Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Home Affairs FDHA
Federal Office of Meteorology and Climatology MeteoSwiss

Performance

Highlighting performance optimization schemes to apply when compiling unstructured DSL stencils

Performance

Overview:

- Some background on GPUs
- Baseline
- Possible optimizations
- Case study: ICON's "diamond" stencil
- State of performance optimization in Dawn

GPUs

- NWP codes are strongly data-level parallel. We therefore see GPUs as the hardware that fits best our needs, as they are throughput-oriented.
- We entirely focus our optimization efforts on producing performant CUDA code.
- Dawn IIR's structure reflects specific needs of the general principles of a vector processor.
- Follows a short (and extremely simplified!) refresher of GPU architectures (using NVIDIA terminology) and some GPU optimization hints.

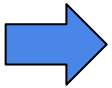
GPUs simplified

In few words:

- **SIMT**: Spawn **a lot** of *threads* executing concurrently a block of code = *kernel*
- But also **SIMD**: same instruction on several data at once
- *Warp* = granularity of SIMD: *vector* of 32 threads, which execute the same instruction in lock-step
- Synchronization (between all the threads) at the end of the kernel

GPUs: parallelizing a loop

```
for(int i=0; i < MAX; ++i) {
    a[i] = b[i];
}
```



Kernel

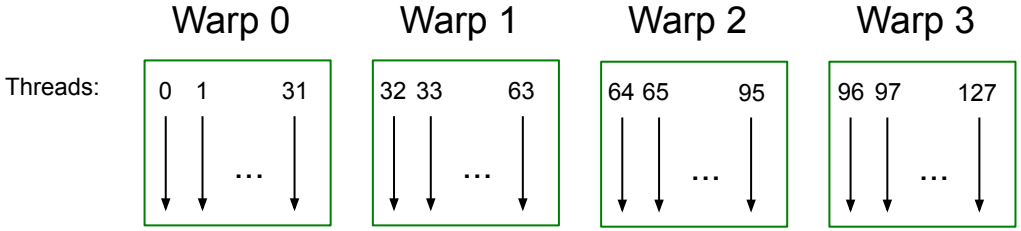
```
__global__ void copy_kernel(double *a, double *b) {
    int pidx = ...
    if(pidx < MAX)
        a[pidx] = b[pidx];
}
```

CPU code

```
void run() {
    ...
    copy_kernel<<<1 + MAX / BLOCK_SIZE, BLOCK_SIZE>>>
        (a_gpu, b_gpu);
}
```

Threads are uniquely indexed, use that index to map memory.

GPUs: warp execution



Code:

```
a = a + 5
b = a
...
```

Data dependency

Showing a very simplified example of warp scheduling.

Threads within each warp must execute the same instruction simultaneously.

A *Warp Scheduler* selects which warp goes into execution from a pool of ready-to-go warps: those for which the next instructions have the operands available (data dependencies resolved).

(Divergence due to conditionals is not considered in this presentation)

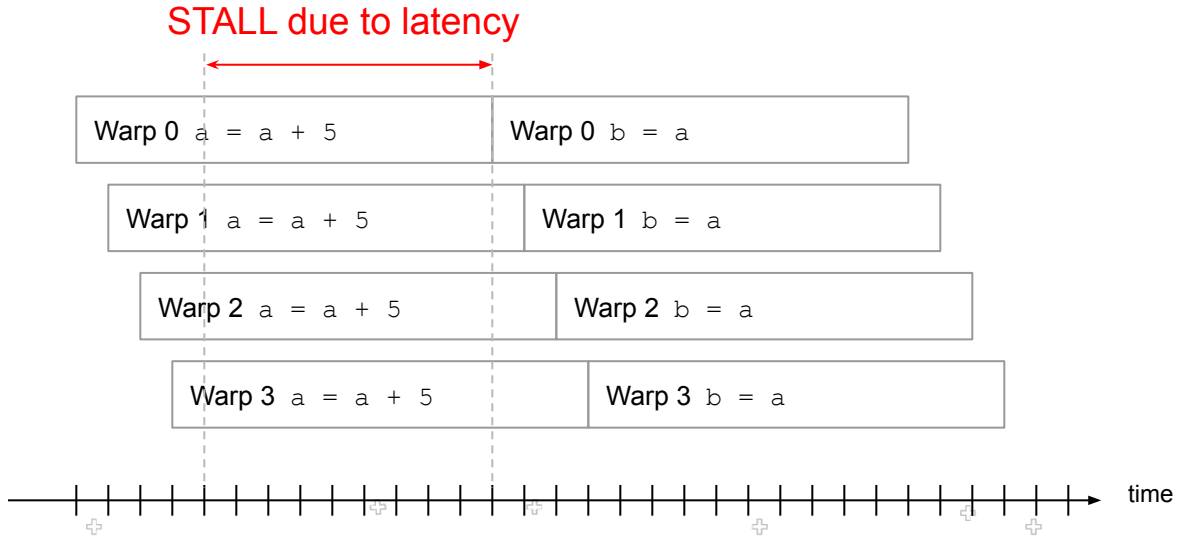
GPUs: warp execution

128 threads (4 warps)

Code:

```
a = a + 5
b = a
...
```

Data dependency



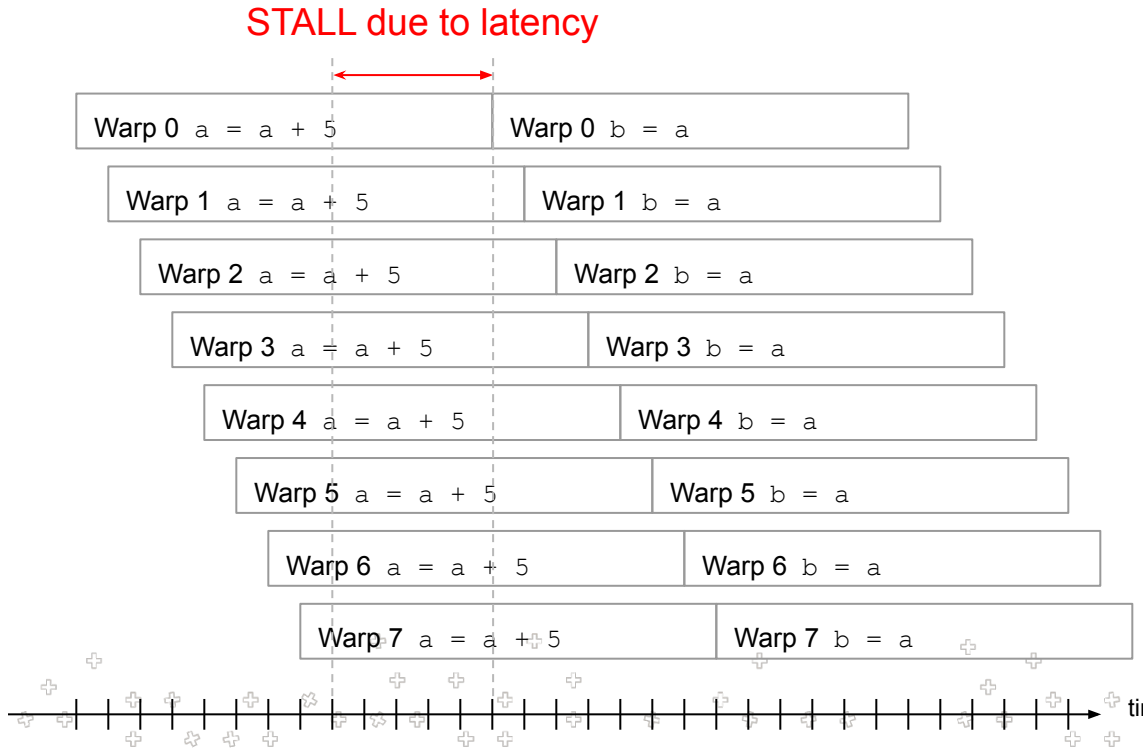
GPUs: latency hiding (more threads)

256 threads (**8** warps)

Code:

```
a = a + 5
b = a
...
```

Data dependency



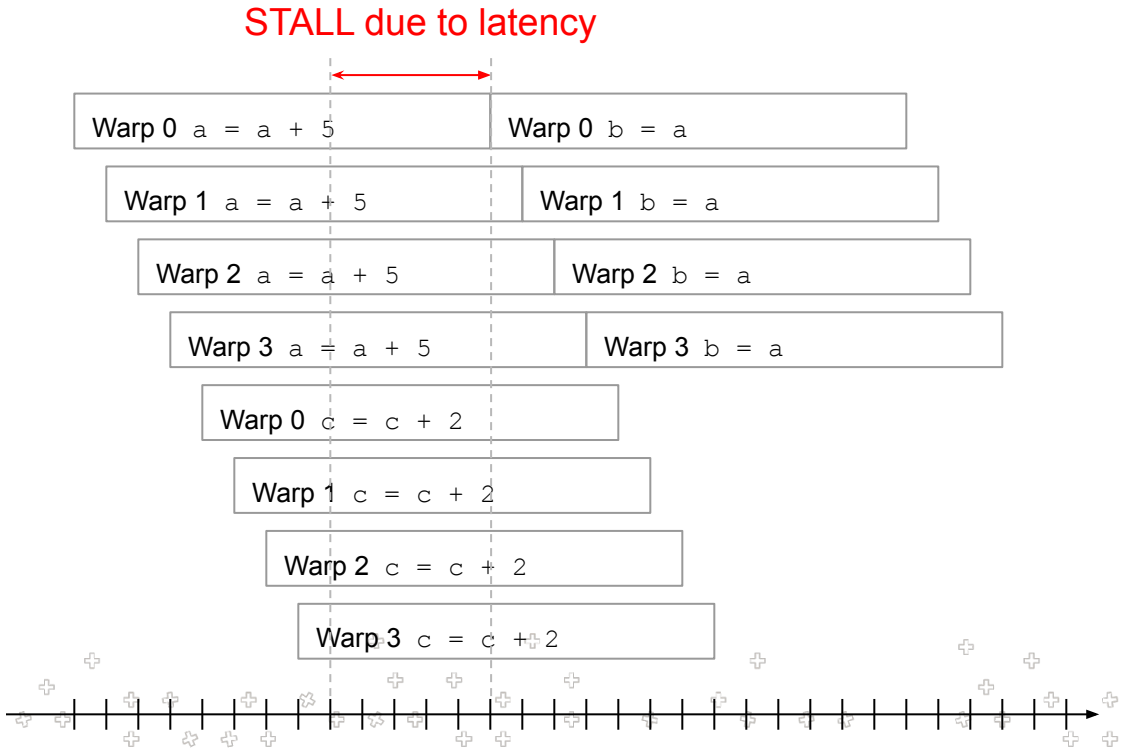
GPUs: latency hiding (ILP)

128 threads (**4** warps)

Code:

```
a = a + 5
c = c + 2
b = a
...
```

Data dependency



GPUs: memory

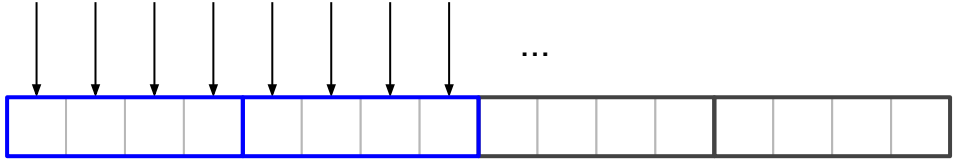
- **Per-thread** *registers*: fast but limited
- *Main memory* (and caches): slow
- … (not relevant for our analysis)

Accessing (load/store) main memory efficiently requires some considerations, e.g. minimizing the number of memory transactions...

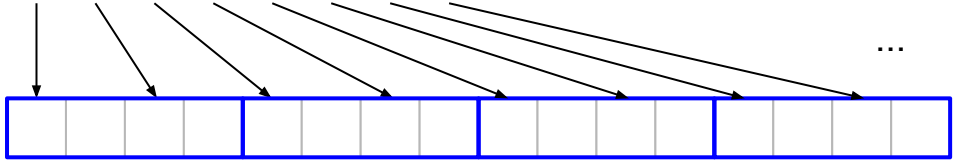
GPUs: access coalescing

GPU tries to coalesce loads/stores of a warp (32 threads) into as few as possible transactions…
but transactions span consecutive segments of memory.

