ENSC 894 PROGRAMMING FOR HETEROGENEOUS COMPUTING SYSTEMS

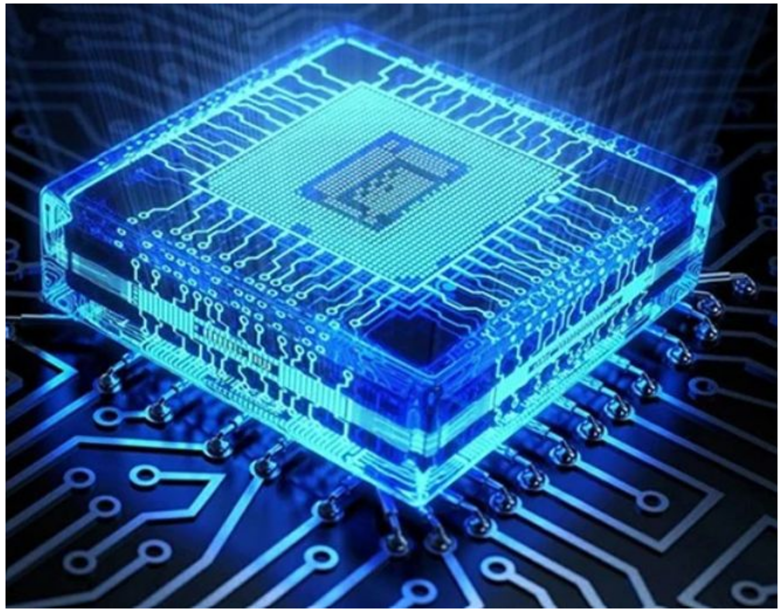
Analysing and Implementing Hotspot 3D

MOHAMMED SHUHAD

Contents

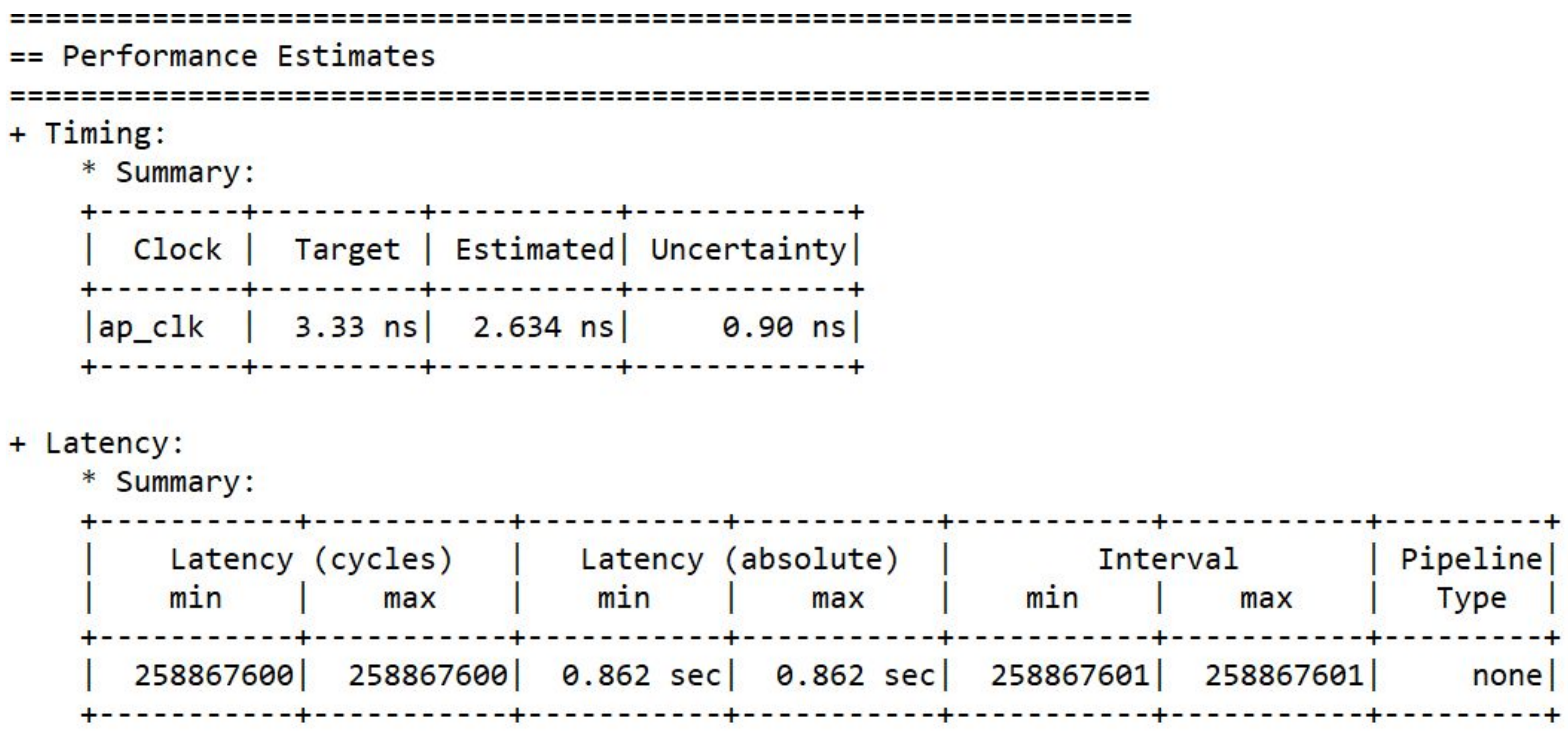
REVISITING- HOTSPOT3D

- Hotspot3D is a transient thermal modeling kernel, which computes the final state of a grid of cells when given the initial conditions (temperature and power dissipation per cell).
- The application iteratively updates the temperature values of all cells of a 3D grid in parallel and usually stops after given number of iterations.



BASELINE PERFORMANCE

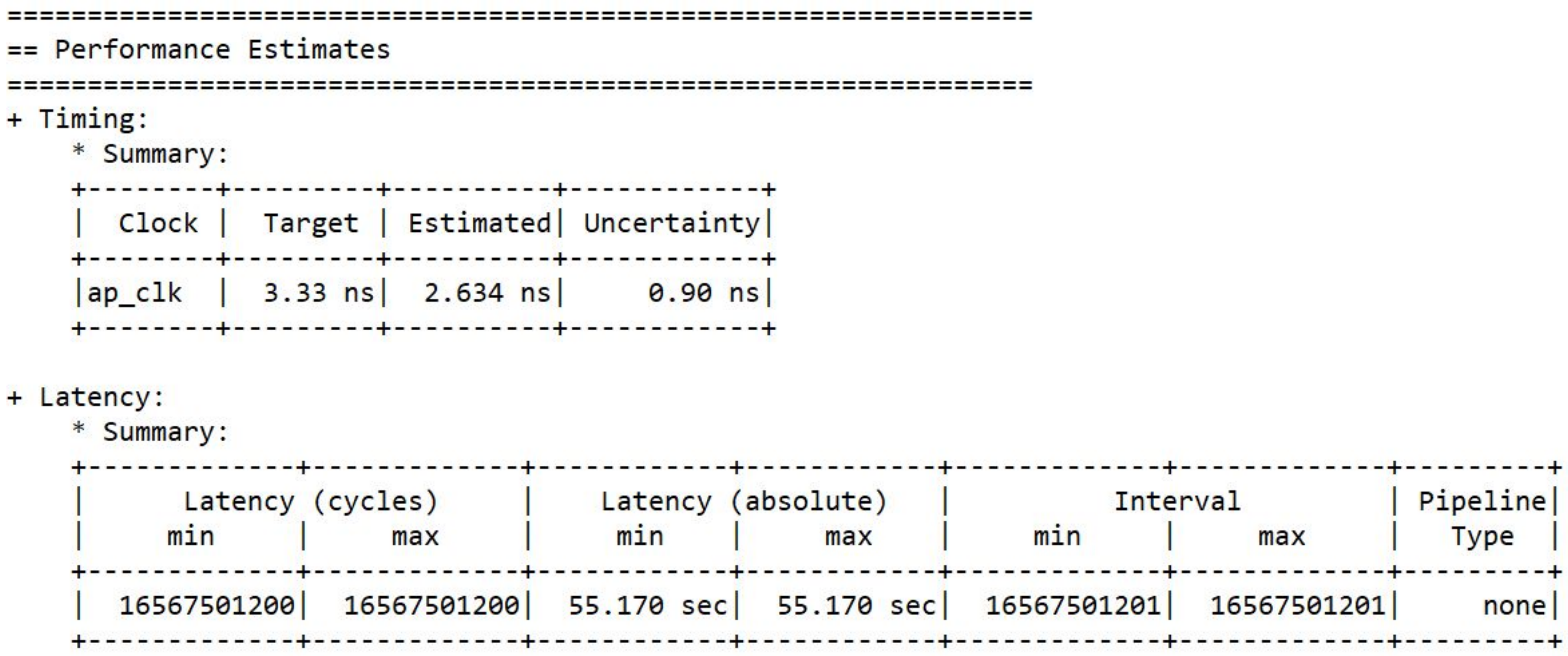
1. For 64x64x8



```
================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  3.33 ns|  2.634 ns|     0.90 ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +-----------+-----------+-----------+-----------+-----------+-----------+---------+
    |    Latency (cycles)   |   Latency (absolute)  |        Interval       | Pipeline|
    |    min    |    max    |    min    |    max    |    min    |    max    |   Type  |
    +-----------+-----------+-----------+-----------+-----------+-----------+---------+
    |  258867600|  258867600|  0.862 sec|  0.862 sec|  258867601|  258867601|     none|
    +-----------+-----------+-----------+-----------+-----------+-----------+---------+
```

