CUDA
Game of Life parallelization

Failla Edoardo

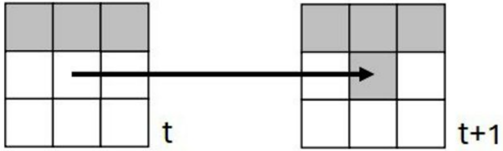
Introduction

Conway’s game of life game → 2D grid of cellular automata

$$\text{Cell state} \begin{cases} 1 \text{ - if cell is alive} \\ 0 \text{ - if cell is dead} \end{cases}$$

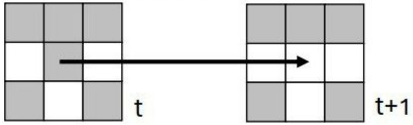
At each generation we apply cell transition rules:

Birth

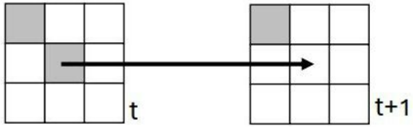


Death

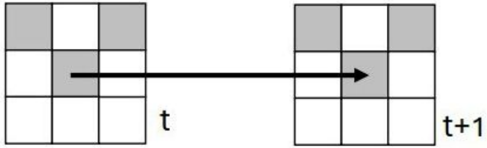
- Overcrowding:



- Exposure:



Survival



Introduction

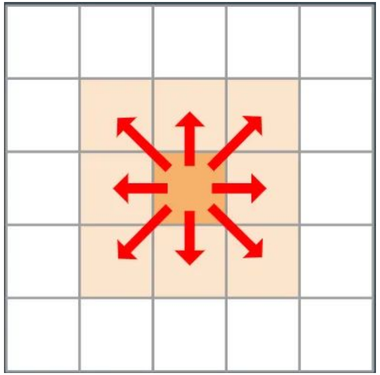
Complexity of Conway's Game of Life

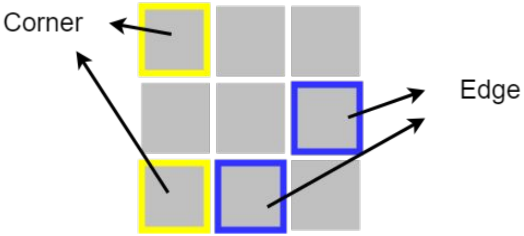
- **Time complexity:** $O(G \times N \times M)$, where G is the number of generations, *M* and *N* the size of the grid
- **Memory Complexity:** $O(N \times M)$, where N × M is the grid size.

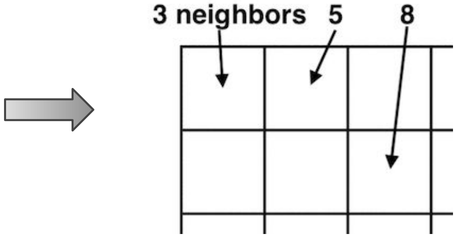
Challenges:

- **Incomplete Neighbors:**
  - ***Corners***: only 3 neighbors instead of 8.
  - ***Edges***: only 5 neighbors.

8 neighbors cell





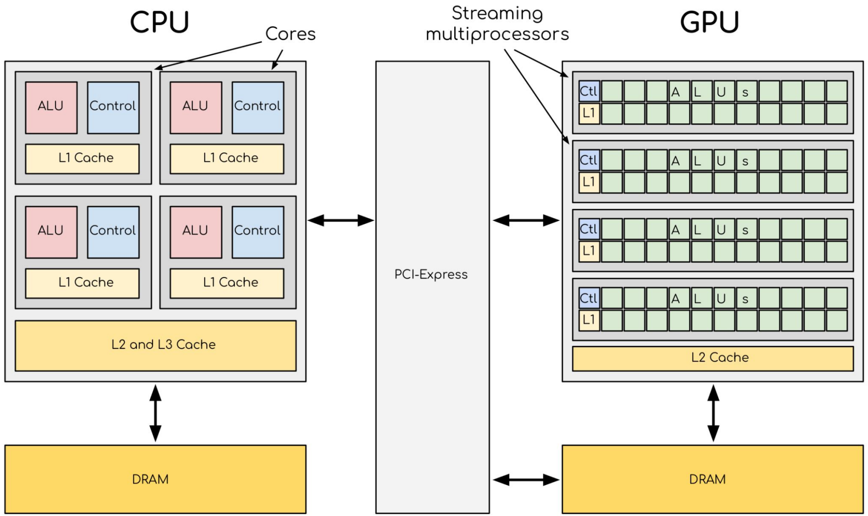


CUDA implementation

Using GPGPU computation

We can benefit from:

- *Natural parallelization architecture* (SIMD)
- *High Throughput*
- *Hardware Acceleration*
- *Scalability*



CUDA (Compute Unified Device Architecture):

- A parallel computing platform
- Developed by NVIDIA for GPU processing
- Supports languages like C, C++, and Fortran.

CUDA implementation

Thread indexing

Game grid 2D → 1D → ***Cell_index*** = row x N + column

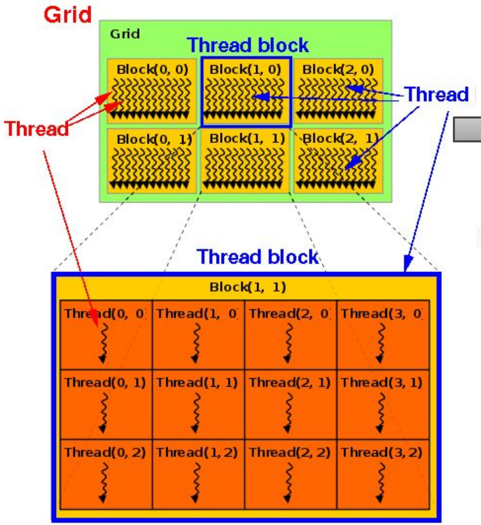
Global indexes with ***Block/Grid*** built-in variable

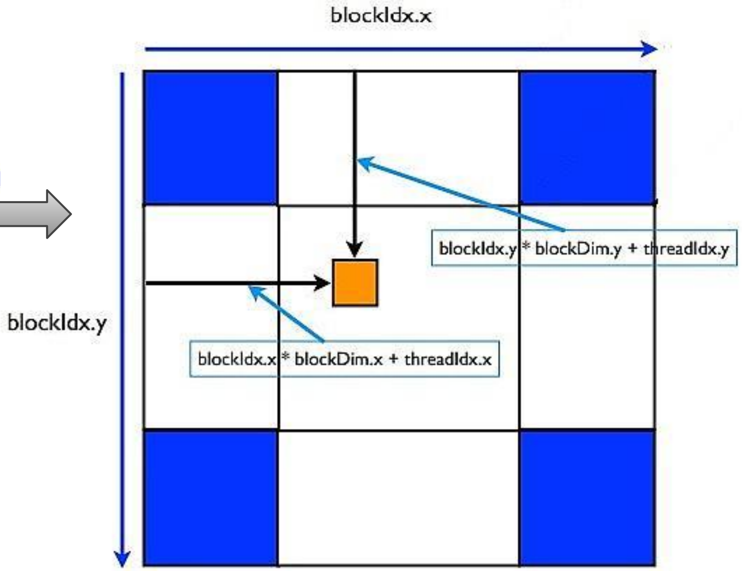
Device Tesla K40m:

- Max threads per block: 1024
- Max dimension (1024, 1024, 64)

We tested different dimensions of Block size

- (4, 4, 1)
- (8, 8, 1)
- (16, 16, 1)
- (32, 32, 1)