Sequential access pattern: sequential threads in a warp access memory that is sequential.

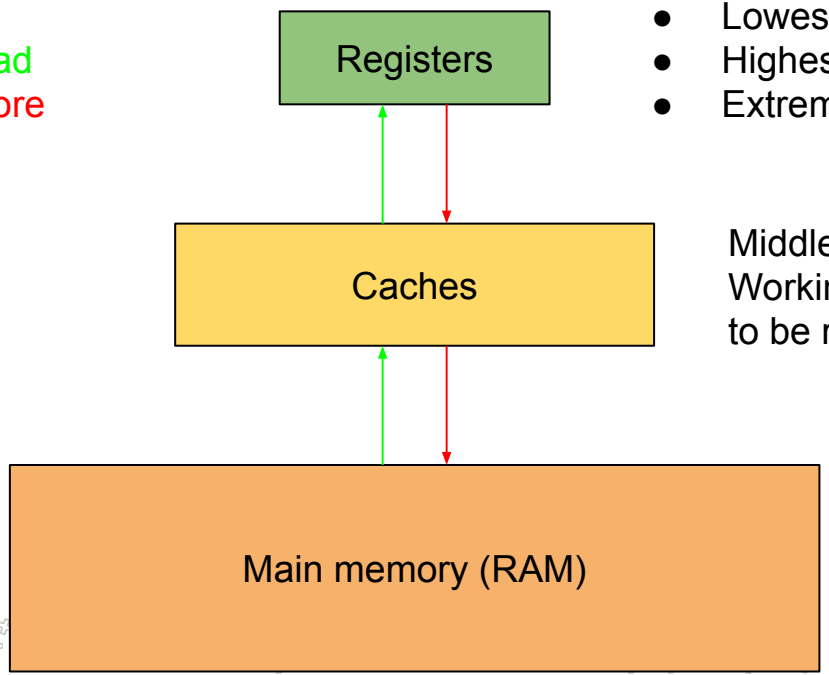


If instead accesses are strided ...



GPUs: memory hierarchy

load
store



- Lowest latency
- Highest bandwidth
- Extremely limited capacity

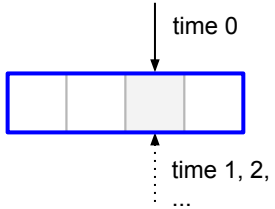
Middle ground… but closer to registers.
Working principle: temporarily holds data that are likely to be reused.

- Highest latency
- Lowest bandwidth
- Enormous capacity

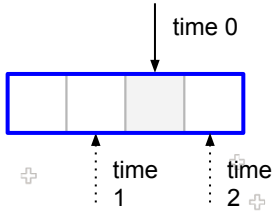
GPUs: locality

Efficiency of caches (how likely a datum is going to be found already in cache), depends on the validity of (at least) one of the following statistical assumptions

Temporal locality: recently accessed memory locations are likely to be accessed again in the near future



Spatial locality (also *data locality*): memory locations with addresses close to those of recently accessed ones are likely to be accessed in the near future.
Works because data is transferred between main memory and caches in *contiguous blocks*.



Contract with the user (parallel model)

```
@stencil
def my_stencil(
 field_a: Field[Edge, K],
 field_c: Field[Edge, K]
) -> None:


 field_b: Field[Edge, K]


 with levels_upward:
   field_b = field_a
   field_c = field_b
```

We guarantee to the user who wrote this Dusk stencil that the resulting generated code will be the equivalent (in terms of *effects* on output fields) of:

- While iterating sequentially through the k-levels from bottom to top,
  - Copying `field_a` over the whole horizontal domain into `field_b`, then
  - Copying `field_b` over the whole horizontal domain into `field_c`.

Baseline: 1 CUDA kernel per statement

```
__global__ void stmt_1_kernel(double *field_a, double *field_b) {
 for(unsigned k = 0; k < K_SIZE; ++k) {
   unsigned idx = k * NUM_EDGES + pidx;
   field_b[idx] = field_a[idx];
 }
}
__global__ void stmt_2_kernel(double *field_b, double *field_c) {
 for(unsigned k = 0; k < K_SIZE; ++k) {
   unsigned idx = k * NUM_EDGES + pidx;
   field_c[idx] = field_b[idx];
 }
}
void run(double *field_a, double *field_b, double *field_c) {
   ...
   stmt_1_kernel<<<1 + NUM_EDGES / BLOCK_SIZE, BLOCK_SIZE>>>
       (field_a_gpu, field_b_gpu);
   stmt_2_kernel<<<1 + NUM_EDGES / BLOCK_SIZE, BLOCK_SIZE>>>
       (field_b_gpu, field_c_gpu);
}
```

- Assumption: there are no *vertical* data dependencies between statements.
- We choose our baseline to be very close to the parallel model. That is: producing one CUDA *kernel* per statement of the original Dusk code and repeating the k-loop inside each kernel.
- 1 CUDA *thread* will perform the computation for one location (in this case one edge) of the dense “iteration” space.

Baseline: Memory organization

Fields as multidimensional arrays linearly stored in *row-major* order.

- Dense fields: `dense[k_idx][dense_idx]`
- Sparse fields: `sparse[k_idx][sparse_idx][dense_idx]`

Since we parallelize over the dense iteration space, to always obtain coalesced accesses we must have the dense (edge/vertex/cell) dimension as last.

Baseline: Sparse iterations

Spare iterations (reductions or sparse loop statements) are translated into *for loops*.

At each iteration we need to query a *neighbor table* which, given the current dense index and neighbor number, gives the index of the current neighbor in the mesh.

A neighbor table is organized in memory as `table[dense_idx][neighbor_iter]`

Example on **Cell > Edge** iteration space:

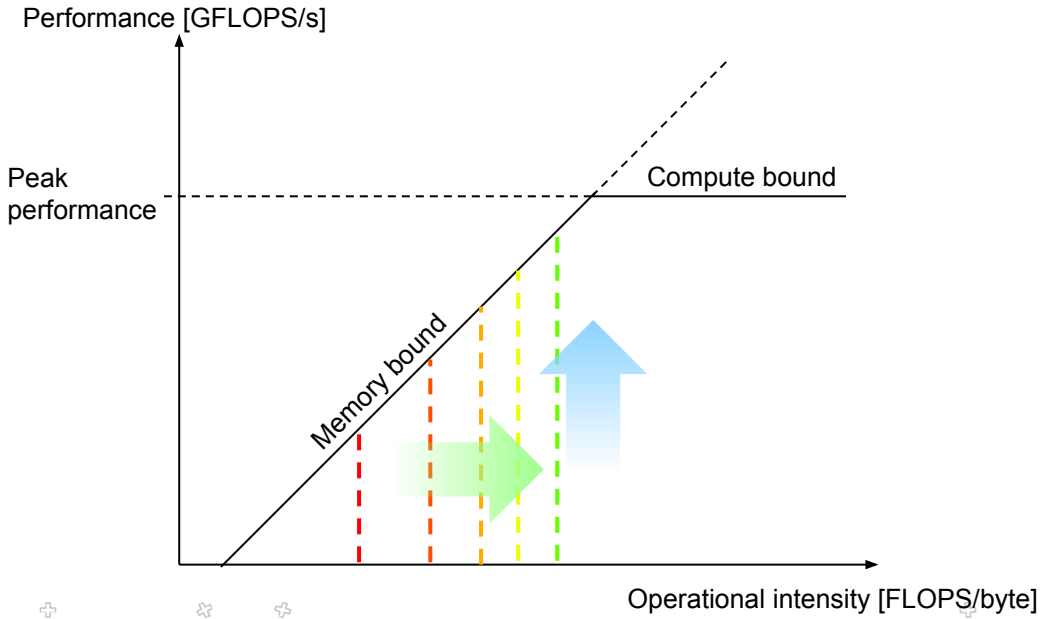
```
__global__ void sparse_stmt_kernel(...) {
 for(unsigned k = 0; k < K_SIZE; ++k) {
 ...
   for (int nbhIter = 0; nbhIter < C_E_SIZE; ++nbhIter)
   {
     int nbhIdx = ceTable[pidx * C_E_SIZE + nbhIter];
     ...
   }
 ...
 }
}
```

Index of current *edge* neighbor

Index of current *cell*

Neighbor number (0, 1 or 2)

Overview on the Roofline model



Stencil-like computations are (sometimes heavily) memory bound.
Some of the optimizations we devised aim at mitigating the limits imposed by memory bandwidth by increasing the *operational intensity* (reducing overall memory traffic).

Another very good way to gain performance is to *hide latency*. In this case, memory traffic remains the same.

Q&A

Questions so far?

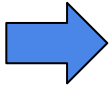
Small break

10 minutes

Optimization: fusing dense “loops”

```
@stencil
def my_stencil(
 field_a: Field[Edge, K],
 field_b: Field[Edge, K],
 field_c: Field[Edge, K],
 field_d: Field[Edge, K]
) -> None:

 with levels_upward:
   field_b = field_a + 1.0
   field_d = field_c + 2.0
```



```
__global__ void fused_kernel(double *field_a, double *field_b,
                             double *field_c, double *field_d) {
  for(unsigned k = 0; k < K_SIZE; ++k) {
    unsigned idx = k * NUM_EDGES + pidx;
    field_b[idx] = field_a[idx] + 1.0;
    field_d[idx] = field_c[idx] + 2.0;
  }
}
void run(double *field_a, double *field_b,
         double *field_c, double *field_d) {
  ...
  fused_kernel<<<NUM_EDGES*K_SIZE/BLOCK_SIZE, BLOCK_SIZE>>>
      (field_a_gpu, field_b_gpu, field_c_gpu, field_d_gpu);
}
```

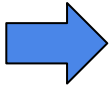
Most simple case: same iteration space (edges) and **no data dependencies**.

Major benefit: hiding latency. Also gain from avoiding kernel-level synchronization and from sharing loop over k-levels. Op. intensity unchanged.

Optimization: fusing dense “loops”

```
@stencil
def my_stencil(
 field_a: Field[Edge, K],
 field_b: Field[Edge, K],
 field_c: Field[Vertex, K],
 field_d: Field[Edge, K]
) -> None:

 with levels_upward:
   field_b = field_a + 1.0
   field_d = sum_over(Edge > Vertex,
                      field_c)
```



```
__global__ void fused_kernel(...) {
 ... // e.g. loop over k
   field_b[k * NumEdges + pidx] =
       field_a[k * NumEdges + pidx]+ 1.0;
   double sum = 0.0;
   for (int nbhIter = 0; nbhIter < E_V_SIZE; nbhIter++) {
     int nbhIdx = evTable[pidx * E_V_SIZE + nbhIter];
     sum += field_c[k * NumVertices + nbhIdx];
   }
   field_d[k * NumEdges + pidx] = sum;
 ...
}
```

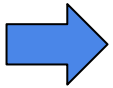
Same case as before, just changed the second assignment into a reduction.

Optimization: fusing dense “loops”

```
@stencil
def my_stencil(
 field_a: Field[Edge, K],
 field_c: Field[Edge, K]
) -> None:

 field_b: Field[Edge, K]

 with levels_upward:
   field_b = field_a
   field_c = field_b
```



```
__global__ void fused_kernel(double *field_a, double *field_c) {
 ... // e.g. loop over k
   double local_field_b = field_a[k * NumEdges + pidx];
   field_c[k * NumEdges + pidx] = local_field_b;
 ...
}
```

Case: same iteration space (edges) and **non-offset data dependency**.

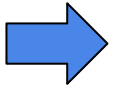
On top of benefits of previous case, here we also spare the memory accesses to `field_b` (temporary). Op. intensity increased.

