Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks

High Performance Computing 2015

Matsuoka Lab.
Tokyo Institute of Technology, Japan
King Mongkut’s Institute of Technology Ladkrabang, Thailand

Piyawath Boukom

Paper

Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks

Chen Zhang, Guangyu Sun, Yijin Guan - Peking University, Beijing, China

Peng Li, Bingjun Xiao - University of California, Los Angeles, USA

Jason Cong - PKU/UCLA Joint Research Institute in Science and Engineering

FPGA’15 - 23rd 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (Pages 161 - 170)

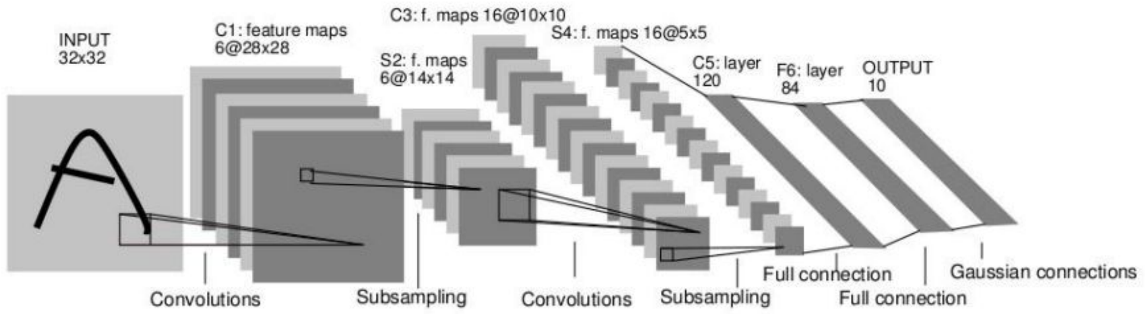
Link: http://dl.acm.org/citation.cfm?id=2689060

Outline

Introduction

Convolutional neural network (CNN) is a deep learning architecture extended from artificial neural network.

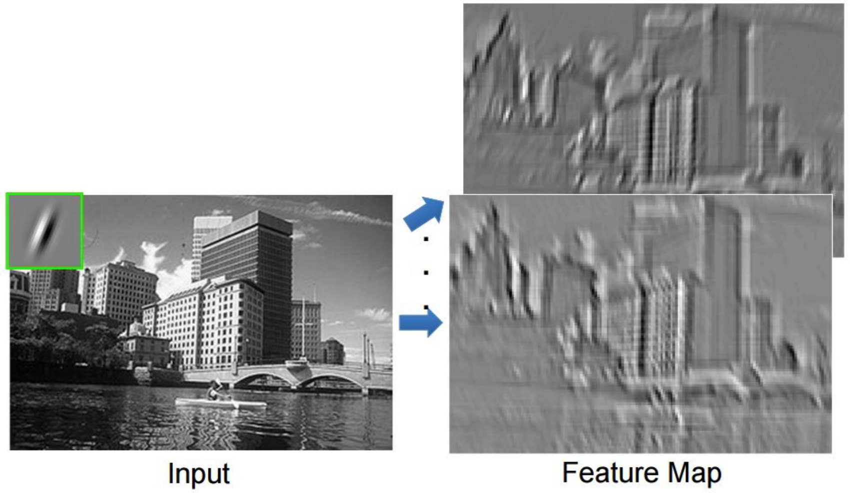
A CNN design processes data with multiple layers of neuron connections to achieve high accuracy in image recognition.



Y. LeCun 1998, Gradient-based learning applied to document recognition

Introduction (cont.)

Why convolution for recognition ?



- Translation invariance
- Fewer parameters
- Stride can be > 1 (accuracy and computational demand tradeoff)
- Dependencies are local

Introduction (cont.)

Due to general processors are not efficient for CNN implementation and can hardly to meet the performance requirement. Thus, FPGA and GPU have been proposed to improve performance of CNN design.

FPGA based accelerators have advantages of good performance, high energy efficiency, fast development round, and capability of reconfiguration.

Introduction (cont.)

If an accelerator structure is not carefully designed, computation throughput cannot match the memory bandwidth, that means performance is degraded due to under-utilization of either logic resource or memory bandwidth.

It is important to find the optimal solution, especially when limitations on computation resource and memory bandwidth are considered.

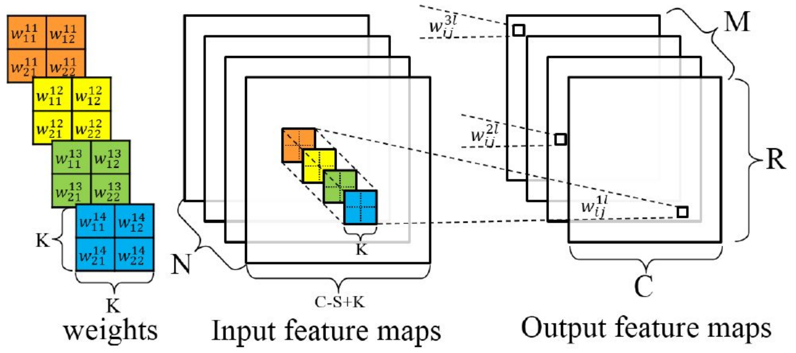
Main Works

- Quantitatively analyze computing throughput and require memory bandwidth of any potential solution of a CNN design on a FPGA platform.
- Identify all possible solutions in the design space using a roofline model and find the optimal solution for each layer in design space.
- Propose CNN accelerator design with uniform loop unroll factors across different convolutional layers.

*This work ONLY aims on speeding up the feedforward computation, an integration with other optional layers, such as sub-sampling or max pooling layers, will be studied in future work.

Outline

Graph of Convolutional Layer


w_{11}^{11} w_{12}^{11} w_{21}^{11} w_{22}^{11}
w_{11}^{12} w_{12}^{12} w_{21}^{12} w_{22}^{12}
w_{11}^{13} w_{12}^{13} w_{21}^{13} w_{22}^{13}
w_{11}^{14} w_{12}^{14} w_{21}^{14} w_{22}^{14}
K
K
weights
N
K
C-S+K
Input feature maps
w_{ij}^{3l}
w_{ij}^{2l}
w_{ij}^{1l}
M
R
C
Output feature maps

