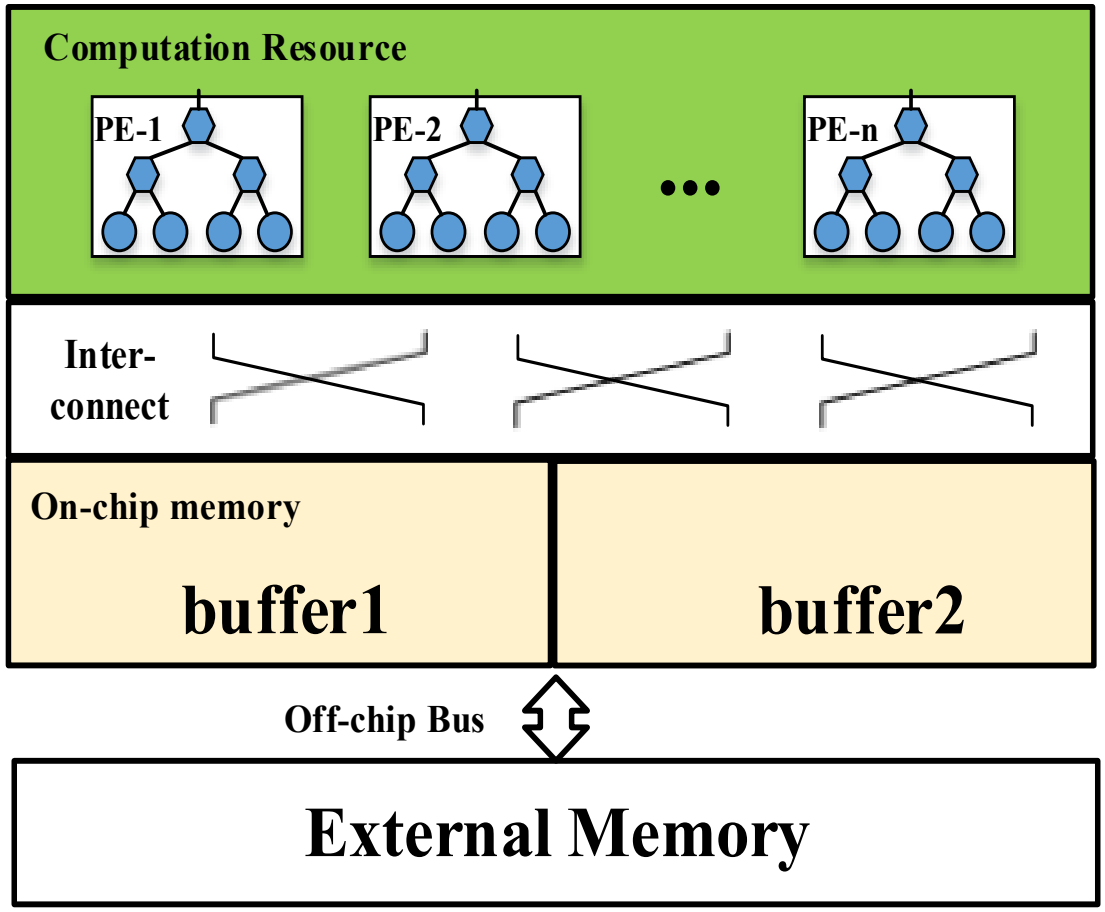


Generated Hardware

Throughput and size of the accelerator determined by tile factors *Ti* and *To*

Number of data transfers largely determined by Tr and Tc

Overall Accelerator Architecture



Design Space Complexity

- **Challenge:** Number of available optimizations present a huge space of possible designs
  - What is the optimal loop order?
  - What tile size to use for each loop?

