

# 实验报告

## 选题

Filter2D

## 摘要

本实验总的设计思路是让每个AIE复用Xilinx官方的filter2D API，使得每个AIE能够对固定大小的图片进行卷积操作。然后利用PL端对输入的任意大小的图片进行分块操作，分块后的矩阵大小与每个AIE能处理的图片大小相同。之后循环调用一组 AIE 对这些分块数据进行处理，最后再利用PL端将计算好的分块数据进行拼接操作即可。这里输入图片的大小需不小于每个AIE能处理的图片大小，另外重复进行上述过程即可处理多张图片。

但由于host端代码始终运行有误等原因，本实验仅进行了仿真。即需手动调用分块操作代码对图片进行分块，得到若干输入文件，然后再通过仿真调用AIE得到若干输出文件，最后运行拼接操作代码得到卷积后的结果。

## 系统设计

### AIE代码

本设计的AIE代码直接利用了已有API，细节不做过多赘述。  
唯一的区别在于，将代码中AIE处理的矩阵大小修改为  $64 * 32$

### 对数据进行分块以及拼接

因为未找到可适配x86平台的矩阵 tile & sticker API，本实验对 tile & sticker 代码进行了复现

需注意这里的 padding 操作采取了filter2D API中的做法，即：

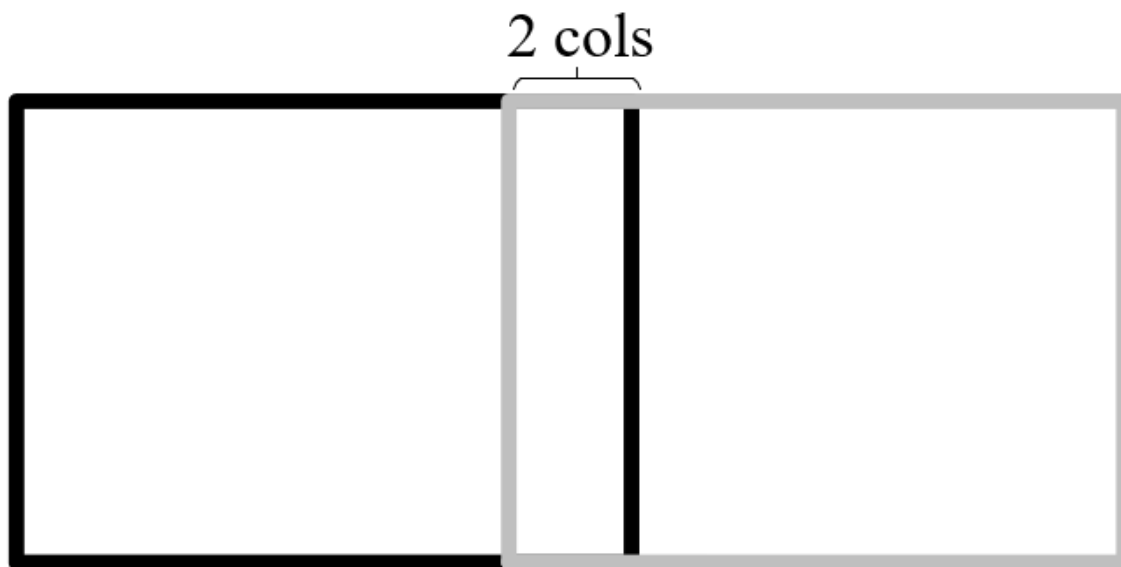
- padding 后矩阵四个角的数据与原矩阵四个角的数据相同
- padding 后矩阵第一列/行的数据与原矩阵第一列/行的数据相同
- padding 后矩阵最后一列/行的数据与原矩阵最后一列/行的数据相同