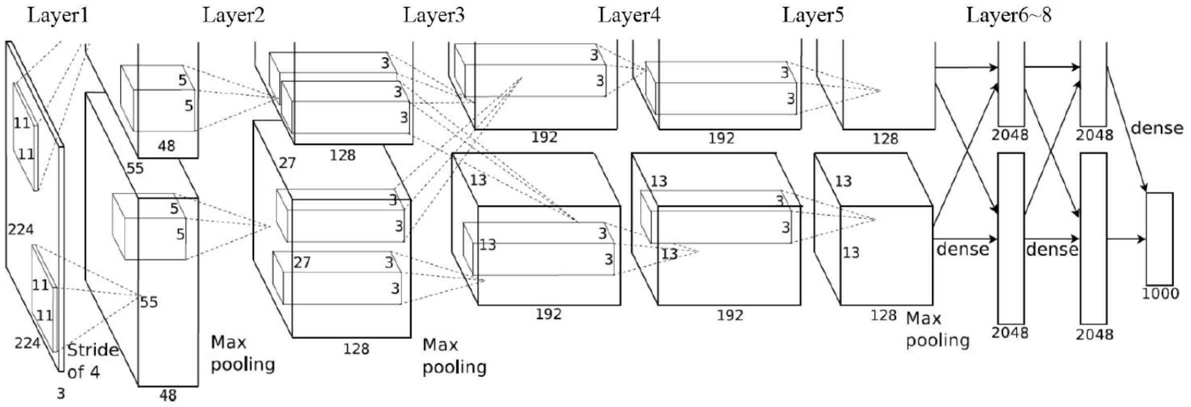
```
for(row=0; row<R; row++) {
 for(col=0; col<C; col++) {
  for(to=0; to<M; to++) {
   for(ti=0; ti<N; ti++) {
    for(i=0; i<K; i++) {
     for(j=0; j<K; j++) {
   L:    output_fm[to][row][col] +=
            weights[to][ti][i][j]*
            input_fm[ti][S*row+i][S*col+j];
} } } } } }
```

The convolutional layers receives N feature maps as input.

Each input feature map is convolved by a shifting window with a K x K kernel to generate one pixel in one output feature map.

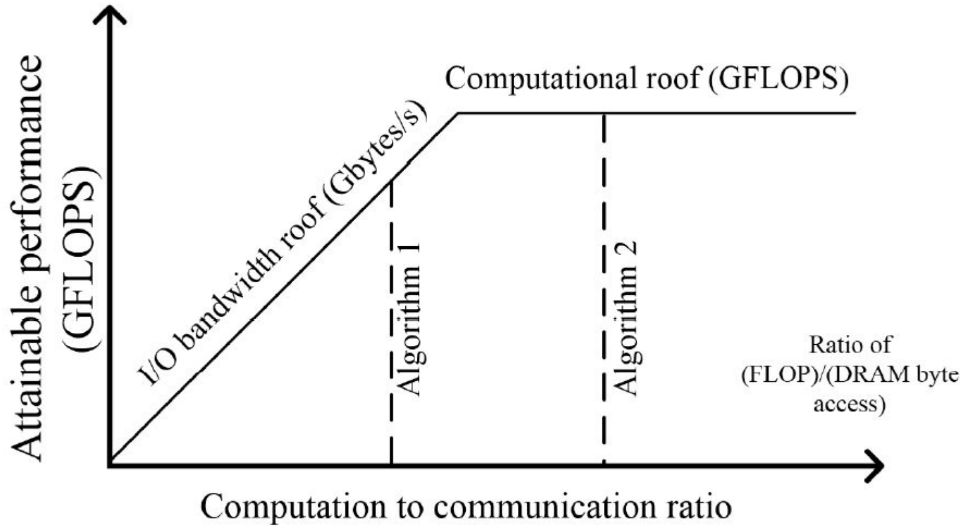
A total of M output feature maps will form set of input feature maps for the next convolutional layer.

Real-life CNN



Layer	1	2	3	4	5
input_fm (N)	3	48	256	192	192
output_fm (M)	48	128	192	192	128
fm row (R)	55	27	13	13	13
fm col. (C)	55	27	13	13	13
kernel (K)	11	5	3	3	3
stride (S)	4	1	1	1	1
set #	2	2	2	2	2

Roofline Model

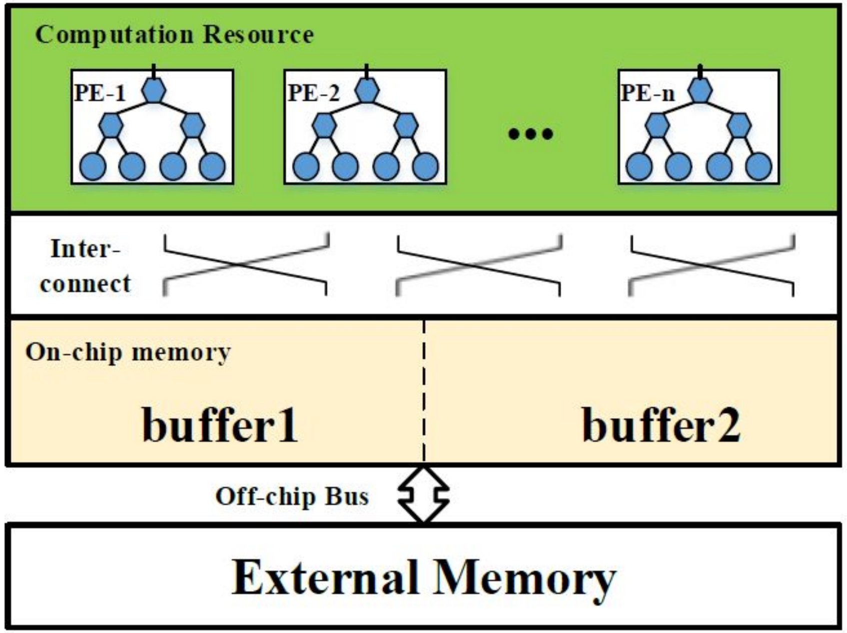

Attainable performance (GFLOPS)
Computational roof (GFLOPS)
I/O bandwidth roof (Gbytes/s)
Algorithm 1
Algorithm 2
Ratio of (FLOP)/(DRAM byte access)
Computation to communication ratio

$$Attainable\ Perf. = min \begin{cases} Computational\ Roof \\ CTC\ Ratio \times BW \end{cases}$$

- CTC ratio or Operations per DRAM traffic
- Computation roof is the peak performance provided by all available computation resources on the platform

Outline

Overview of Accelerator Design



- PE is the basic computation unit for convolution
- All data for processing are stored on the external memory (DRAM)
- Data are cached in on-chip buffers before being fed to PE
- Buffers are used to cover computation time with data transfer time
- On-chip interconnect is dedicated for data communication between PEs and on-chip buffers

Design Overview

- Loop tiling is needed to fit a small portion of data on chip, an improper tilling may degrade the efficiency of data reuse.
- Processing elements and buffer banks organizing should be carefully considered to process on-chip data efficiently.
- Throughput of processing elements should match with the FPGA's off-chip interface bandwidth (the platform itself has PCI-Express 2.0 x8 ~4GB/s).