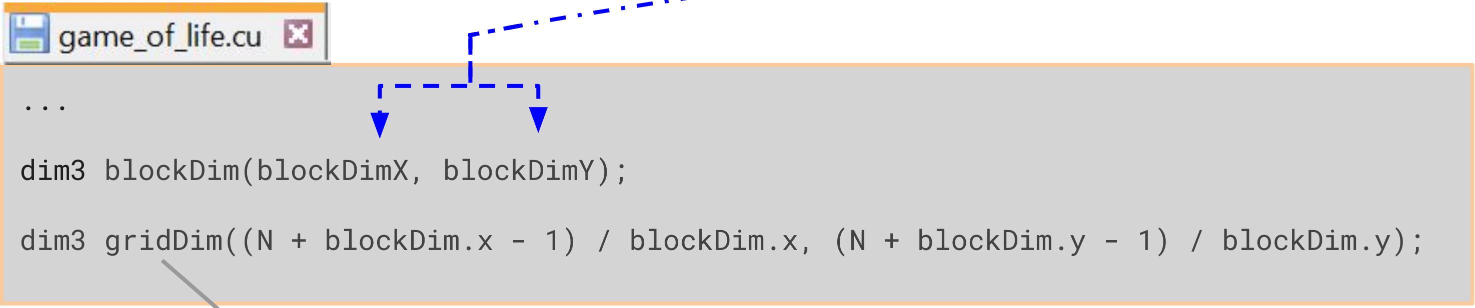

blockIdx.x
blockIdx.y
blockIdx.y * blockDim.y + threadIdx.y
blockIdx.x * blockDim.x + threadIdx.x

CUDA implementation

Input and parameters

```
./game_of_life <initial_state> <grid-size> <BlockDim> <Num-of-generations> --options
```

```
list options
--verbose “print result in a .txt file”
--check  “(for one version only) check the correctness of the result”
```



```
...

dim3 blockDim(blockDimX, blockDimY);

dim3 gridDim((N + blockDim.x - 1) / blockDim.x, (N + blockDim.y - 1) / blockDim.y);
```

CUDA grid dimensions calculated

CUDA implementation

First idea:

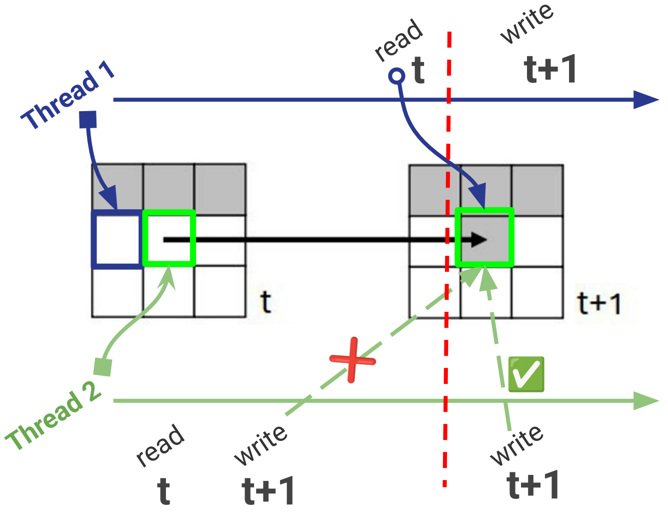
- Only one kernel
- *for* generation cycle inside kernel
- One function call

→

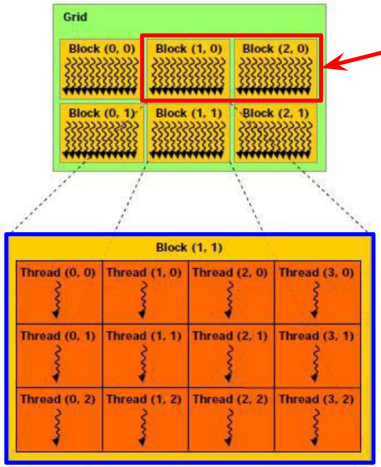
Problems:

- Threads synchronization
- Race condition

Race condition



Synchronization has many limitations



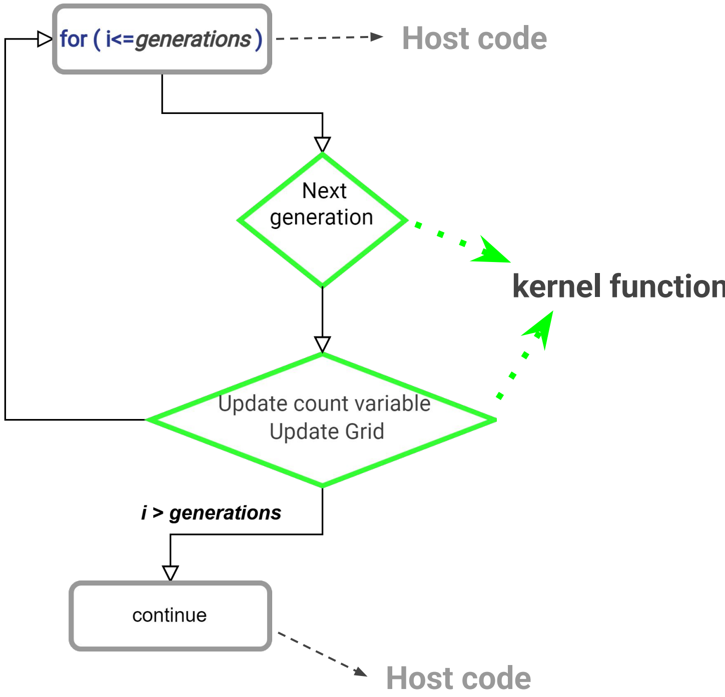
?