Optimization: fusing dense “loops”

```
@stencil
def my_stencil(
 field_a: Field[Vertex, K]
 field_c: Field[Edge, K]
) -> None:

 field_b: Field[Edge, K]

 with levels_upward:
    field_b = sum_over(Edge > Cell > Vertex,
                       field_a)
    field_c = field_b + 1.0
```



```
__global__ void fused_kernel(double *field_a, double *field_c) {
 ... // e.g. loop over k
   double sum = 0.0;
   for (int nbhIter = 0; nbhIter < E_C_V_SIZE; nbhIter++) {
     int nbhIdx = ecvTable[pidx * E_C_V_SIZE + nbhIter];
     sum += field_a[k * NumVertices + nbhIdx];
   }
   double local_field_b = sum;
   field_c[k * NumEdges + pidx] = local_field_b +1.0;
 ...
}
```

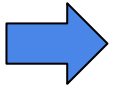
Same case as before, just changed the first assignment into a reduction.

Optimization: fusing dense “loops”

```
@stencil
def my_stencil(
 field_a: Field[Edge, K],
 field_c: Field[Cell, K],
 field_d: Field[Edge, K]
) -> None:

 field_b: Field[Edge, K]

 with levels_upward:
   field_b = field_a + 1.0
   field_d = sum_over(Edge > Cell,
                      field_b * field_c)
```



```
__global__ void fused_kernel(...) {
 ... // e.g. loop over k
   double local_field_b = field_a[k * NumEdges + pidx] +1.0;
   double sum = 0.0;
   for (int nbhIter = 0; nbhIter < E_C_SIZE; nbhIter++) {
     int nbhIdx = ecTable[pidx * E_C_SIZE + nbhIter];
     sum += local_field_b * field_c[k * NumCells + nbhIdx];
   }
   field_d[k * NumEdges + pidx] = sum;
 ...
}
```

Same case as before, now second assignment is a reduction instead.

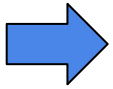
Notice that `field_b` is still accessed non-offset from within the reduction, because it’s on edges.

Optimization: one-time stencil inlining

```
@stencil
def my_stencil(
 field_a: Field[Vertex, K],
 field_c: Field[Edge, K]
) -> None:

 field_b: Field[Vertex, K]

 with levels_upward:
   field_b = field_a + 1.0
   field_c = sum_over(Edge > Vertex,
                      field_b)
```



```
__global__ void fused_kernel(double *field_a, double *field_c) {
 ... // e.g. loop over k
   double sum = 0.0;
   for (int nbhIter = 0; nbhIter < E_V_SIZE; nbhIter++) {
     int nbhIdx = evTable[pidx * E_V_SIZE + nbhIter];
     double local_field_b = field_a[k * NumVertices + nbhIdx] +1.0;
     sum += local_field_b;
   }
   field_c[k * NumEdges + pidx] = sum;
 ...
}
```

Also when there's an **offset data dependency** we can get rid of accesses to temporary `field_b` and remove its pre-computation.

Similar benefits of fusing dense loops with non-offset data dependencies.

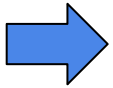
Op. intensity increased.

Optimization: one-time stencil inlining

```
@stencil
def my_stencil(
 field_a: Field[Vertex, K],
 field_c: Field[Edge, K]
) -> None:

field_b: Field[Cell, K]

with levels_upward:
  field_b = sum_over(Cell > Vertex,
                     field_a)
  field_c = sum_over(Edge > Cell,
                     field_b)
```



```
__global__ void fused_kernel(double *field_a, double *field_c) {
 ... // e.g. loop over k
 double sum_0 = 0.0;
 for (int nbhIter_0 = 0; nbhIter_0 < E_C_SIZE; nbhIter_0++) {
   int nbhIdx_0 = ecTable[pidx * E_C_SIZE +nbhIter_0];
   double sum_1 = 0.0;
   for (int nbhIter_1 = 0; nbhIter_1 < C_V_SIZE; nbhIter_1++) {
     int nbhIdx_1 = cvTable[nbhIdx_0 * C_V_SIZE + nbhIter_1];
     sum_1 += field_a[k * NumVertices +nbhIdx_1];
   }
   double local_field_b = sum_1;
   sum_0 += local_field_b;
 }
 field_c[k * NumEdges + pidx] =sum_0;
 ...
}
```

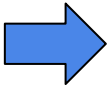
Same optimization. Now first statement is a reduction.

Need to nest reduction loops in the generated code.

Optimization: fusing reductions

```
@stencil
def my_stencil(
 field_a: Field[Edge, K],
 field_b: Field[Edge, K],
 field_c: Field[Edge, K],
 field_d: Field[Vertex, K],
 field_e: Field[Vertex, K]
) -> None:

 with levels_upward:
   field_d = sum_over(Vertex > Edge,
                      field_a * field_b)
   field_e = sum_over(Vertex > Edge,
                      field_a * field_c)
```



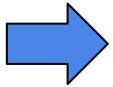
```
__global__ void fused_kernel(...) {
 ... // e.g. loop over k
  double sum_d = 0.0;
  double sum_e = 0.0;
  for (int nbhIter = 0; nbhIter < V_E_SIZE; nbhIter++) {
    int nbhIdx = veTable[pidx * V_E_SIZE + nbhIter];
    double local_field_a = field_a[k * NumEdges + nbhIdx];
    sum_d += (local_field_a * field_b[k * NumEdges + nbhIdx]);
    sum_e += (local_field_a * field_c[k * NumEdges + nbhIdx]);
  }
  field_d[k * NumVertices + pidx] = sum_d;
  field_e[k * NumVertices + pidx] = sum_e;
 ...
}
```

Can fuse reductions on same iteration space (here Vertex > Edge).
Multiple benefits, sharing: accesses to input fields in common, accesses to the neighbor table and the loop over neighbors. Op. intensity increased.

Optimization: fusing also sparse loops

```
@stencil
def my_stencil(
 field_a: Field[Edge, K],
 field_b: Field[Vertex, K],
 field_c: Field[Edge, K],
 field_d: Field[Vertex, K],
 field_sparse: Field[Vertex > Edge, K]
) -> None:

 with levels_upward:
   with sparse[Vertex > Edge]:
     field_sparse = field_a * field_b
   field_d = sum_over(Vertex > Edge,
                      field_b * field_c)
```



```
__global__ void fused_kernel(...)
{
 ... // e.g. loop over k
 double sum = 0.0;
 double local_field_b = field_b[k * NumVertices + pidx];
 for (int nbhIter = 0; nbhIter < V_E_SIZE; nbhIter++) {
  int nbhIdx = veTable[pidx * V_E_SIZE + nbhIter];
  field_sparse[k*V_E_SIZE*NumVertices + nbhIter*NumVertices + pidx] =
                                    field_a[k * NumEdges + nbhIdx] *
                                    local_field_b;
  sum += (local_field_b * field_c[k * NumEdges + nbhIdx]);
 }
 field_d[k * NumVertices + pidx] = sum;
 ...
}
```

Can fuse 2 sparse loops or a sparse loop with a reduction.

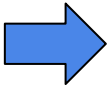
Same benefits as previous optimization.

Optimization: fusing also sparse loops

```
@stencil
def my_stencil(
 field_a: Field[Edge, K],
 field_b: Field[Vertex, K],
 field_c: Field[Edge, K],
 field_d: Field[Vertex, K]
) -> None:

 field_sparse: Field[Vertex > Edge, K]

 with levels_upward:
   with sparse[Vertex > Edge]:
     field_sparse = field_a * field_b
   field_d = sum_over(Vertex > Edge,
                      field_sparse * field_c)
```



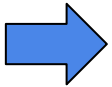
```
__global__ void fused_kernel(double *field_a, double *field_b,
                             double *field_c, double *field_d)
{
 ... // e.g. loop over k
   double sum = 0.0;
   for (int nbhIter = 0; nbhIter < V_E_SIZE; nbhIter++) {
     int nbhIdx = veTable[pidx * V_E_SIZE + nbhIter];
     double local_field_sparse = field_a[k * NumEdges + nbhIdx] *
                                 field_b[k * NumVertices + pidx];
     sum += (local_field_sparse * field_c[k * NumEdges + nbhIdx]);
   }
   field_d[k * NumVertices + pidx] = sum;
 ...
}
```

Data dependency between loops being fused. Saving also accesses to `field_sparse`. This corresponds to inlining the computation of `field_sparse`.

Optimization: recurrent stencil inlining

```
...
field_b: Field[Vertex, K]

with levels_upward:
  field_b = field_a + 1.0
  field_c = sum_over(Edge > Vertex,
                     field_b)
  field_d = sum_over(Cell > Vertex,
                     field_b)
```



```
__global__ void fused_kernel_1(double *field_a, double *field_c) {
  ... // e.g. loop over k
    double sum = 0.0;
    for (int nbhIter = 0; nbhIter < E_V_SIZE; nbhIter++) {
      int nbhIdx = evTable[pidx * E_V_SIZE + nbhIter];
      double local_field_b = field_a[k * NumVertices + nbhIdx] +1.0;
      sum += local_field_b;
    }
    field_c[k * NumEdges + pidx] = sum;
__global__ void fused_kernel_2(double *field_a, double *field_d) {
  ... // e.g. loop over k
    double sum = 0.0;
    for (int nbhIter = 0; nbhIter < C_V_SIZE; nbhIter++) {
      int nbhIdx = cvTable[pidx * C_V_SIZE + nbhIter];
      double local_field_b = field_a[k * NumVertices + nbhIdx] +1.0;
      sum += local_field_b;
    }
    field_d[k * NumCells + pidx] = sum;
  ...
}
```

Temporary is inlined multiple times.
Under some circumstances, it can improve performance.
Benefits of this transformation are not captured by the Roofline model.

Optimization: recurrent stencil inlining

```
...
field_b: Field[Vertex, K]

with levels_upward:
  field_b = field_a + 1.0
  field_c = sum_over(Edge > Vertex,
                     fieldb)
  field_d = sum_over(Cell > Vertex,
                     fieldb)
```

vs.

```
...
with levels_upward:
  field_c = sum_over(Edge > Vertex,
                     field_a +1.0)
  field_d = sum_over(Cell > Vertex,
                     field_a +1.0)
```

2 memory accesses per vertex +
3 memory accesses per edge +
7 memory accesses per cell

0 memory accesses per vertex +
3 memory accesses per edge +
7 memory accesses per cell

Ignoring effects of cache and of indexing pattern (explained later on).

Optimization: recurrent stencil inlining

```
...
field_tmp: Field[Vertex, K]

with levels_upward:
  field_tmp = (field_a + field_b + field_c)**2
  field_d = sqrt(field_tmp + 5.0)
  field_e = sum_over(Edge > Vertex,
                     field_tmp * 4.0)
  field_f = field_tmp / 2.0
```

8 memory accesses per vertex +
3 memory accesses per edge

vs.

```
...
with levels_upward:
  field_d = sqrt((field_a + field_b + field_c)**2 + 5.0)
  field_e = sum_over(Edge > Vertex,
                     (field_a + field_b + field_c)**2 + 5.0)
  field_f = (field_a + field_b + field_c)**2 / 2.0
```

8 memory accesses per vertex +
7 memory accesses per edge

Ignoring effects of cache and of indexing pattern (explained later on).

Optimization: recurrent stencil inlining

```
...
field_tmp: Field[Edge > Vertex, K]

  with levels_upward:
    with sparse[Edge > Vertex]:
      field_tmp = (field_edge * field_vert1)*2
    field_x = sum_over(Edge > Vertex,
                       field_vert2 +field_tmp)
    field_y = sum_over(Edge > Vertex,
                       field_x *field_tmp)
```

14 memory accesses per edge

VS.

```
...

with levels_upward:
  field_x = sum_over(Edge > Vertex,
             field_vert2 + (field_edge * field_vert1)*2)
  field_y = sum_over(Edge > Vertex,
             field_x * (field_edge * field_vert1)*2)
```

11 memory accesses per edge

In this case the temporary is a **sparse field**.

Limitations: register pressure

Short digression:

The optimizations presented so far tend to *increase the number of necessary hardware registers* per thread, as a natural consequence of keeping memory transactions to the minimum.

At some point registers will be spilled into main memory!

How much can we fuse kernels together until it's not convenient anymore? Clearly we need to keep an eye on register pressure.

