目录

[1. 简介 1](#_Toc454220388)

[2. Conv Loop Structure 1](#_Toc454220389)

[2.1 Tile计算和通信的同步 2](#_Toc454220390)

[2.2 Tile 内部的计算流水 3](#_Toc454220391)

[2.3资源和性能量化分析 4](#_Toc454220392)

[2.4 硬件实现 5](#_Toc454220393)

[2.5 Corner Case Analysis 6](#_Toc454220394)

[2.6 Performance Estimation 7](#_Toc454220395)

[3. Conclusion 9](#_Toc454220396)

[Reference 9](#_Toc454220397)

**FPGA Accelerator Design for CNN**

# 简介

考虑到Zedboard上的资源情况，目前选择了资源消耗少，计算密度高的，性能较好的实现方式。基本的思路就是对Conv进行tiling处理，Tile的数据集要能够被FPGA完整的缓存。每个tile又分为两部分，tile的核心部分由FPGA来开发数据级并行，tile核心的计算要能方便支持stride并适应不同的kernel Size，tile的内部的数据级并行方案将在下一节详述。Tile内部的核心之间流水实现。Tile之间主要是串行执行，可以考虑进一步使用Ping-Pong Buffer，实现Tile序列的load/compute/store流水。实现方案主要参照论文[4][5]，并在此基础上，借鉴[1]增加了分析模型，以协助确定设计参数和性能预估。

# Conv Loop Structure

for(to = 0; to < M; to += Tm){ //Output feature map

for(ti = 0; ti < N; ti += Tn){ // Input feature map

for(row = 0; row < R; row += Tr) { // # of rows

for(col = 0; col < C; col += Tc){ # of columns

// load output feature maps

// load weights

// load input feature maps

for(too=to; too < min(to+Tm, M); too++){ // Output feature tiling

for(tii = ti; tii < min(ti+Tn, N); tii++){ // input feature tiling

for(trr = row; trr < min(row+Tr, R); trr++){ // Row tiling

for(tcc = col; tcc < min(col+Tc, C); tcc++){ // Column tiling

for(i = 0; i < K; i++){

for(j = 0; j<K; j++){

output\_fm[too][trr][tcc] += weight[too][tii][i][j] \*

input\_fm[tii][S\*trr+i][S\*tcc+j];

}

} //compute kernel inner the tile

}

} **One computing iteration**

}

} **Tile**

}

}

}

} **Overall computing task**

Fig.1 Tiling Structure of Conv

Fig.1 给出了Tiling策略所对应的代码，Fig.2 给出了相应的设计框图。

每个Tile内部虽然有Tn个输入通道，但是由于资源限制，Tile的计算单元仅仅实现X个并行输入，类似的情况，可以每个Tile内部的并行实现了Y个输出，这样硬件一共需要X\*Y个MAC单元，流水情况下每个MAC单元一个周期可以完成一次计算。 通过控制控制输入输出并行的路数，可以调整整体计算的并行度。通过输入数据布局，可以将每个输入feature进一步分块，假如分为Z块，那么并行计算单元可以增加到X\*Y\*Z，同时input\_fm所需要的存储空间也略有增加，增加的量跟进一步分块带来的数据复制有关。

Weight [to=0:Tm-1][ti=0:X-1][k][k]

Potential partition

(t=Tm/Y)

input\_fm[ti=0:X-1][Tr][Tc]

Input\_fm

**X**

**Y**

**X**

Weight

input\_fm[ti=0][Tr][Tc]

input\_fm[ti=X-1][Tr][Tc]

out\_fm[to=Y-1][Tr][Tc]

W[t] [ti=0][k][k]

Adder

Tree

Adder

Tree

W[t][ti=X-1][k][k]

W[t][ti=0][k][k]

W[t][ti=X-1][k][k]

MUL

MUL

out\_fm[to=0][Tr][Tc]

MUL

MUL

Accum

Accum

Fig.2 Tiling Based Parallel Conv Computing Diagram

## 2.1 Tile计算和通信的同步

Conv大体共由5个部分组成，结构如图Fig.3 所示。软件配置Conv加速器，配置信息通过Conv Control控制Tile内部计算以及Tile与DDR的输入输出：