BASELINE PERFORMANCE

1. For 512x512x8



```
================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  3.33 ns|  2.634 ns|     0.90 ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +-------------+-------------+-------------+-------------+-------------+-------------+---------+
    |     Latency (cycles)      |   Latency (absolute)      |          Interval         | Pipeline|
    |    min      |     max     |    min      |     max     |     min     |     max     |   Type  |
    +-------------+-------------+-------------+-------------+-------------+-------------+---------+
    |  16567501200|  16567501200|  55.170 sec|  55.170 sec|  16567501201|  16567501201|     none|
    +-------------+-------------+-------------+-------------+-------------+-------------+---------+
```

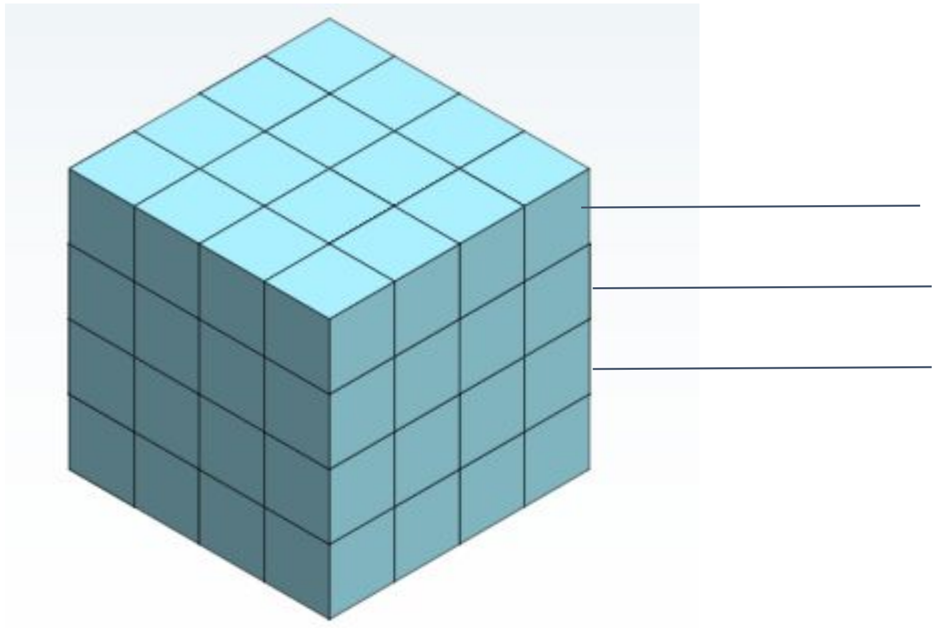
SFU

FPGA OPTIMIZATIONS

- BUFFERING
- COMPUTE OPTIMIZATIONS
- MEMORY OPTIMIZATIONS
- PING PONG BUFFER

BUFFERING

- Off chip data is moved to smaller sized buffers on on-chip memory.
- The data is transferred to on-chip memory by blocks of layers at a time.



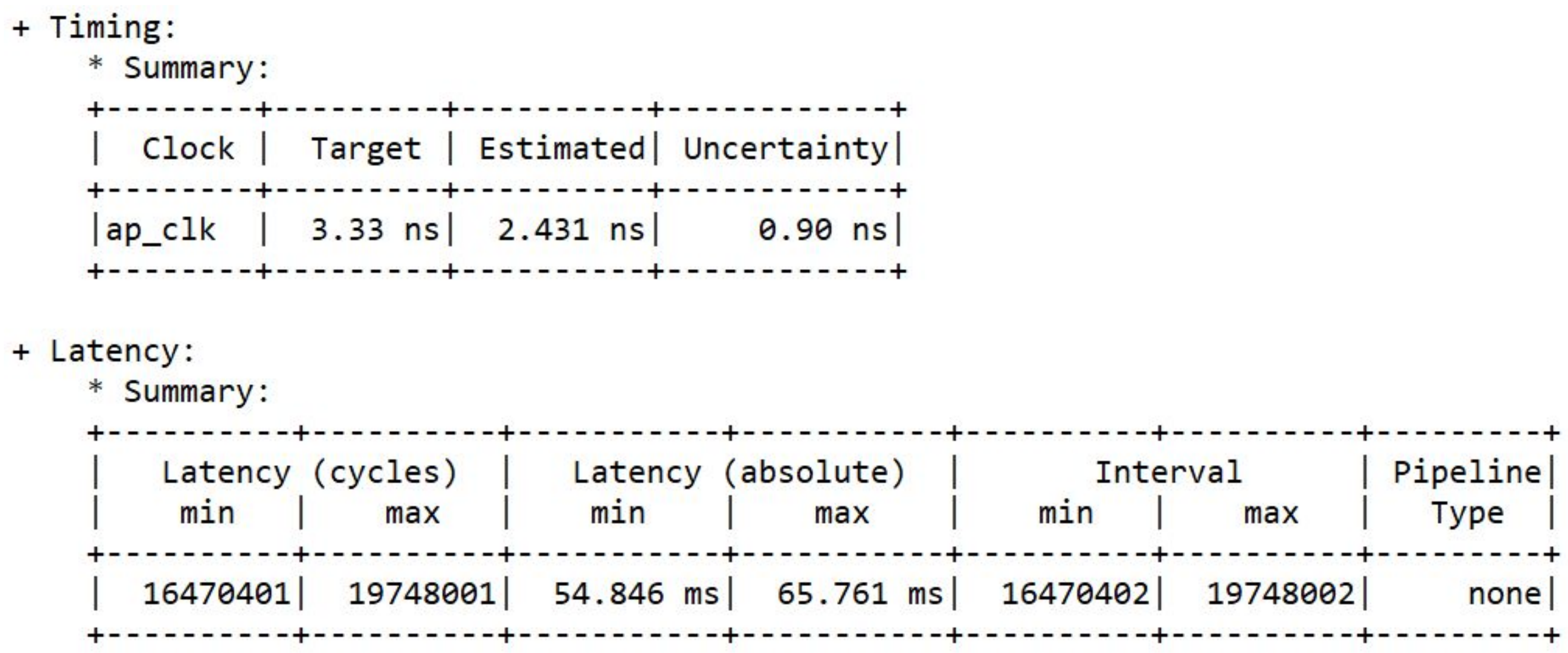
Instead of fetching the entire grid consisting of multiple layers, few layers are fetched at a time.
An extra layer from top and bottom are also fetched for computation.
Hence total layers fetched = TILE_SIZE + 2

BUFFERING - ALGORITHM

```
1. tile_size : Number of layers to be processed
2. grid_layers : Maximum number of layers
3. for i = 0 to grid_layers:
       2.1 Copy (tile_size + 2) units of data from base address + i.   LOAD
       2.2 Perform computation on the tile.   COMPUTE
       2.3 Store result to back to off chip at start address base address + i.   STORE
       2.4 Increment i by tile_size
```

BUFFERING - RESULT

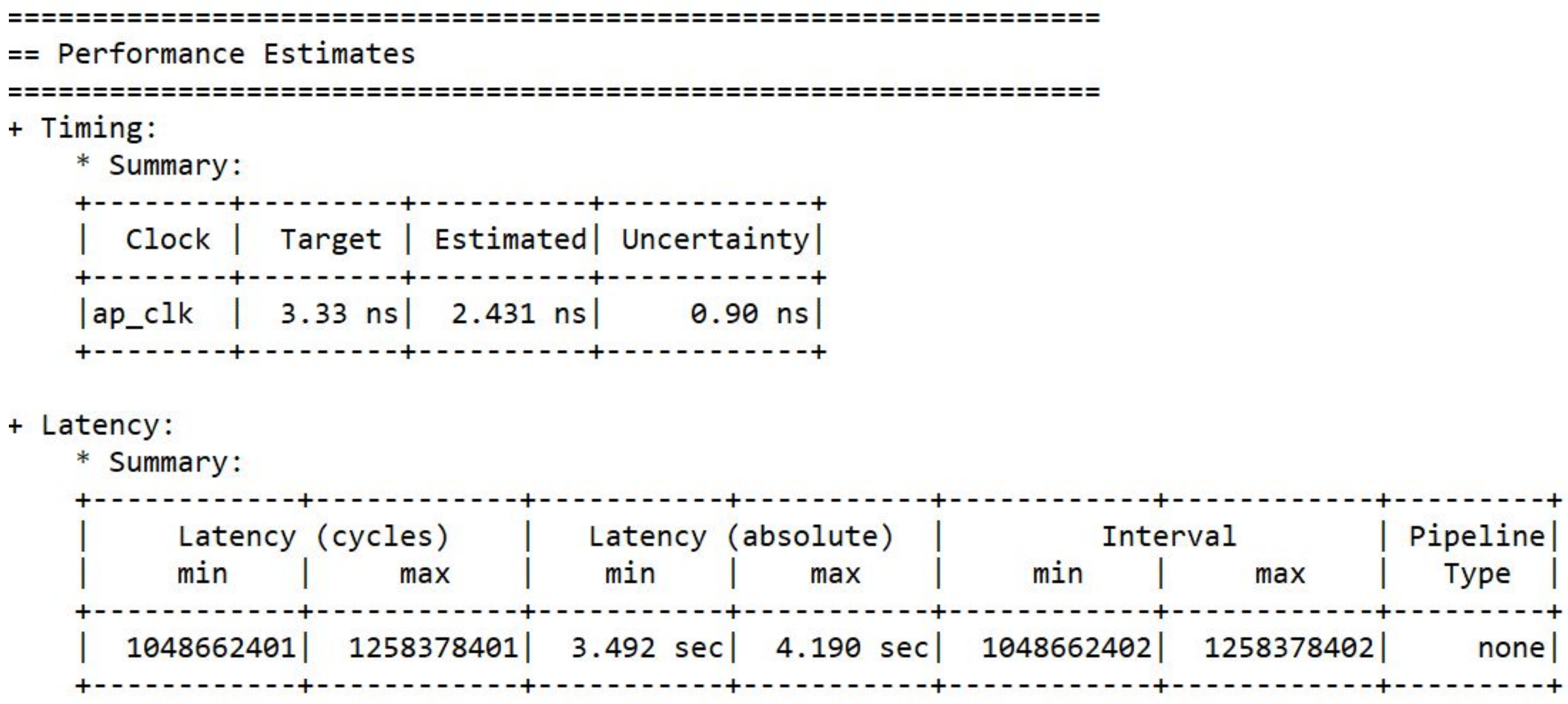
1. For 64x64x8



```
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  3.33 ns|  2.431 ns|     0.90 ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +----------+----------+-----------+-----------+----------+----------+---------+
    |   Latency (cycles)  |  Latency (absolute)   |       Interval      | Pipeline|
    |    min   |    max   |    min    |    max    |    min   |    max   |   Type  |
    +----------+----------+-----------+-----------+----------+----------+---------+
    |  16470401|  19748001| 54.846 ms | 65.761 ms |  16470402|  19748002|     none|
    +----------+----------+-----------+-----------+----------+----------+---------+
```