Computation Optimization

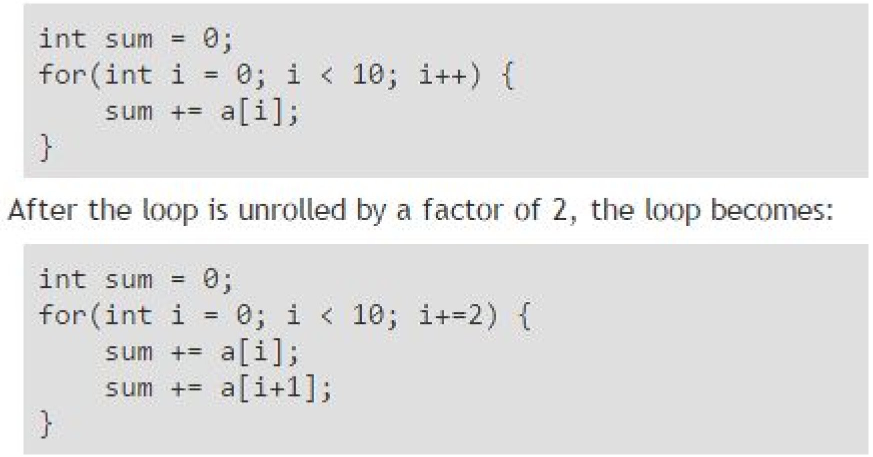
The objective of optimization is to fully utilize all of computation resources on FPGA hardware platform by techniques as follows,

- Loop unrolling
- Loop pipelining
- Tile size selection

Loop Unrolling

Loop unrolling is a technique to exploit parallelism between loop iterations and reduce iteration overhead by creating multiple copies of the loop body and adjusting loop iteration counter.

A normal rolled loop:

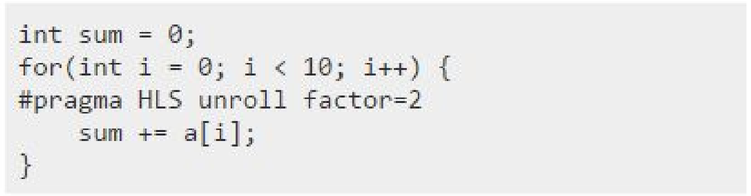


```
int sum = 0;
for(int i = 0; i < 10; i++) {
    sum += a[i];
}
```

After the loop is unrolled by a factor of 2, the loop becomes:

```
int sum = 0;
for(int i = 0; i < 10; i+=2) {
    sum += a[i];
    sum += a[i+1];
}
```

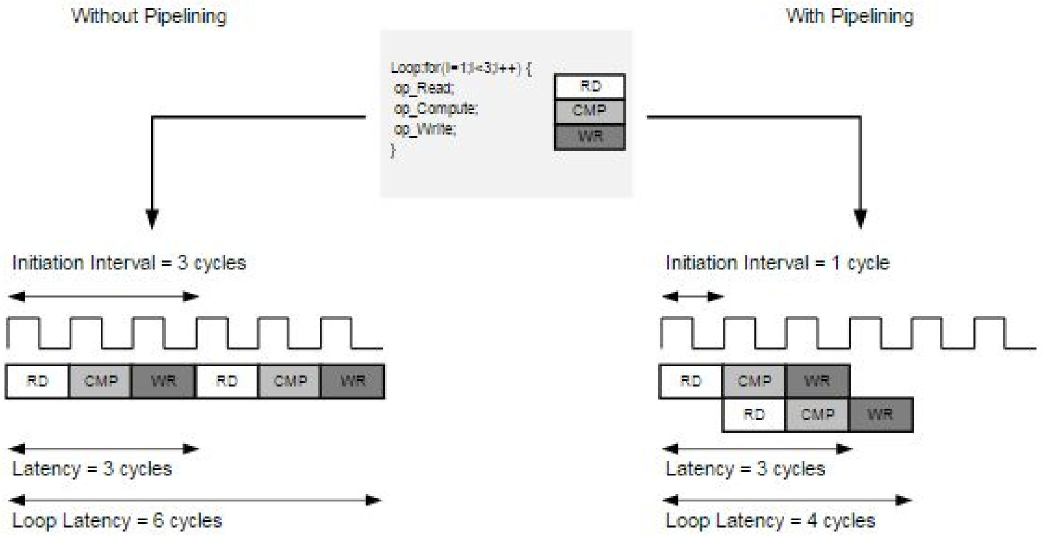
Unrolled by Vivado HLS with unroll factor = 2:



```
int sum = 0;
for(int i = 0; i < 10; i++) {
#pragma HLS unroll factor=2
    sum += a[i];
}
```

Loop Pipelining

Loop pipelining is a technique in high-level synthesis to improve system throughput by overlapping the execution of operations from different loop iteration.



Tile Size Selection

On-chip memory is limited. To help ensure data reuse, loop tiling is needed to make a proper portion of data chunks, design variants with different loop tile size will also have significantly different performance.

$$\begin{cases} 0 < Tm \times Tn \leq (\# \ of \ PEs) \\ 0 < Tm \leq M \\ 0 < Tn \leq N \\ 0 < Tr \leq R \\ 0 < Tc \leq C \end{cases}$$

The space of all legal tile sizes

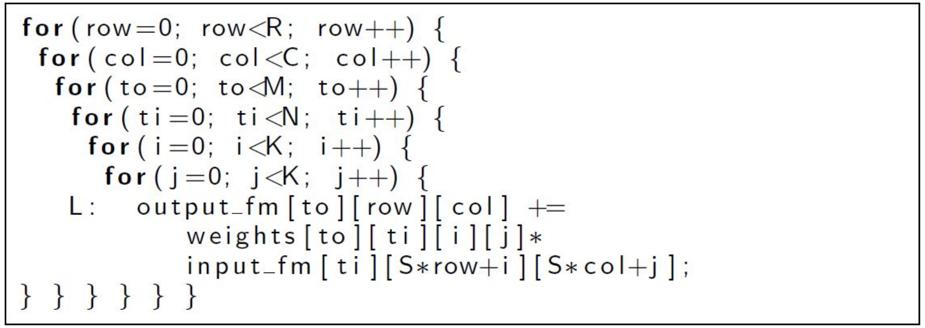
Computational Performance

Computational performance (computational roof) can be calculated by the below equation.

$$
\begin{aligned}
& computational\ roof \\
= & \frac{total\ number\ of\ operations}{number\ of\ execution\ cycles} \\
= & \frac{2 \times R \times C \times M \times N \times K \times K}{\lceil \frac{M}{T_m} \rceil \times \lceil \frac{N}{T_n} \rceil \times \frac{R}{T_r} \times \frac{C}{T_c} \times (T_r \times T_c \times K \times K + P)} \\
\approx & \frac{2 \times R \times C \times M \times N \times K \times K}{\lceil \frac{M}{T_m} \rceil \times \lceil \frac{N}{T_n} \rceil \times R \times C \times K \times K}
\end{aligned}
$$

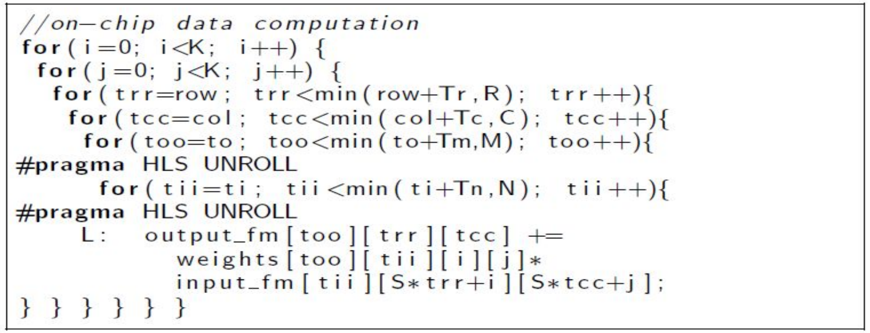
where $P = pipeline\ depth - 1$.

