report

September 19, 2023

Asynchronous and concurrent execution on GPUs - Room for improvement in High Performance Computing for Weather and Climate Models?

Melina Abeling, Julian Aeissen and Michele Pagani High Performance Computing for Weather and Climate Supervisor Oliver Fuhrer Summer term 2023 melina.abeling@students.unibe.ch, julian.aeissen@students.unibe.ch, paganimi@student.ethz.ch

---

0.1 Introduction

In order to better represent earth´s complex climate the resolution of climate models is becoming continuously higher which improves the skill of climate and weather models forecasts. In numerical weather predictions the grid consisting of cells or points, for example an icosahedron (ICON-model) sphere, is projected on the globe to discretize and solve the governing, often differential equations, for the whole grid size (Zängel et al., 2015, Wahib 2014). Associated with the higher spatial resolution is a need for higher computing power, as it means a finer grid consisting of more cells and points for which calculations have to be done in greater detail. Here, high performance computing and especially Graphical Process Units (GPU´s), hardware that accellerates computing also due to the ability to process multiple tasks in parallel, are important to meet the demand for better computing performance. One method commonly used to solve the necessary partial differential equations in climate models are stencil motivs (Schäfer & Fey, 2011; Wahib 2014). Those algorithms are space and time discrete that uniformly compute the value of a specific grid point based on its own value and those of its neigbouring grid points (Schäfer & Fey, 2011). This project aims to investigate how the maxmimum efficiency for various sized tasks, here the mentioned stencil motifs, differ for asynchronous and concurrent execution on GPUs or if it is even possible to reach a total utilization of the available hardware. For this a simple 2D Jacobian stencil and a Gaussian stencil will be utilized to investigate in which scenarios which execution leads to maximum efficiency or if maximum efficiency could not be reached.

0.2 Methods

To investigate whether, or better, when asynchronous or concurrent execution on GPUs is most preferable, different scenarios where applied (Figure 1). The starting point for this is an initially unperturbed field (a square) or grid of fields to better be able to observe changes and correctness of computation (Figure 2). Figure 1: Project draft for asyncronous vs. concurrent execution on GPU´s. The y-axis represents the elapsed time after submitting tasks of various size. Therefore two stencils were chosen in order to see the influence they have on the preformance of GPU execution.