BUFFERING - RESULT

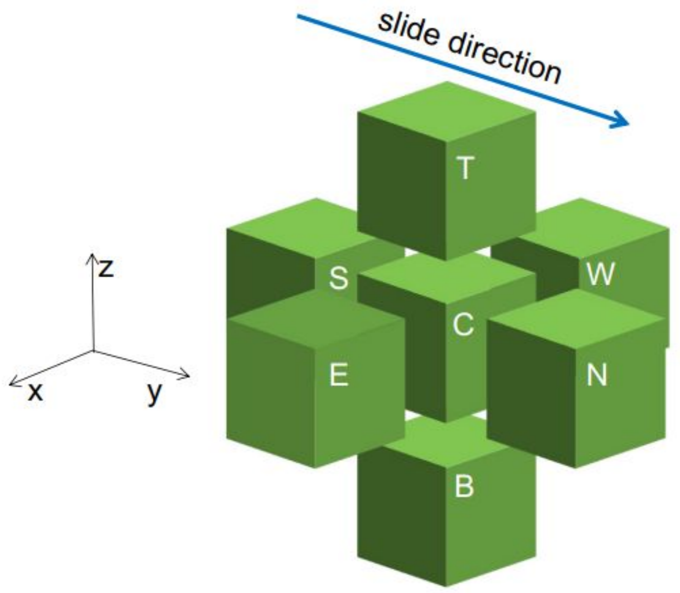
1. For 512x512x8



```
================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  3.33 ns|  2.431 ns|     0.90 ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +-------------+-------------+------------+------------+-------------+-------------+---------+
    |     Latency (cycles)      |  Latency (absolute)     |          Interval         | Pipeline|
    |     min     |     max     |    min     |    max     |     min     |     max     |   Type  |
    +-------------+-------------+------------+------------+-------------+-------------+---------+
    |   1048662401|   1258378401|  3.492 sec |  4.190 sec |   1048662402|   1258378402|     none|
    +-------------+-------------+------------+------------+-------------+-------------+---------+
```

COMPUTE OPTIMIZATIONS

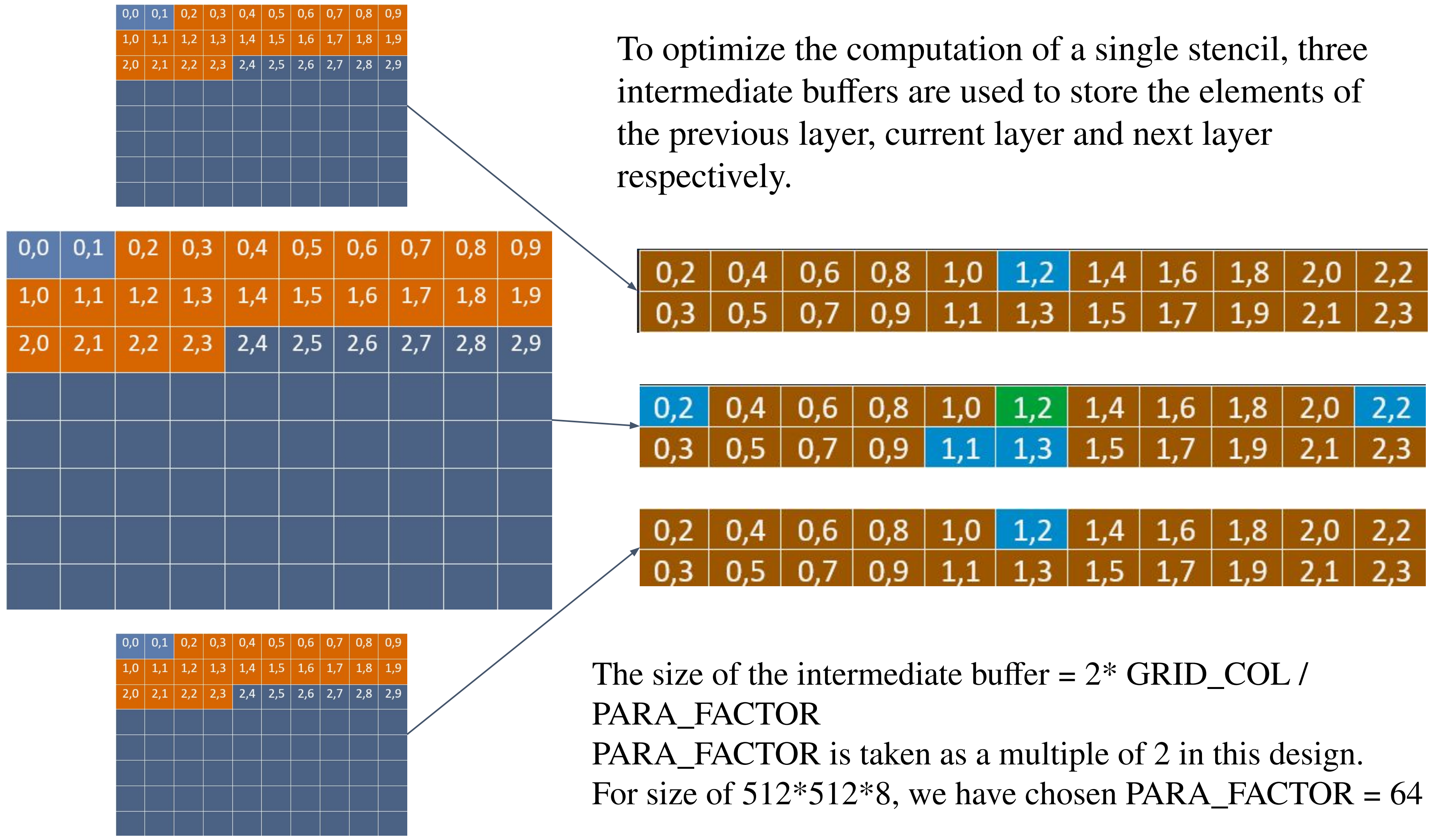
- In the compute method, further optimization is done to process the data in a smaller tile. This further parallelize the compute operation.
- Loop unroll, pipeline and array partition is applied on the compute method.
- Hotspot3D computation is a stencil computation, where the adjacent positions are used in calculating each stencil.



For the computation of a single stencil, elements from top and bottom layer are required, in addition to the neighbours in same row and column of the same layer. In the Hotspot3D program, the 3D model is linearised and elements in the previous/next layers are at an offset GRID_ROW * GRID COL and the maximum offset for neighbours in the same layer is GRID COL. This fact is utilised to fetch only the required number of cells for a single stencil computation.

GRID_COL = Number of columns , GRID_ROWS = Number of rows

COMPUTE OPTIMIZATIONS



To optimize the computation of a single stencil, three intermediate buffers are used to store the elements of the previous layer, current layer and next layer respectively.

The size of the intermediate buffer = 2* GRID_COL / PARA_FACTOR
PARA_FACTOR is taken as a multiple of 2 in this design.
For size of 512*512*8, we have chosen PARA_FACTOR = 64

The position of the element to be processed is chosen as (GRID_COL / PARA_FACTOR)/2. This is the center position. The neighboring positions can be easily obtained. For example, if the center element is BUFF[k][c], the element in the previous and next rows BUFF[k][0] and BUFF[k][GRID_COL / PARA_FACTOR - 1]

Sliding Window Mechanism is used to fetch the next element. In the example shown in the figure, elements in position (2,4) and (2,5) are fetched next after shifting the buffer array to the left.