- Implementing and testing each design by hand will be slow and error-prone
  - Some designs will exceed the on-chip compute/memory capacity

- **Solution:** Performance modeling + automated design space exploration

Performance Modeling

- Three design metrics are estimated:
  - **Total number of operations (FLOP)**
    - Depends on the CNN model parameters
  - **Total external memory access (Byte)**
    - Depends on the CNN weight and activation size
  - **Total execution time (Sec)**
    - Depends on the hardware architecture (e.g., tile factors To and Ti)

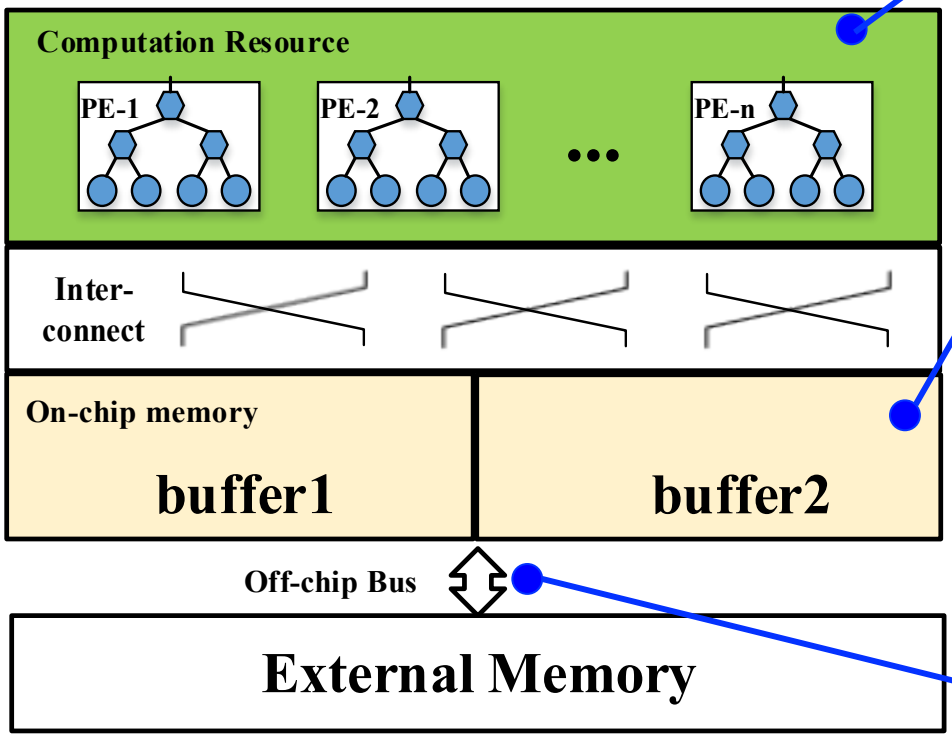
Performance Modeling

- Total operations: FLOPs $\approx 2\times O\times I\times R\times C\times K^2$

- Execution time = Number of Cycles × Clock Period
  - Number of cycles $\approx \left\lceil\frac{O}{To}\right\rceil\times\left\lceil\frac{I}{Ti}\right\rceil\times\left\lceil\frac{R}{Tr}\right\rceil\times\left\lceil\frac{C}{Tc}\right\rceil\times(Tr\times Tc\times K^2)$
    $\approx \left\lceil\frac{O}{To}\right\rceil\times\left\lceil\frac{I}{Ti}\right\rceil\times R\times C\times K^2$

- External memory accesses $= ao\times Bo + ai\times Bi + aw\times Bw$
  - Size of output fmap buffer: ***Bo*** $= To\times Tr\times Tc$
  - Size of input fmap buffer: ***Bi*** $= Ti\times(Tr+K-1)(Tc+K-1)$ with stride=1
  - Size of weight buffer: ***Bw*** $= Ti\times To\times K^2$
  - External access times: ***ao*** $= \left\lceil\frac{O}{To}\right\rceil\times\left\lceil\frac{R}{Tr}\right\rceil\times\left\lceil\frac{C}{Tc}\right\rceil$, ***ai*** $=$ ***aw*** $= \left\lceil\frac{I}{Ti}\right\rceil\times ao$
    - ***ao***, ***ai***, and ***aw*** denote the number of transactions between external memory and the output, input, and weight buffers
    - Input features and weights are reused across multiple output tiles

Revisiting the Roofline Model



Computational Throughput

$$= \frac{\text{Total number of operations}}{\text{Total execution time}}$$

(OPs / Sec)

Operational Intensity* (OI)

$$= \frac{\text{Total number of operations}}{\text{Total external memory accesses}}$$

(OPs / Byte Accessed)