Limitations: data dependencies

```
@stencil
def my_stencil(
 field_a: Field[Vertex, K],
 field_b: Field[Cell, K],
 field_c: Field[Vertex, K],
 field_d: Field[Cell, K]
) -> None:

 with levels_upward:
   field_b = sum_over(Cell > Vertex,
                      field_c)
   field_c = field_a + 1.0
   field_d = sum_over(Cell > Vertex,
                      field_a * field_c)
```

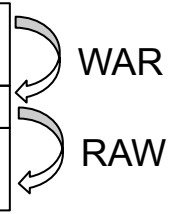
Would like to fuse these 2 statements together in the same kernel, because they are both *on cells*.

Limitations: data dependencies

```
@stencil
def my_stencil(
 field_a: Field[Vertex, K],
 field_b: Field[Cell, K],
 field_c: Field[Vertex, K],
 field_d: Field[Cell, K]
) -> None:

 with levels_upward:
```

```
field_b = sum_over(Cell > Vertex,
                   field_c)
field_c = field_a + 1.0
field_d = sum_over(Cell > Vertex,
                   field_a * field_c)
```

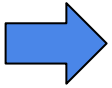


Can't. There is a write-after-read dependency between first and second statements and a read-after-write dependency between second and third.

Optimization: parallelize k-loop

```
@stencil
def my_stencil(
 field_a: Field[Edge, K],
 field_b: Field[Edge, K]
) -> None:

 with levels_upward:
   field_b = field_a
```



```
__global__ void my_stencil_kernel(double *field_a, double *field_b) {
 unsigned int pidx = blockIdx.x * blockDim.x + threadIdx.x;
 unsigned int kidx = blockIdx.y * blockDim.y + threadIdx.y;

 field_b[kidx * NumEdges + pidx] = field_a[kidx * NumEdges + pidx];
}
void run(double *field_a, double *field_b) {
 ...
 dim3 dB(BLOCK_SIZE, BLOCK_SIZE,1);
 dim3 dG(K_SIZE / BLOCK_SIZE, NUM_EDGES / BLOCK_SIZE,1);
 my_stencil_kernel<<<dG, dB>>>(field_a_gpu, field_b_gpu);
}
```

Assumption: order of iteration over k-levels doesn't matter.

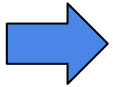
1 CUDA *thread* will perform the computation for one location (here edge) **and for one k level**.

More threads: helps in hiding latency. Op. intensity unchanged.

Optimization: vector packing

```
@stencil
def my_stencil(
 field_u: Field[Edge, K],
 field_v: Field[Edge, K],
 field_res: Field[Edge, K]
) -> None:

 with levels_upward:
   field_res = field_u ** 2 +
               field_v ** 2
```



```
__global__ void my_stencil_kernel(double2 *field_uv,
                                  double *field_res) {
 ...
   double2 local_field_uv = field_uv[k * NumEdges + pidx];
   field_res[k * NumEdges + pidx] =
       (pow(local_field_uv.u, 2) +
        pow(local_field_uv.v, 2));
 ...
}
```

Pack fields which are always accessed together (because they are vectors in the mathematical sense) using `float2, float3, double2, double3,` … types provided by CUDA.

Gain from larger memory access with same load instruction. Op. intensity unchanged.

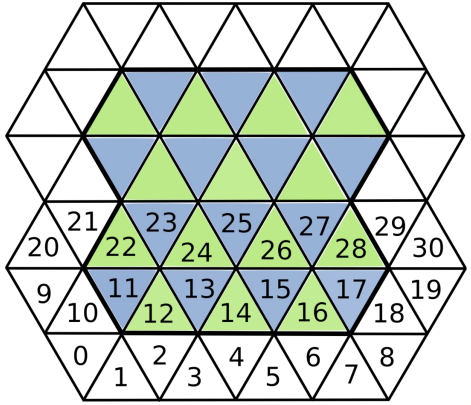
Q&A

Questions so far?

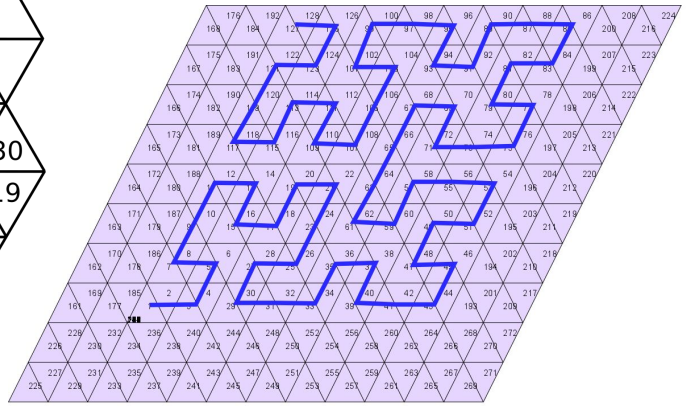
Coffee break

30 minutes

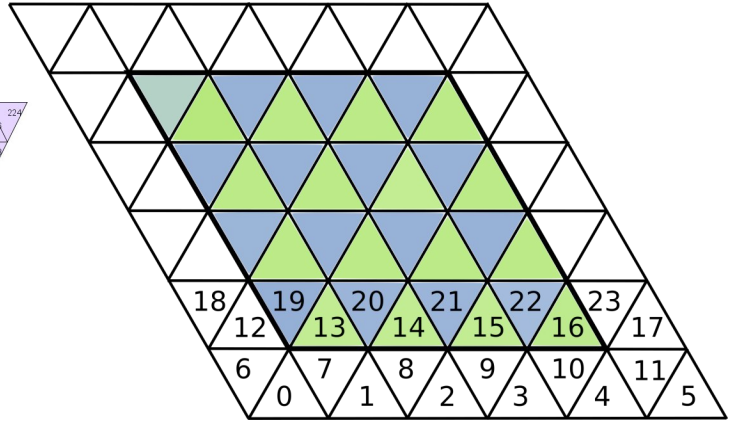
Optimization: indexing patterns



Row Major



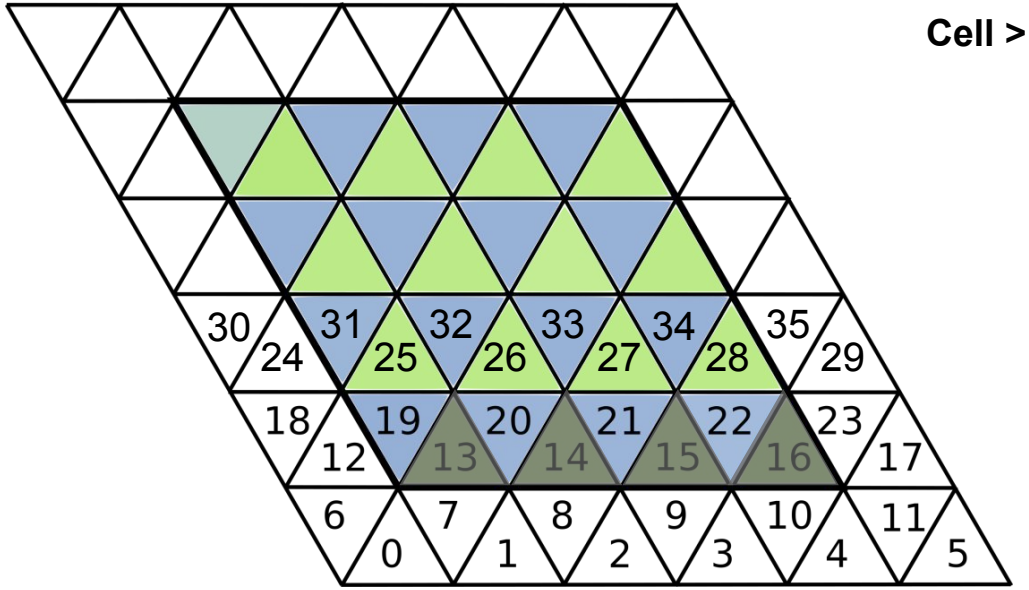
Space Filling Curve



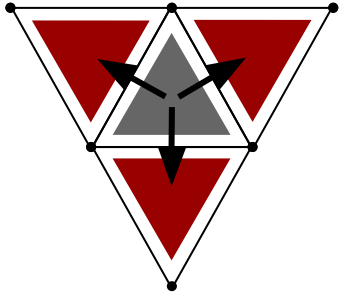
Structured Numbering

Order of elements of the dense dimension impacts performance when accessing neighbors. It has implications on locality (thus cache efficiency) and the overall number of memory transactions.

Optimization: indexing patterns

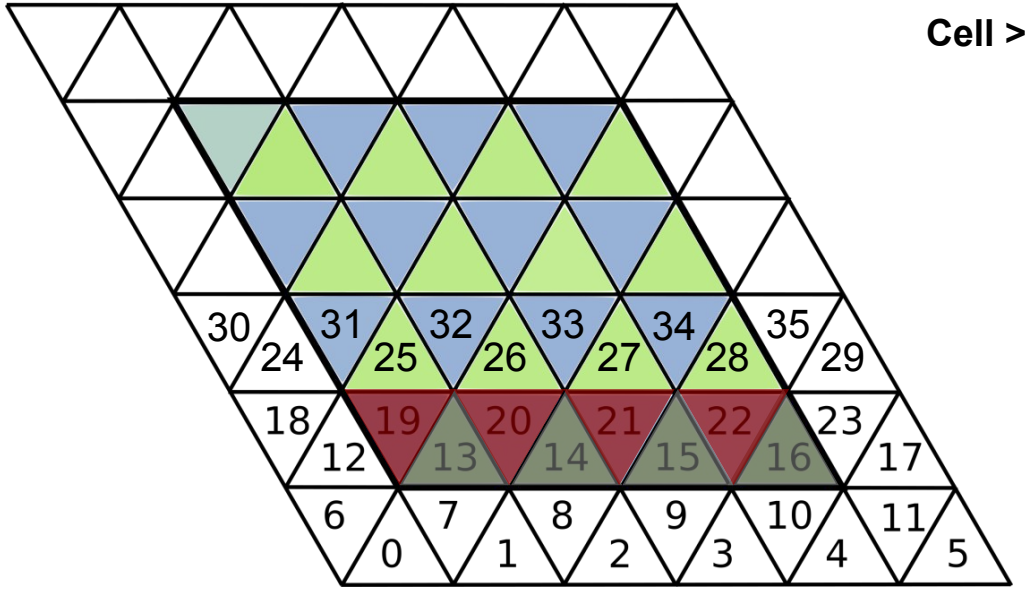


Cell > Edge > Cell

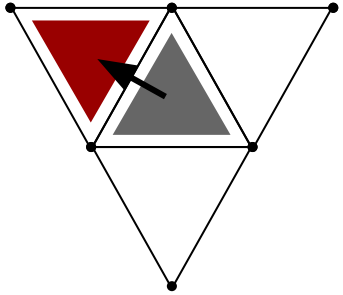


Structured numbering. Always coalesced accesses, worst locality.