COMPUTE ALGORITHM

1. Initialize buffer arrays: Fetch GRID_COLS * 2 / PARA_FACTOR elements from each layer.

2. Repeat GRID_ROWS*GRID_COLS/PARA_FACTOR times:

   2.1 Obtain center and neighbour elements : center, east, west, north, south, top and bottom

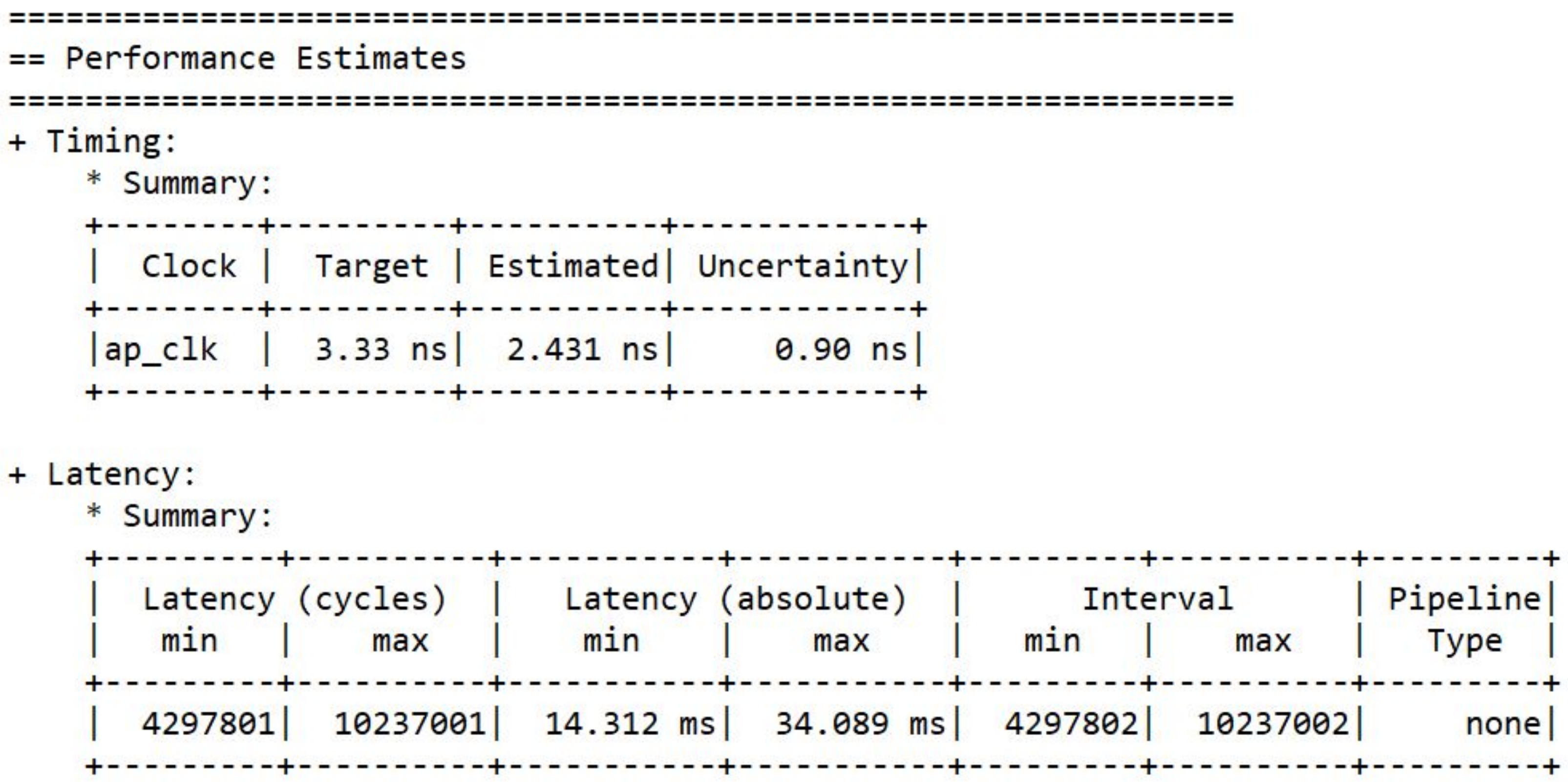
   2.2 Do stencil computation for the elements fetched.

   2.3 Left-Shift the buffer arrays by 1.

   2.4 Fetch PARA_FACTOR number of elements into the buffer at the last position.

COMPUTE OPTIMIZATIONS - RESULT

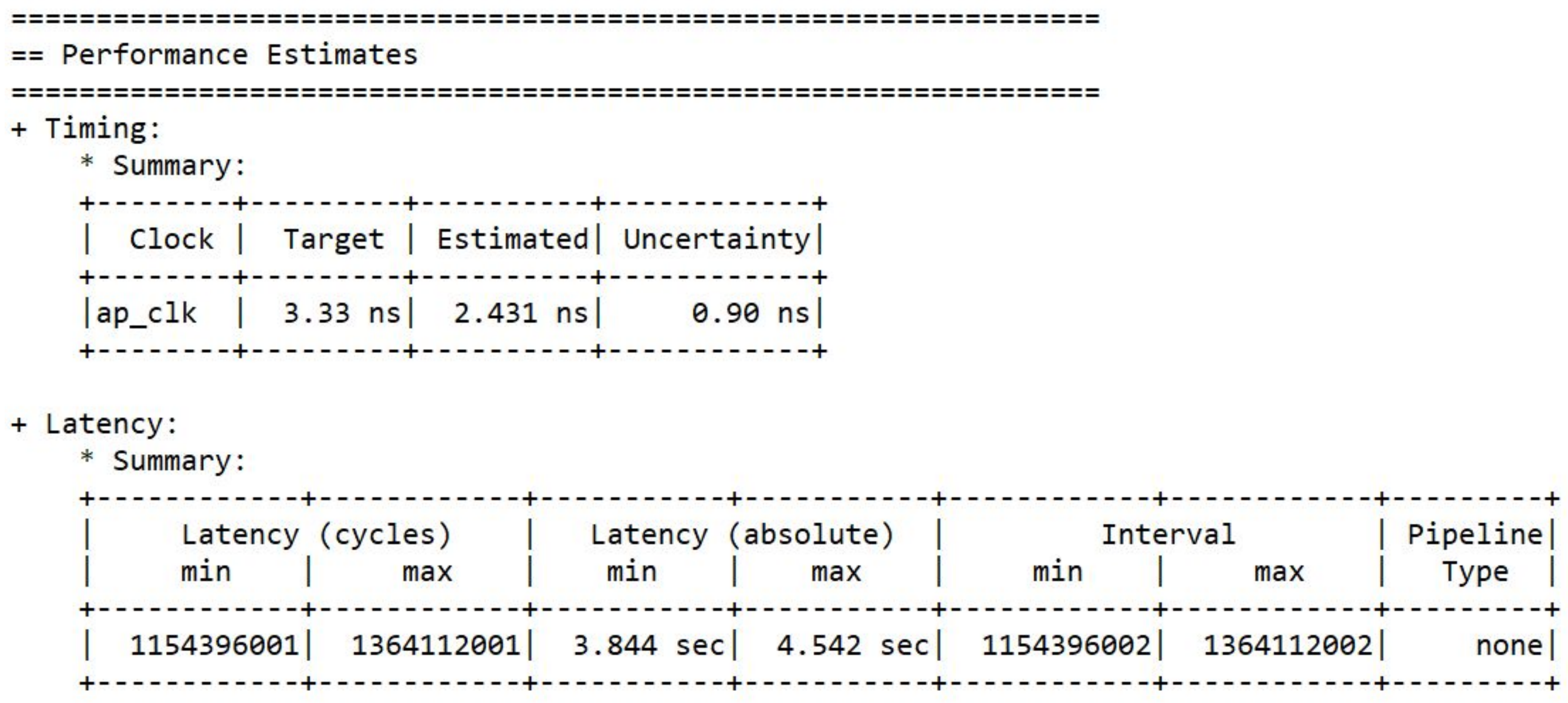
1. 64x64x8



```
================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  3.33 ns|  2.431 ns|     0.90 ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +---------+---------+-----------+-----------+---------+----------+---------+
    |  Latency (cycles) |  Latency (absolute)   |      Interval      | Pipeline|
    |   min   |   max   |    min    |    max    |   min   |    max   |   Type  |
    +---------+---------+-----------+-----------+---------+----------+---------+
    |  4297801| 10237001| 14.312 ms | 34.089 ms|  4297802|  10237002|     none|
    +---------+---------+-----------+-----------+---------+----------+---------+
```

COMPUTE OPTIMIZATIONS - RESULT

1. 512x512x8



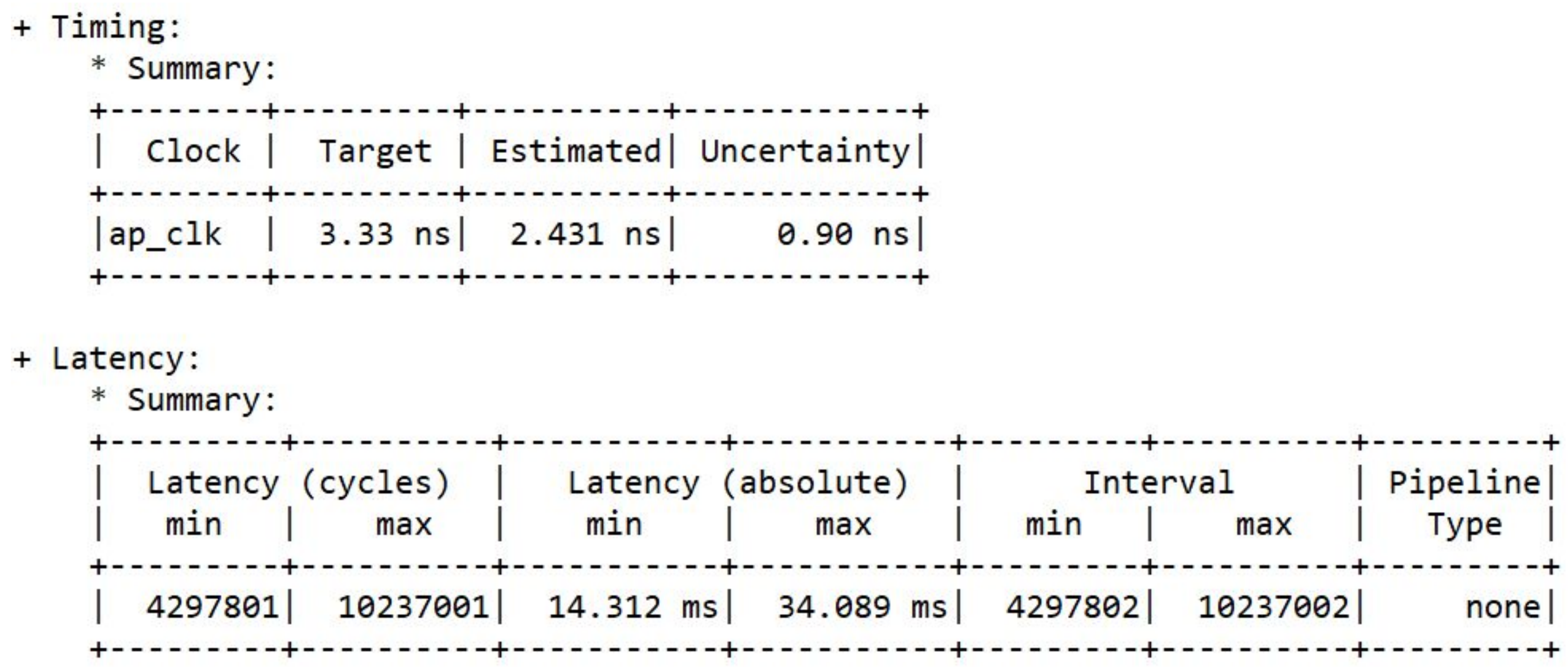
```
================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  3.33 ns|  2.431 ns|     0.90 ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +------------+------------+-----------+-----------+------------+------------+---------+
    |     Latency (cycles)    |   Latency (absolute)  |         Interval        | Pipeline|
    |    min     |    max     |    min    |    max    |    min     |    max     |   Type  |
    +------------+------------+-----------+-----------+------------+------------+---------+
    |  1154396001|  1364112001|  3.844 sec|  4.542 sec|  1154396002|  1364112002|     none|
    +------------+------------+-----------+-----------+------------+------------+---------+
```