Proposed Accelerator Structure



```
for(row=0; row<R; row++) {
 for(col=0; col<C; col++) {
  for(to=0; to<M; to++) {
   for(ti=0; ti<N; ti++) {
    for(i=0; i<K; i++) {
     for(j=0; j<K; j++) {
   L:   output_fm[to][row][col] +=
           weights[to][ti][i][j]*
           input_fm[ti][S*row+i][S*col+j];
} } } } } }
```

Original



```
//on-chip data computation
for(i=0; i<K; i++) {
 for(j=0; j<K; j++) {
  for(trr=row; trr<min(row+Tr,R); trr++){
   for(tcc=col; tcc<min(col+Tc,C); tcc++){
    for(too=to; too<min(to+Tm,M); too++){
#pragma HLS UNROLL
     for(tii=ti; tii<min(ti+Tn,N); tii++){
#pragma HLS UNROLL
    L:  output_fm[too][trr][tcc] +=
          weights[too][tii][i][j]*
          input_fm[tii][S*trr+i][S*tcc+j];
} } } } } }
```

After unrolled, pipelined and tiled

Note that loop iterators i and j are not tiled because of the relatively small size of convolution window size K in CNN.

*without the optional factor = N , loop will be fully unrolled and pipelined.

Limitations of Loop Unrolling and Pipelining

Both loop pipelining and loop unrolling exploit the parallelism between loop iterations, but parallelism between loop iterations is limited by two main factors:

- Data dependencies between loop iterations
- Number of available hardware resources

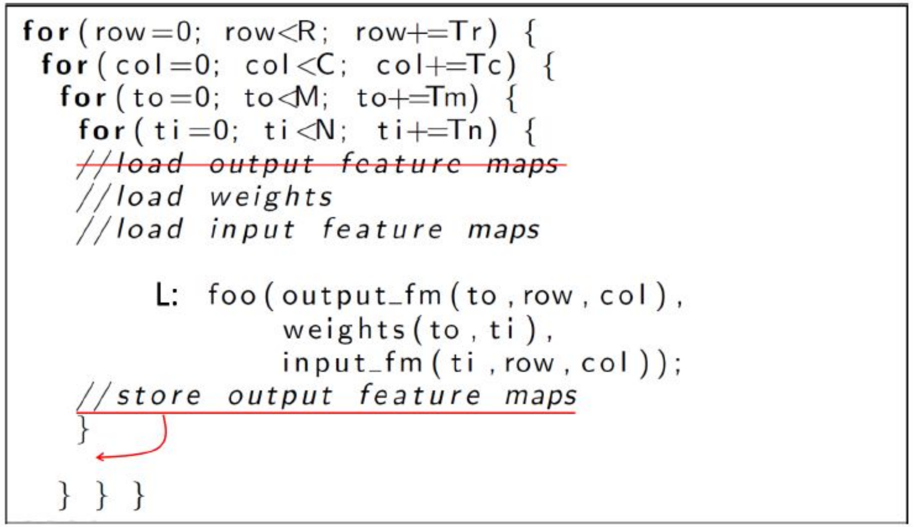
Memory Access Optimization

Design variants with higher computation roof doesn’t necessarily achieve the higher performance under memory bandwidth constraints.

So, it is a must to reduce communication volume with higher data reuse and data locality by loop transformation techniques, this work uses polyhedral-based optimization to identify all of legal loop transformations.

*The polyhedral model is a geometry as well as a linear algebra, its detail is deep in linear algebra & geometry that the paper doesn’t say about it much.

Memory Transfer and Local Promotion of CNN Layer



```
for(row=0; row<R; row+=Tr) {
 for(col=0; col<C; col+=Tc) {
  for(to=0; to<M; to+=Tm) {
   for(ti=0; ti<N; ti+=Tn) {
   ~~//load output feature maps~~
   //load weights
   //load input feature maps

       L: foo(output_fm(to,row,col),
              weights(to,ti),
              input_fm(ti,row,col));
   //store output feature maps
   }

  } } }
```

1. Input/output feature maps and weights are loaded
2. Computation engines start to compute
3. Output feature maps are stored back to main memory

If innermost loop is irrelevant to any array, local memory promotion can be used to reduce the redundant operation between loop iterations.

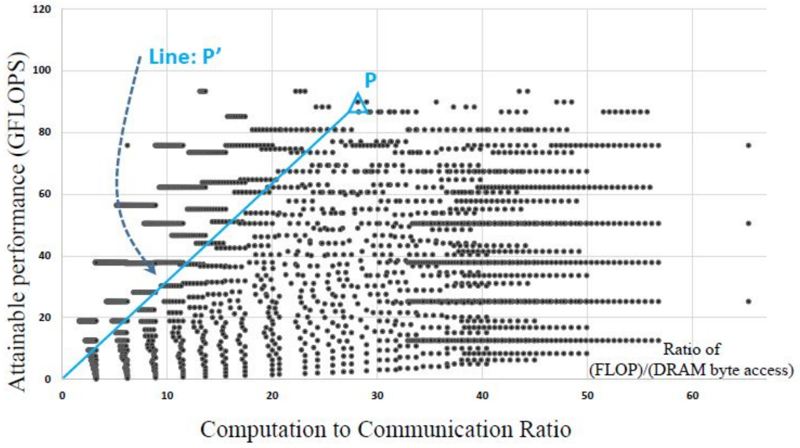
$$2 \times \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c}$$

Before

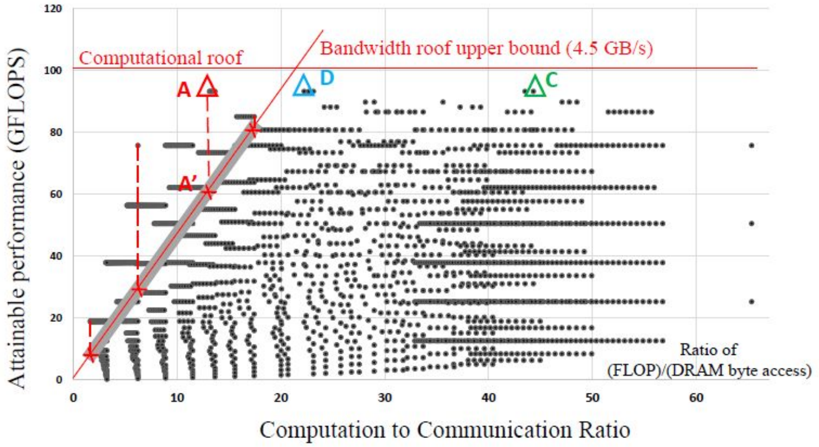
$$\frac{M}{T_m} \times \frac{R}{T_r} \times \frac{C}{T_c}$$