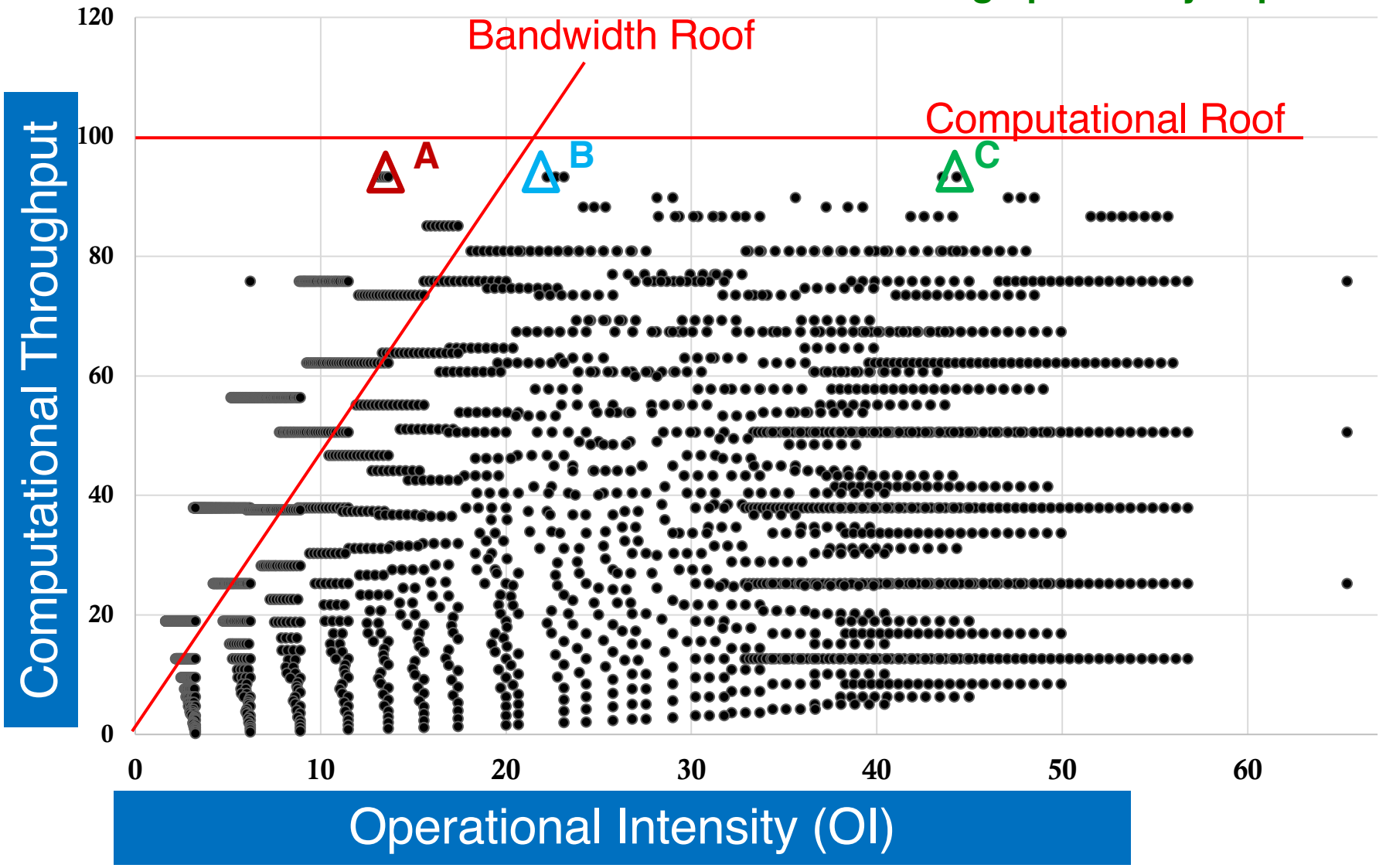
Memory Bandwidth

(Bytes / Sec)