MEMORY OPTIMIZATIONS

1. **Coalescing** is done in the load and store operations. The data from the off chip is sent using a wider type which enables faster reads in each cycle. We have chosen <ap_int<512>. The load and store methods now maps the data from <ap_int<512> to float data type.
2. **Array Partitioning** with loop pipelining and loop unrolling is done to enable parallel access in various for loops. We have done cyclic array partitioning is done with a factor of 64.

MEMORY OPTIMIZATIONS - RESULT

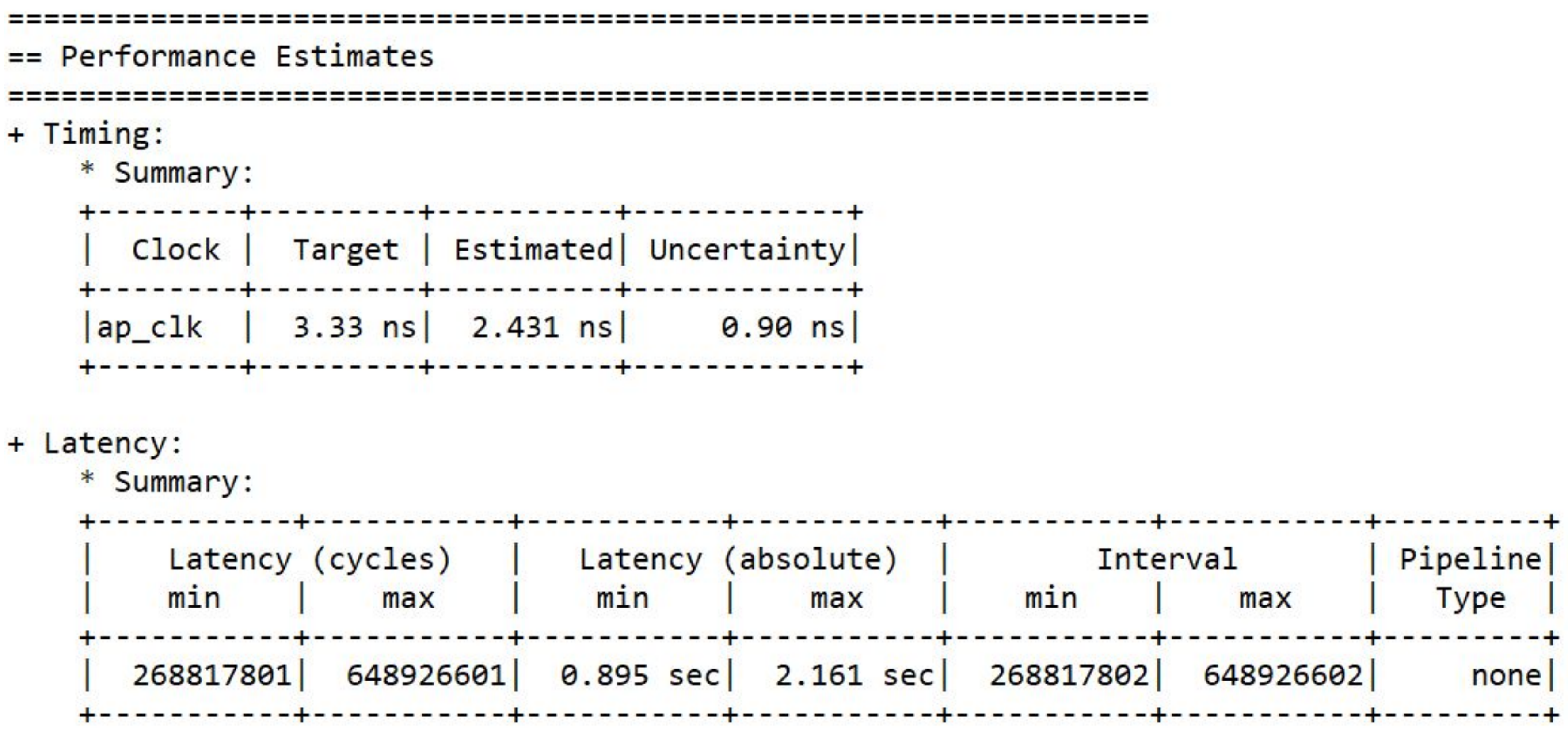
1. 64x64x8



```
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  3.33 ns|  2.431 ns|     0.90 ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +---------+----------+-----------+-----------+---------+----------+---------+
    |  Latency (cycles)  |  Latency (absolute)   |      Interval      | Pipeline|
    |   min   |    max   |    min    |    max    |   min   |    max   |   Type  |
    +---------+----------+-----------+-----------+---------+----------+---------+
    |  4297801|  10237001| 14.312 ms | 34.089 ms|  4297802|  10237002|     none|
    +---------+----------+-----------+-----------+---------+----------+---------+
```

MEMORY OPTIMIZATIONS - RESULT

1. 512x512x8



```
================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  | 3.33 ns|  2.431 ns|     0.90 ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +---------+---------+----------+----------+---------+---------+---------+
    |    Latency (cycles) |  Latency (absolute) |      Interval     | Pipeline|
    |   min   |   max   |    min   |    max   |   min   |   max   |   Type  |
    +---------+---------+----------+----------+---------+---------+---------+
    |  268817801|  648926601|  0.895 sec|  2.161 sec|  268817802|  648926602|     none|
    +---------+---------+----------+----------+---------+---------+---------+
```

PING PONG BUFFER

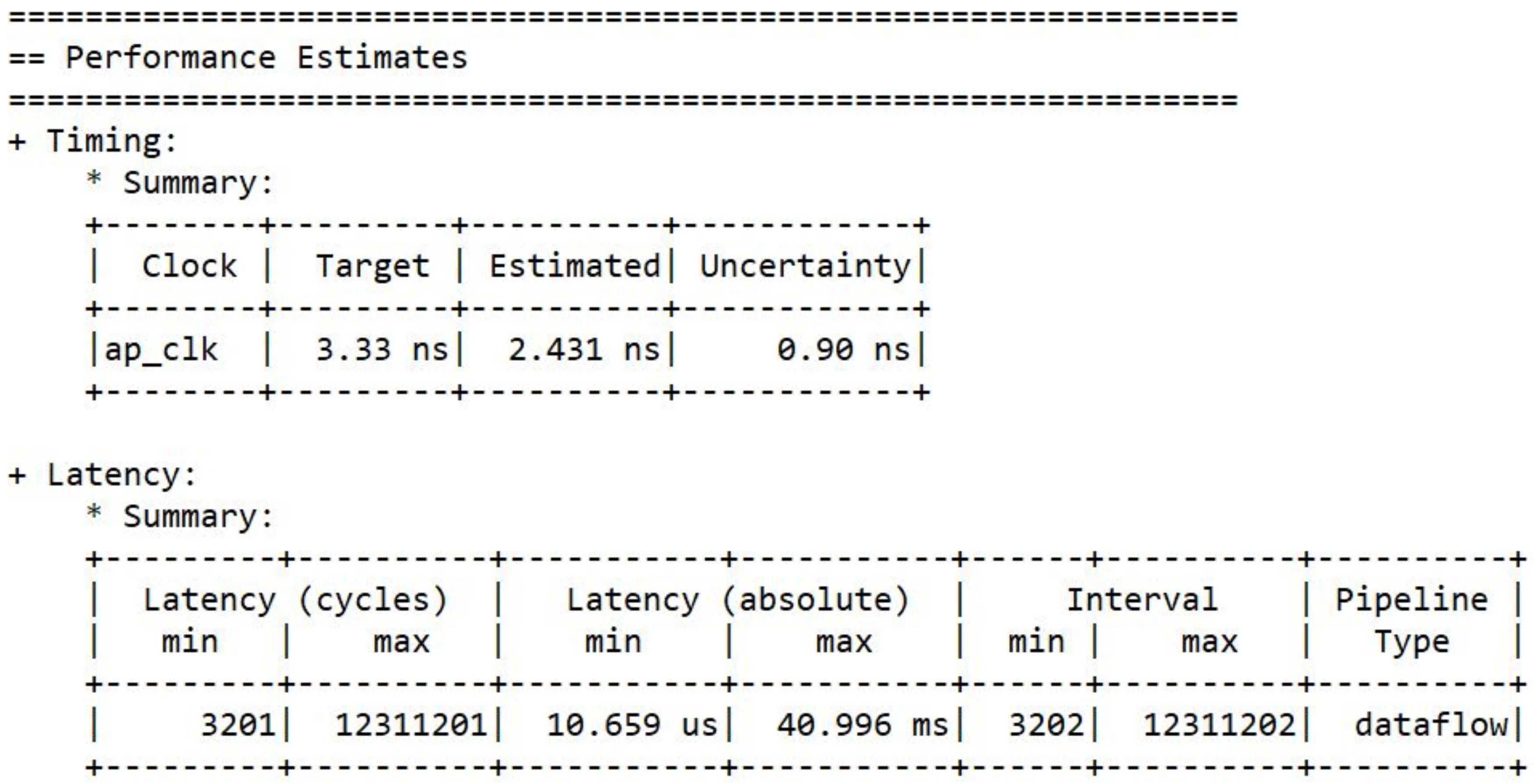
- Double buffering is done in load-compute-store to enable pipelining between the three methods.