After

Design Space Exploration



(a) Design space of all possible designs



(b) Design space of platform-supported designs

P denotes the minimal bandwidth requirement

CTC Ratio

$$\begin{aligned}
& \mathit{Computation\ to\ Communication\ Ratio} \\
= \; & \frac{\mathit{total\ number\ of\ operations}}{\mathit{total\ amount\ of\ external\ data\ access}} \\
= \; & \frac{2 \times R \times C \times M \times N \times K \times K}{\alpha_{in} \times B_{in} + \alpha_{wght} \times B_{wght} + \alpha_{out} \times B_{out}}
\end{aligned}$$

where

$$\begin{aligned}
& B_{in} = T_n(ST_r + K - S)(ST_c + K - S) \\
& B_{wght} = T_m T_n K^2 \\
& B_{out} = T_m T_r T_c \\
& 0 < B_{in} + B_{wght} + B_{out} \leq BRAM_{capacity} \\
& \alpha_{in} = \alpha_{wght} = \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c}
\end{aligned}$$

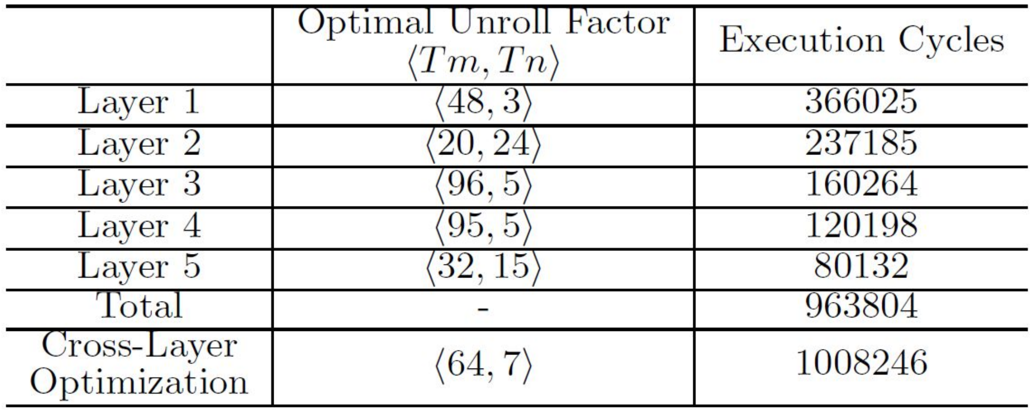
Without *output_fm*'s data reuse,

$$\alpha_{out} = 2 \times \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c}$$

With *output_fm*'s data reuse,

$$\alpha_{out} = \frac{M}{T_m} \times \frac{R}{T_r} \times \frac{C}{T_c}$$

Layer Optimal Solution and Cross-Layer Optimization



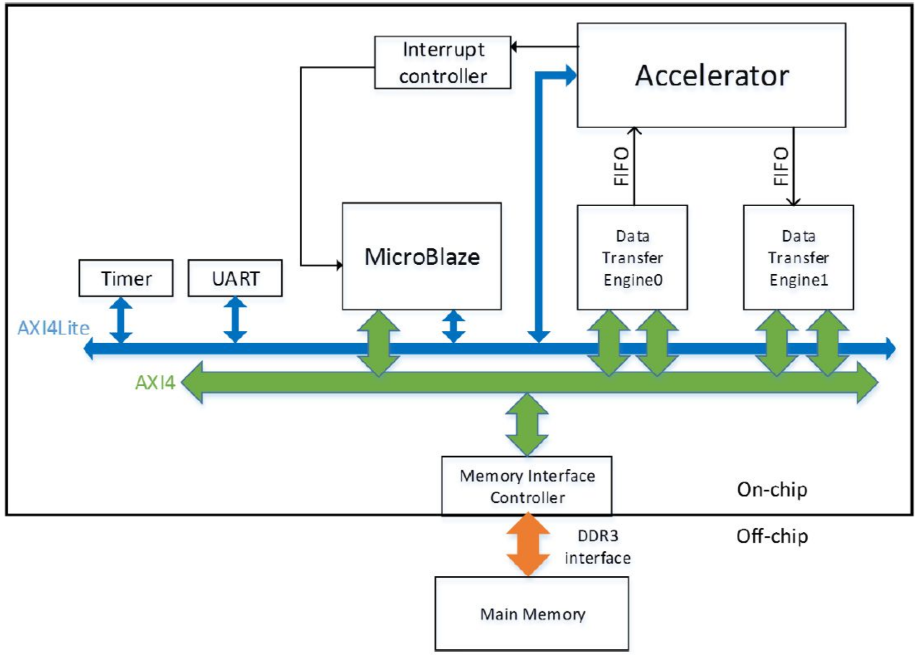
	Optimal Unroll Factor $\langle Tm, Tn \rangle$	Execution Cycles
Layer 1	$\langle 48, 3 \rangle$	366025
Layer 2	$\langle 20, 24 \rangle$	237185
Layer 3	$\langle 96, 5 \rangle$	160264
Layer 4	$\langle 95, 5 \rangle$	120198
Layer 5	$\langle 32, 15 \rangle$	80132
Total	-	963804
Cross-Layer Optimization	$\langle 64, 7 \rangle$	1008246

Designing an accelerator with different unroll factor across different convolutional layers requires re-configure FPGA for interconnects and computation engines.

An alternative approach is to find the cross-layer optimal unroll factor, the unified unroll factor <64, 7> has only 5% degradation compared to total execution cycles of each optimized layers.