Conv Control

Load tile input

Tile computing

Store tile output

DDR

cfg

Fig.3 Conv Accelerator Architecture

Tile计算和通信的同步有三种可行的方式，一种是简单的以Tile为粒度的串行输入数据load, 数据计算computing，以及结果数据store。即每次从DDR读取一个tile的输入数据，数据读完之后，开始进行计算，计算完成之后，将结果写到DDR。优点是控制简单，可以增加一倍的输入输出缓存，设计成Ping-Pong Buffer，也能进一步挖掘load/computing/store的并行，并行程度受限于load或者computing中较长的处理步骤。当computing成为瓶颈时，可以通过增加MAC的数量来减少computing时间，以实现流水的平衡。

当并行粒度不大的时候，存在大量的分块，当流水线平衡的时候，能够取得最优的计算通信效率，达到接近理论最优的计算时间，但是Tile之间的流水很难平衡。当Tile的粒度较大，单个tile的所需时间长，总体的tile数量不多，流水的收益不大。

第二种则是一种细粒度的流水方式，在weight Buffer和input\_fm buffer中设置同步标志（主要是内部RAM的读写地址比较），一旦两者同步，则可以启动相应的PE进行计算，中间计算结果可以被立即缓存，达到一定的数量，则可以写回DDR。同步的粒度可以为单个数据到一块数据，同时缓存和DDR之间的通信也可以采用相同的粒度进行同步。

第三种则是完全分离计算和缓存，将输入数据重新组合为数据包的格式，每个PE处理一个数据包（包含一组weight数据，一组input\_fm数据以及控制信息），每个PE维持一个处理队列，DDR访存逻辑以数据包为单位传输数据，从而可以集中优化PE的实现，使得设计很容易被更改和升级。不过数据复用还需要仔细考虑。

在目前的系统上，资源相对较少，分块的粒度应该不会太大，因此计划采用第一种方案。

## 2.2 Tile 内部的计算流水

由于片上资源的局限（主要是DSP的限制），并行的输入输出并不等同于相应的Tiling的结果XTi, YTo,这样每个物理的input\_fm缓存以及weight缓存要存储input\_fm和Weight的不同部分。目前为了简化设计，每个weight Buffer存储了To/Y组weight向量，每个input\_fm buffer存储了Ti/X个不同channel的数据。硬件上同时有X\*Y组kernel计算，每个计算作为一个iteration，每个Tile内部有(Ti/X)\*(To/Y)个计算的iteration, 这些iteration之间是流水实现的。

## 2.3资源和性能量化分析

RAM资源:

input\_fm[tii][S\*trr+i][S\*tcc+j]:

（1）

weight[too][i][j]: （2）

output\_fm[too][trr][tcc]: （3）

DSP资源：

*X\*Y* 浮点MAC，logX个加法完成*X*个输入的树形加法。

MAC计算总量：

计算时间：

（5）

通信总量：

（6）

（7）

（8）

（9）

从公式（4）可以看出，计算时间主要和所能提供的并行计算的MAC数量(X\*Y)有关，能够比较容易的扩展，扩展的主要受限于输入输出所能并行的路数。

设计假设片上能够存储整个tile计算所需要的数据，所需要的片上RAM资源主要取决于分块的大小。如公式（1-3）所示，分块越大片上RAM需求越多。

在给定的通信接口下（假设通信带宽能得到充分利用），通信总量将会直接影响通信时间，从（9）可以看出，随着分块增大，通信数据量会成倍降低，此外stride对输入数据的通信也有很大的影响。

由于每个Tile的计算是重复的，比较计算时间和通信数据量，可以看出带宽的需求，当带宽需求大于平台所能提供的带宽时，应该增加分块，或者根据（9）选择性增加影响最大的维度的分块，从而在保证RAM允许的情况下，减少通信带宽。当带宽需求小于平台提供的带宽时，可以减小分块，从而节省片上缓存资源。