Algorithm

```
1.  For i = 0 to N
    1.1 For layer = 0 to (GRID_LAYER - 1) + 2
        1.1.1 If layer%2 == 0
              i. Load tiles - temp_inner0, power_inner0
              ii. Compute tile -  result_inner1 using temp_inner1, power_inner1
              iii. Store result_inner0
        1.1.1 If layer%2 == 0
              i. Load tiles - temp_inner1, power_inner1
              ii. Compute tile -  result_inner0 using temp_inner0, power_inner0
              iii. Store result_inner1
```

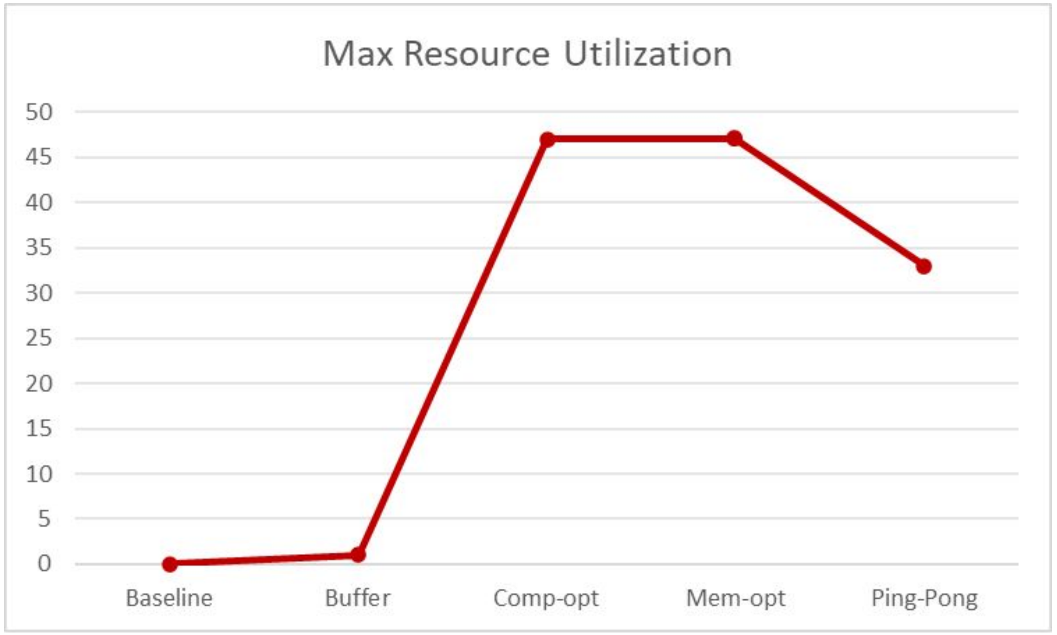
PING PONG BUFFER - RESULT

1. 64x64x8

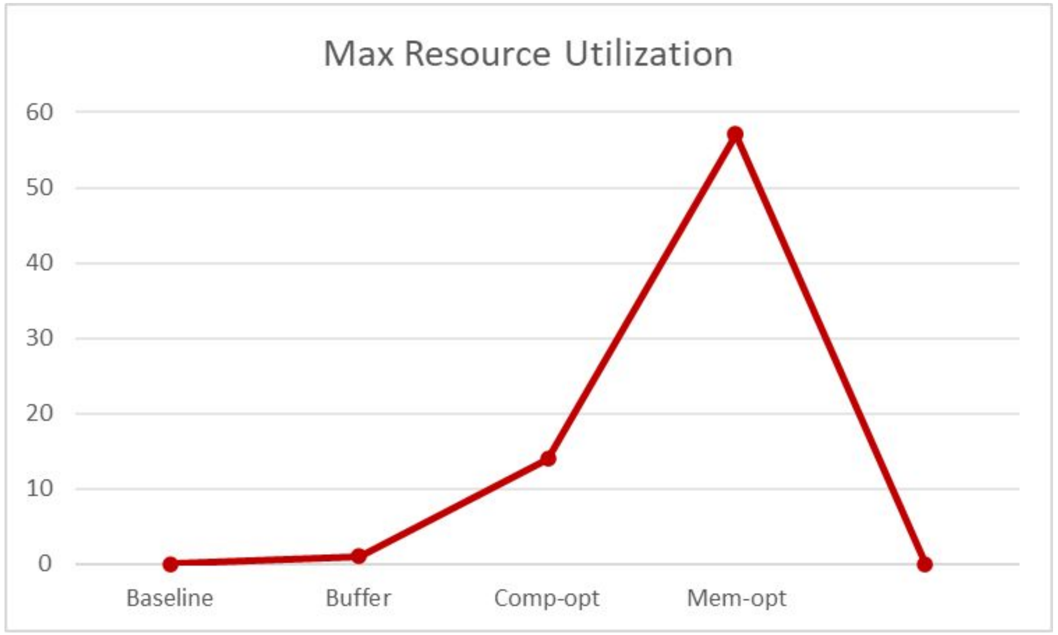


```
================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  3.33 ns|  2.431 ns|     0.90 ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +---------+----------+-----------+-----------+------+----------+----------+
    |  Latency (cycles)  |  Latency (absolute)   |     Interval    | Pipeline |
    |   min   |    max   |    min    |    max    |  min |    max   |   Type   |
    +---------+----------+-----------+-----------+------+----------+----------+
    |     3201|  12311201| 10.659 us | 40.996 ms|  3202|  12311202|  dataflow|
    +---------+----------+-----------+-----------+------+----------+----------+
```