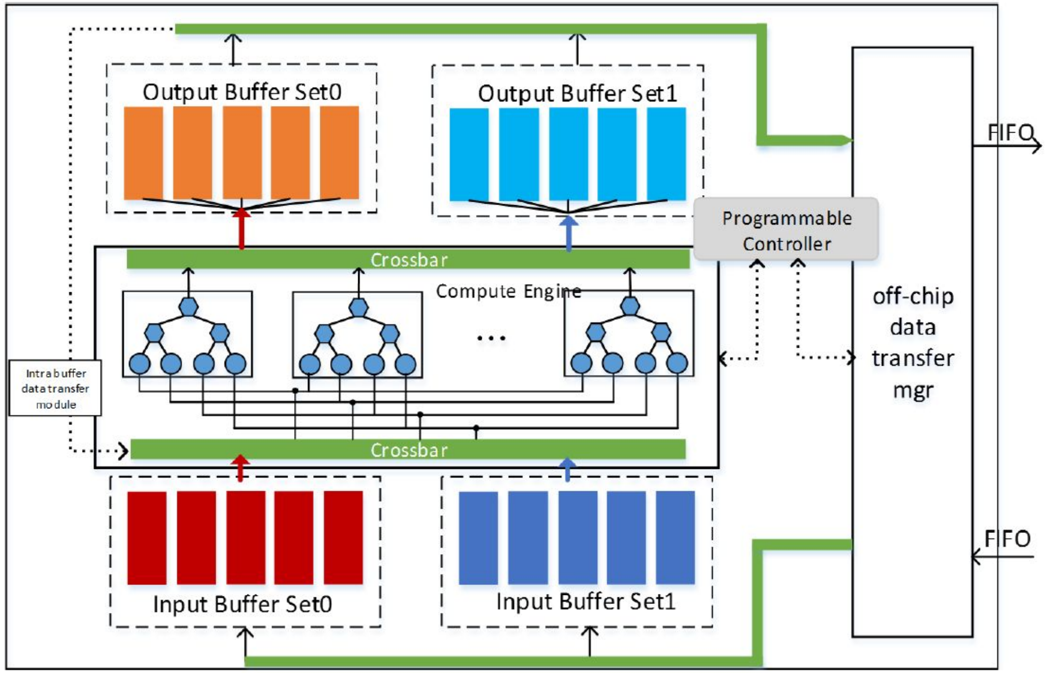
Outline

Implementation Overview

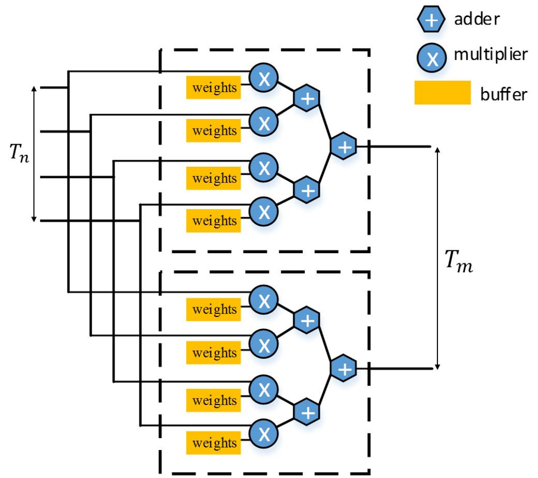


- The whole system fits in single FPGA chip and uses data from DDR3 for external storage
- MicroBlaze is used to assist CNN startup communication and time measurement
- AXI4Lite bus is for command transfer
- AXI4 bus is for data transfers
- Accelerator receives commands from MicroBlaze through AXI4Lite bus
- Data transfer engine transfers data between accelerator and AXI4 bus through FIFO interfaces, it uses 2 IP for each engine

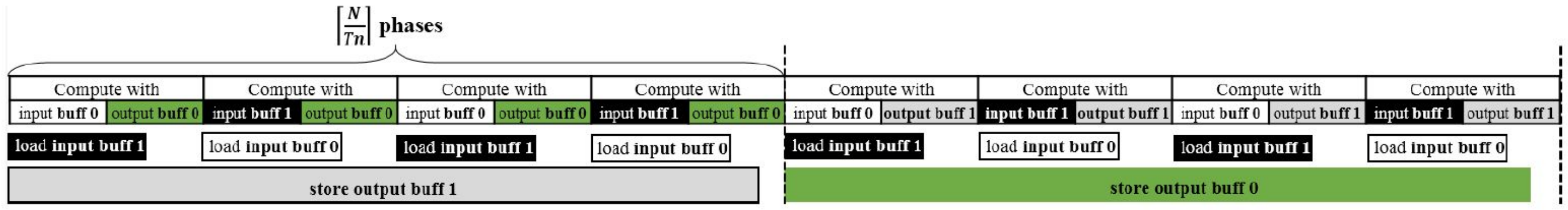
Block Diagram of Proposed Accelerator



- Two-level unrolled loops are implemented as concurrently executing computation engines
  - 64 poly structures are duplicated for unrolling loop Tm



Memory Sub-System

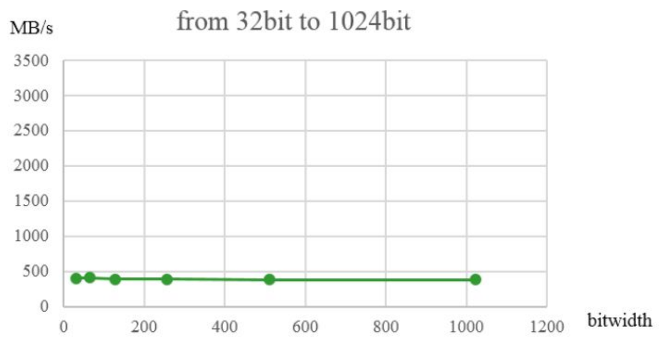

$\left\lceil \frac{N}{Tn} \right\rceil$ phases
Compute with input buff 0 output buff 0
Compute with input buff 1 output buff 0
Compute with input buff 0 output buff 0
Compute with input buff 1 output buff 0
Compute with input buff 0 output buff 1
Compute with input buff 1 output buff 1
Compute with input buff 0 output buff 1
Compute with input buff 1 output buff 1
load input buff 1
load input buff 0
load input buff 1
load input buff 0
load input buff 1
load input buff 0
load input buff 1
load input buff 0
store output buff 1
store output buff 0

Double buffer set are used to realize ping-pong operations, it has 2 independent data channels.

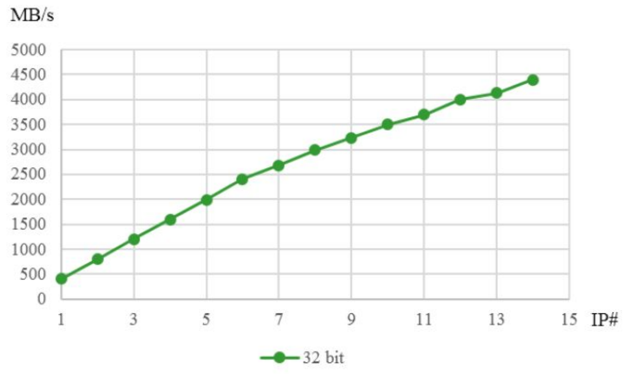
1) First phase, computation engine is processing with input buffer set 0 while copying the next phase data to input buffer set 1.
2) The next phase will do the opposite operation.
3) When [N/Tn] phase is done, the result of output feature maps are stored to DRAM.

This is the ping-pong operation, it holds results in output buffer set 0/1 until the end of each [N/Tn] phase before buffer sets generate new results, to increase bandwidth utilization and data reuse.

IP-DRAM Bandwidth (Vivado 2013.4)



Single IP’s bandwidth-bitwidth relation



Bandwidth-IP numbers relation

Increase of IP’s bitwidth has no effect on bandwidth (400 MB/s under 100 MHz).

Bandwidth almost linearly increases by adding IP interfaces. In the design, a minimal bandwidth of 1.55 GB/s is required. Therefore, 4 IP interfaces are sufficient for this design.

