

## Vitis Accel Examples: Systolic Array

## Reference:

[https://github.com/Xilinx/Vitis\\_Accel\\_Examples/tree/master/cpp\\_kernels/systolic\\_array](https://github.com/Xilinx/Vitis_Accel_Examples/tree/master/cpp_kernels/systolic_array)

## 1. Introduction

The systolic array is a hardware-efficient implementation for matrix multiplication. Compared to a direct-mapped design, the architecture of a systolic array adopts the parallel processing technique in a pipelined way. The network of computing units rhythmically processes and passes data through the system. Below is an illustration of a systolic array processor.

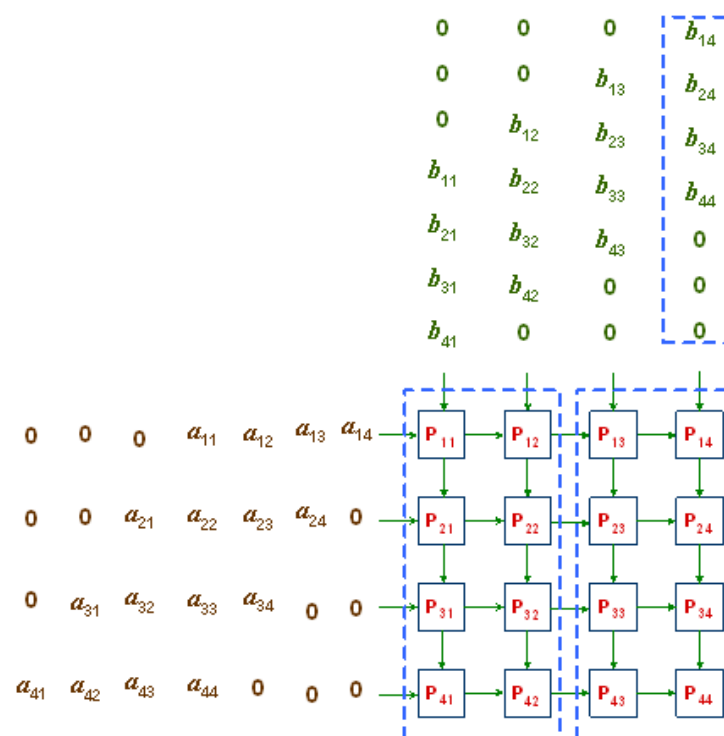


Figure. A 4 x 4 systolic array of processors for matrix multiplication.

(Source: <http://ecelabs.njit.edu/ece459/lab3.php>)

The two matrices A and B are shifted into the boundary processors in column 1 and row 1, respectively, as shown in Figure. The leading and trailing 0s in rows and columns are employed so that elements  $a_{ir}$  and  $b_{rj}$  arrive at processor  $P_{ij}$  simultaneously for the operation  $a_{ir} \times b_{rj}$  to be performed. In the array,  $P_{ij}$  is

initialized to 0 for all  $i, j = 1, 2, 3, 4$ . In the end, processor  $P_{ij}$  will contain  $c_{ij}$ , for  $1 \leq i, j \leq 4$ . This algorithm takes time  $O(n)$  for  $n \times n$  matrices.

## 2. Vitis HLS Implementation

Source Code

Kernel code:

--- mmult.cpp                      Matrix multiplication kernel

Host code:

--- host.cpp                      Host program

--- xcl2.cpp                      xcl2 function file

--- xcl2.hpp                      xcl2 header file

In the matrix multiplication kernel, the matrices A and B are first loaded into the kernel buffer to enable burst mode. In the for loop, we can use `#pragma HLS LOOP_TRIPCOUNT` to specify the total number of iterations performed by a loop, so that timing analysis can be performed. The min and max value is set to 16, aligned with the input matrix specification defined in the host program.

```
readA:
    for (int loc = 0, i = 0, j = 0; loc < a_row * a_col; loc++, j++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size* c_size max = c_size * c_size
        if (j == a_col) {
            i++;
            j = 0;
        }
        localA[i][j] = a[loc];
    }

// Read Input B
readB:
    for (int loc = 0, i = 0, j = 0; loc < b_row * b_col; loc++, j++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size* c_size max = c_size * c_size
        if (j == b_col) {
            i++;
            j = 0;
        }
        localB[i][j] = b[loc];
    }
```