另外卷积核的步长固定为 1

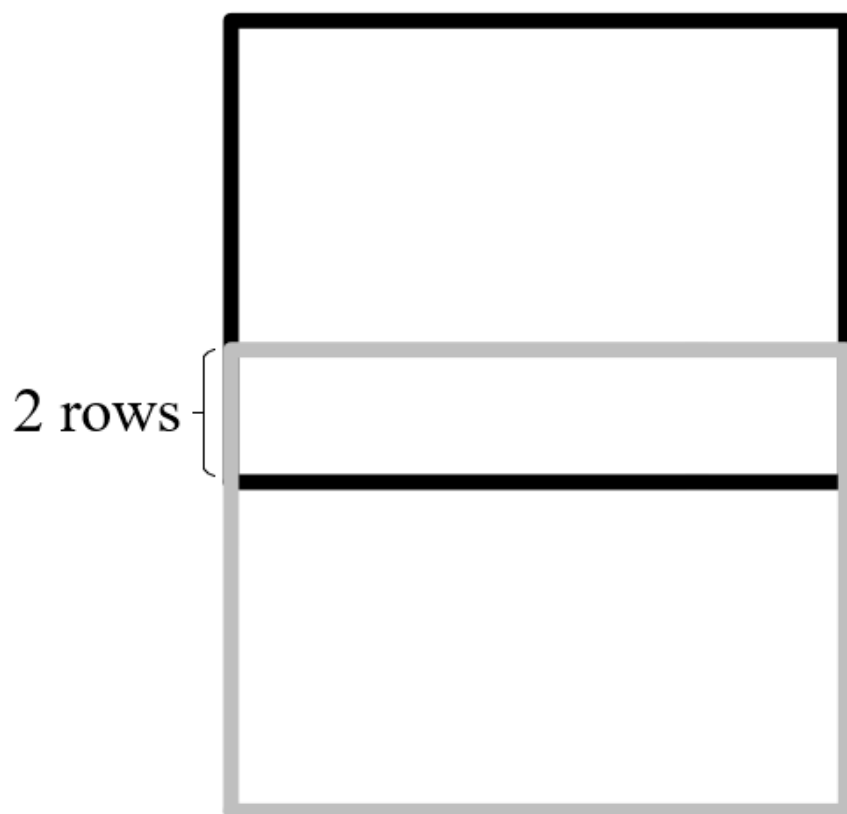
### 分块

分块的重点是需要处理相邻分块矩阵数据的 overlap，根据 padding 策略和卷积核的步长，这里的 overlap 策略为：

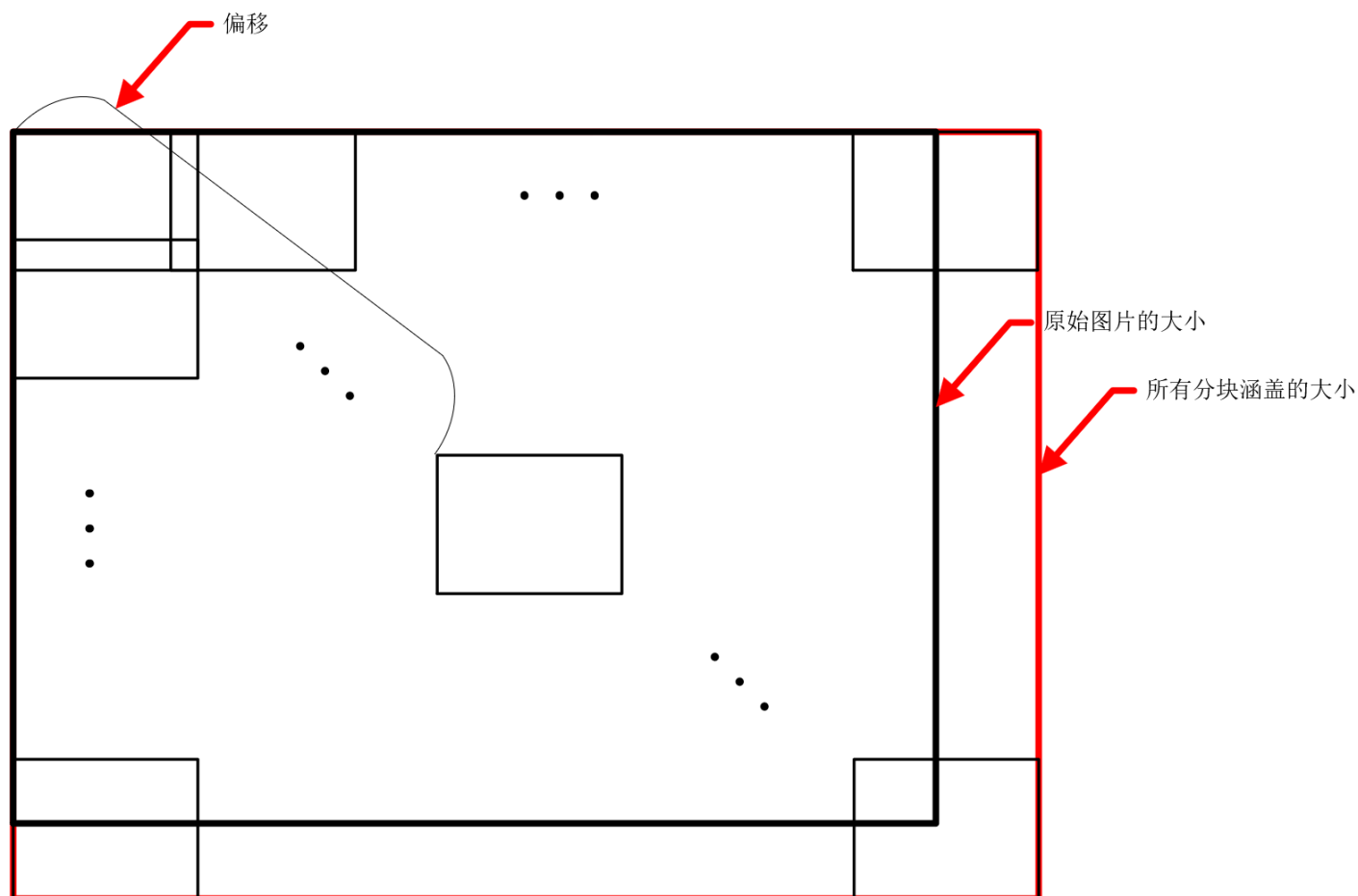
- 左右两个分块矩阵需要重叠左侧矩阵的最后两列数据，如图1所示



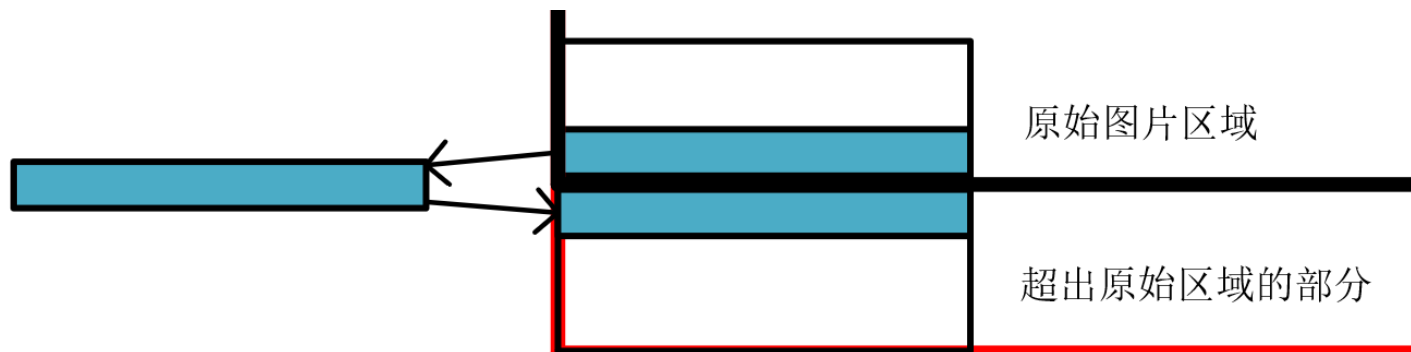
- 上下两个分块矩阵需要重叠上方矩阵的最后两行数据，如图2所示



完整图片的分块示意图如下，截取每个分块数据的关键就在于获得这个分块相对于原点的偏移