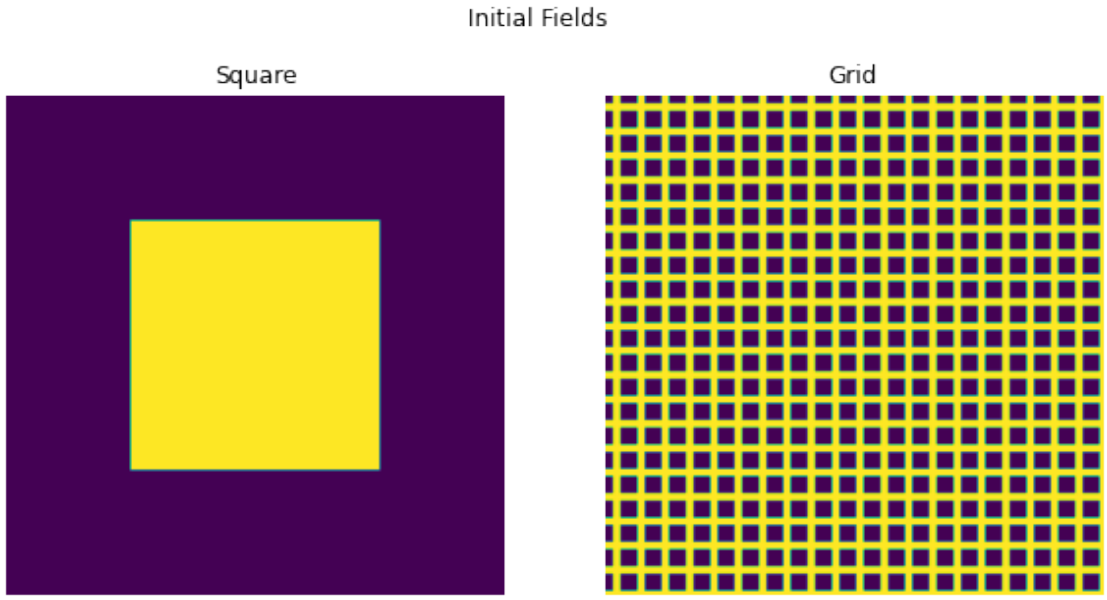

Figure 2: Initial fields, whithout any blurr or perturbations.

2D Jacobi stencil The Jacobi Stencil calculates a weighted average of nearest neighbours and the center grid points and is therefore a five point stencil (Figure 3). In one time step two multiplication and 4 additions are performed. With double precision numbers at least 8 bytes have to be read or written (single-digit), yielding an Arithmetic Intensity of <0.5 FLOP/Byte. Figure 3: The illustration on the left shows a Jacobian Stencil. The figures on the right display the effect of 1000 iterations (blurr) on the field and over the whole grid.

Gaussian 5x5 stencil The gaussian 5x5 stencil on the other hand is much larger, being a 25 point stencil (Figure 4). It is a discrete approximation of the 2D Gaussian filter/blurr. In one time step/grid update 24 FLOP are performed per grid point (addition or multiplication or addition followed by multiplication). Again at least one new number is written or read (with double precision) yielding an intensity of < 3 FLOP/Byte which can be considerably larger than for the smaller Jacobi stencil. Figure 4: The left image shows an illustration of the Gaussian Stencil (5x5) and the right images portray the effect 1000 iterations (blurr) have on the field and the whole grid.

GPU parallelization To investigate the impact on performance of different levels of concurrency on the GPU two approaches were applied. First, the difference in performance with different levels of parallelization was compared, i.e. the total field was divided into a varying number of tiles. Each tile is assigned a different stream on the GPU. In the second approach the field is also divided into tiles, but the execution of the tiles is now done sequentially via a for loop which represents different tasks for the CPU. Figure 5: Sketch of the division of the fields into tiles, in parallel execution each tile is executed in one seperate stream by the GPU.

0.3 Results and Discussion

0.3.1 GPU versus Streams

Performance over concurrency The streams come with overhead so dividing the field in tiles and computing each tile in a different stream leads, as expected, to a worse performance than

having the whole field in a single stream.

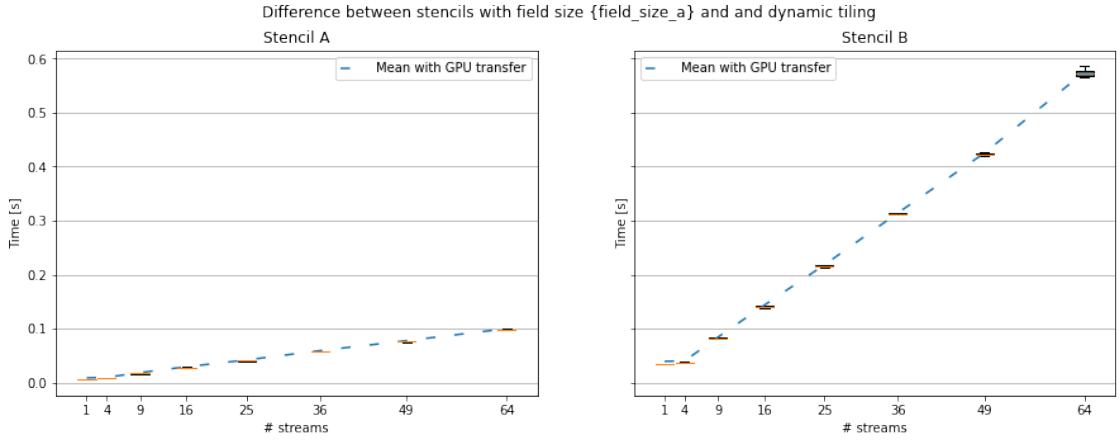

Figure 6: Execution time for parallel execution of one stream per tile. For the 5-point stencil (left) and the 25-point stencil (right).