Optimization: indexing patterns

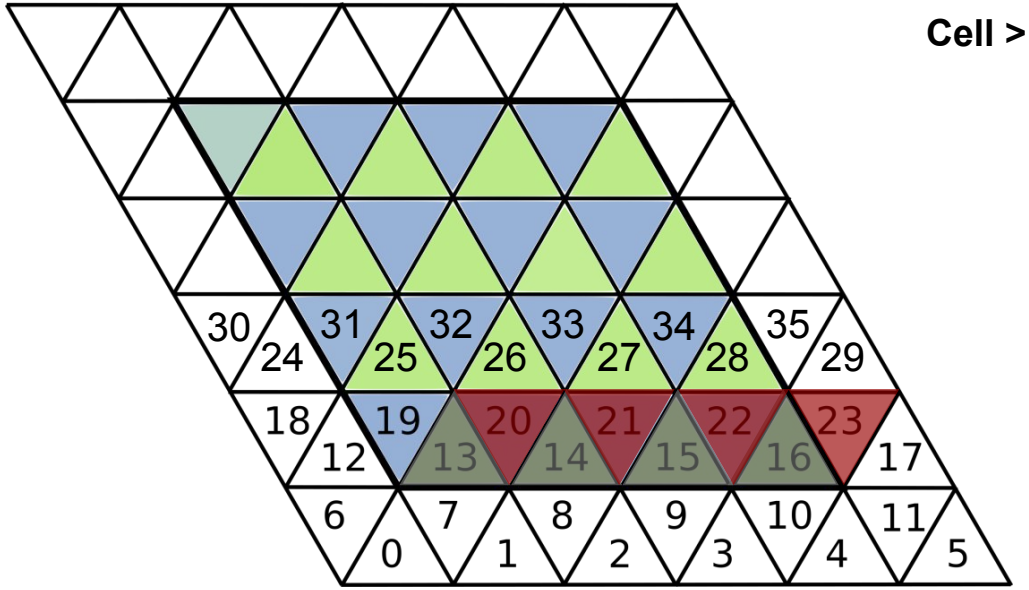


Cell > Edge > Cell

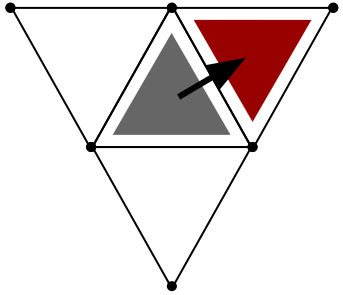


Structured numbering. Always coalesced accesses, worst locality.

Optimization: indexing patterns

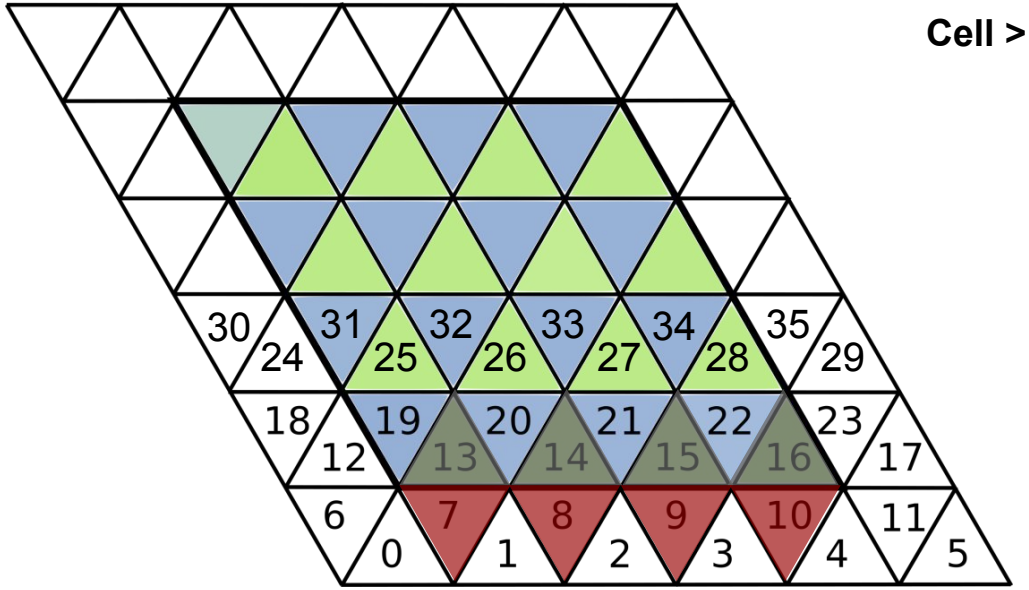


Cell > Edge > Cell

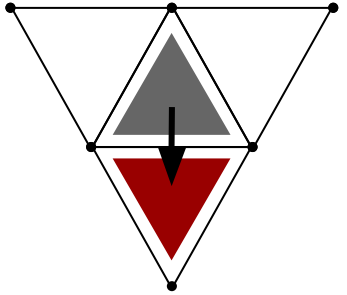


Structured numbering. Always coalesced accesses, worst locality.

Optimization: indexing patterns

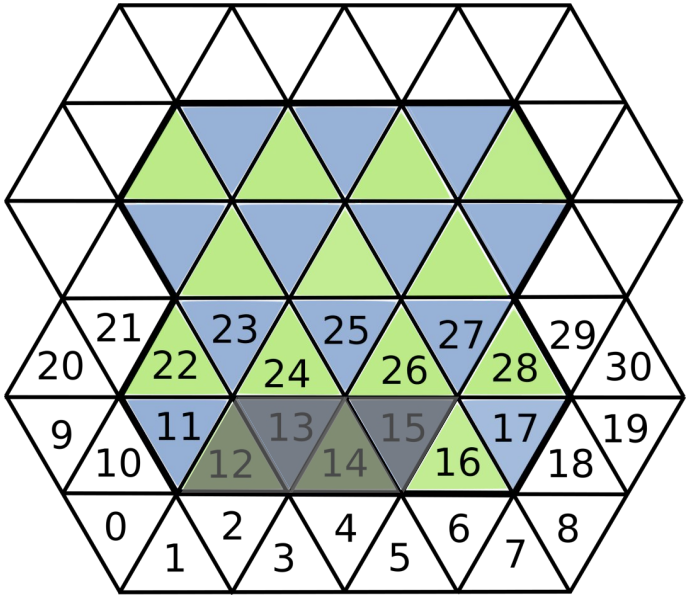


Cell > Edge > Cell

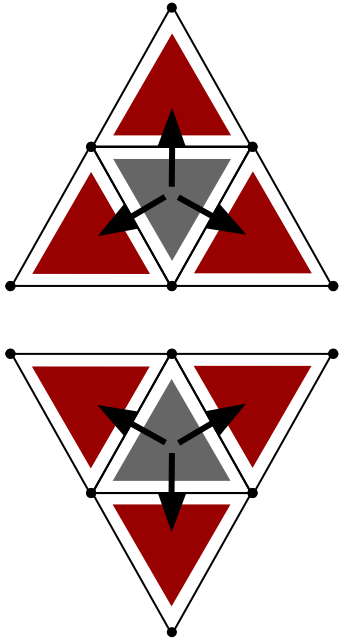


Structured numbering. Always coalesced accesses, worst locality.

Optimization: indexing patterns

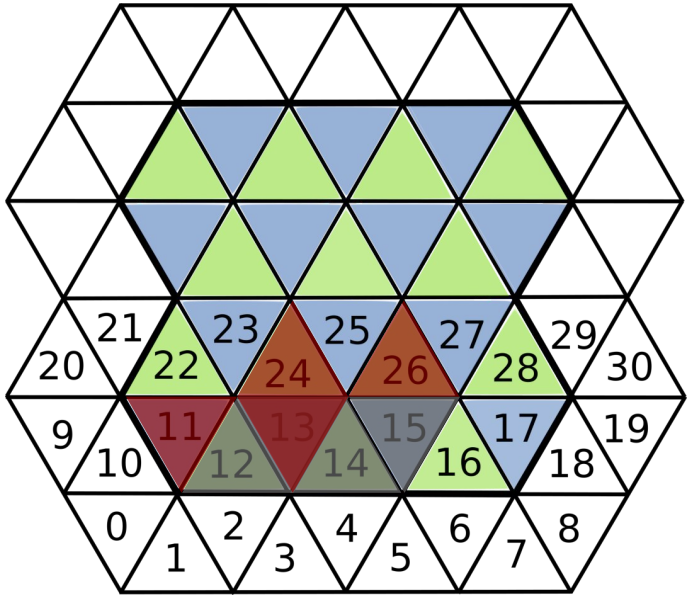


Cell > Edge > Cell

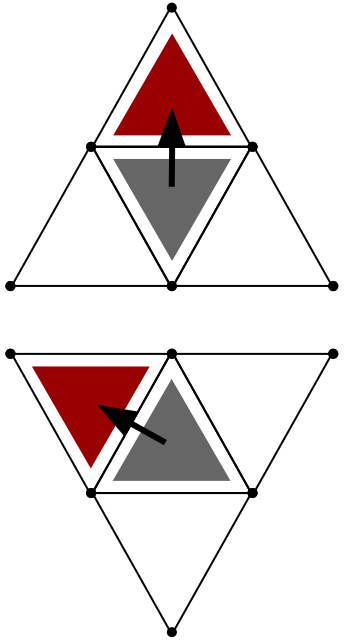


Row Major numbering. Compromise between access coalescing and locality.

Optimization: indexing patterns

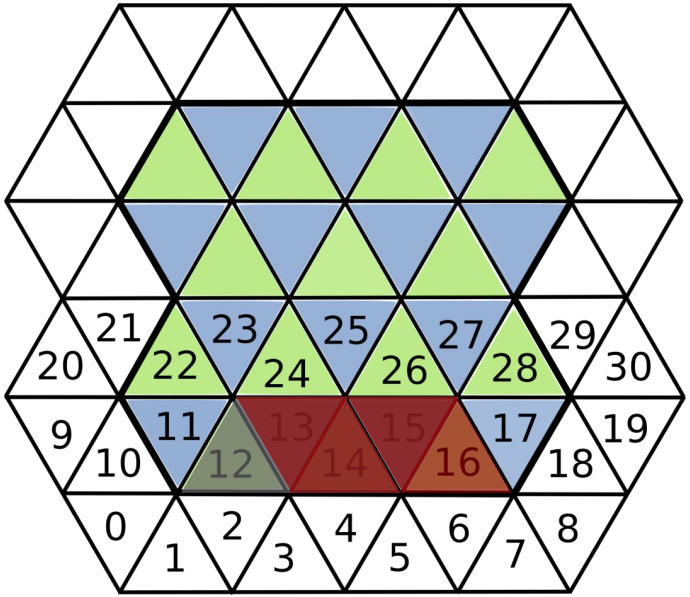


Cell > Edge > Cell

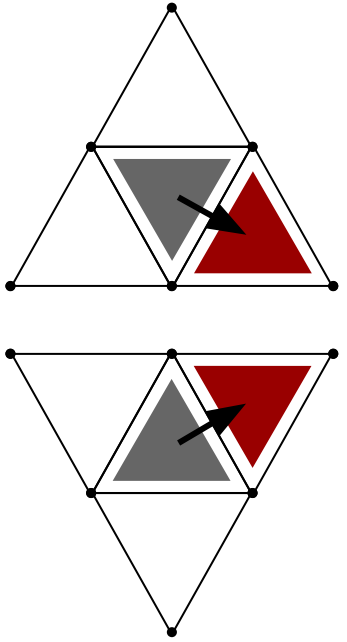


Row Major numbering. Compromise between access coalescing and locality.

Optimization: indexing patterns

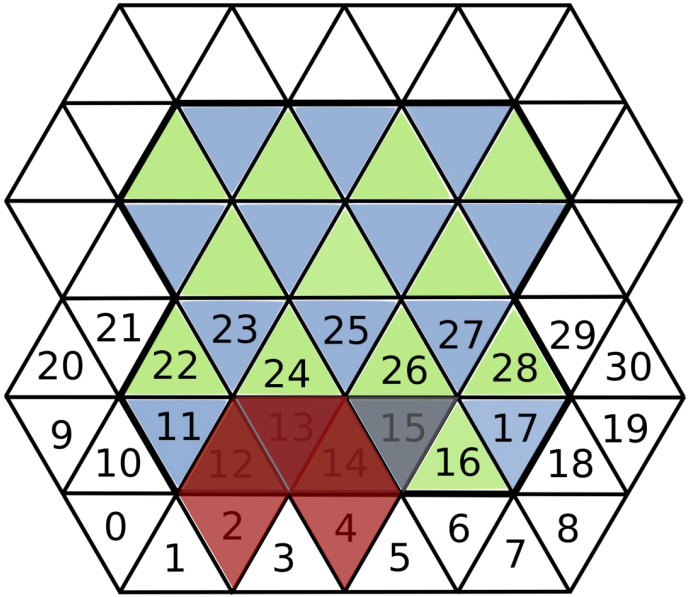


Cell > Edge > Cell

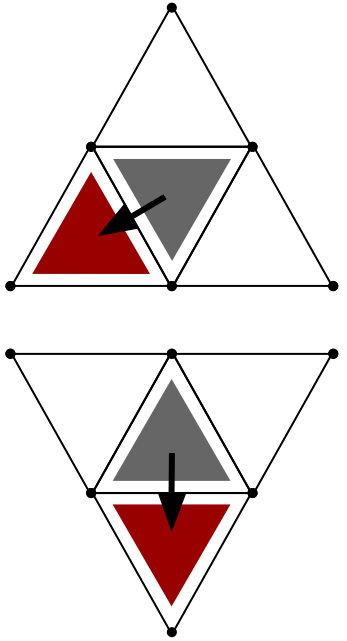


Row Major numbering. Compromise between access coalescing and locality.