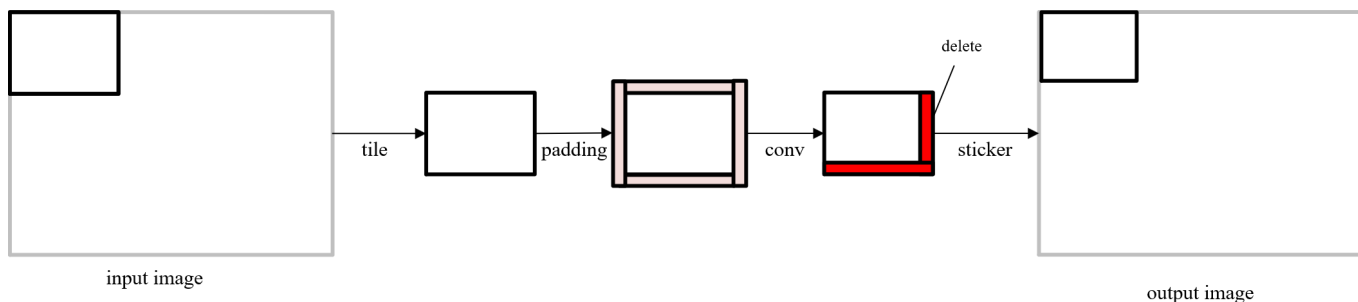
另外对于最后一行和最后一列的分块，它们涵盖的范围有部分超过了原始图片区域。根据 padding 策略，只需将原始图片的最后一行/列的数据复制给这部分区域的第一行/列即可，如下图所示



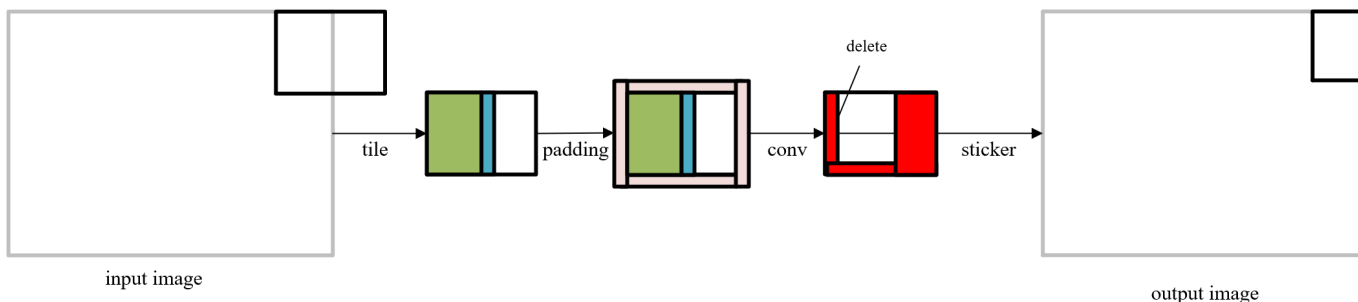
## 拼接

AIE会对每个分块数据进行 padding 后再卷积，因此AIE计算所得的部分分块结果在拼接时需要丢弃。这里的算法将计算所得的各个分块数据按照其在原始图片中的顺序进行拼接（从左至右，从上至下），具体算法描述如下：