Performance over grid size Figures 7 and 8 show the execution time of 10 iterations with the Jacobi Stencil and the Gaussian stencil for different total field sizes. Since the curves flatten towards small grid sizes (up to field sizes of 1024x1024 the time stays almost constant over time), we can readily read off the overhead of the programm. The overhead becomes - unsurprisingly - bigger with more streams. The actual execution time of the stencil calculations only becomes relevant, depending on the number of streams, for grid sizes bigger than (1024x1024).

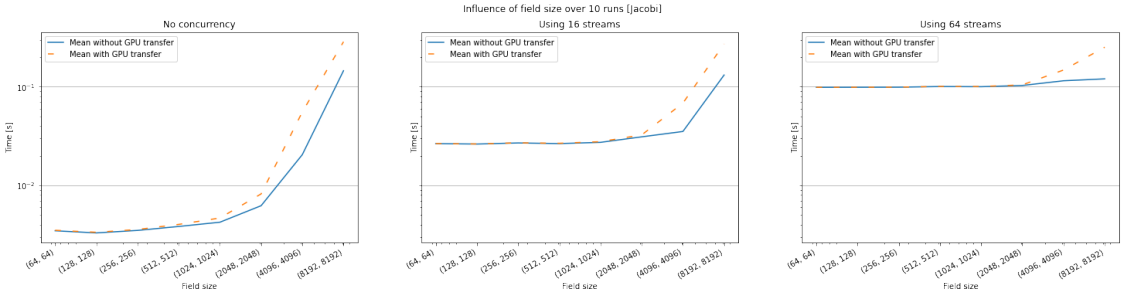

Figure 7: Execution time vs different field sizes with the 5-point stencil for varying number of streams.

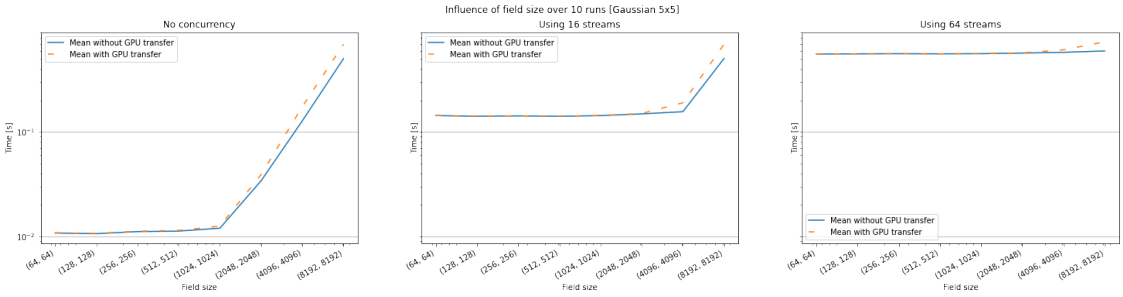

Figure 8: Execution time vs different field sizes with the 25-point stencil for varying number of streams.

0.3.2 Fixed tiling

Varying streams number Figure 9 shows the execution time averaged over 10 runs of the execution time, varying the number of streams. The tile size is kept constant at (256,256). As we

can see from the two plots, as the number of streams increase, the performance slightly worsen. This is very probably because of the overhead of each stream, which is in line with the results of the previous section.