Cooperative Groups API:
Nº of blocks limitation

`__synchtreads()`

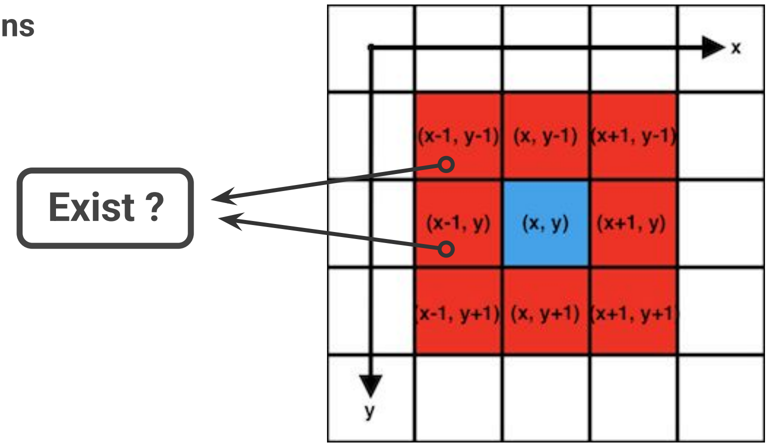
CUDA implementation

First implementation



Calculate next generation:

- Check if neighbors' indexes are in range
- Count alive neighbors
- Apply rules

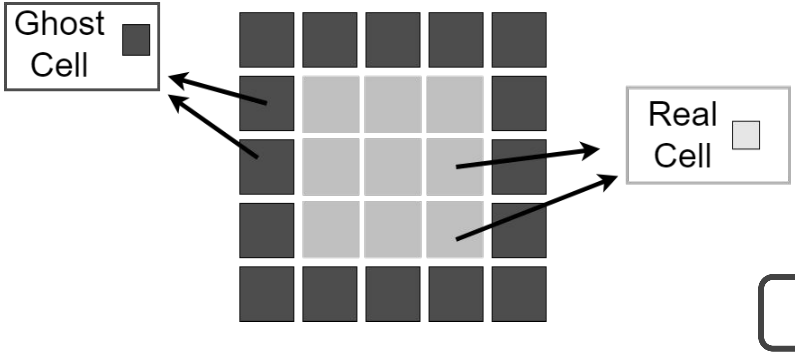


CUDA implementation

Second implementation

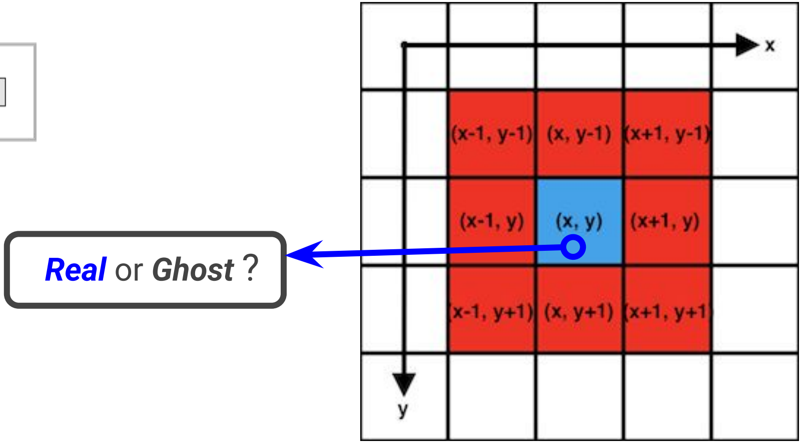
Add Ghost cells to perimeter:

- ***Ghost*** cells are always **dead**
- Do not influence the rules



Calculate next generation:

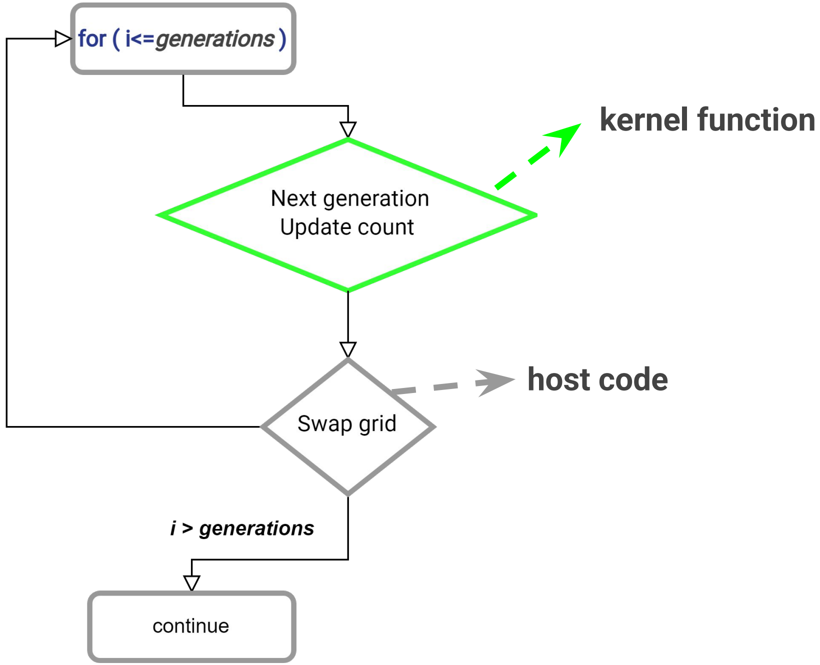
- Check if cells are ***Real*** or ***Ghost***
- Count alive neighbors
- Apply rules



CUDA implementation

Some improvements

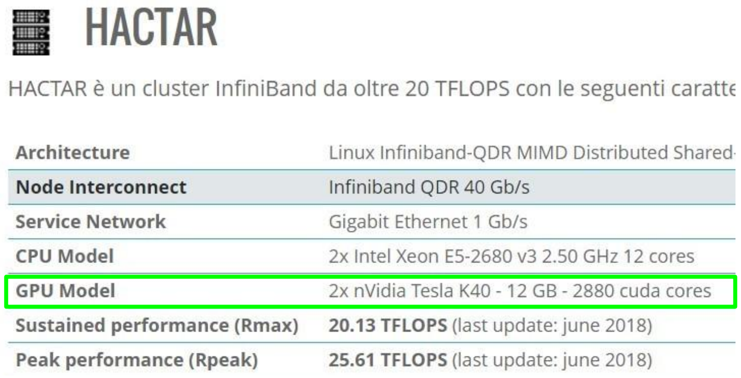
We can modify the code for optimizing grid swapping



- ➢ Only one kernel
- ➢ Reduced function call
- ➢ Swap pointer operation

CUDA implementation

Can we use multiple GPUs ?



HACTAR

HACTAR è un cluster InfiniBand da oltre 20 TFLOPS con le seguenti caratte

Architecture	Linux Infiniband-QDR MIMD Distributed Shared-
Node Interconnect	Infiniband QDR 40 Gb/s
Service Network	Gigabit Ethernet 1 Gb/s
CPU Model	2x Intel Xeon E5-2680 v3 2.50 GHz 12 cores
GPU Model	2x nVidia Tesla K40 - 12 GB - 2880 cuda cores
Sustained performance (Rmax)	**20.13 TFLOPS** (last update: june 2018)
Peak performance (Rpeak)	**25.61 TFLOPS** (last update: june 2018)