## 2.4 硬件实现

硬件设计如Fig.4 所示。

DPath0

DPath1

DPath2

DPath3

TileCtrl

in-ld-fifo

w-ld-fifo

out-ld-fifo

out-st-fifo

ConvCtrl

in-fm

weight

out-fm

avlon-rmst

avlon-rmst

avlon-wmst

avlon-rmst

Config

DDR

DDR

Fig. 4 Conv Diagram

硬件端口需要的配置参数：

**基本的Conv配置：M, N, R, C, K, S, Channel\_Offset, Row\_Offset**

**分块的配置：Tm, Tn, Tr, Tc, Tile\_Row\_Offset, Tile\_Col\_Offset**

**计算单元配置：X, Y**

硬件主要包括Tile的计算逻辑，输入缓存，树形累加器，输出缓存。

计算逻辑由基本的MAC组成，流水实现。

树形累加器取决于并行输入的数量X。

输入输出缓存难点在于地址控制逻辑，地址控制逻辑如下：

Weight缓存的地址：

for(input\_it=0; input\_it<Tn/X; input\_it++) //同一个input\_fm buffer不同channel输入数据索引

for(w\_it = 0; w\_ it < Tm/Y; W\_ it++){ **//**同一个weight buffer的不同filter系数矩阵索引

for(trr=0; trr<Tr; trr++){

for(tcc=0; tcc<Tc; tcc++){

**for(i=0; i<K; i++){**

**for(j=0; j<K; j++){**

**read\_addr = w\_it\*K\*K+i\*K+j;**

**}**

**}**

}}}}}

Input\_fm缓存的地址：

for(input\_it=0; input\_it<Tn/X; input\_it++)

for(trr=0; trr<Tr; trr++){

for(tcc=0; tcc<Tc; tcc++){

for(i=0; i<K; i++){

for(j=0; j<K; j++){

**read\_addr =input\_it\*Tr\*Tc+ (S\*trr+i)\*Tc+(S\*tcc+j)**

}}}}}

输出缓存的地址：

for(trr=0; trr<Tr; trr++){

for(tcc=0; tcc<Tc; tcc++){

**write\_addr = trr\*Tc+tcc;**

}

}

## 2.5 Corner Case Analysis

当Tile的输入通道Tm或者输出通道Tn不能被X以及Y整除的时候，空闲的通道数据无效或者被当作常数0处理即可。但是建议Tm,Tn要能正好分别被X和Y整除，避免计算资源的浪费。

当输入通道M或者输出通道N不能被Tile的Tm与Tn整除时，最终会导致输入Buffer和输出Buffer空闲，进行输出无效处理即可。

当输入图像R和C不能被分块参数Tr与Tc整除时， 需要同时修改输入通道input\_fm Buffer和weight Buffer的读地址逻辑即可，设计的其他部分保持不变。

Padding需要访问DDR读写片上RAM的位置进行处理，对于计算单元是透明的。

## 2.6 Performance Estimation

Typical convolution configuration:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | In Channel | Out Channel | Height | Width | Kernel Size | Stride |
| C1 | 8 | 16 | 512 | 512 | 3 | 1 |

Hardware configuration parameters:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Tiled In Channel | Tiled Out Channel | Tiled Height | Tiled Width | Physical input path | Physical output path | Bandwidth & design frequency |
| HW1 | 4 | 8 | 16 | 10 | 4 | 4 | Load:4GB/s  Store:4GB/s  Freq: 100MHz |
| HW2 | 8 | 8 | 16 | 18 | 4 | 8 |
| HW3 | 8 | 8 | 16 | 34 | 8 | 8 |
| HW4 | 8 | 16 | 16 | 34 | 8 | 16 |
| HW5 | 8 | 16 | 32 | 34 | 8 | 16 |

Opt：基于ping-pong buffer的计算和IO重叠，weight缓存优化。

没有优化的情况下，读写分立，优化之后，读写并发，因此所消耗的带宽增加了一倍。当输入输出的channel数量比较大的时候，可以通过调度tile的顺序，降低带宽需求。不过输入输出channel数量较少时，数据复用的机会比较少。