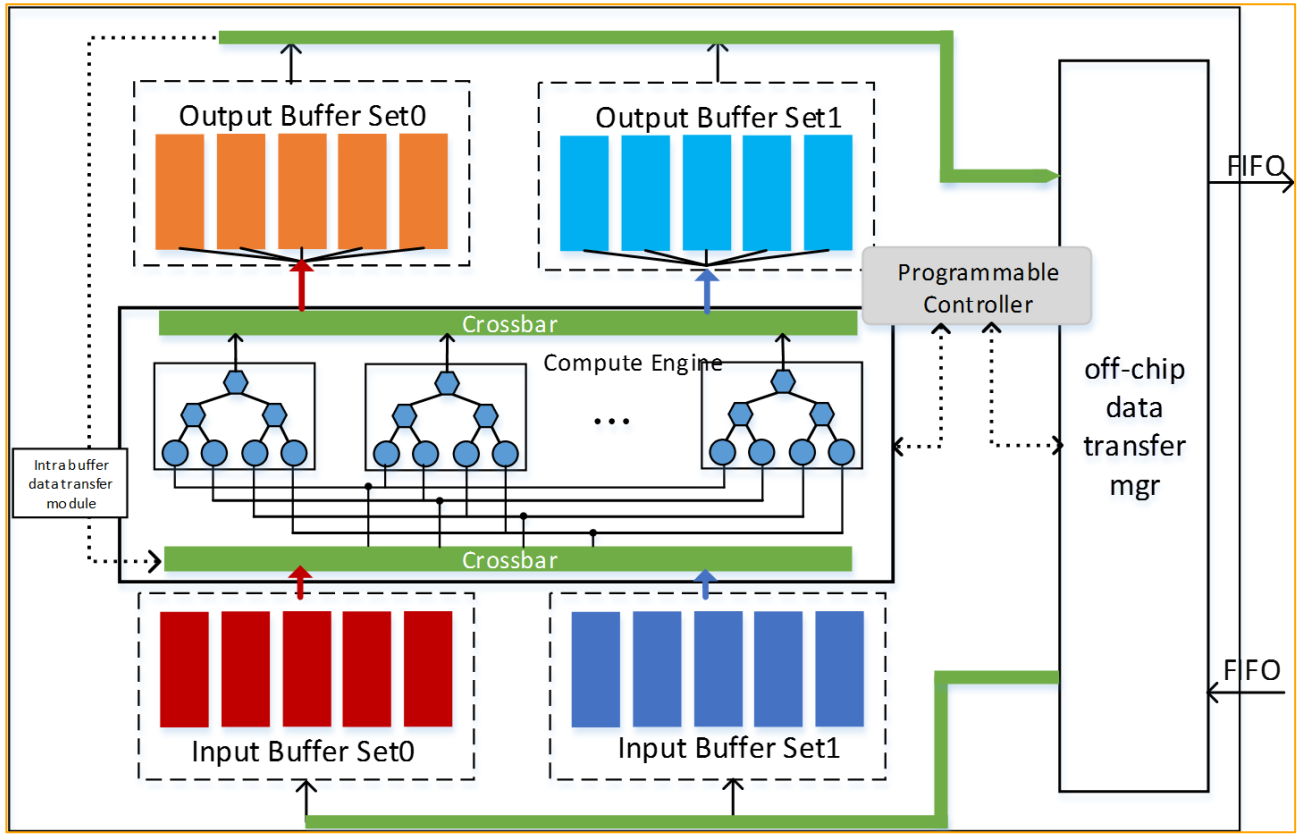
* **OI** is also known as computation to communication ratio (CTC) or arithmetic intensity (AI)

Design Space Exploration with Roofline

Which design point do you prefer?



FPGA Implementation



- All values in floating-point
- Only handles Conv layers, not dense or pooling
- Target FPGA: Virtex7-485t (28nm)

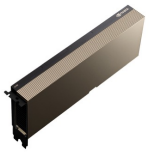
Understanding the Potential of FPGA-Based Spatial Acceleration for Large Language Model Inference

Hongzheng Chen[1], Jiahao Zhang[2,1], Yixiao Du[1], Shaojie Xiang[1], Zichao Yue[1], Niansong Zhang[1], Yaohui Cai[1], Zhiru Zhang[1]

[1] Cornell University
[2] Tsinghua University

ACM TRETS 2024 (FCCM Journal Track)

The Era of Large Language Models (LLMs)