Optimization: indexing patterns

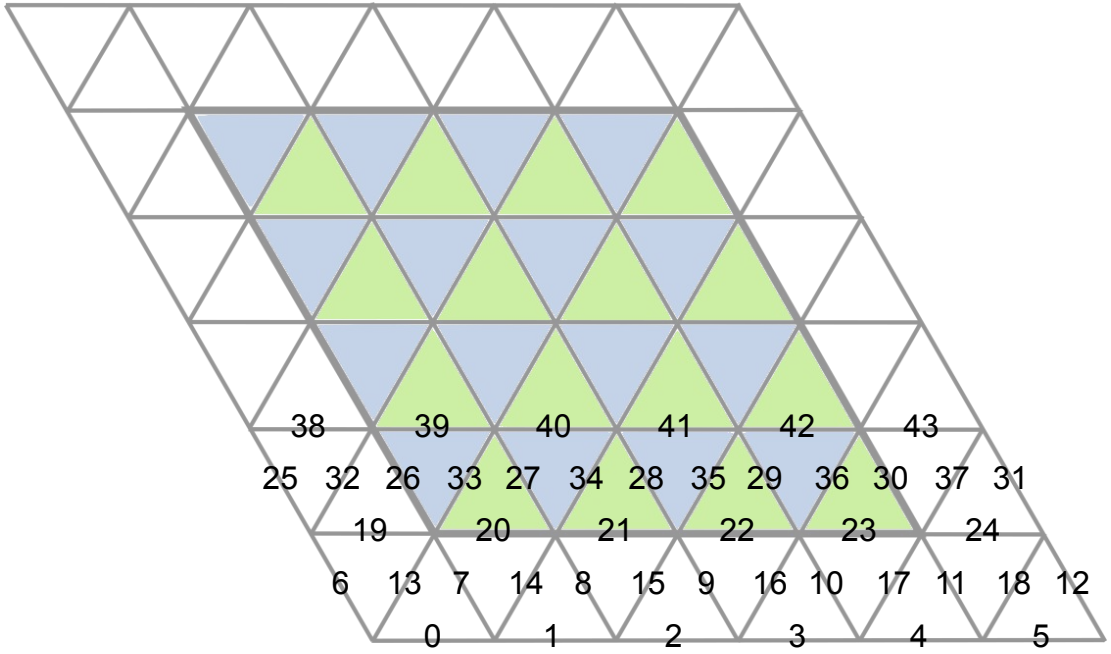


Cell > Edge > Cell



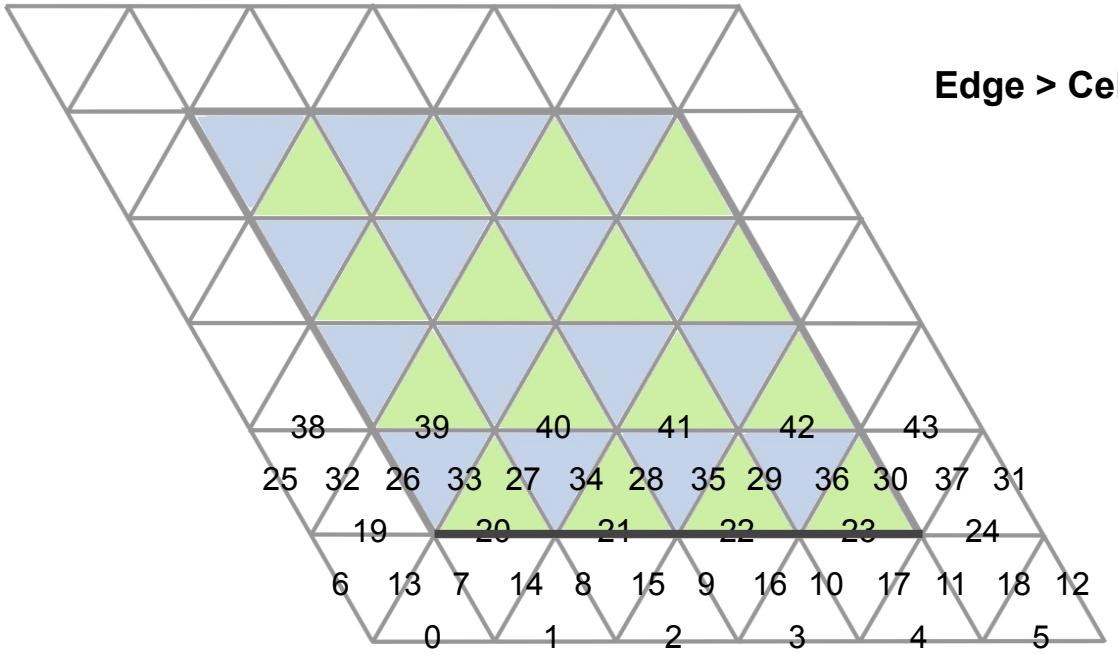
Row Major numbering. Compromise between access coalescing and locality.

Optimization: indexing patterns

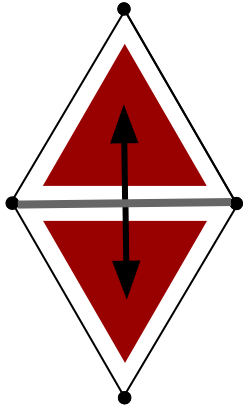


Structured numbering. Always coalesced accesses, worst locality.

Optimization: indexing patterns

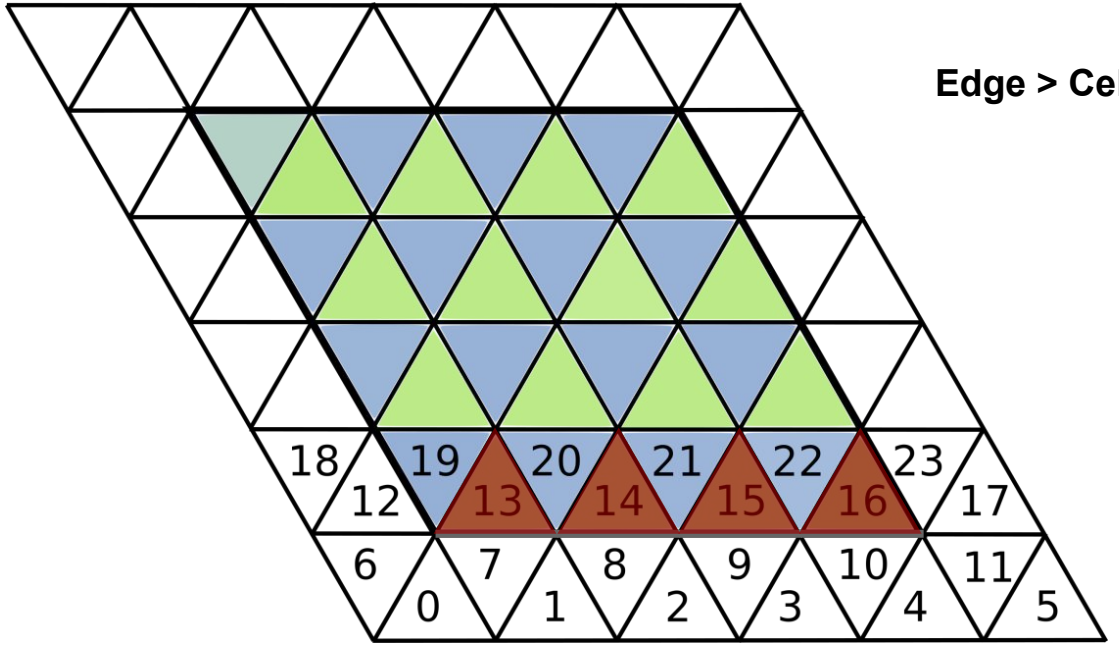


Edge > Cell

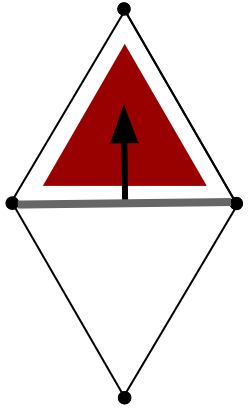


Structured numbering. Always coalesced accesses, worst locality.

Optimization: indexing patterns

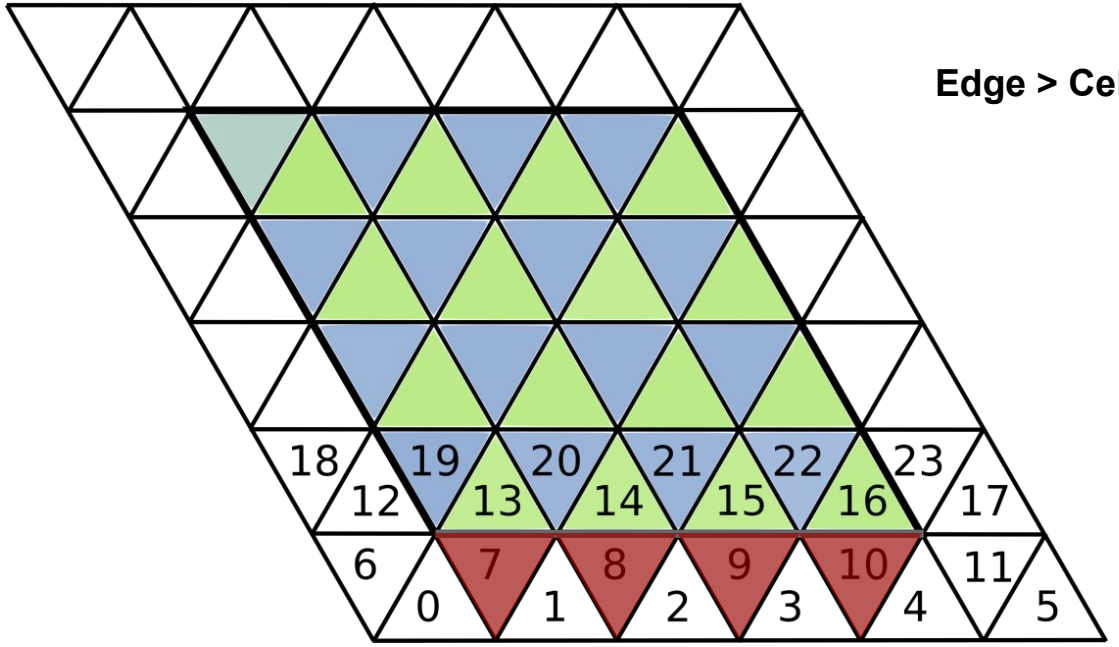


Edge > Cell

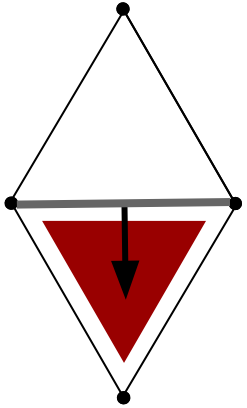


Structured numbering. Always coalesced accesses, worst locality.