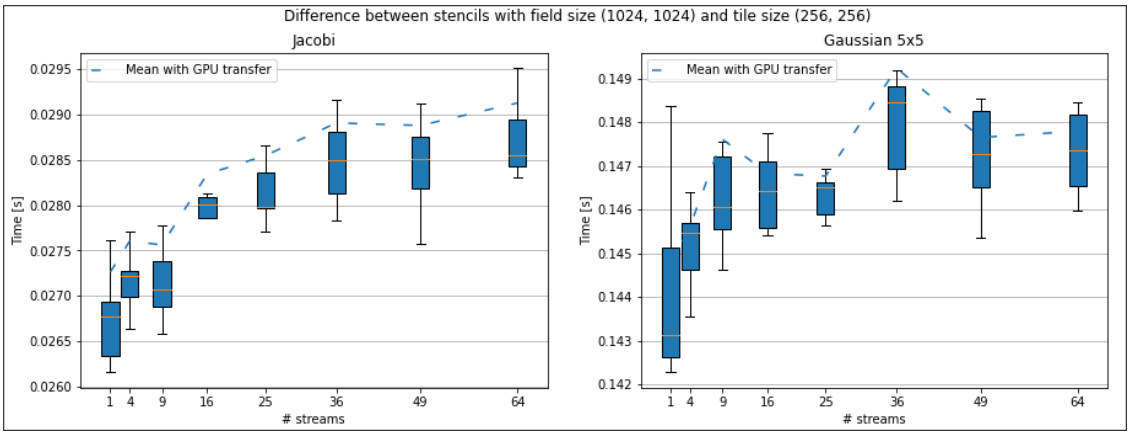

Figure 9: Execution time for parallel execution for fixed tile size (256,256) and varying number of streams. For the 5-point stencil (left) and the 25-point stencil (right).

Varying tile size The size of the tiles dividing the field has virtual no influence, as it can be seen in Figures 10 and 11. Independently of the number of streams or the stencil, performance increases exponentially for increased tile size. This means the overhead for tiling and sending packets to the GPU is predominant here.

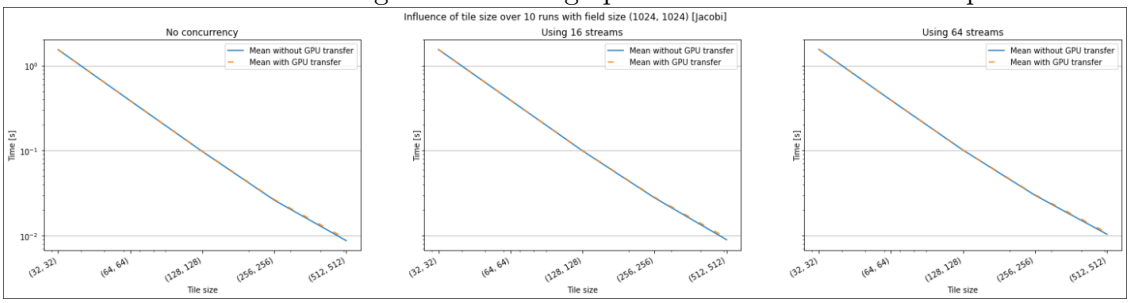

Figure 10: Execution time vs different number of streams with the 5-point stencil for varying tile sizes.

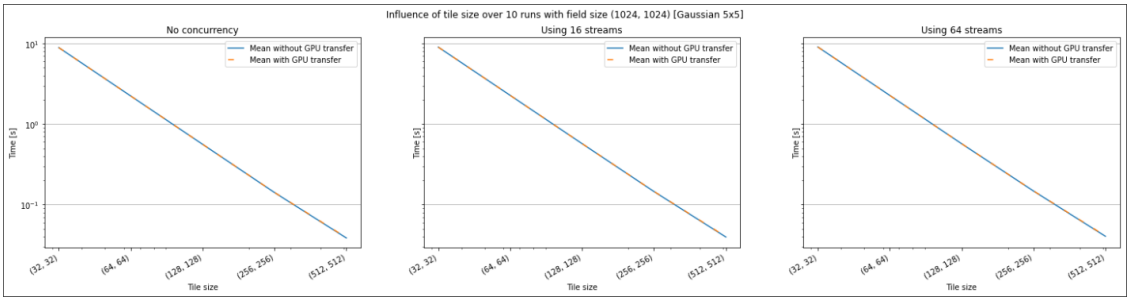

Figure 11: Execution time vs different number of streams with the 25-point stencil for varying tile sizes.