Devices	Cost	Energy
		
A100 GPU (7nm)	~$20K	~250W
		
U280 FPGA (16nm)	~$8K	~30W

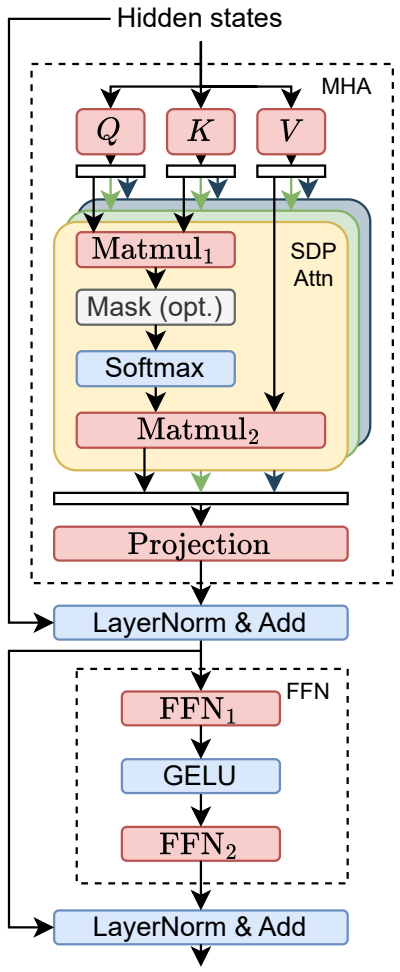
Is FPGA suitable for efficient LLM inference?

Transformer GPU Execution Breakdown

- LLMs aren't just about matrix multiplications (MatMul)
- Non-linear & element-wise operators also play a significant role
  - Low compute-to-memory ratio (namely, low OI)
  - High kernel launch overheads

Operator Class	Representative Op	% FLOP	% Run Time
Tensor contraction	MM, MV	**99.80**	61.0
Stat. normalization	softmax, layernorm	0.17	**25.5**
Element-wise	bias, dropout	0.03	**13.5**

Proportions for operator classes in the BERT model, implemented in PyTorch, profiled with an NVIDIA V100 GPU
(MM: matrix-matrix multiply; MV: matrix-vector multiply)

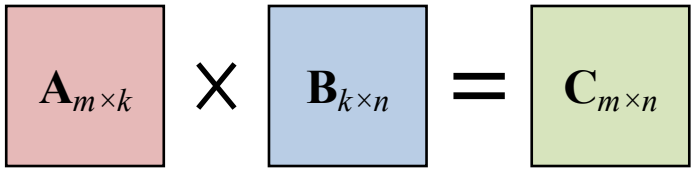


Data source: Andrei Ivanov et al., "Data Movement is All You Need: A Case Study on Optimizing Transformers", arXiv:2007.00072, 2021.

Two-Stage LLM Inference: OI Analysis

Stage 1: **Prefill**
Takes in user prompts and generates the first token

MM



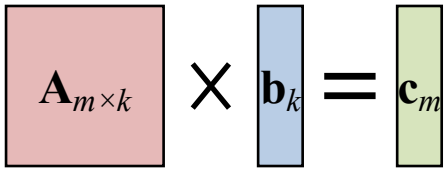
$$\frac{\text{compute}}{\text{memory footprint}} = \frac{mkn}{mk + kn + mn}$$

MM OI: Cubic / Quadratic

Compute bound: performance dictated by the compute roof

Stage 2: **Decode**
Processes the previously generated token to produce new tokens one at a time in an auto-regressive manner