We can use 2 GPUs for one node

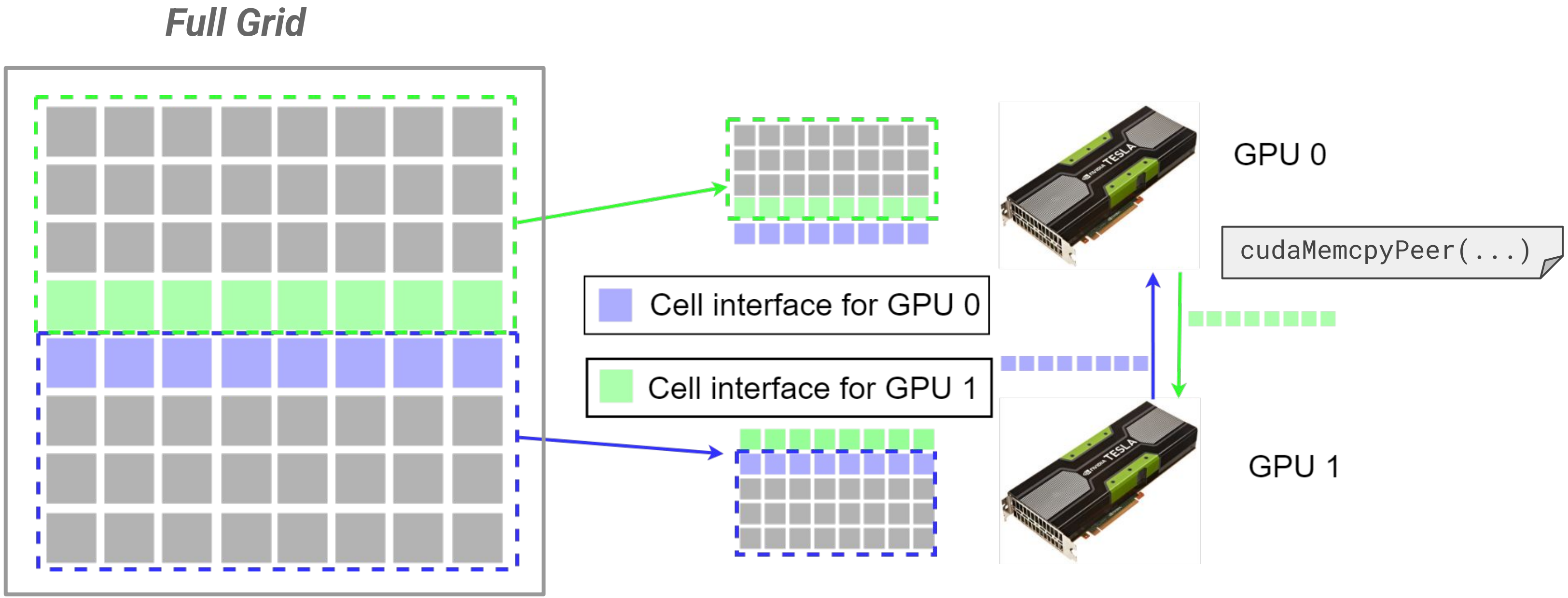
We need to:

1. Split the grid
2. Exchange border information
3. Reassemble the grid

How can we do such task?