- *The `LOOP_TRIPCOUNT` pragma or directive is for analysis only, and does not impact the results of synthesis. Note that the absence of this pragma causes an incorrect timing result in the synthesis report!!!*

The systolic matrix multiplication is performed in a three-level hierarchy of the for loop. In the loop, a single multiply-and-accumulate (MAC) operation is performed.

```
systolic1:
    for (int k = 0; k < a_col; k++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
        systolic2:
            for (int i = 0; i < MAX_SIZE; i++) {
#pragma HLS UNROLL
                systolic3:
                    for (int j = 0; j < MAX_SIZE; j++) {
#pragma HLS UNROLL
                        // Get previous sum
                        int last = (k == 0) ? 0 : localC[i][j];

                        // Update current sum
                        // Handle boundary conditions
                        int a_val = (i < a_row && k < a_col) ? localA[i][k] : 0;
                        int b_val = (k < b_row && j < b_col) ? localB[k][j] : 0;
                        int result = last + a_val * b_val;

                        // Write back results
                        localC[i][j] = result;
                    }
            }
    }
```

Finally, the output matrix C is written to global memory in burst mode.

Here we introduce the optimization pragma for the mmult kernel:

```
HLS ARRAY_PARTITION variable = localA dim = 1 complete
```

The matrix localA is partitioned into multiple arrays in the row direction

```
HLS ARRAY_PARTITION variable = localB dim = 2 complete
```

The matrix localB is partitioned into multiple arrays in the column direction

```
HLS ARRAY_PARTITION variable = localC dim = 0 complete
```

The matrix localC is partitioned completely. That is to say, it will be implemented using a register array with a size of 16 x 16 x 32 bits.

```
HLS UNROLL
```

This allows the MAC operation in the for loop to be processed in parallel.


### 3. Vitis HLS Build Flow

The build flow is the same as in Lab 3, except the program arguments in run


[illegible]

Kernels & Compute Units


Kernel Execution (includes estimated device times)

Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
 mmult	1	0.021	0.021	0.021	0.021

Top Kernel Execution

Kernel	Kernel Instance Address	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)
 mmult	0x55d5d52a2bd0	0	0	xilinx_u50_gen3x16_xdma_5_202210_1-0	0.219	0.021

Compute Unit Utilization (includes estimated device times)

Compute Unit	Kernel	Calls	Dataflow Execution	Max Parallel Executions	Dataflow Acceleration	CU Device Utilization (%)	CU Kernel Utilization (%)	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)	Clock Freq (MHz)
 mmult_1	mmult	1	Yes	1	1.000000x	17.404	17.404	0.004	0.004	0.004	0.004	300.000

## Effect of array partition

In the Bind Storage Report, we can see that after array partition, the BRAM for A and B in the original solution is replaced with 16 (storing each row or each column) register files each. In addition, the BRAM for C is replaced with all registers, due to the **complete** partition argument in the pragma. Therefore, the FF (Flip-Flop) and LUT in the Resource Estimate of the synthesis report are drastically increased.