MV



$$\frac{\text{compute}}{\text{memory footprint}} = \frac{mk}{mk + k + m}$$

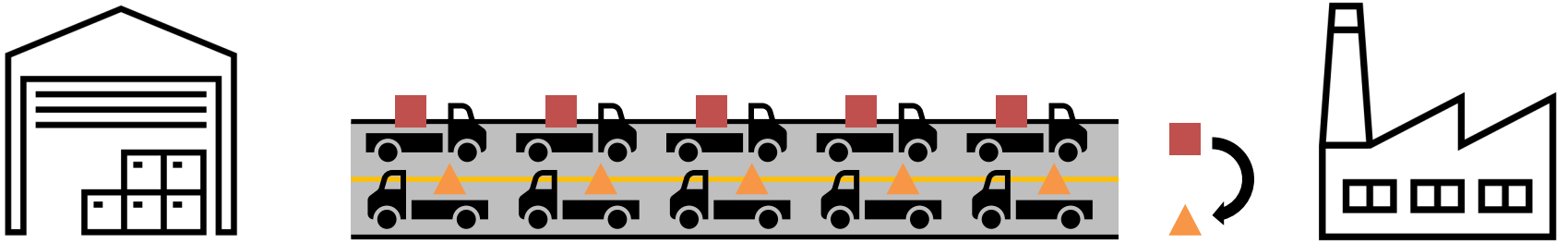
MV OI: Quadratic / Quadratic

Memory bound: performance dictated by the bandwidth roof

Typical GPU Execution Cycle for a Single Operator

- **Analogy**: Compute is like a factory, requiring instructions (kernel launch overhead) and a steady data supply (memory bandwidth) to run efficiently

- If compute efficiency grows faster than data supply, it limits the system's ability to operate at peak performance



Warehouse
(**memory**)

Factory
(**compute units**)

Diagrams adopted from Horace He, "Making Deep Learning Go Brrrr From First Principles", https://horace.io/brrr_intro.html

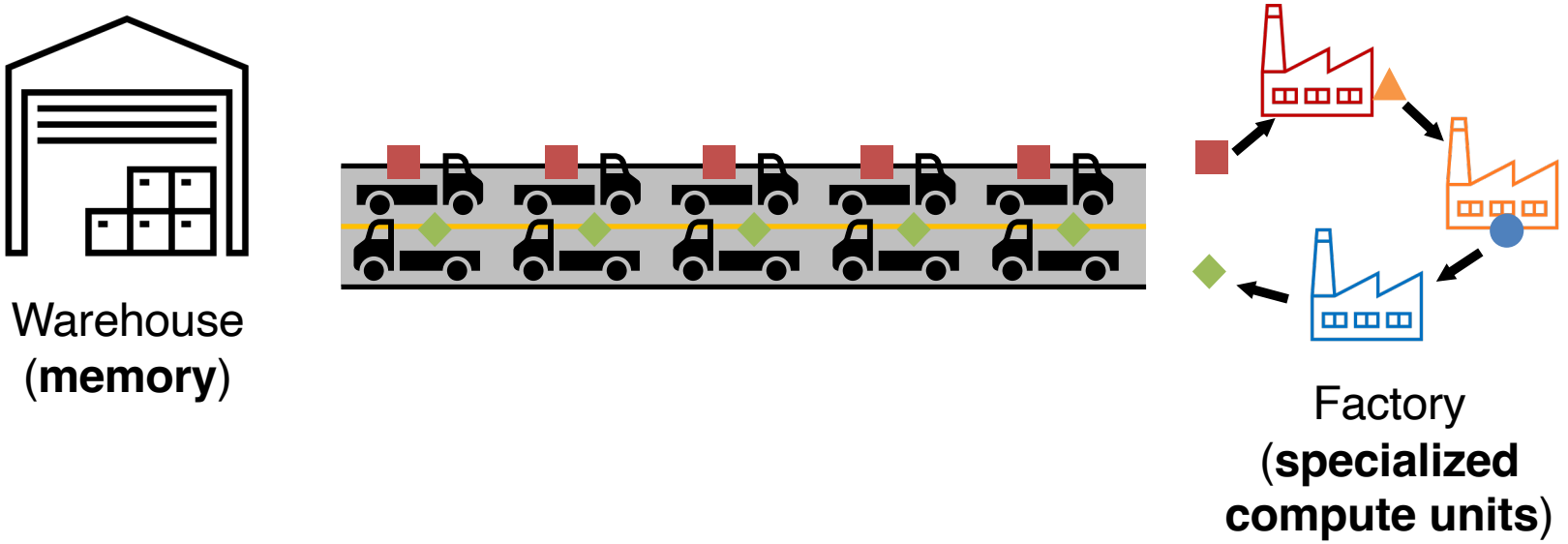
Typical GPU Execution Cycle for Multiple Operators

- Moving data to and from GPU compute units incurs a high memory bandwidth cost
- For memory-bound operations, more time is spent moving data than performing computation