Comparisons-Resources

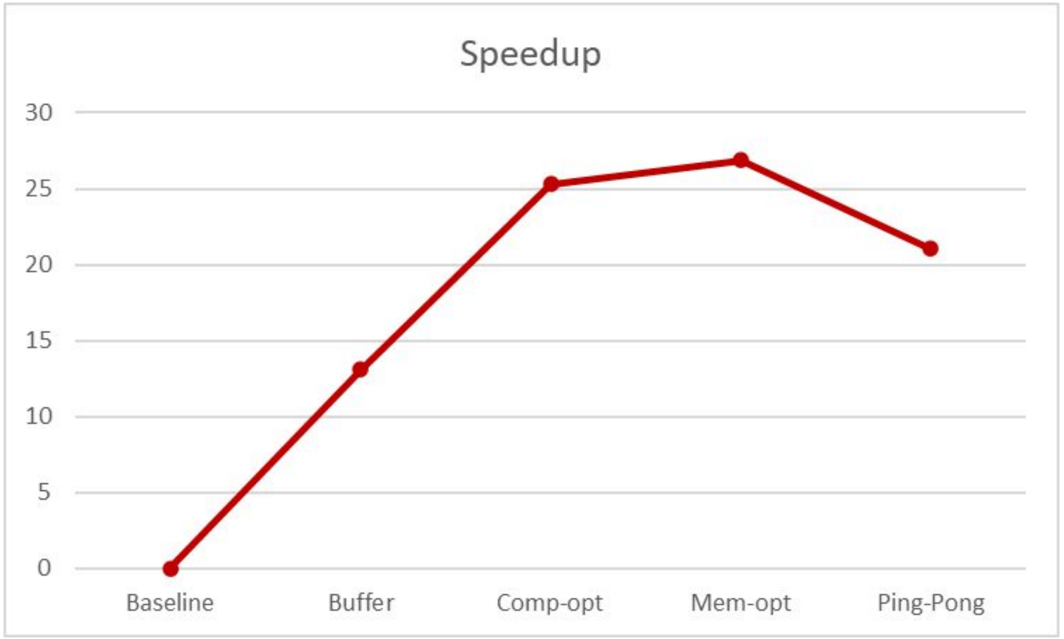


64x64x8

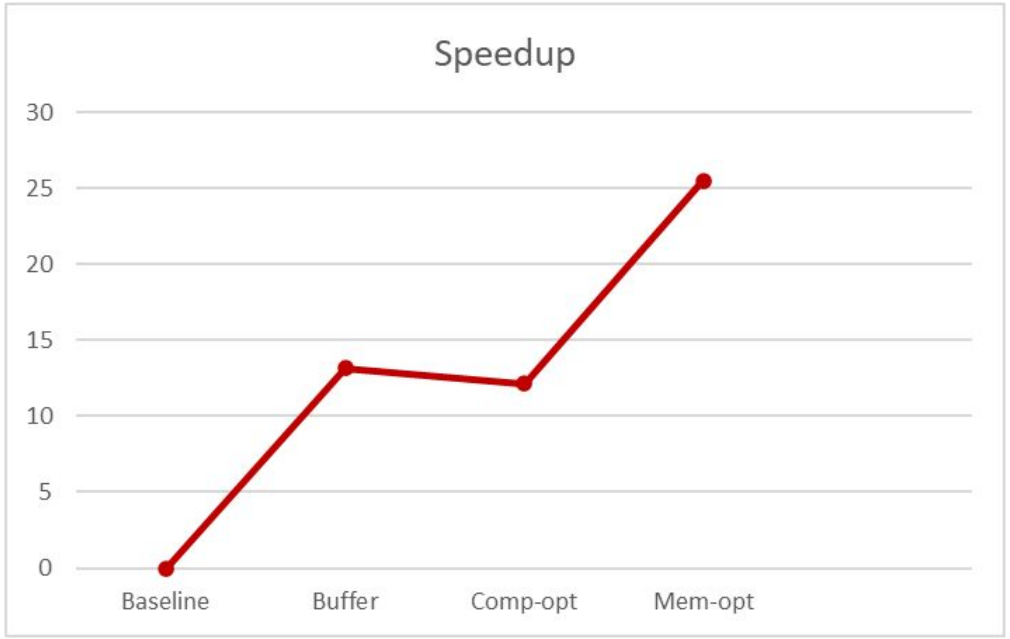


512x512x8

Comparisons-Speedup

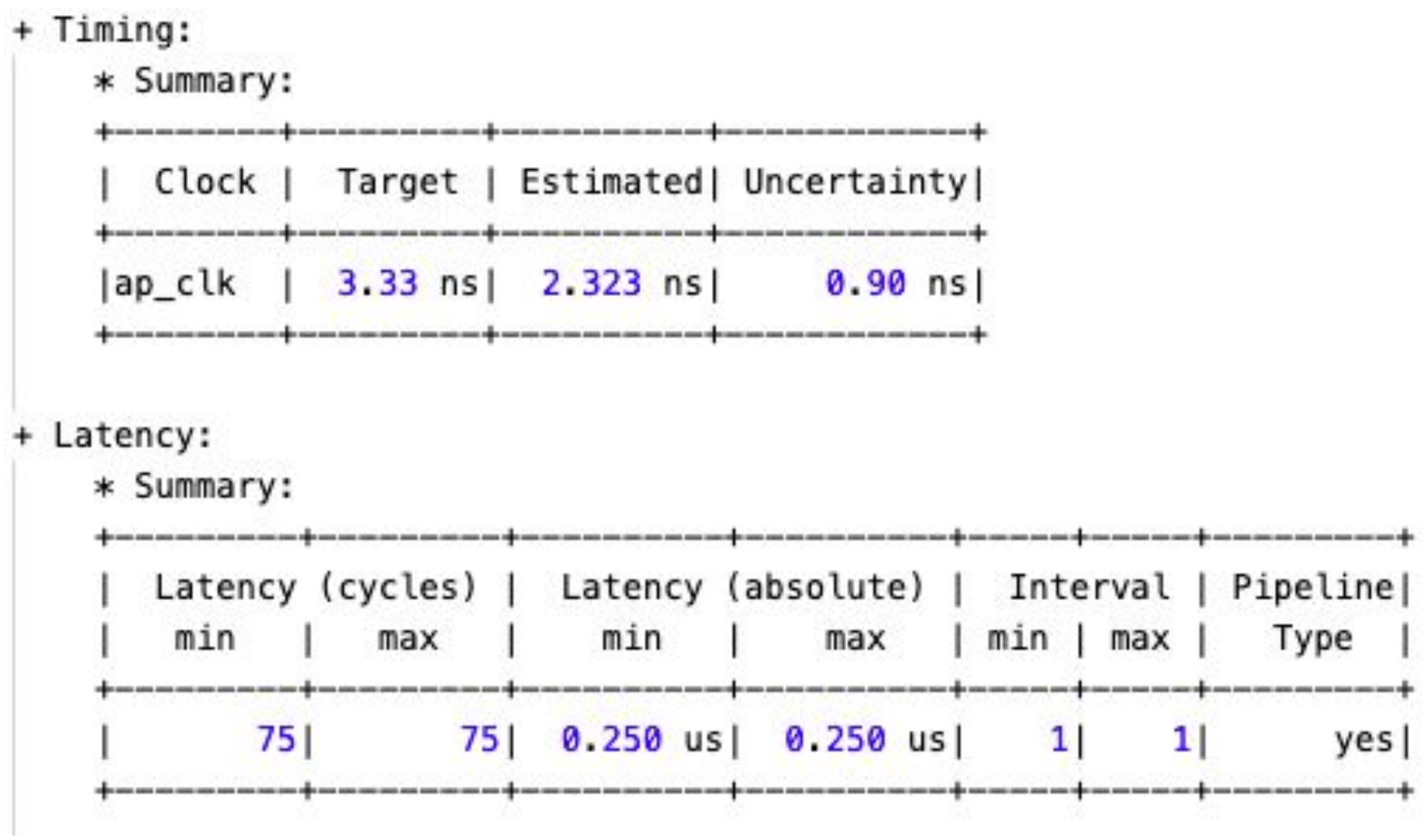


64x64x8



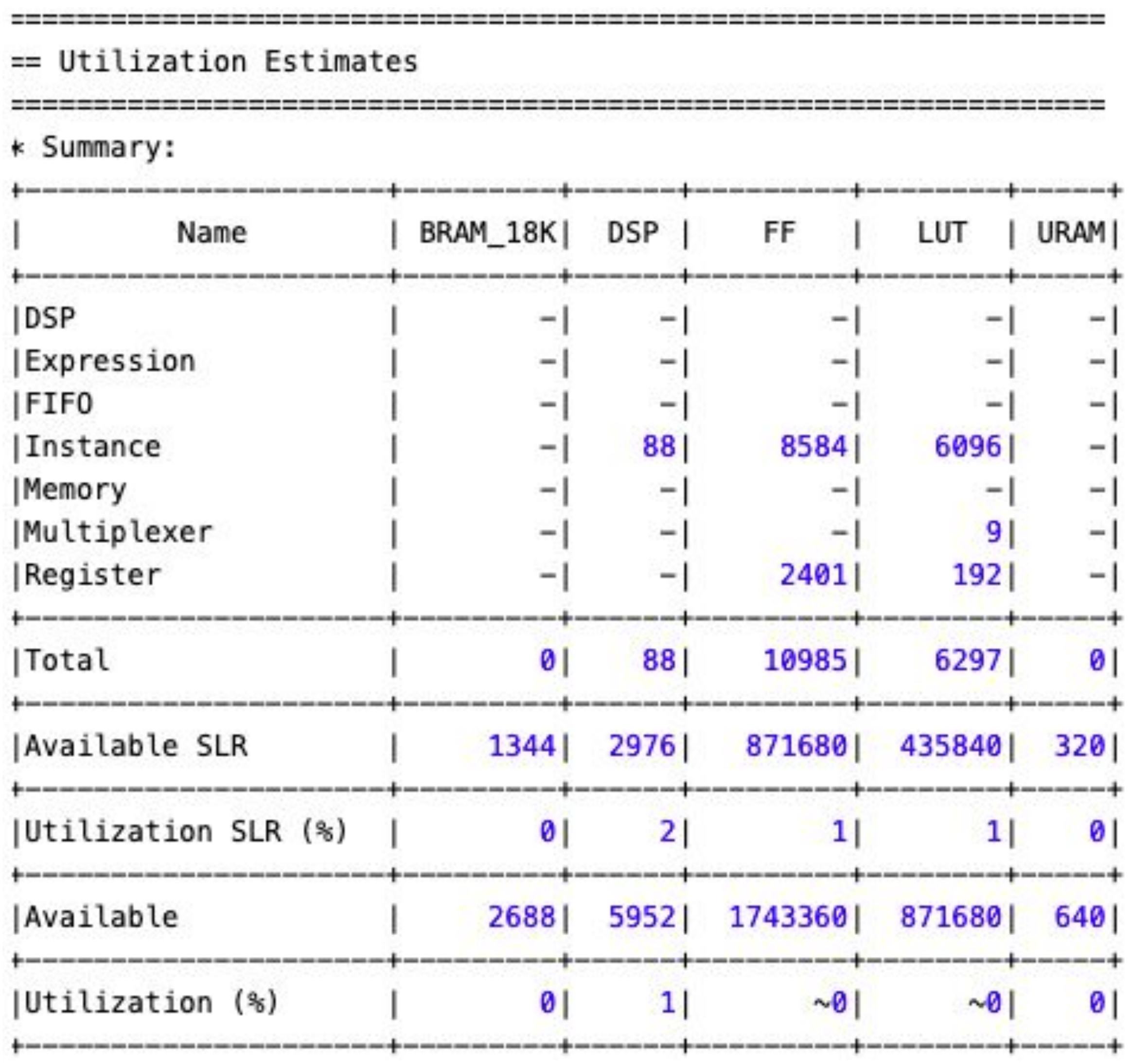
512x512x8

FPGA FINAL HOST CODE EXECUTION



```
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  3.33 ns|  2.323 ns|     0.90 ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +---------+---------+----------+----------+-----+-----+---------+
    |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
    |   min   |   max   |    min   |    max   | min | max |   Type  |
    +---------+---------+----------+----------+-----+-----+---------+
    |       75|       75|  0.250 us|  0.250 us|    1|    1|      yes|
    +---------+---------+----------+----------+-----+-----+---------+
```

FPGA FINAL HOST CODE EXECUTION



```
================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+-------+---------+--------+-----+
|         Name        | BRAM_18K|  DSP  |    FF   |   LUT  | URAM|
+---------------------+---------+-------+---------+--------+-----+
|DSP                  |        -|      -|        -|       -|    -|
|Expression           |        -|      -|        -|       -|    -|
|FIFO                 |        -|      -|        -|       -|    -|
|Instance             |        -|     88|     8584|    6096|    -|
|Memory               |        -|      -|        -|       -|    -|
|Multiplexer          |        -|      -|        -|       9|    -|
|Register             |        -|      -|     2401|     192|    -|
+---------------------+---------+-------+---------+--------+-----+
|Total                |        0|     88|    10985|    6297|    0|
+---------------------+---------+-------+---------+--------+-----+
|Available SLR        |     1344|   2976|   871680|  435840|  320|
+---------------------+---------+-------+---------+--------+-----+
|Utilization SLR (%)  |        0|      2|        1|       1|    0|
+---------------------+---------+-------+---------+--------+-----+
|Available            |     2688|   5952|  1743360|  871680|  640|
+---------------------+---------+-------+---------+--------+-----+
|Utilization (%)      |        0|      1|       ~0|      ~0|    0|
+---------------------+---------+-------+---------+--------+-----+
```