0.4 Conclusion

This work shows the performance for various-sized tasks and stencil motifs, using different approaches to parallelization: at GPU level (single call to GPU, single stream), at thread level

(multiple calls to GPU, single stream), and at stream level. This was done both varying the overall problem size (first result section) and fixing it, but varying the size per GPU thread (second result section). In general, the best results were obtained by letting the GPU handle the parallelization, with as little external inputs such has forcing streams or tiles.

Indeed, the execution leaves room for improvement in HPC in Weather and Climate Modeling but it these very complex structures a perfect usage of hardware might not be possible. Further investigations on how these differences in efficiency can be utilized for different purposes in high performance computing for weather and climate modelling and It will be interesting to be able to witness how this might change with new hardware even higher computational power and the ever evolving code of Earth System Models.

0.5 References

Alizadeh, O. (2022). Advances and challenges in climate modeling. Climatic Change, 170(1), 18. https://doi.org/10.1007/s10584-021-03298-4 Schäfer, A., & Fey, D. (2011). High performance stencil code algorithms for GPGPUs [Proceedings of the International Conference on Computational Science, ICCS 2011]. Procedia Computer Science, 4, 2027--2036. https://doi.org/https://doi.org/10.1016/j.procs.2011.04.221 Wahib, M., & Maruyama, N. (2014). Scalable kernel fusion for memory-bound GPU applications. SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 191--202. https://doi.org/10.1109/SC.2014.21 Zängl, G., Reinert, D.,Rípodas, P., & Baldauf, M.(2015).The ICON(ICOsahedral Non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core. Quarterly Journal of the Royal Meteorological Society, 141(687), 563--579. https://doi.org/10.1002/qj.2378

0.6 Appendix

0.6.1 Code snippets

For more information, please refer to ***computation.ipypnb*** *and* ***utils.py*** #### Stencils

[ ]:
```python
# Jacobian stencil
def jacobi_stencil_2d(in_field, out_field, n_halo, alpha=0.5, beta=0.125):
    # Checks
    assert len(in_field.shape) == 2
    assert len(out_field.shape) == 2
    h,w = out_field.shape
    h_in_,w_in_ = in_field.shape
    assert h_in_ == h + 2*n_halo
    assert w_in_ == w + 2*n_halo
    # IMPORTANT always have an expected halo
    assert n_halo == 1

    # Computation
    out_field[:,:] = (
        alpha * in_field[1:-1, 1:-1]
        + beta * ( in_field[2:, 1:-1]+  in_field[:-2, 1:-1]
        +  in_field[1:-1, 2:]+  in_field[1:-1, :-2] )
```

```
    )

# Simple 5x5 Gaussian filter
def gaussian_5x5_stencil_2d(in_field, out_field, n_halo):
    # Checks
    assert len(in_field.shape) == 2
    assert len(out_field.shape) == 2
    h,w = out_field.shape
    h_in_,w_in_ = in_field.shape
    assert h_in_ == h + 2*n_halo
    assert w_in_ == w + 2*n_halo

    # IMPORTANT always have an expected halo
    assert n_halo == 2

    # Computation
    out_field[:,:] = (
        (
            in_field[0:-4, 0:-4]
            + 4.0 * in_field[0:-4, 1: -3]
            + 6.0 * in_field[0:-4, 2: -2]
            + 4.0 * in_field[0:-4, 3: -1]
            + in_field[0:-4, 4: ]
        )
        + 4.0 * (
            in_field[1:-3, 0:-4]
            + 4.0 * in_field[1:-3, 1: -3]
            + 6.0 * in_field[1:-3, 2: -2]
            + 4.0 * in_field[1:-3, 3: -1]
            + in_field[1:-3, 4: ]
        )
        + 6.0 * (
            in_field[2:-2, 0:-4]
            + 4.0 * in_field[2:-2, 1: -3]
            + 6.0 * in_field[2:-2, 2: -2]
            + 4.0 * in_field[2:-2, 3: -1]
            + in_field[2:-2, 4: ]
        )
        + 4.0 * (
            in_field[3:-1, 0:-4]
            + 4.0 * in_field[3:-1, 1: -3]
            + 6.0 * in_field[3:-1, 2: -2]
            + 4.0 * in_field[3:-1, 3: -1]
            + in_field[3:-1, 4: ]
        )
        + (
            in_field[4:, 0:-4]
```

```
            + 4.0 * in_field[4:, 1: -3]
            + 6.0 * in_field[4:, 2: -2]
            + 4.0 * in_field[4:, 3: -1]
            + in_field[4:, 4: ]
        )
    ) / 256.0
```