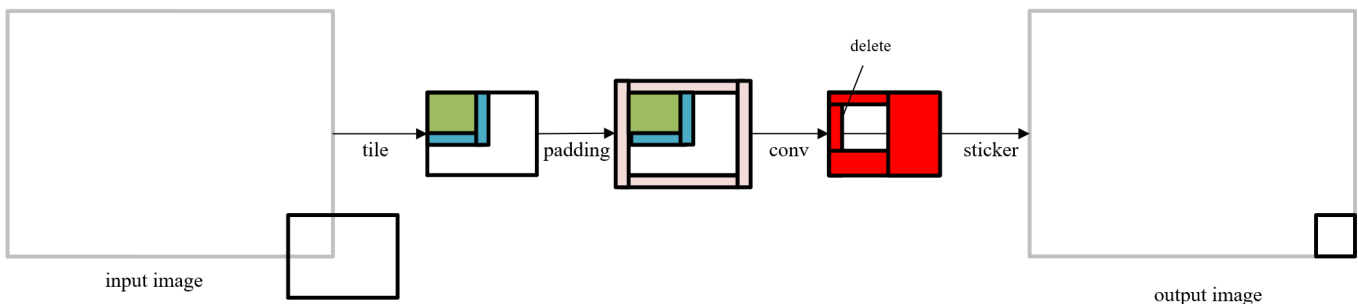
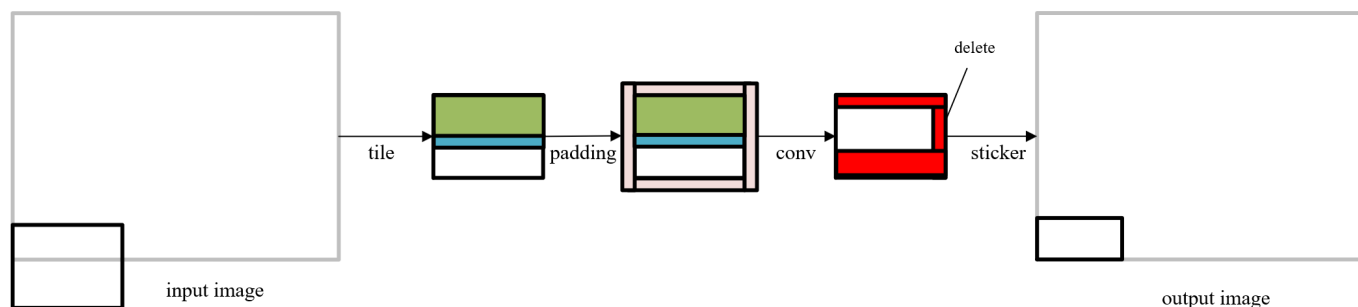
- 位于左上角的分块，其最后一列和最后一行的计算结果是错误的（除了完整图片的大小和AIE能处理的图片大小一致等特殊情况）。如下图所示，粉色区域代表AIE进行 padding 后得到的数据，红色区域代表错误的结果。但是为了满足 完整图片大小与AIE能处理的图片大小一致 等特殊情况，这里直接将这个分块所有的计算结果都写入最终输出中。这样最终输出中并不会包含这些错误的结果，因为按照从左至右，从上至下对分块结果进行拼接时，后续的分块会用正确的结果覆盖掉这些错误数据。



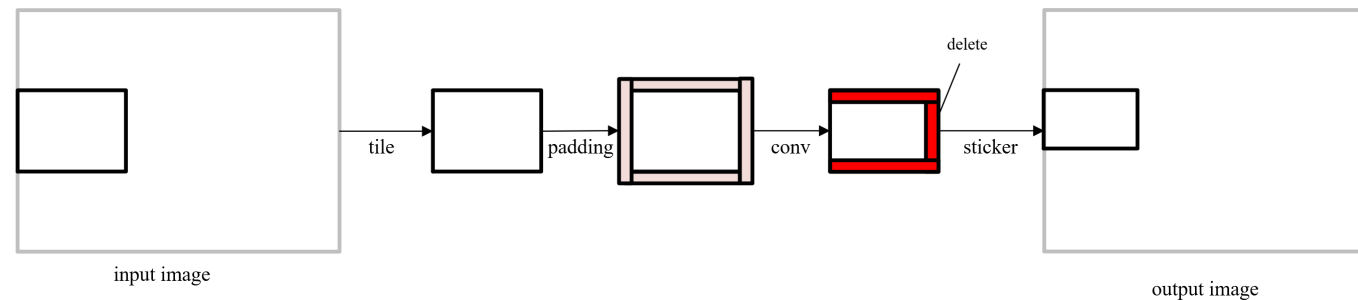
- 右上角分块的处理方式如下图所示，此分块的一部分数据不属于原始图片。绿色区域数据是属于原始图片的，蓝色区域数据与原始图片最后一列的数据相同，粉色和红色区域与之前的描述一致。这里写回时，依然会将最后一列没有超过原始图片区域的数据写入最终结果中，因为后续的分块会用正确的结果覆盖掉错误的数据。



