# 总体架构



CNN IP Core的开发涉及三个平台。

Develop（PC）：编译Caffe model，生成CNN所需配置数据；

Host（ARM）：应用程序运行平台，兼容ARMv7/ARMv8指令集。支持Altera/Xilinx两家平台，不同平台使用不同的驱动库，统一Library API接口；

Target（FPGA）：CNN加速IP核。对外分别采用厂商标准IP核接口，Altera-Avalon-MM，Xilinx-AXI3/4接口。IP核实现内外两套低速配置、高速数传接口。低速配置接口在IP核内部做时钟域同步。每个功能层仅包含两类接口，自定义配置接口(configure)，数据传输接口(data)，时钟域和IP核时钟保持一致。配置接口与CNN Scheduler模块相连，数传接口与Data Scheduler模块相连。

# 模块设计

CU计算单元由Convolution、relu、pooling组成，Pipeline实现，总耗时基本等价于Convolution耗时。当前版本的理论性能为W\*H\*Nin\*Nout个时钟周期。其中，W，H，Nin分别为输入数据的宽、高、通道数；Nout为输出数据通道数。在带宽足够的情况下，测试性能可达99%以上。对于当前版本的Convolution，单就性能而言，可以满足一定的应用需求。但还是有两个比较大的设计不足：① Kernel size支持不够灵活。如果按照5\*5的实现，在跑3\*3的kernel size时，会有很大的资源浪费；② 未充分利用内部RAM资源，对带宽需求太高，导致性能瓶颈。下一版本着重解决这两个问题。

CNN Scheduler：根据Compiler生成的数据，自动配置CU，FC等网络层的连接。

Data Scheduler：数据调度，关乎整个IP核的计算效率。对外输出一路标准IP核接口总线，由仲裁模块实现；对内多路自定义数据接口，降低CNN各模块的访存复杂度。

时钟域设计。预计划分三个时钟域，config\_clk, data\_clk, IP\_clk。所有的CNN layer模块同步于同一时钟域IP\_clk。

# 模块接口

配置接口(configure)：写数据前，必须拉高cfg\_ena信号。cfg\_rdata只需对地址译码。

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Directrion | Data width | Description |
| cfg\_ena | input | 1 | Configure enable |
| cfg\_addr | input | UCAW(default=5) | Configure address（read/write） |
| cfg\_wdata | input | UCDW(default=32) | Configure write data |
| cfg\_rdata | output | UCDW(default=32) | Module status return |

数传接口(data)：读写接口分离。为提高数据写入效率，必须在所有数据写入FIFO之后，再拉高相应的控制信号。

表1 控制信号接口

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Directrion | Data width | Description |
| control\_base | input | UDAW(default=32) | Word aligned byte address where the master will begin transferring data |
| control\_length | input | UDAW(default=32) | Number of bytes to transfer.This number must be a multiple of the data  width in bytes (e.g. 32 bit data requires a multiple of 4) |
| control\_go | input | 1 | One clock cycle strobe that instructs the master to begin transferring.The base, and length values are registered on this clock cycle. |
| control\_done | output | 1 | Asserted and held when the master has  transferred the last word of data. This occurs when the last write transfer completes or the last pending read returns. You can start the master again on the next cycle after done is asserted |

表2 读数据信号接口

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Directrion | Data width | Description |
| user\_buffer\_data | Output | UDDW(default=128) | Contains the next valid buffered data when user\_data\_available is asserted. |
| user\_data\_available | output | 1 | When asserted the user buffer contains valid data that has been read by the read master. You must not assert ‘user\_read\_buffer’ when this signal is de-asserted as this will cause a buffer underflow condition and the read master will fail to complete the entire transfer. |
| user\_read\_buffer | input | 1 | Acts as a read acknowledge. When this signal is asserted the master component assumes your user logic has registered the data. On the next clock cycle the output of ‘user\_buffer\_data’ will be the next read value (if it has already been buffered). |

表3 写数据信号接口

|  |  |  |  |
| --- | --- | --- | --- |
| Signal name | Directrion | Data width | Description |
| user\_buffer\_data | Input | UDDW(default=128) | Valid data word that your logic writes into the user buffer. Use ‘user\_write\_buffer’ to qualify it as valid data when ‘user\_buffer\_full’ is de-asserted |
| user\_buffer\_full | output | 1 | When asserted the user buffer is full and you must not write any more data. Asserting ‘user\_write\_buffer’ while this signal is asserted may lead data being lost and the write master failing to complete the entire transfer. |
| user\_write\_buffer | Input | 1 | Acts as a write qualifier. Assert this signal to write valid data into the user buffer. You must not assert this signal if ‘user\_buffer\_full’ is asserted otherwise data overflow will occur. |

# 3D卷积

3D卷积实现表达式：



是输入通道数据，是卷积核系数，是前一输入通道的卷积输出。



不考虑存储带宽因素，流水线工作之后，可以在单周期实现一个输入通道的单点卷积运算。考虑WxH的单通道输入，padding为0，stride为1，则其理论耗时为（W-2）\*（H-2）+ N（pipeline setup）个时钟周期。此实现方式的不足，前面已有提及。