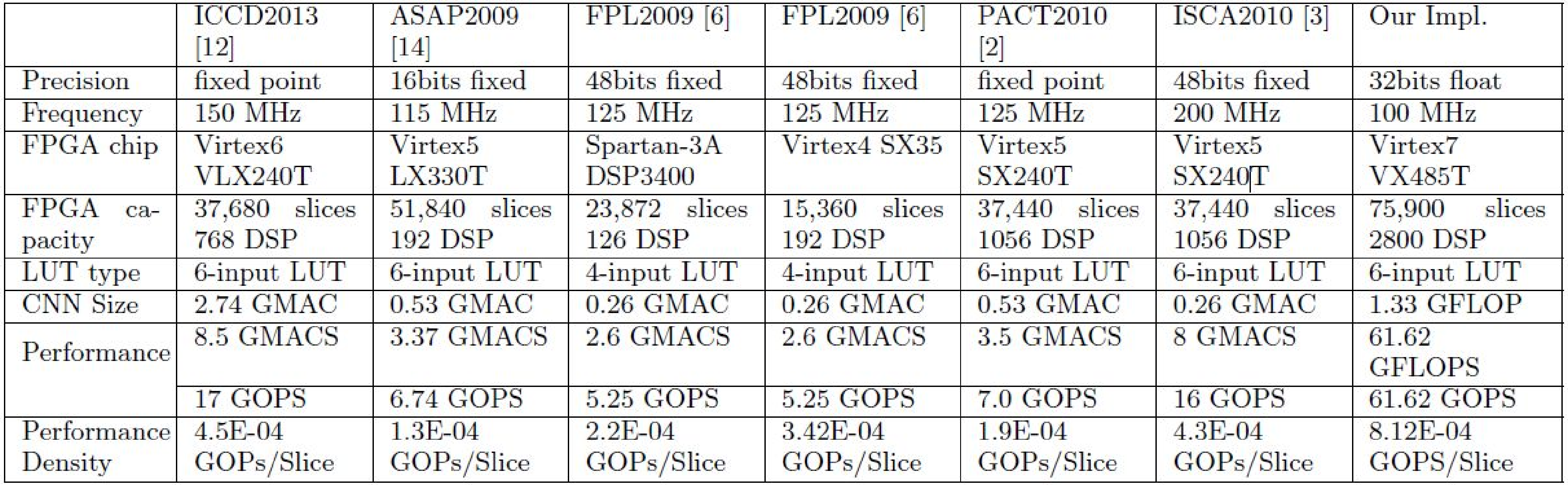
Outline

Experiment Setup

- CPU E5-2430 (@2.20GHz) with 15MB cache
- Xilinx VC707 board with FPGA chip Virtex7 485t (@100MHz)

1. Implemented with Vivado HLS (v2013.4). It enables implementing the accelerator with C and exporting the RTL as a Vivado's IP core.
2. C code of CNN is parallelized by adding HLS-defined pragma.
3. Validated by timing analysis tool.
4. Pre-synthesis simulation with C simulation and C/RTL co-simulation.
5. Exported RTL is synthesized and implemented in Vivado (v2013.4).

Comparison to Previous Implementation

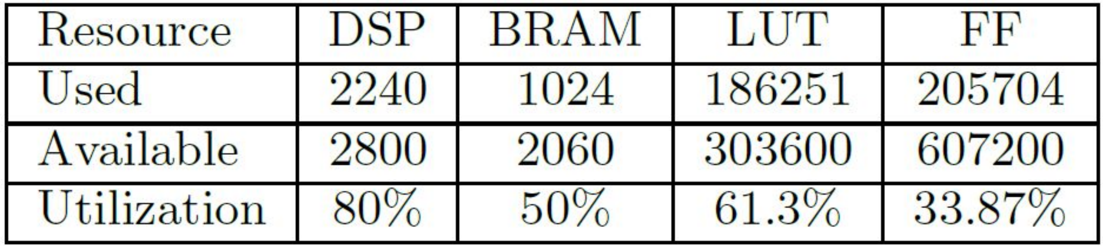


	ICCD2013 [12]	ASAP2009 [14]	FPL2009 [6]	FPL2009 [6]	PACT2010 [2]	ISCA2010 [3]	Our Impl.
Precision	fixed point	16bits fixed	48bits fixed	48bits fixed	fixed point	48bits fixed	32bits float
Frequency	150 MHz	115 MHz	125 MHz	125 MHz	125 MHz	200 MHz	100 MHz
FPGA chip	Virtex6 VLX240T	Virtex5 LX330T	Spartan-3A DSP3400	Virtex4 SX35	Virtex5 SX240T	Virtex5 SX240T	Virtex7 VX485T
FPGA capacity	37,680 slices 768 DSP	51,840 slices 192 DSP	23,872 slices 126 DSP	15,360 slices 192 DSP	37,440 slices 1056 DSP	37,440 slices 1056 DSP	75,900 slices 2800 DSP
LUT type	6-input LUT	6-input LUT	4-input LUT	4-input LUT	6-input LUT	6-input LUT	6-input LUT
CNN Size	2.74 GMAC	0.53 GMAC	0.26 GMAC	0.26 GMAC	0.53 GMAC	0.26 GMAC	1.33 GFLOP
Performance	8.5 GMACS	3.37 GMACS	2.6 GMACS	2.6 GMACS	3.5 GMACS	8 GMACS	61.62 GFLOPS
	17 GOPS	6.74 GOPS	5.25 GOPS	5.25 GOPS	7.0 GOPS	16 GOPS	61.62 GOPS
Performance Density	4.5E-04 GOPs/Slice	1.3E-04 GOPs/Slice	2.2E-04 GOPs/Slice	3.42E-04 GOPs/Slice	1.9E-04 GOPs/Slice	4.3E-04 GOPs/Slice	8.12E-04 GOPS/Slice

* GMACS (giga multiplication and accumulation per second)

** GOPS (giga operations per second)

Resource Utilization



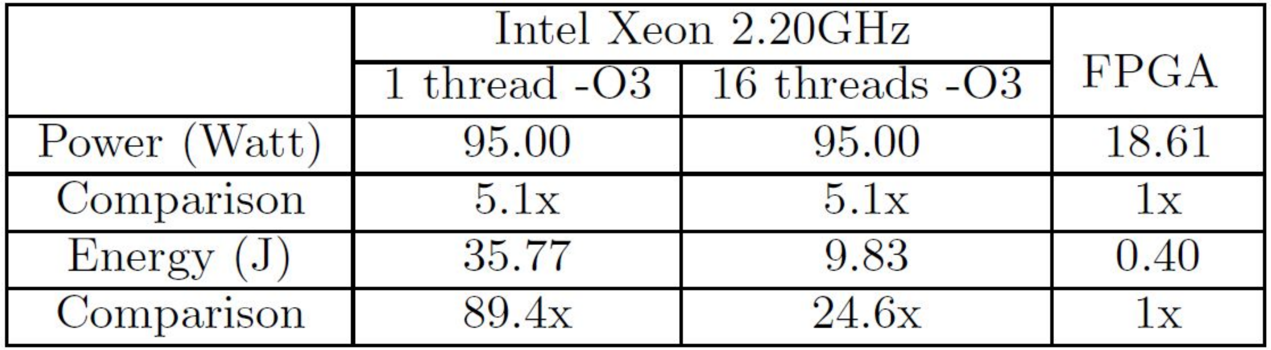
Resource	DSP	BRAM	LUT	FF
Used	2240	1024	186251	205704
Available	2800	2060	303600	607200
Utilization	80%	50%	61.3%	33.87%