CUDA implementation

Two-GPUs



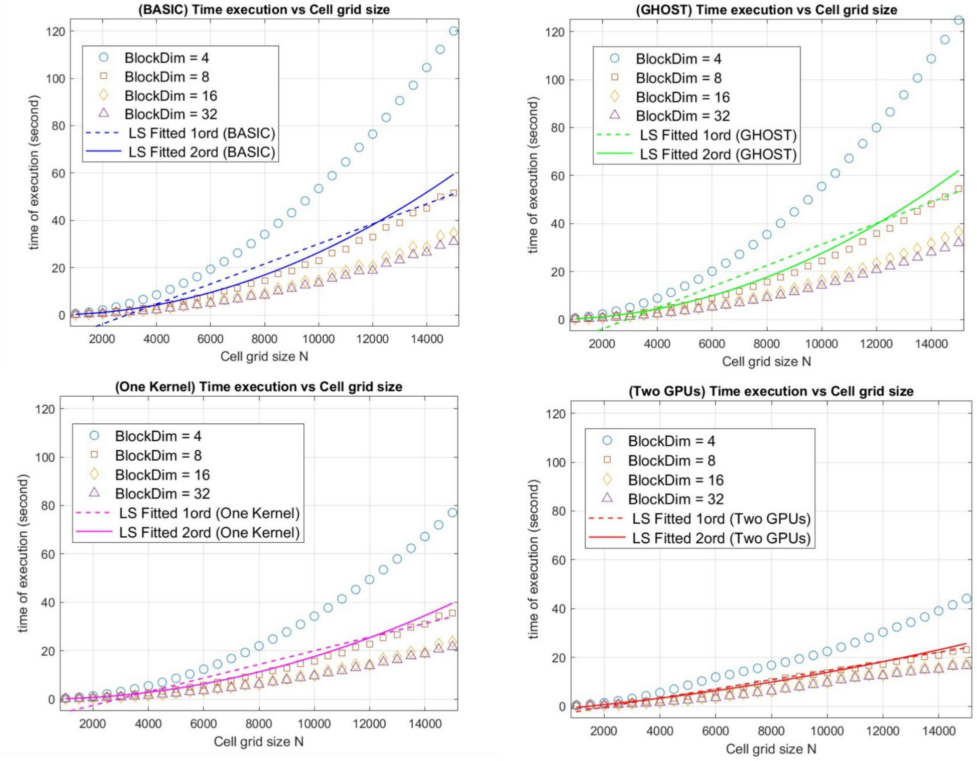
🟦 Cell interface for GPU 0

🟩 Cell interface for GPU 1

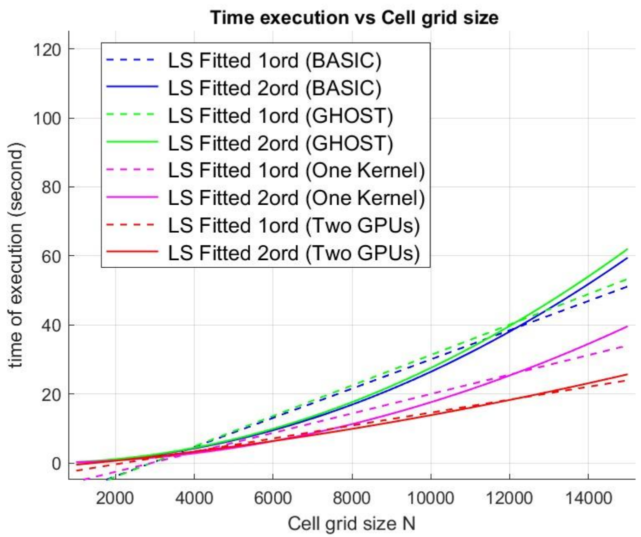
`cudaMemcpyPeer(...)`

CUDA implementation

Time of execution



Results Summary



CUDA implementation

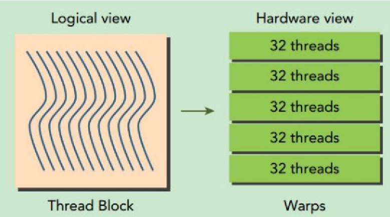
CUDA occupancy

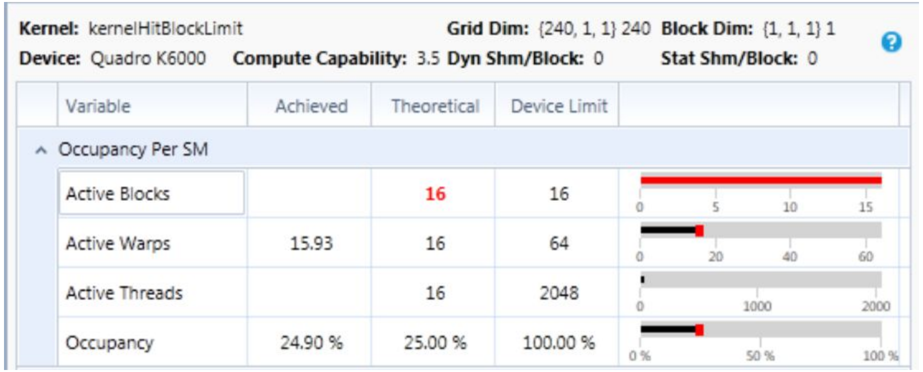
“*Occupancy* is defined as the ratio of active warps on a SM to the maximum number of active warps supported by the SM.”