[ ]:
```
# Jacobian stencil
def jacobi_stencil_2d(in_field, out_field, n_halo, alpha=0.5, beta=0.125):
    # Checks
    assert len(in_field.shape) == 2
    assert len(out_field.shape) == 2
    h,w = out_field.shape
    h_in_,w_in_ = in_field.shape
    assert h_in_ == h + 2*n_halo
    assert w_in_ == w + 2*n_halo
    # IMPORTANT always have an expected halo
    assert n_halo == 1

    # Computation
    out_field[:,:] = (
        alpha * in_field[1:-1, 1:-1]
        + beta * ( in_field[2:, 1:-1]+  in_field[:-2, 1:-1]
        +  in_field[1:-1, 2:]+  in_field[1:-1, :-2] )
    )

# Simple 5x5 Gaussian filter
def gaussian_5x5_stencil_2d(in_field, out_field, n_halo):
    # Checks
    assert len(in_field.shape) == 2
    assert len(out_field.shape) == 2
    h,w = out_field.shape
    h_in_,w_in_ = in_field.shape
    assert h_in_ == h + 2*n_halo
    assert w_in_ == w + 2*n_halo

    # IMPORTANT always have an expected halo
    assert n_halo == 2

    # Computation
    out_field[:,:] = (
        (
            in_field[0:-4, 0:-4]
            + 4.0 * in_field[0:-4, 1: -3]
            + 6.0 * in_field[0:-4, 2: -2]
            + 4.0 * in_field[0:-4, 3: -1]
            + in_field[0:-4, 4: ]
```

```
        )
        + 4.0 * (
            in_field[1:-3, 0:-4]
            + 4.0 * in_field[1:-3, 1: -3]
            + 6.0 * in_field[1:-3, 2: -2]
            + 4.0 * in_field[1:-3, 3: -1]
            + in_field[1:-3, 4: ]
        )
        + 6.0 * (
            in_field[2:-2, 0:-4]
            + 4.0 * in_field[2:-2, 1: -3]
            + 6.0 * in_field[2:-2, 2: -2]
            + 4.0 * in_field[2:-2, 3: -1]
            + in_field[2:-2, 4: ]
        )
        + 4.0 * (
            in_field[3:-1, 0:-4]
            + 4.0 * in_field[3:-1, 1: -3]
            + 6.0 * in_field[3:-1, 2: -2]
            + 4.0 * in_field[3:-1, 3: -1]
            + in_field[3:-1, 4: ]
        )
        + (
            in_field[4:, 0:-4]
            + 4.0 * in_field[4:, 1: -3]
            + 6.0 * in_field[4:, 2: -2]
            + 4.0 * in_field[4:, 3: -1]
            + in_field[4:, 4: ]
        )
    ) / 256.0
```