Placement and routing is completed by Vivado tool set

Performance Comparison to CPU

float	CPU 2.20GHz (ms)		FPGA	
32 bit	1thd -O3	16thd -O3	(ms)	GFLOPS
layer 1	98.18	19.36	7.67	27.50
layer 2	94.66	27.00	5.35	83.79
layer 3	77.38	24.30	3.79	78.81
layer 4	65.58	18.64	2.88	77.94
layer 5	40.70	14.18	1.93	77.61
Total	376.50	103.48	21.61	-
Overall GFLOPS	**3.54**	**12.87**	**61.62**	
Speedup	**1.00x**	**3.64x**	**17.42x**	

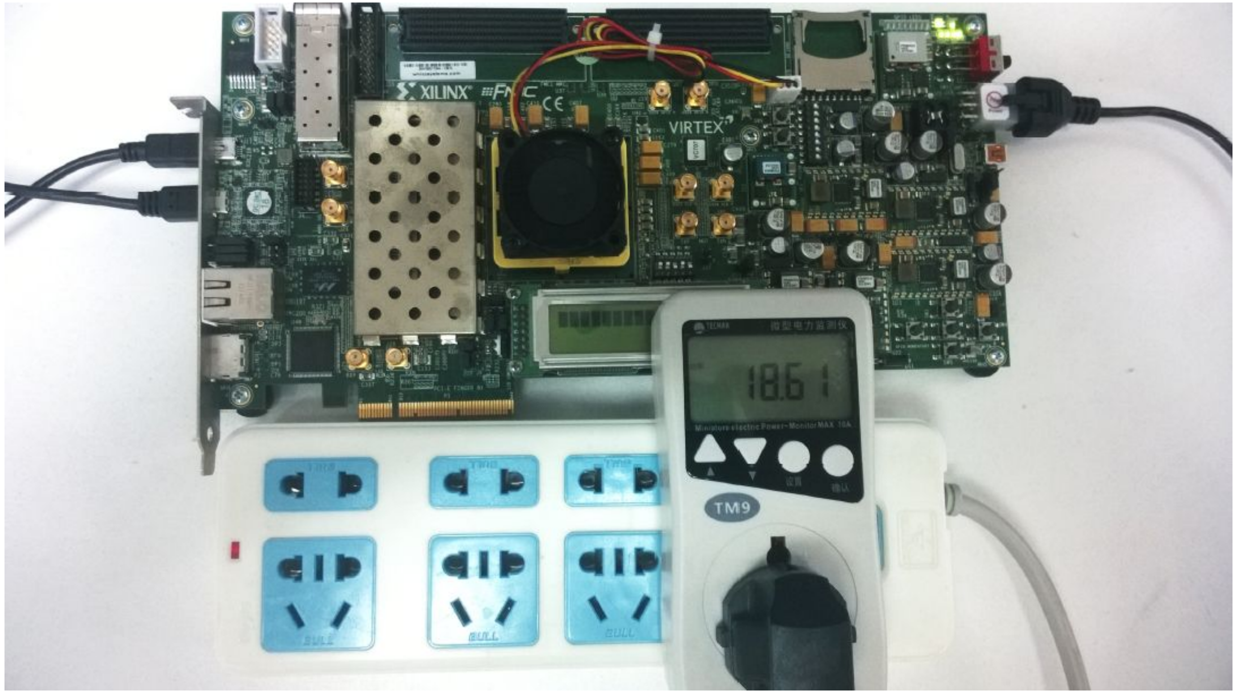
Power Consumption and Energy



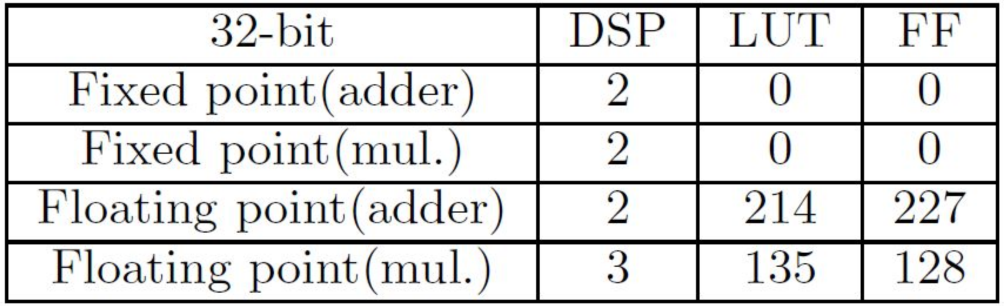
	Intel Xeon 2.20GHz		FPGA
	1 thread -O3	16 threads -O3	
Power (Watt)	95.00	95.00	18.61
Comparison	5.1x	5.1x	1x
Energy (J)	35.77	9.83	0.40
Comparison	89.4x	24.6x	1x

Measure CPU and FPGA runtime power performance

Power Measurement



Resource Occupation



32-bit	DSP	LUT	FF
Fixed point(adder)	2	0	0
Fixed point(mul.)	2	0	0
Floating point(adder)	2	214	227
Floating point(mul.)	3	135	128

Outline

Related Works

There are two different methods work on FPGA-based CNN accelerator.

First, CNN applications are focused on optimizing computation engines,

- Building CNN application mainly by software implementation while using hardware systolic architecture to do the filtering convolution job
- Implementing CNN application mainly uses the parallelism within feature maps and convolution kernel

Second, considering CNN's communication issue, and chooses to maximize data reuse and reduce memory bandwidth requirement to the minimum.

Conclusion

This work proposes Roofline-model-based method for CNN’s FPGA acceleration.

- Optimize both CNN’s computation and memory access
- Model all possible designs in roofline model to find the best design for each layer
- Find the best cross-layer design by enumeration
- Realize an implementation on Xilinx VC707 board

Opinion

- This work is such a platform-specific; it needs to re-design for every single change of the platform.
  - Enumeration of an optimal solutions in a new design space
  - Tile size selection (depends on on-chip memory capacity)
  - Memory subsystem (timing of data transfer and compute phase)
  - Off-chip data communication (# of IP interfaces, PCI-Express lanes)
- Better to have a comparison to GPU
- Need to pay for some of development tools (synthesizer, etc.)

Inspiration

This work inspired me to look deeper into applications of deep CNN for:

- Facial/object recognition
- Image classification

These are not a new sort of field, but I think they are still interesting in several ways. Many institutions have been working to increase accuracy of classifier and other improvements.

Also, optimizing accelerator design is remarkable.

Additional References

Xilinx Vivado high-level synthesis tool: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

Loop unrolling & pipelining: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html

Loop parallelism & tranformation: http://parlab.eecs.berkeley.edu/wiki/_media/patterns/loop_parallelism.pdf

FPGA platform: http://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html#overview