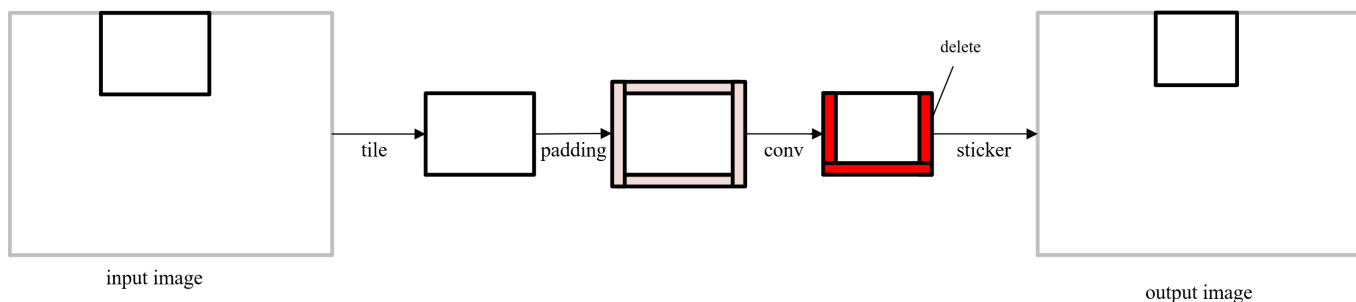
- 右下角和右上角分块的处理方式都类似，分别如下图所示



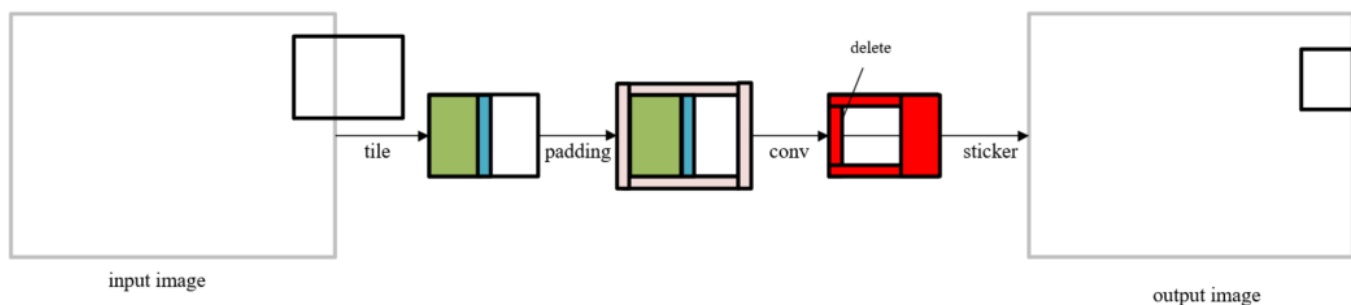
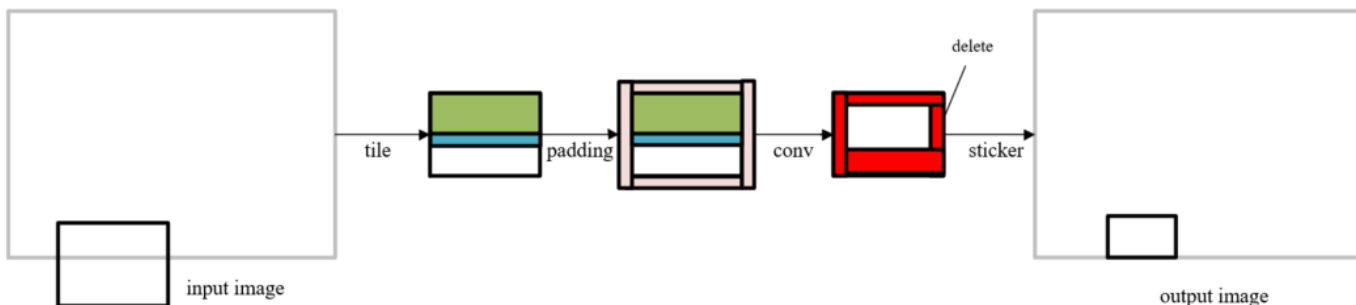
- 位于第一列（除左上角和左下角）的分块，其第一行、最后一行和最后一列的计算结果是错误的。但是这里依然会将此分块最后一列和最后一行的数据写入最终结果中，同样因为后续的分块会用正确的结果覆盖掉这些错误数据。