Optimization: indexing patterns

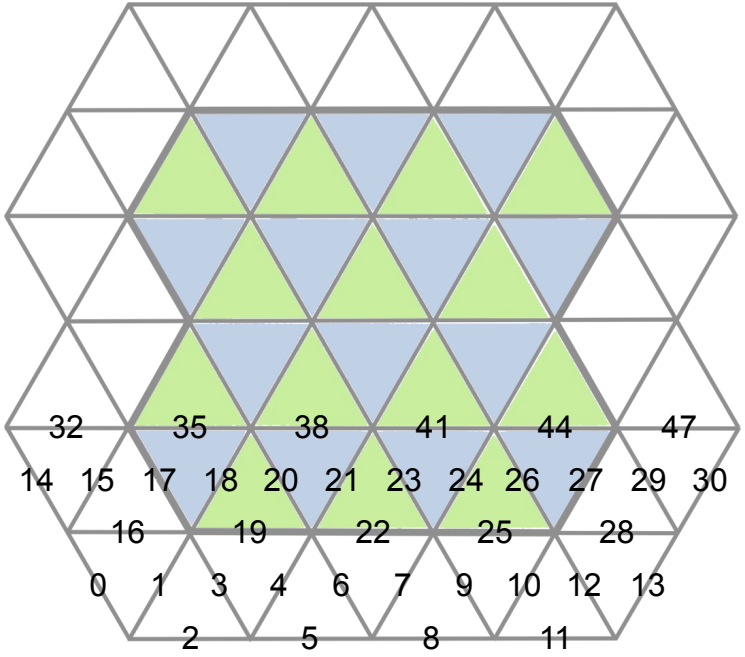


Edge > Cell



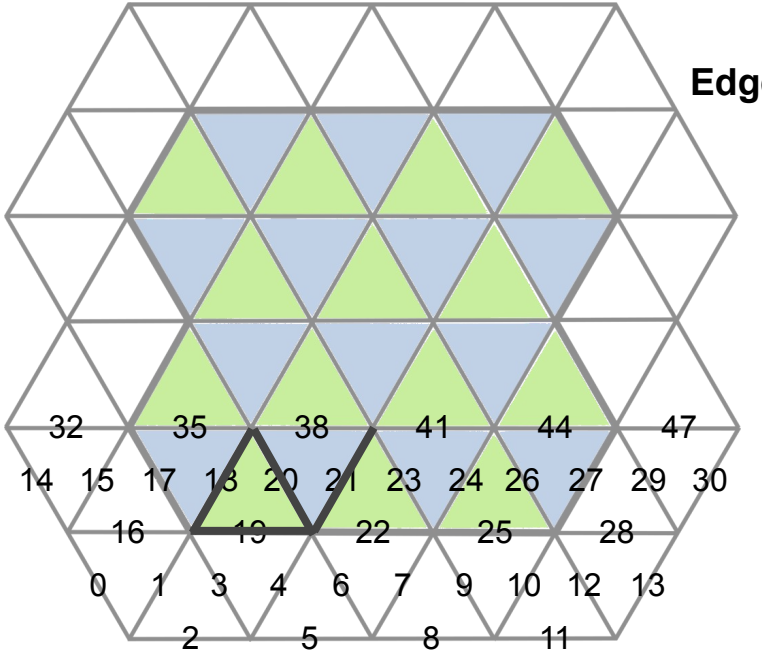
Structured numbering. Always coalesced accesses, worst locality.

Optimization: indexing patterns

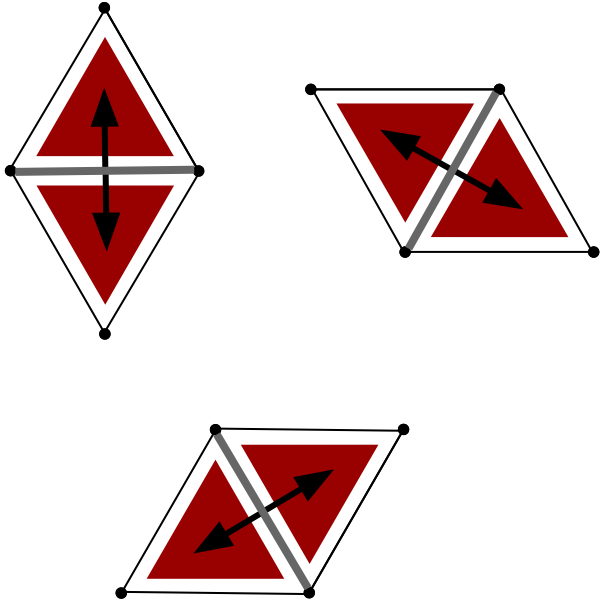


Row Major numbering. Compromise between access coalescing and locality.

Optimization: indexing patterns

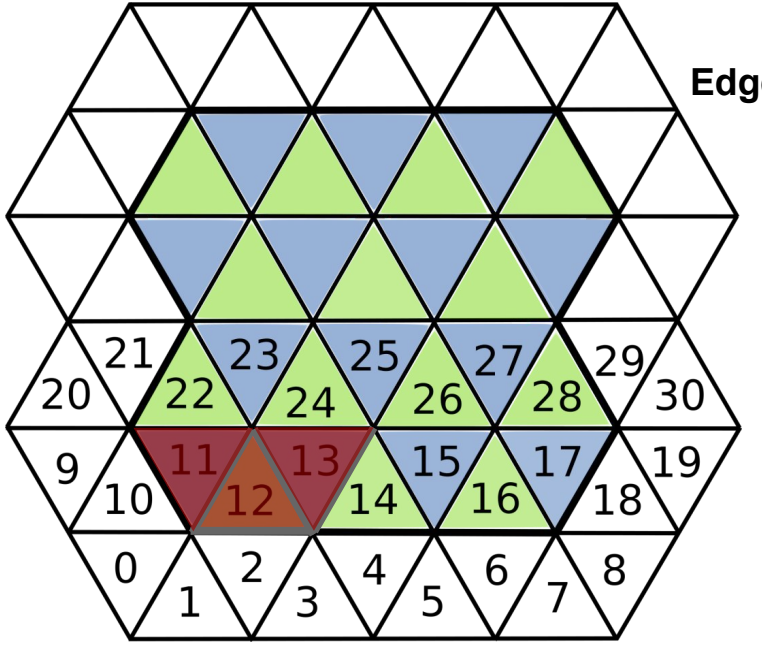


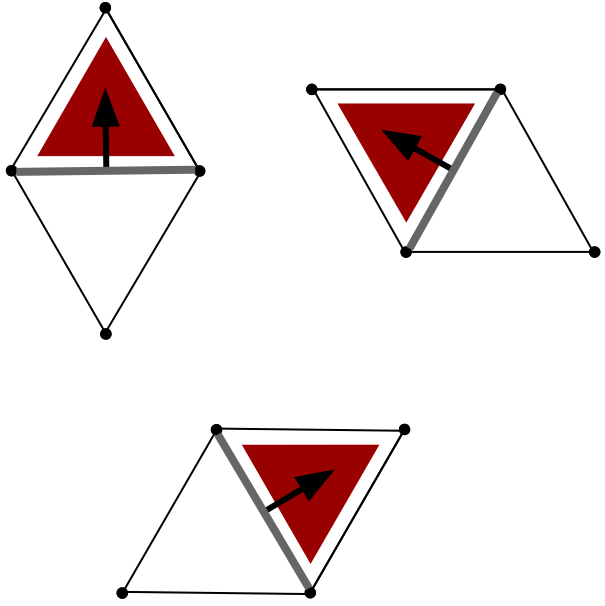
Edge > Cell



Row Major numbering. Compromise between access coalescing and locality.

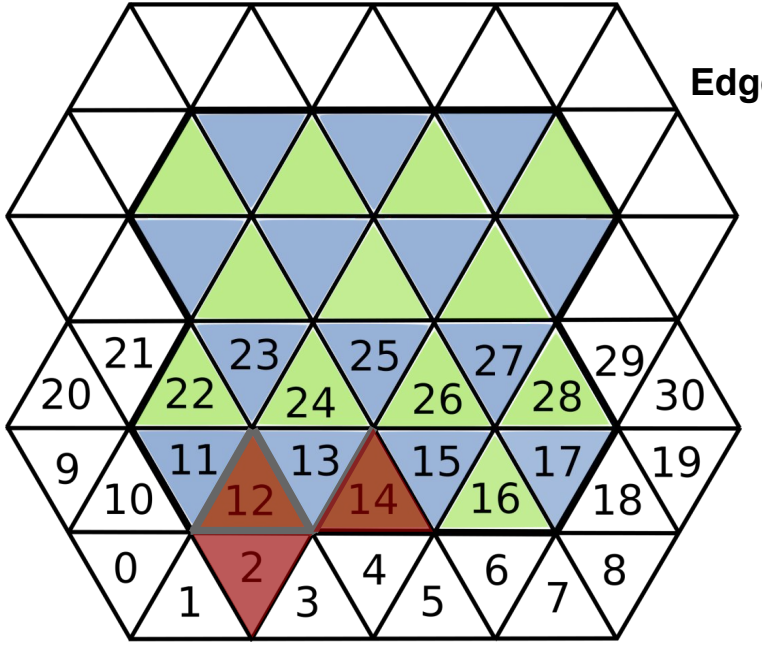
Optimization: indexing patterns

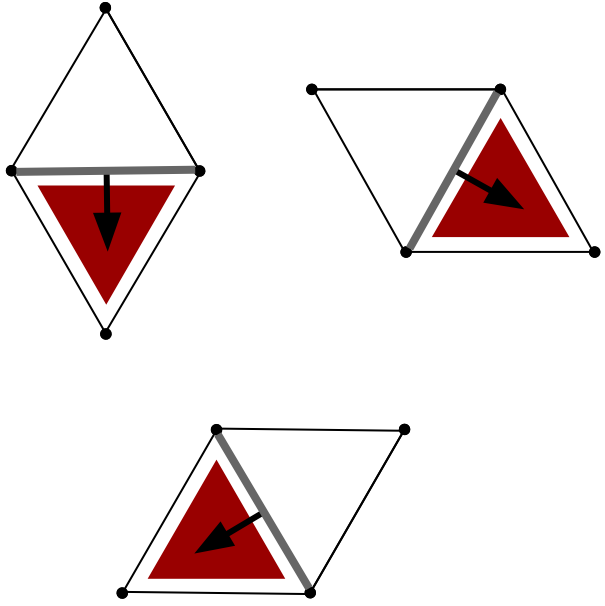




Row Major numbering. Compromise between access coalescing and locality.

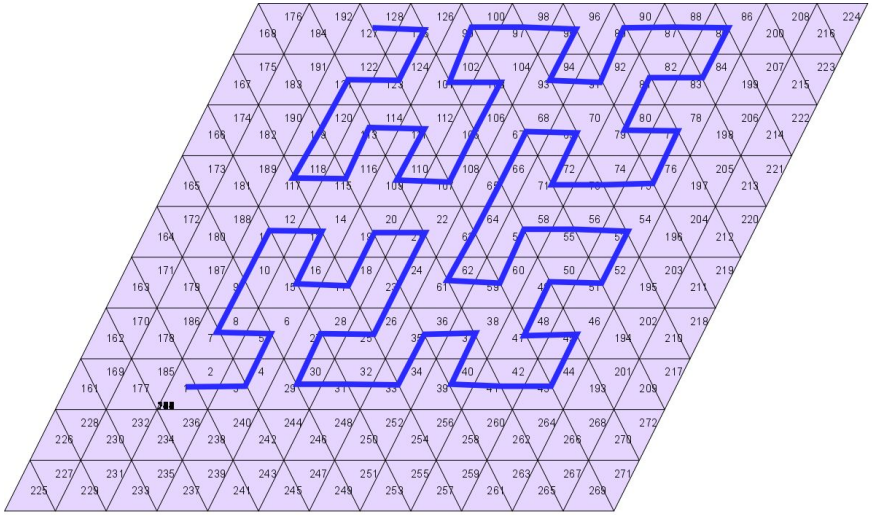
Optimization: indexing patterns





Row Major numbering. Compromise between access coalescing and locality.

Optimization: indexing patterns



A space filling curve provides the maximum data locality, to the benefit of cache efficiency. However access coalescing is almost absent.

Case Study: ICON's "diamond" stencil

- Code extracted from ICON's dycore computing a Laplacian and a Smagorinsky coefficient.
- Only one local neighborhood is used: **Edge > Cell > Vertex** (graphically looks like a *diamond*).
- Taking timings with a 340x340x80 grid, i.e. ~174k edges and 80 k-levels
- 1 NVIDIA V100, compiling with CUDA Toolkit release 10.1
- Baseline of 13 CUDA kernels
- Manually applying optimizations one at a time
- Keep in mind that it's a single, limited example. Other stencils might not give the same results.