Before:

Name	BRAM	URAM	Pragma	Variable	Storage	Impl
mmult		4	0			
localA_U		1	0		localA	rom_np
localB_U		1	0		localB	ram_1p
localC_U		1	0		localC	ram_s2p

After:

Name	BRAM	URAM	Pragma	Variable	Storage	Impl
mmult		0	0			
localA_U		0	0		localA	ram_1p
localA_1_U		0	0		localA_1	ram_1p
localA_2_U		0	0		localA_2	ram_1p
localA_3_U		0	0		localA_3	ram_1p
localA_4_U		0	0		localA_4	ram_1p
localA_5_U		0	0		localA_5	ram_1p
localA_6_U		0	0		localA_6	ram_1p
localA_7_U		0	0		localA_7	ram_1p
localA_8_U		0	0		localA_8	ram_1p
localA_9_U		0	0		localA_9	ram_1p
localA_10_U		0	0		localA_10	ram_1p
localA_11_U		0	0		localA_11	ram_1p
localA_12_U		0	0		localA_12	ram_1p
localA_13_U		0	0		localA_13	ram_1p
localA_14_U		0	0		localA_14	ram_1p
localA_15_U		0	0		localA_15	ram_1p
localB_U		0	0		localB	ram_1p
localB_1_U		0	0		localB_1	ram_1p
localB_2_U		0	0		localB_2	ram_1p
localB_3_U		0	0		localB_3	ram_1p
localB_4_U		0	0		localB_4	ram_1p
localB_5_U		0	0		localB_5	ram_1p
localB_6_U		0	0		localB_6	ram_1p
localB_7_U		0	0		localB_7	ram_1p
localB_8_U		0	0		localB_8	ram_1p
localB_9_U		0	0		localB_9	ram_1p
localB_10_U		0	0		localB_10	ram_1p

## Effect of loop unrolling

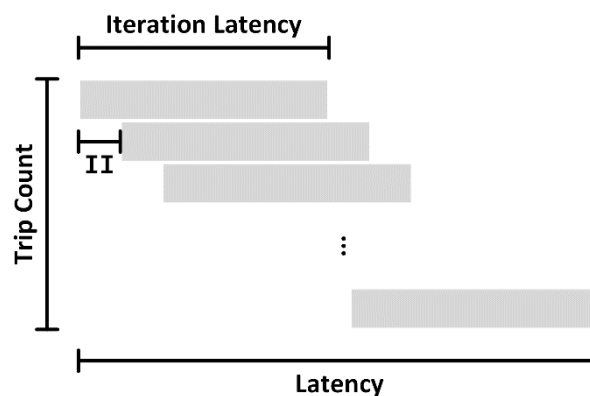
In the original solution, the system calculates a single MAC at each cycle, so the trip count is 4096 (16 x 16 x 16). This part accounts for 80% of the total latency and must be parallelized to increase throughput.

- *Note that in the original solution, the computation is automatically pipelined, so the computing resource is actually more than 1 MAC unit.*

Due to the UNROLL pragma, the computation is performed in parallel. In the optimized design, the systolic module is performed in a pipelined parallel fashion. The initiation interval (II), which is the number of clock cycles before the function can accept new input data, is 1. We can therefore know that the design is **fully pipelined** to increase throughput.

The latency of systolic computation can be calculated by leveraging the iteration latency, trip count, and interval. An interval of 19 is obtained by the following equation:

$$Latency - II + II \times (TC - 1) = 5 - 1 + (16 - 1) = 19$$



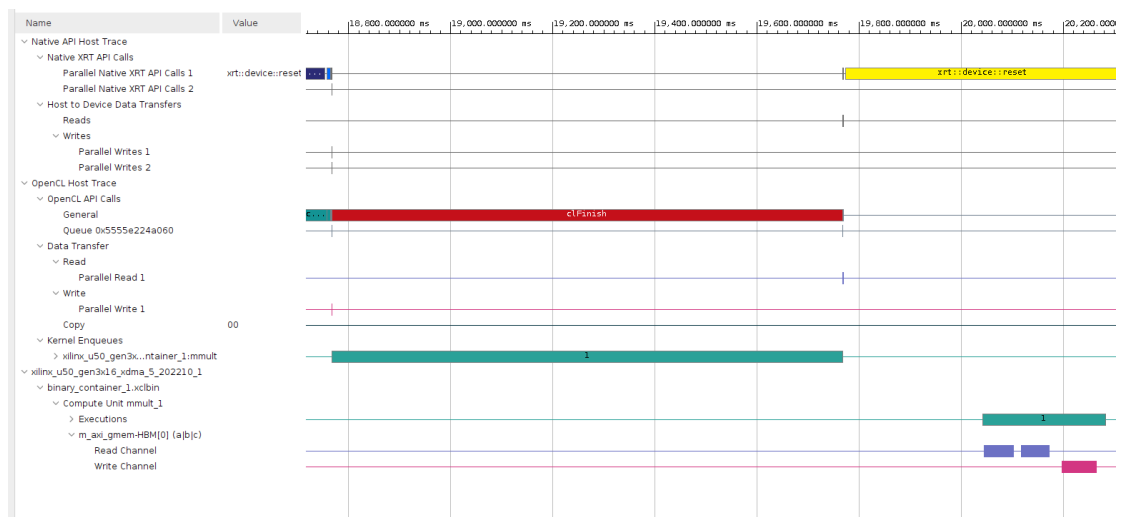
The latency after loop unrolling is reduced by over 95.5% (from 4100 to 19 cycles).

## 5. GitHub Link:

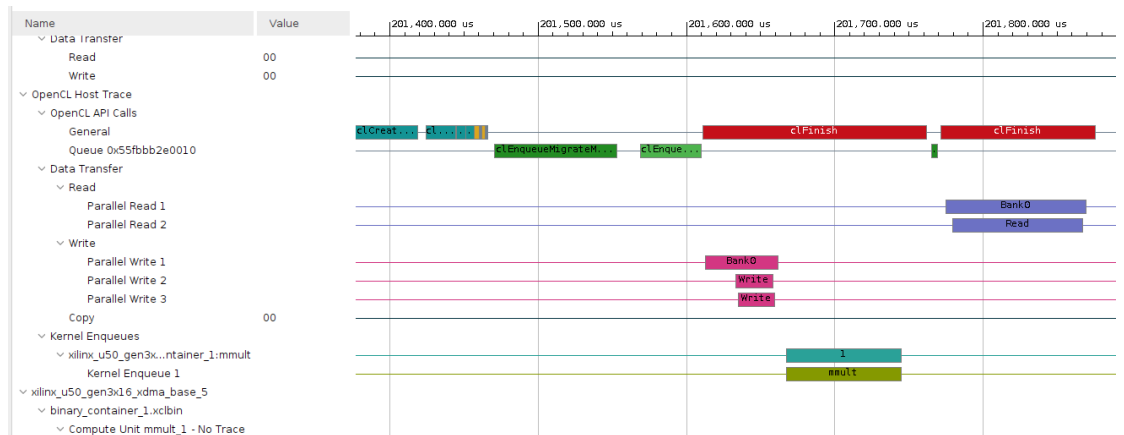
[https://github.com/yochenglin/HLS\\_LabB\\_systolic\\_array](https://github.com/yochenglin/HLS_LabB_systolic_array)

## 6. Appendix

### Run Hardware Emulation



## Run Hardware



```
[Console output redirected to file:/mnt/HLSNAS/03.RJWQGv/labB_yocheng/systolic_array_opt/systolic_array/Hardware/SystemDebug]
Found Platform
Platform Name: Xilinx
INFO: Reading /mnt/HLSNAS/03.RJWQGv/labB_yocheng/systolic_array_opt/systolic_array_system/Hardware/binary_container_1.xclbin
Loading: '/mnt/HLSNAS/03.RJWQGv/labB_yocheng/systolic_array_opt/systolic_array_system/Hardware/binary_container_1.xclbin'
Trying to program device[0]: xilinx_u50_gen3x16_xdma_base_5
Device[0]: program successful!
TEST PASSED
```

Run hardware success!