- 位于第一行（除左上角和右上角）的分块类似，如下图所示



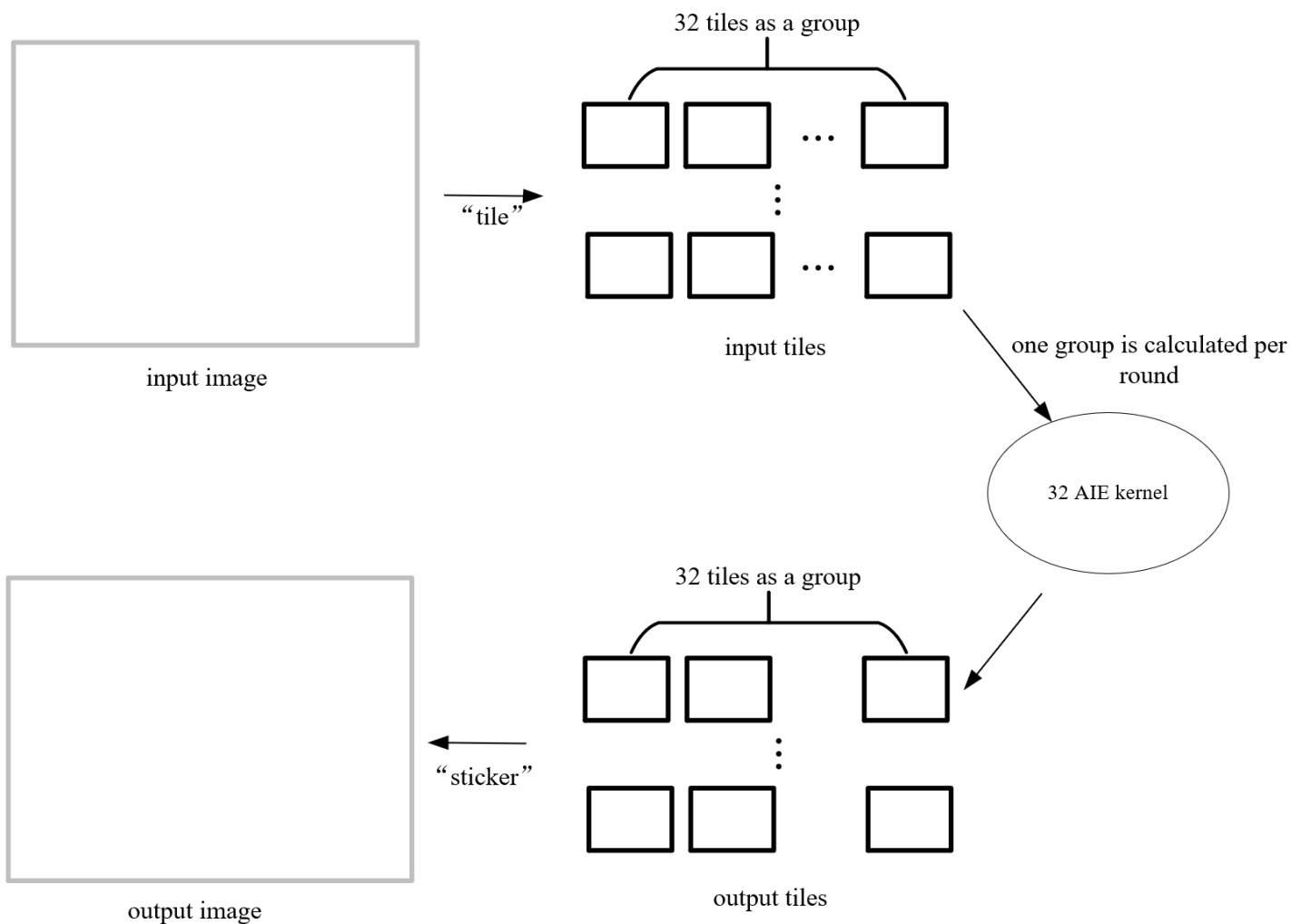
- 位于最后一列和最后一行（除左下角、右下角和右上角）的分块与右上角分块的处理方式类似，这些分块的一部分数据不属于原始图片，其处理方式分别如下图所示



## Graph

本实验定义了 32 个 kernel，每个 kernel 可以处理  $64 \times 32$  大小的图片。根据仿真结果，总共消耗了 57 个 AIE（在资源够用的情况下，可以增加 kernel 的个数，提高系统的吞吐量）

当输入的图片较大时（例如 1080P 图片， $1920 \times 1080$ ），可以将分块后得到的数据按照顺序（从左至右，从上至下）以每 32 个分块为一组进行分组。每个组内将分块按照顺序依次分给 32 个 kernel 进行计算，组内第  $i$  个分块的计算结果通过追加写写到“outputi.txt”文件中。当输入图片的总分块数量不能整除 kernel 个数时，向最后一个组添加若干全零的输入文件即可。总的来说，通过多次启动 graph 便可达到对连续输入的若干较大图片进行卷积运算的目的。单张图片的运算流程如下图所示



Graph 代码只是简单的将上述分块好的输入文件传输给每个 kernel，然后将每个 kernel 的输出分别存到各自的 output 文件中。仿真得到的 graph 示意图如下所示