Diagrams adopted from Horace He, "Making Deep Learning Go Brrrr From First Principles", https://horace.io/brrr_intro.html

The Potential of Dataflow Execution

- Model-specific dataflow accelerator
  - Pass intermediate results directly to the next operator
  - Reduced memory traffic leads to higher performance and higher energy efficiency



Warehouse
(**memory**)

Factory
(**specialized compute units**)

Contributions of This Work

- An analytical model to estimate performance and resource/bandwidth usage for model-specific dataflow accelerator on FPGAs
  - **Compute resource**: M is compute power in MACs/cycle and C is the # of layers per FPGA

$$\sum M_i C < M_{\text{tot}}, i \in \{q,k,v,a_1,a_2,p,f_1,f_2\}$$

  - **Memory capacity**: S is buffer size

$$S_{\text{param}} C < \text{DRAM}_{\text{tot}},$$
$$\sum S_i C < \text{SRAM}_{\text{tot}}, i \in \{\text{tile,KV,FIFO}\}$$

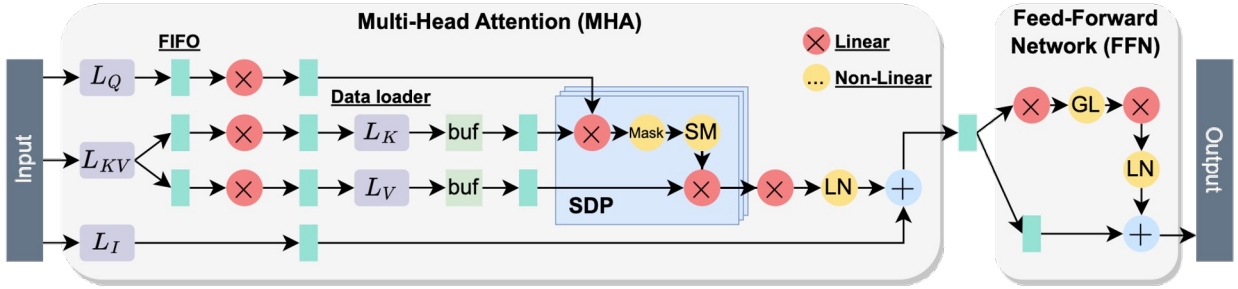
  - **Memory port**: s is tensor size and b is bitwidth

$$R_i = \left\lceil \frac{s_i b_{BRAM}}{M_i/r_i \times S_{BRAM}} \right\rceil \times \frac{M_i/r_i}{k}$$
$$\sum_i C R_i + 2C(R_{a_1} + R_{a_2}) < \text{SRAM}_{\text{tot}}, i \in \{q,k,v,p,f_1,f_2\}$$

  - **Memory bandwidth**: B is bandwidth

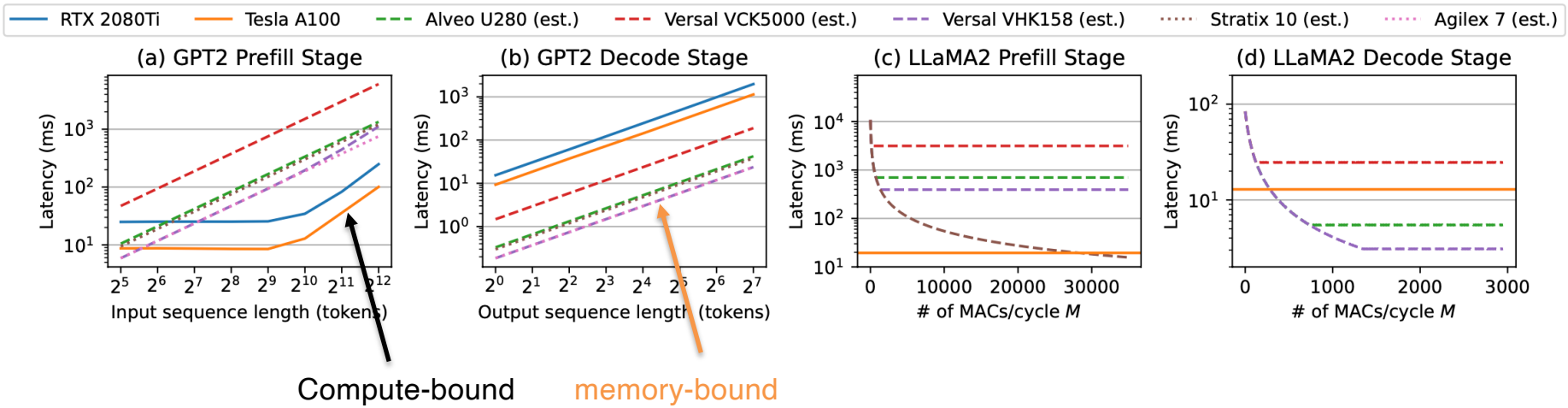
$$\sum_i C B_i < B_{\text{tot}}, i \in \{q,k,v,p,f_1,f_2\}$$