B4: 4GB/s

B8: 8GB/s

B16: 16GB/s

B32: 32GB/s

采用计算阵列流水实现，所需的带宽为8GB/s，理论性能为32ms

Estimated performance and resource requirement (stride = 1):

红色区域，计算瓶颈，绿色区域，通信瓶颈，黑色属于相对比较平衡的配置。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | C1-B4  (no opt) | C1-B8  (opt) | C1-B8  (no opt) | C1-B16  (opt) | C1-B16  (no opt) | C1-B32  (opt) |
| HW1 | Buffer (K-word) | **2** | **4** | **2** | **4** | **2** | **4** |
| Runtime (ms) | **425** | **121** | **260** | **95** | **178** | **95.48** |
| # of FP Multiplier | **16** | **16** | **16** | **16** | **16** | **16** |
| IO/Computing | **1.27** | **1.27** | **0.634** | **0.634** | **0.31** | **0.32** |
| HW2 | Buffer (K-word) | **5** | **10** | **5** | **10** | **5** | **10** |
| Runtime (ms) | **225** | **54.57** | **136.39** | **47.74** | **92** | **47.74** |
| # of FP Multiplier | **32** | **32** | **32** | **32** | **32** | **32** |
| IO/Computing | **1.14** | **1.14** | **0.57** | **0.57** | **0.29** | **0.29** |
| HW3 | Buffer (K-word) | **9** | **18** | **9** | **18** | **9** | **18** |
| Runtime (ms) | **209** | **51.54** | **128.44** | **47.74** | **88** | **47.74** |
| # of FP Multiplier | **64** | **64** | **64** | **64** | **64** | **64** |
| IO/Computing | **1.07** | **1.08** | **0.54** | **0.54** | **0.27** | **0.26** |
| HW4 | Buffer (K-word) | **13** | **26** | 13 | 26 | **13** | **26** |
| Runtime (ms) | **159** | **51.54** | 91.69 | 25.77 | **57.78** | **23.87** |
| # of FP Multiplier | **64** | **64** | 64 | 64 | **64** | **64** |
| IO/Computing | **2.15** | **2.15** | 1.07 | 1.07 | **0.503** | **0.539** |
| HW5 | Buffer (K-word) | **26** | **52** | 26 | 52 | **26** | **52** |
| Runtime (ms) | **145** | **47.36** | 84.25 | 23.68 | **53.88** | **23.50** |
| # of FP Multiplier | **128** | **128** | 128 | 128 | **128** | **128** |
| IO/Computing | **2.01** | **2.01** | 1.007 | 1.007 | **0.503** | **0.503** |

# Conclusion

目前的设计方案比较灵活，可以容忍不同的stride和kernel size配置，支持Tiling优化，针对不同的规模的Conv，寻找最优的设计方案，还需要做更完善的分析模型和设计空间探索。对于每一组Conv的配置，分别有对应的最优的实现方案，当多个不同的Conv共享同一个硬件实现时，优化的硬件设计也需要更长时间的设计探索。

# Reference

[1] Chen Zhang, Jason Cong, etc., “Optimizing FPGA based accelerator design for deep conventional neural networks”, FPGA, 2015

[2] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. “Polyhedral-based data reuse optimization for configurable computing”, FPGA, 2013

[3] Jason Cong, and Bingjun Xiao, “Minimizing computing in Conventional Neural Networks”, Springer 2014

[4] Maurice Peemen, Arnaud A. A. Setio, etc., “Memory-centric Accelerator Design for Conventional Neural Networks”, ICCD, 2013

[5] Maurice Peemen, Arnaud A. A. Setio, etc., Optimal Iteration Scheduling for Intra- and Inter- Tile Reuse in Nested Loop Accelerators, Eindhovern University of Technology, ES Report, 2013

[6]<https://forums.xilinx.com/t5/Xcell-Daily-Blog/How-do-you-use-FPGAs-to-accelerate-machine-learning/ba-p/696411>

[7] Jiantao Qiu, Yu Wang, etc, “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network”, FPGA, 2016