CUDA Analysis & Optimizations

ALGORITHM ANALYSIS

KERNEL EXECUTION USING (GRID_COLS*GRID_ROWS) THREADS

```
for (int i = 0; i < numiter; ++i) {
      hotspotOpt1<<<grid_dim, block_dim>>>
         (p_d, tIn_d, tOut_d, stepDivCap, nx, ny, nz, ce,
cw, cn, cs, ct, cb, cc);
      float *t = tIn_d;
      tIn_d = tOut_d;
      tOut_d = t;
   }
```

HOST

In CUDA, a group of threads is called a **block**.CUDA blocks are grouped into a **grid**. A kernel is executed as a grid of blocks of threads

For size 512*512*8:

Each block is 64*4*1

Each grid is nx / 64 * ny / 4 * 1

For 512*512*8, the size of the grid is 8 * 128 * 1

Total number of threads = (64*4*1) * (8 * 128 * 1) = 262144 = 512 * 512 (number of elements in each layer)

KERNEL

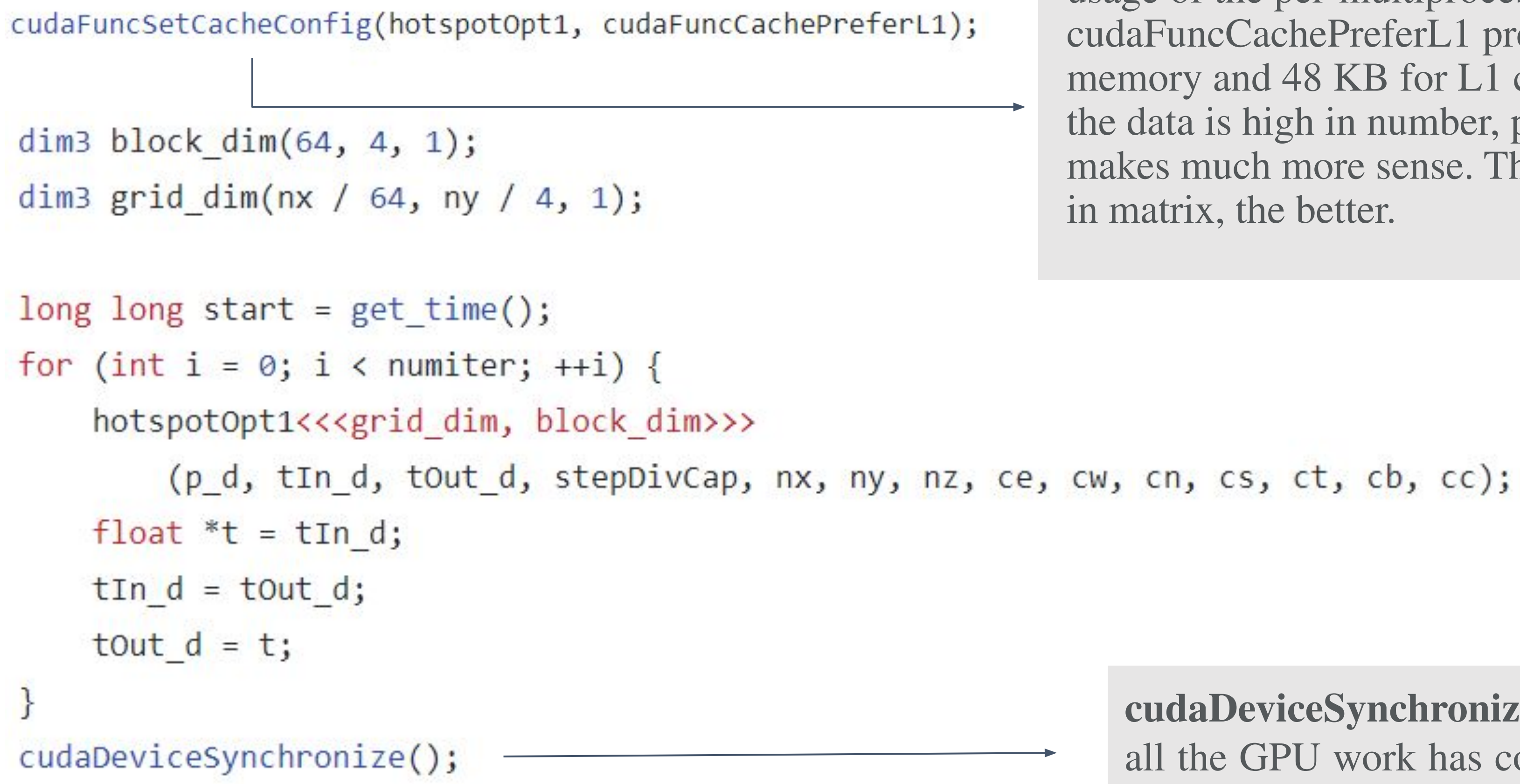
1. Obtain unique thread id using the below calculations:

   i = blockDim.x * blockIdx.x + threadIdx.x;

   j = blockDim.y * blockIdx.y + threadIdx.y;
2. The thread is mapped to one cell c as c = i + j * nx and the neighboring elements in same layer are obtained using c+1, c-1,c - nx, c + nx,;
3. Calculation of stencil for elements of 0th layer is done by obtaining top element using c+xy, where xy= nx*ny.
4. Calculation of stencil for elements from 1st layer to the second last layer is done by using three variables temp1, temp2, temp3 to indicate elements in bottom layer, center, and top layer respectively which are iteratively swapped. The neighboring elements are updated by a factor of xy.
5. Calculation of stencil for elements of last layer is done.

Dimension : nz * ny * nx

ALGORITHM ANALYSIS



```
cudaFuncSetCacheConfig(hotspotOpt1, cudaFuncCachePreferL1);

dim3 block_dim(64, 4, 1);
dim3 grid_dim(nx / 64, ny / 4, 1);

long long start = get_time();
for (int i = 0; i < numiter; ++i) {
    hotspotOpt1<<<grid_dim, block_dim>>>
        (p_d, tIn_d, tOut_d, stepDivCap, nx, ny, nz, ce, cw, cn, cs, ct, cb, cc);
    float *t = tIn_d;
    tIn_d = tOut_d;
    tOut_d = t;
}
cudaDeviceSynchronize();
```

cudaFuncSetCacheConfig: Sets the hotspotOpt1 kernel to prefer the usage of the per-multiprocessor memory for L1 cache.. cudaFuncCachePreferL1 prefers L1, that is, it sets 16 KB for shared memory and 48 KB for L1 cache. Since the calculation is repetitive and the data is high in number, preferring L1 cache over shared memory makes much more sense. The closer to each other the memory reads are in matrix, the better.

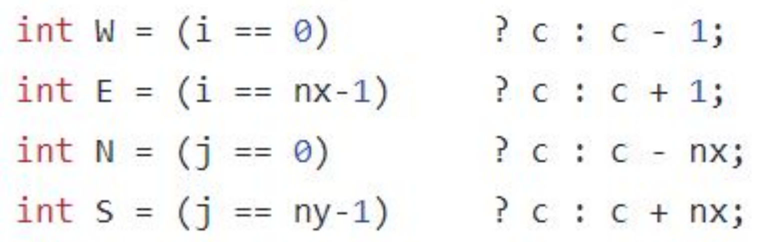
cudaDeviceSynchronize(): CPU is forced to idle until all the GPU work has completed before doing anything else.Blocks Host till all operations are finished.

ALGORITHM ANALYSIS

Point of divergence:

Hotspot 3D code has multiple points of divergence since it uses many conditional statements. Each border condition would be a point of divergence.

Block size = 64x4x1
Elements in one layer = 512x512



```
int W = (i == 0)        ? c : c - 1;
int E = (i == nx-1)     ? c : c + 1;
int N = (j == 0)        ? c : c - nx;
int S = (j == ny-1)     ? c : c + nx;
```

ALGORITHM ANALYSIS

Divergence Calculation

The following conditions are checked only once, for the first layer. (i refers to the column and j to the row)

```
int W = (i == 0)       ? c : c - 1;
int E = (i == nx-1)    ? c : c + 1;
int N = (j == 0)       ? c : c - nx;
int S = (j == ny-1)    ? c : c + nx;
```

Block size = 64*4 = 256; Number of warps = 8. Since array is linearised, this will cover 512/2 columns.

All threads in blocks 1 & 2 as well as the last 2 blocks will go through the same path.

Hence there will be no control divergence for the 4 blocks.

For all the other blocks -> there will be one thread each which is divergent, which will belong to 1 warp.

Control divergence over the 1020 remaining blocks ->

Total number of divergent warps / Total number of warps = 1020/1020*8 = 1/8

Control divergence % = 12.5

CUDA OPTIMIZATIONS

1. **Asynchronous multi - streaming :**

   The 3D grid can be divided into smaller grids and the kernel can be called multiple times. This can be done asynchronously to enable pipelining efficiently.

```
for (int i = 0; i < numiter; ++i) {
      // Code to partition tIn_d and tOut
      for(int k=0; k<num_of_tiles; k++
           hotspotOpt1<<<grid_dim, block_dim>>>
          (p_d, tIn_buff, tOut_buff, stepDivCap, nx, ny, nz, ce, cw, cn, cs, ct, cb, cc); -> Call kernel for each smaller grid
      // Code to copy partial output
       float *t = tIn_d;
       tIn_d = tOut_d;
       tOut_d = t;
    }
```

2. Using **shared memory in kernel** using the keyword __shared__
3. Reduce control divergence

Comparison of CPU, FPGA and CUDA

Comparison

CPU	FPGA	CUDA
Time: 0.014 s	Time: 0.25 us	Time: ~0 us
Clock frequency: 200 MHz	Clock frequency: 300 MHz	Clock frequency : 200 MHz
Maximum Resource utilization: ~0%	Maximum Resource utilization: 1%	Maximum Resource utilization:0%

Data size: 64 * 64 * 8

Q & A