Initial fields

[ ]:
```
# Creates an empty initial field with a square of size / 2 at the center of␣
 ↪value `value`
def get_initial_field_square(size, n_halo, value = 1.0) -> cp.ndarray:
    # Check parameters
    assert type(size) == tuple
    dim = len(size)
    # assert dim in [2,3]
    assert dim in [2]

    # Init
    h, w = size[-2], size[-1]

    # Add halo
    h += 2*n_halo
    w += 2*n_halo
```

```
    # 2d
    if dim == 2:
        field = cp.zeros((h, w), dtype=cp.float32)
        field[ h//4 : 3*h//4,
               w//4 : 3*w//4 ] = value

    # 3d
    elif dim == 3:
        field = cp.zeros((size[0], h, w), dtype=cp.float32)
        field[ :,
               h//4 : 3*h//4,
               w//4 : 3*w//4 ] = value

    return field

# Creates an grid of spacing `spacing`
def get_initial_field_grid(size, n_halo, value = 1.0, spacing=50) -> cp.ndarray:
    # Check parameters
    assert type(size) == tuple
    dim = len(size)
    # assert dim in [2,3]
    assert dim in [2]

    # Init
    h, w = size[-2], size[-1]

    # Add halo
    h += 2*n_halo
    w += 2*n_halo

    # 2d
    if dim == 2:
        field = cp.zeros((h, w), dtype=cp.float32)

        # Horizontal strides
        for i in range(h // spacing + 2):
            if i * spacing >= h:
                break
            idx = spacing // 3 + i * spacing
            field[ idx : idx + spacing // 3, :] = value

        # Vertical strides
        for i in range(w // spacing + 2):
            if i * spacing >= w:
                break
            idx = spacing // 3 + i * spacing
```

```
            field[ :, idx : idx + spacing // 3] = value

    # 3d
    elif dim == 3:
        field = cp.zeros((size[0], h, w), dtype=cp.float32)

        # Horizontal strides
        for i in range(h // spacing + 2):
            if i * spacing >= h:
                break
            idx = spacing // 3 + i * spacing
            field[ :, idx : idx + spacing // 3, :] = value

        # Vertical strides
        for i in range(w // spacing + 2):
            if i * spacing >= w:
                break
            idx = spacing // 3 + i * spacing
            field[ :, :, idx : idx + spacing // 3] = value

    return field
```

Tiling and GPU computation

[ ]:
```
# Compute
def compute_gpu_2d(in_field, stencil, n_stream, n_iter, n_halo, tile_size =␣
 ↪None):
    # Init
    out_field = cp.copy(in_field)

    # Check in_field
    dim = len(in_field.shape)
    assert dim == 2
    h,w = in_field.shape
    h -= 2*n_halo
    w -= 2*n_halo

    # Chech force tile_size
    if tile_size is None:
        assert math.sqrt(n_stream).is_integer()
        tiles_per_side = (int(math.sqrt(n_stream)), int(math.sqrt(n_stream)))
        h_tile = h // tiles_per_side[0]
        w_tile = w // tiles_per_side[1]
    else:
        assert h % tile_size[0] == 0
        assert w % tile_size[1] == 0
        h_tile, w_tile = tile_size
        tiles_per_side = (h // tile_size[0], w // tile_size[1])
```

```
    # Create streams
    streams = [ cp.cuda.Stream() for _ in range(n_stream) ]

    for iter in range(n_iter):
        # Init
        e = cp.cuda.Event()
        e.record()

        update_halo(in_field, n_halo)

        # Iterate over tiles
        for i in range(tiles_per_side[0]):
            for j in range(tiles_per_side[1]):
                # # Indeces
                idx_s = (i*tiles_per_side[0] + j) % n_stream
                with streams[idx_s]:
                    # Stencil iteration
                    stencil(
                        in_field[
                            i*h_tile: 2*n_halo + (i+1)*h_tile,
                            j*w_tile: 2*n_halo + (j+1)*w_tile
                        ],
                        out_field[
                            n_halo + i*h_tile: n_halo + (i+1)*h_tile,
                            n_halo + j*w_tile: n_halo + (j+1)*w_tile
                        ],
                        n_halo
                    )

        # Syncronize all streams
        e.synchronize()

        # Update out_field
        if iter < n_iter - 1:
            in_field, out_field = out_field, in_field

    return out_field
```