“The SM has a maximum number of ***warps*** that can be active at once.”

“The SM has a maximum number of ***blocks*** that can be active at once.”

“The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called ***warps***”





Kernel: kernelHitBlockLimit **Grid Dim:** {240, 1, 1} 240 **Block Dim:** {1, 1, 1} 1
Device: Quadro K6000 **Compute Capability:** 3.5 **Dyn Shm/Block:** 0 **Stat Shm/Block:** 0

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		16	16
Active Warps	15.93	16	64
Active Threads		16	2048
Occupancy	24.90 %	25.00 %	100.00 %

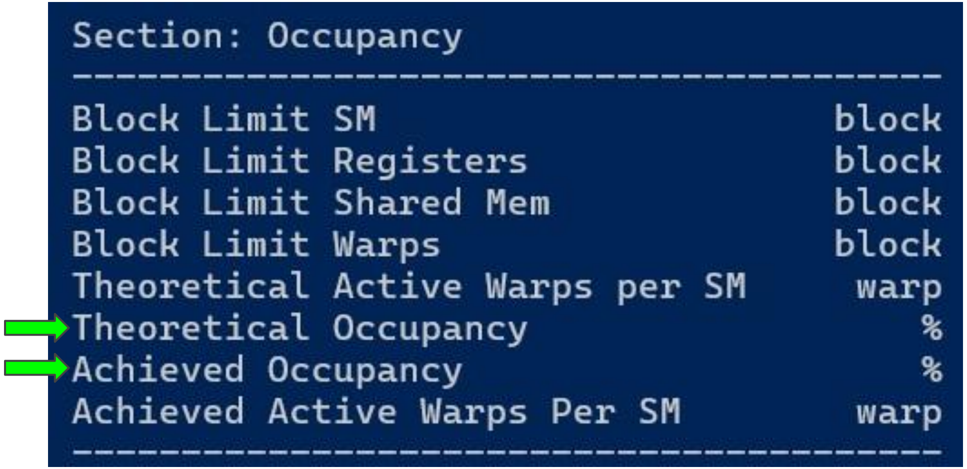
(**NVIDIA** *doc. example*)

CUDA implementation

CUDA occupancy

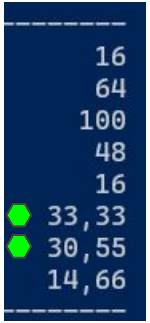
Using the GPU on my laptop

```
ncu --set full ./Okern_game_of_life <...>
```



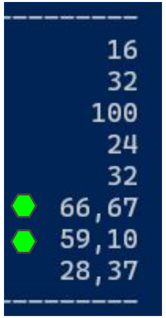
```
  Section: Occupancy
  ----------------------------------------
  Block Limit SM                     block
  Block Limit Registers              block
  Block Limit Shared Mem             block
  Block Limit Warps                  block
  Theoretical Active Warps per SM     warp
→ Theoretical Occupancy                  %
→ Achieved Occupancy                     %
  Achieved Active Warps Per SM        warp
  ----------------------------------------
```

BlockDim(4,4)



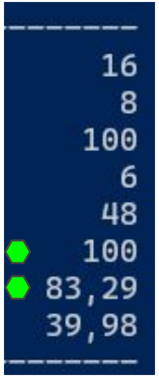
```
--------
      16
      64
     100
      48
      16
⬢  33,33
⬢  30,55
   14,66
--------
```

BlockDim(8,8)



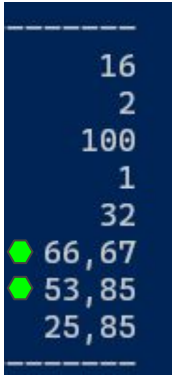
```
--------
      16
      32
     100
      24
      32
⬢  66,67
⬢  59,10
   28,37
--------
```

BlockDim(16,16)



```
--------
      16
       8
     100
       6
      48
⬢    100
⬢  83,29
   39,98
--------
```

BlockDim(32,32)



```
--------
      16
       2
     100
       1
      32
⬢  66,67
⬢  53,85
   25,85
--------
```

Thank you for the attention!