Improving Memory Performance Array Partitioning in LegUp

Progress Sep 18, 2015

Joy Chen[1]

[1]Department of Electrical and Computers Engineering
University of Toronto

January 28, 2016

Outline

1. Previous Implementation
   - Hard-coded
2. Multiple Configurations
   - Available Configurations
   - Config File Specifications

Outline

Previous Implementation

- Hard-coded for partitioning array named 'array' completely in the first dimension
- Need to add a configuration file for additional partitioning options

Outline

Available Configurations

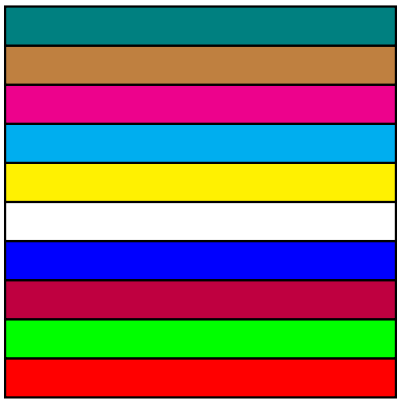
Overview

- Partitioning can only occur along **one** of the dimensions
- The available options are **COMPLETE**, **BLOCK**, **CYCLIC**, and **BLOCK CYCLIC**
- Note that when complete partitioning is used, the dimensionality of the array reduces by 1, not true for other types of partitioning
- In the following examples, we will assume a 10x10 2D array where we would partition in the highest dimension (rows)

Available Configurations

Complete

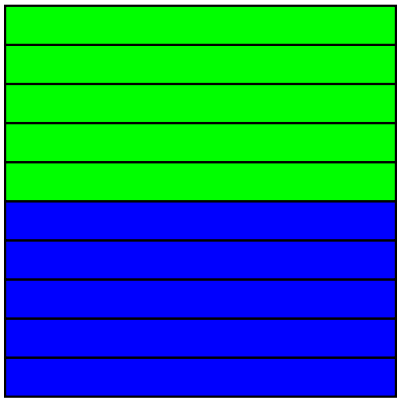
Each element of the partitioned dimension is separated into a different partition



Available Configurations

Block Partitioning

Given the number of elements of the partitioned dimension is N, the number of partitions in this dimension is p, the blocksize of each partition is N/p

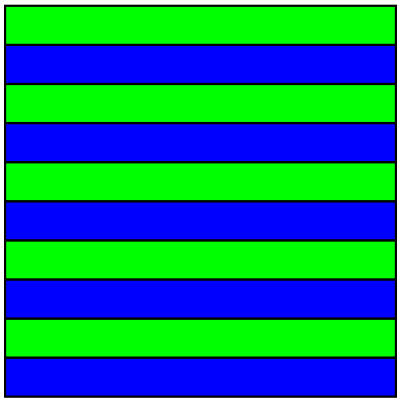


$N = 10,\ p = 2$

Available Configurations

Cyclic Partitioning

Cyclically partition the dimension. Given the number of elements of the partitioned dimension is N, the number of partitions in this dimension is p, the blocksize is always 1 and the number of times the cycle repeats is N/p

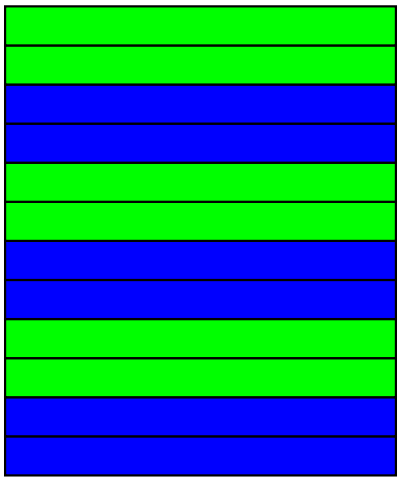


$N = 10$, $p = 2$

Available Configurations

Block Cyclic Partitioning

Cyclically partition the dimension by blocks. Combines block and cyclic partitioning. Given the number of elements of the partitioned dimension is N, the number of partitions in this dimension is p, the blocksize is k, the number of times the cycle repeats is N/(p*k)



$N = 12$, $p = 2$, $k = 2$

Outline

1. Previous Implementation
   - Hard-coded
2. Multiple Configurations
   - Available Configurations
   - Config File Specifications

Config File Specifications

- Each line in config file contains 4 columns
  1. Indicate global or local array: **global**, **local**
  2. Function name if local array: **FuncName**, -
  3. Name of array to be partitioned: **ArrayName**
  4. Configuration specification of how to partition the array

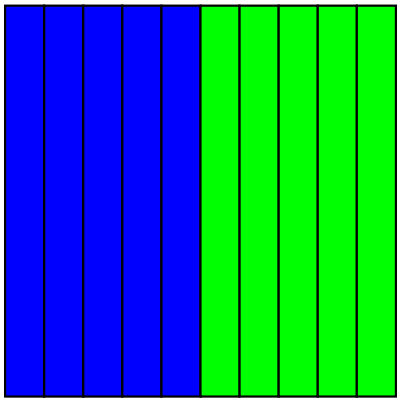
Config File Specifications

- String input where each dimension's partitioning scheme is individually specified as one of the following indicators
- '|' Is used as a separator between dimensions
- '-' No partitioning in dimension
- '*' Completely partition all the elements in that dimension
- 'b<#>' Block partition into <#> partitions
- 'c<#>' Cyclic partition into <#> partitions
- 'bc<#1>,<#2> Block cyclic partition into <#1> partitions, each having blocksize <#2>

Config File Specifications

Example

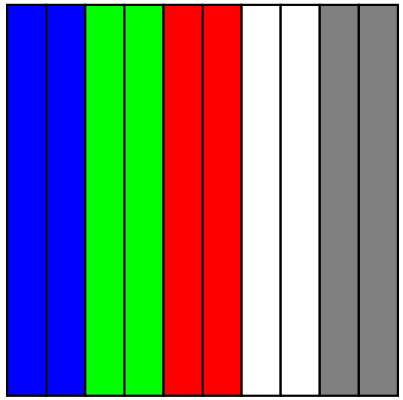
- Given a global 10x10 2D array 'matrix', split it column-wise into 2 separate arrays where the first sub-array is the left half and the second sub-array is the right half
- global - matrix -|b2|



Config File Specifications

Equivalent Configurations

A partition can be specified in multiple ways:



N = 10, p = 5
Configuration 'global - matrix -|b5|' also 'global - matrix -|bc5,2|' etc...

Array Separation Algorithm

Recursive algorithm for initializing new sub-arrays

```
1  // Returns sub-array initialization for partition as specified
2  getSubarrayInit(array, n) {
3      declare newInit
4      if dim not split {
5          for each element i=0:N in dim {
6              getSubarrayInit(array->elem(i), n)
7              concatenate to newInit
8          }
9      } else split dim {
10         for each repetition j=0:repCycle {
11             for each element i=0:blocksize in dim {
12                 concatenate to newInit
13             }
14         }
15     }
16     return newInit
17 }
```

Testing it out!

- Visually check that partitioned sub-arrays are as expected.
- Ran a simple test to access and sum up the values of global array **arr[12][12][12]**. At each access, the value at **arr[i][j][k]** is printed and matched with expected value. Sum is also compared.

Next Steps

- Create a parallel testbench to demonstrate that partitioning can provide speed up
- Apply array partitioning to loop pipelining cases
- Apply array partitioning to regular HLS in the case where multiple read/write accesses are required per clock cycle