- An optimized HLS kernel library and compose accelerator designs for LLM on FPGAs



Estimation on Different Models and Devices

- Is FPGA capable of efficient LLM inference?
  - Latency estimation of GPT2 and LLaMA2 on different FPGAs
  - GPU results are obtained through actual profiling



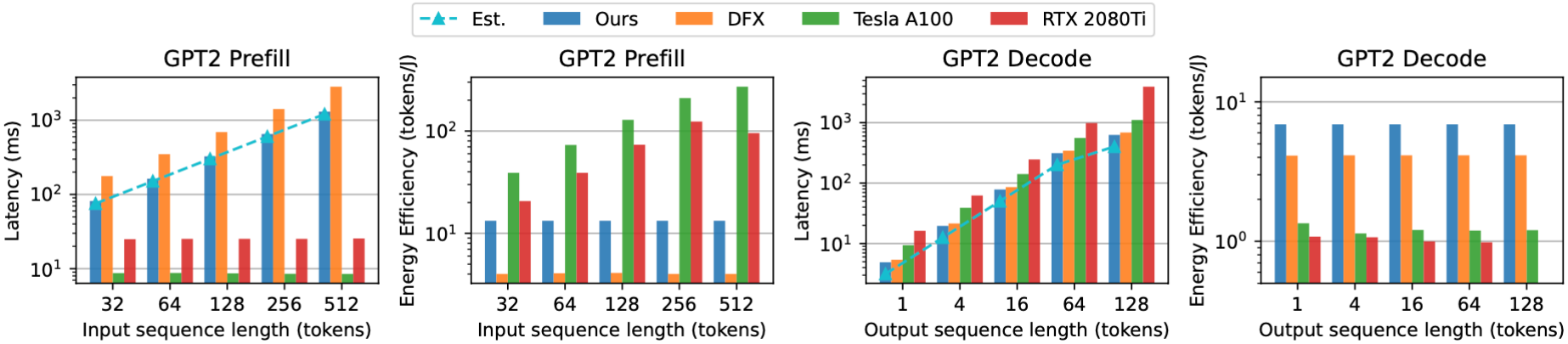
Existing FPGAs are inferior in the compute-intensive prefill stage, but can outperform GPUs in the memory-intensive decode stage

FPGA Implementation of an GPT2 Accelerator

- GPT2: single-batch, low-latency settings, adjust input/output sequence length
  - Implemented on AMD Alveo U280, W8A8 quantization, 250MHz
  - 92% estimation accuracy in the prefill stage using the analytical framework
  - 2.2x speedup in prefill over DFX (an FPGA-based temporal arch.)
  - **1.7x speedup in decode stage, 5.4x more energy-efficient vs A100**

Alveo U280(ours) *16nm*
Max M: 1289
Bandwidth: **498 GB/s**

A100 GPU *7nm*
Max M: 16415 (temporal)
Bandwidth: **1935 GB/s**



https://github.com/cornell-zhang/allo/tree/main/examples

Acknowledgements

- This tutorial contains/adapts materials developed by
  - Ritchie Zhao (Cornell ECE PhD, now NVIDIA)
  - Authors of the following papers
    - Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks (FPGA'15, PKU-UCLA)
    - Understanding the Potential of FPGA-Based Spatial Acceleration for Large Language Model Inference (ACM TRETS, FCCM'24 Journal Track)

Next Lecture

- Prelim Review