Case Study: ICON's "diamond" stencil

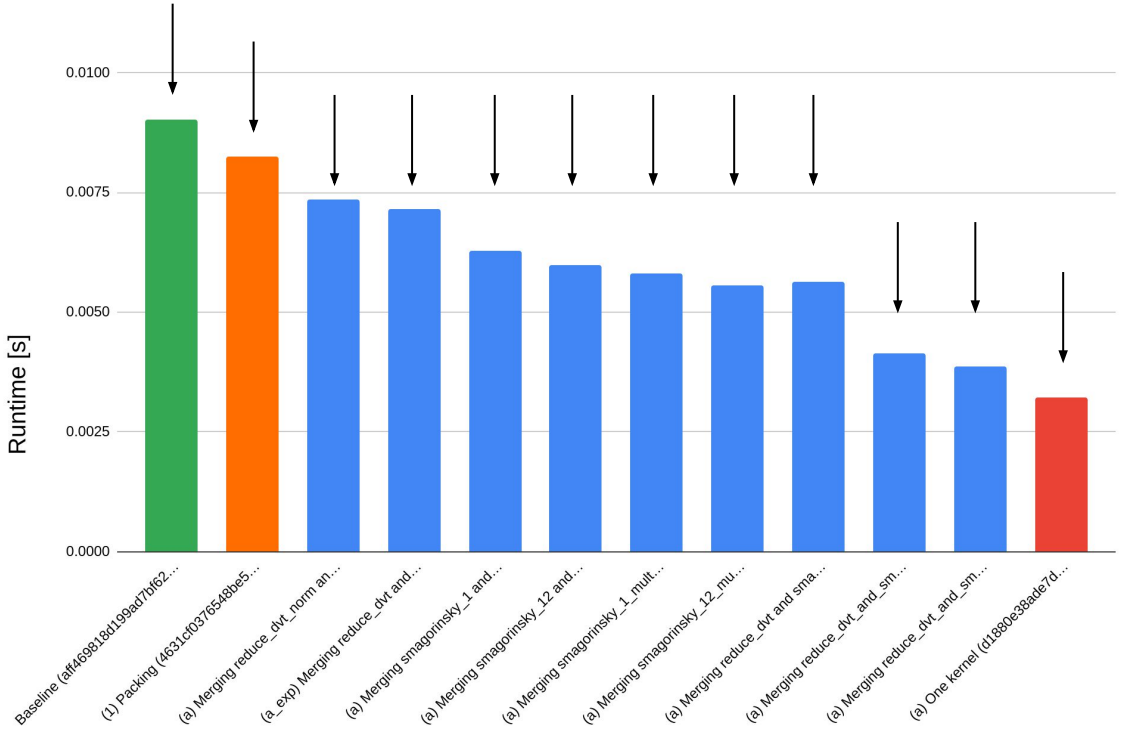
Baseline

Pack vectors

Fuse dense loops

Fuse reductions

Fuse reductions and sparse loop



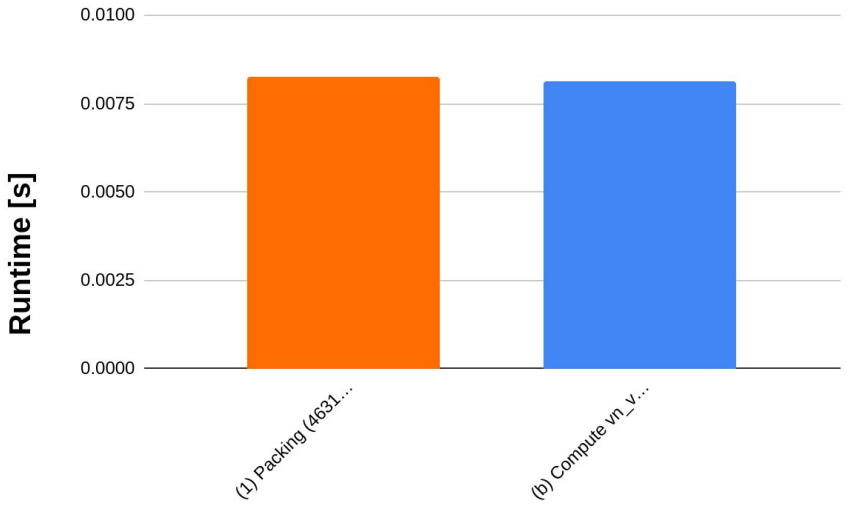
Std dev of measurements around 10^{-5}

Case Study: ICON's "diamond" stencil

Recurrent stencil inlining of sparse temporary field.

Computation is a dot product between 2 vectors and its result is required by 3 kernels.

Very little improvement, maybe an unlucky case.



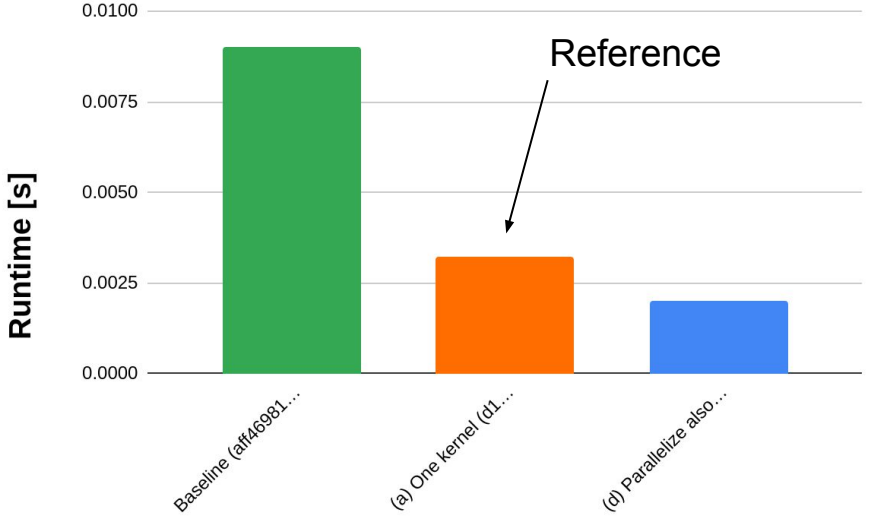
Case Study: ICON's "diamond" stencil

Parallelizing also the k-loop.

Great improvement despite the fact that number of edges (and thus of threads before the opt.) is very big.

Improvement due to a better warp scheduling.

Avg inst/cycle almost doubled.

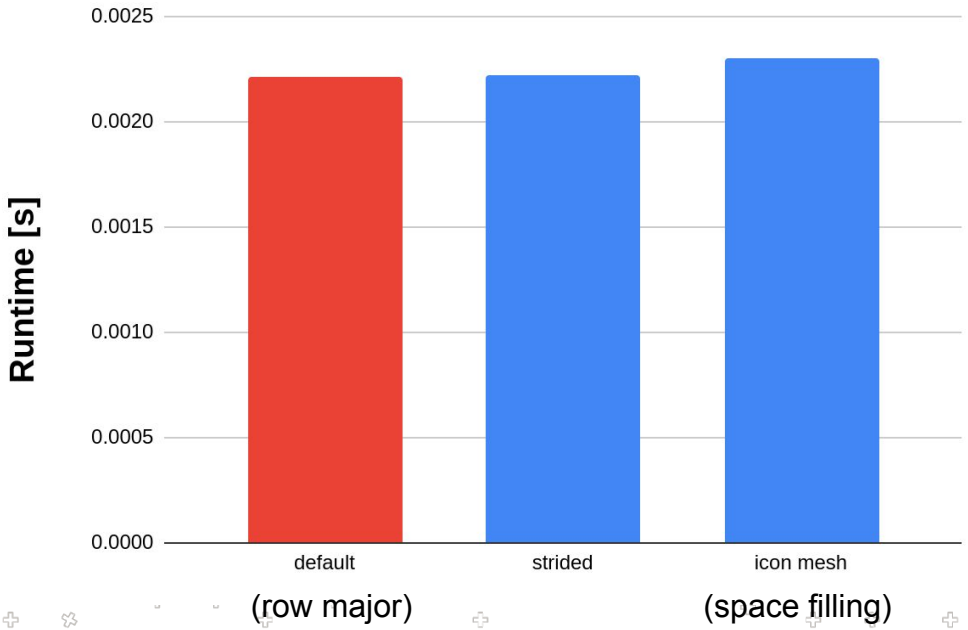


Case Study: ICON's "diamond" stencil

Trying different indexing patterns.

Space filling curve pattern is ICON's one. Performing slightly worse than the others.

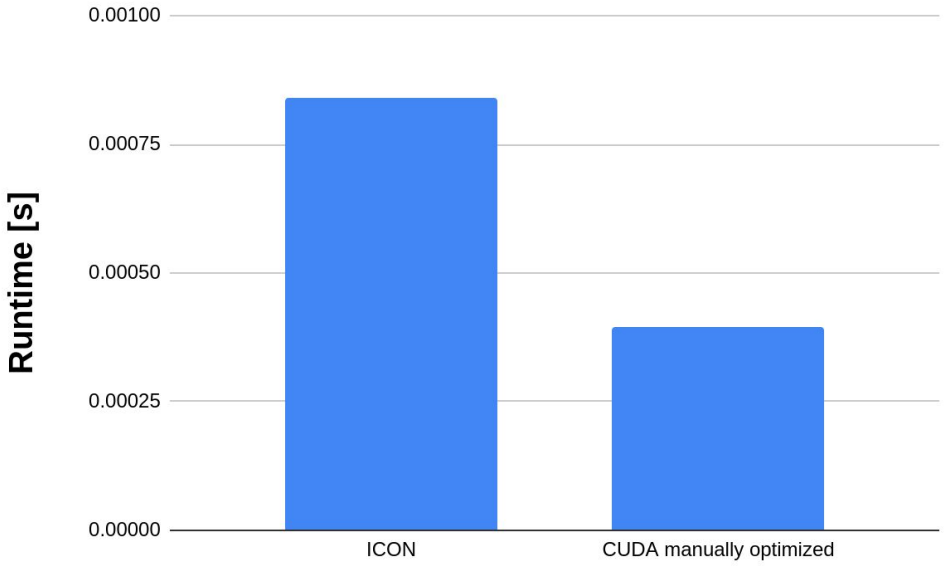
Overall, differences not very noticeable.



Case Study: ICON's "diamond" stencil

P100, ~28k edges, 64 k-levels

All optimizations combined (packing + fusing + parallelize k-loop + row-major indexing) vs ICON OpenACC original stencil performance.



State of Dawn's optimizer

Currently supports:

- Fusing dense "loops"
- Parallelizing k-loops

To be added:

- Fusing reductions and sparse loops
- One-time stencil inlining
- Recurrent stencil inlining
- Vector packing

Indexing patterns are implicit in the fields' storages, which are provided externally (transparent to Dawn).

Outlook

- Results got so far are promising
- There's still a lot to experiment: trying other stencils, testing all the optimizations devised and coming up with others
- Dawn's optimizer is still work in progress, e.g. need appropriate data structures to represent fusion of reductions/sparse loops
- Still need to consider splitting the compute domain to run on several GPUs/nodes, halo exchanges and so on...

Q&A

Questions?

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Home Affairs FDHA
Federal Office of Meteorology and Climatology MeteoSwiss

MeteoSwiss
Operation Center 1
CH-8058 Zurich-Airport
T +41 58 460 91 11
www.meteoswiss.ch

MeteoSvizzera
Via ai Monti 146
CH-6605 Locarno-Monti
T +41 58 460 92 22
www.meteosvizzera.ch

MétéoSuisse
7bis, av. de la Paix
CH-1211 Genève 2
T +41 58 460 98 88
www.meteosuisse.ch

MétéoSuisse
Chemin de l'Aérologie
CH-1530 Payerne
T +41 58 460 94 44
www.meteosuisse.ch