# 系统测试

复现步骤：

```
# Build project
make all
# Run AIE Emulation
make aieemu
```

输入图片张数：2

输入图片宽度：1920

输入图片高度：1080

注意这里的图片默认只有一维，若需处理一张正常的RGB图片，需要手动将其展成三张图片

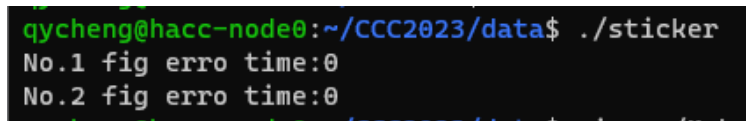
若需增加输入图片的个数，需要修改[生成数据](#)、[拼接数据](#)代码中图片个数的参数，并且增加 AIE graph 启动的次数（此启动次数通过运行[生成数据](#)程序会自动计算回显）

若想改变图片大小，需要对生成数据、拼接数据代码中的参数进行修改，AIE 中 graph 启动次数也需要进行响应的调整

## 正确性验证

本实验首先用简单的循环计算出输入图片对应的参考卷积结果，然后与 AIE 的计算结果（输入图片--->分块--->调用 AIE 仿真--->拼接结果）进行对比。

当输入两张 1080P 图片时，可以看到 AIE 的计算结果与软件计算出的参考结果一致



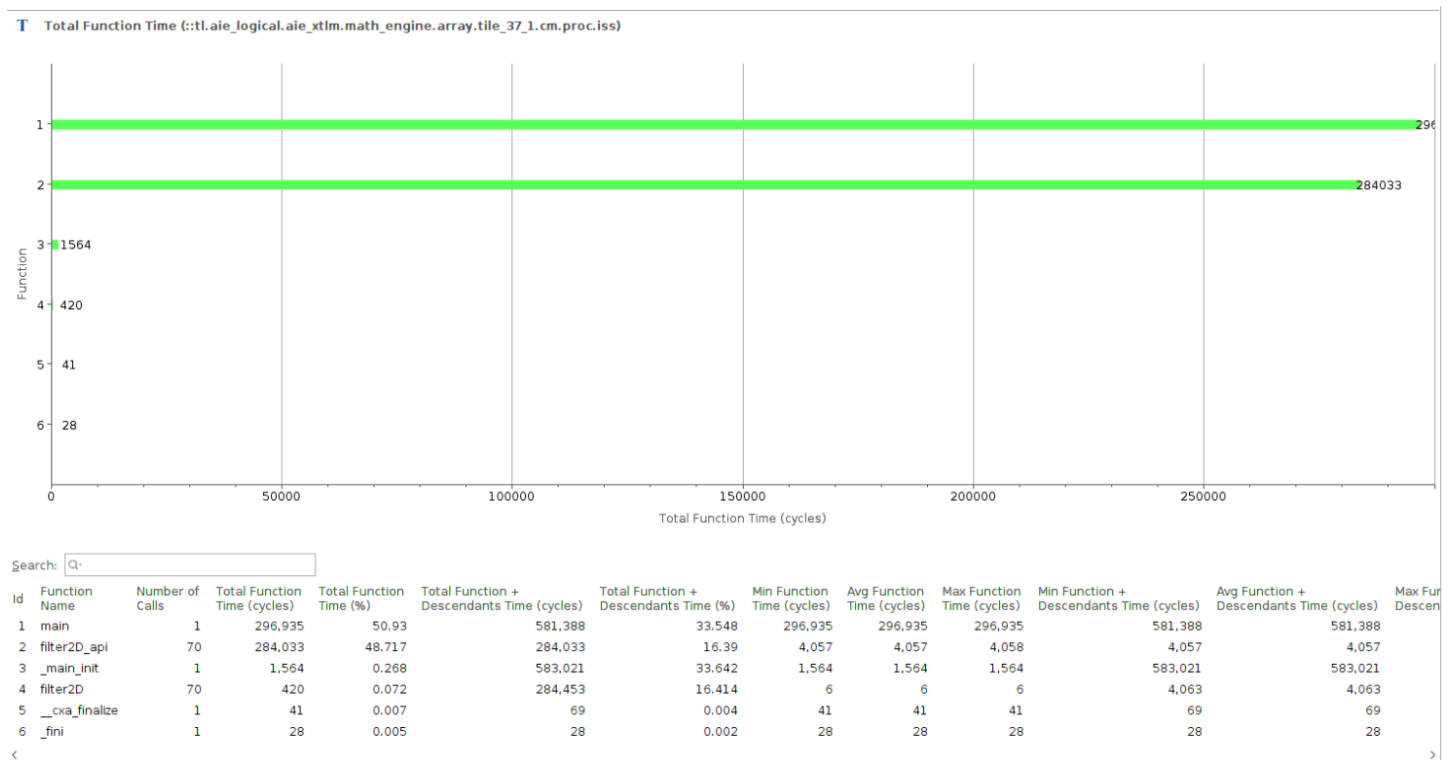
```
qycheng@hacc-node0:~/CCC2023/data$ ./sticker
No.1 fig erro time:0
No.2 fig erro time:0
```

## 系统性能分析

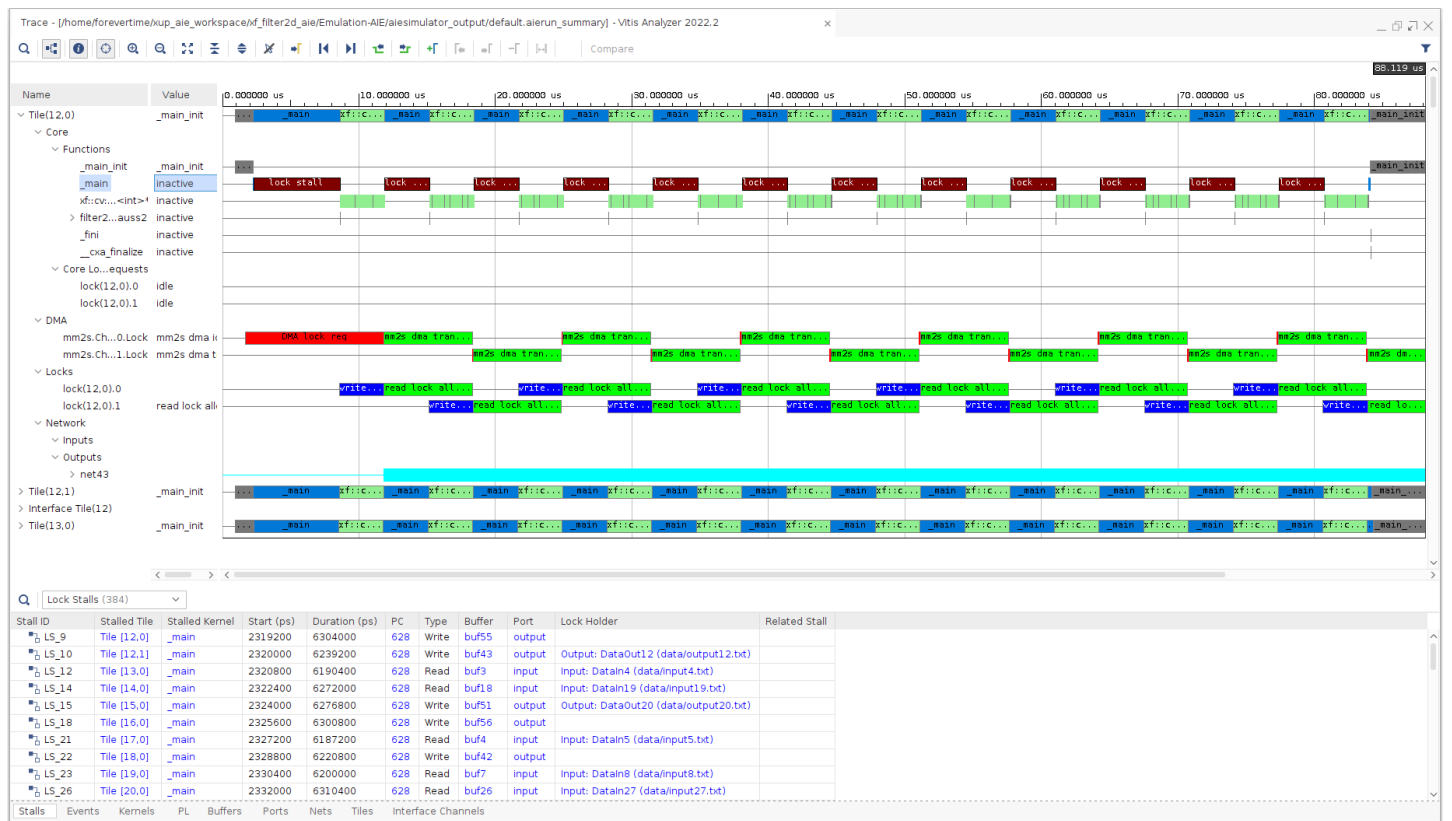
这里的分块和拼接操作均由软件端完成，即需要给每个 AIE 提前准备好对应的输入文件，并且利用软件端对每个 AIE 的输出文件做进一步处理。

故此小节的性能仅仅通过仿真结果分析在输入文件准备好且不需要对输出文件进行拼接时 AIE 的吞吐量

处理两张一维的 1080P 图片时，输入的数据总量=15.8MB 总共花费的时钟周期=296935 cycles 时钟频率=300MHz  
故吞吐量 = 127.7Gbps



下图截取至仿真的 trace 分析，不难看出 AIE 的绝大部分运行时间都花费在了数据传输上，未来可以从增加数据复用的角度出发，优化 AIE 的设计



注意本实验只定义了 32 个kernel，总共利用了 57 个AIE，在带宽、AIE个数允许的条件下可以增大 kernel 的个数来提升系统的性能

## 总结展望

总的来说，本实验利用仿真软件可实现对任意张数任意大小（图片为一维，大小不小于单个AIE kernel 能处理的图片大小）的图片进行filter2D计算



由于对开发流程和开发工具的不熟练，本次实验未能调试完毕 host 端的代码。未来可以将分块和拼接操作移植到 PL 端并且进一步修改 host 端代码。