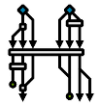
A High-level API for End-to-end Data Compression in Multi-GPU Cluster Applications

Gabriel Mitterrutzner, Peter Thoman, Philipp Gschwandtner



HLPP 2025

Motivation & Background

Motivation

- Integration of GPUs in HPC clusters apparent
  - Efficient usage challenging (MPI + X)
    - Deep knowledge of hardware-specific optimizations

  → Higher level programming models or runtime systems
    - Eg. SkePu, Müsli, Celerity, …
    - Mostly focus on compute-bound task

- Bandwidth-bound tasks can be challenging

  → Compression

Celerity

- Goal: Create a high-level, user-friendly API, simplifying programming for multi-GPU clusters, while keeping a competitive performance.

- Some prior research:
  - Celerity: High-Level C++ for Accelerator Clusters [Euro-Par 2019]
  - An Asynchronous Dataflow-Driven Execution Model For Distributed Accelerator Computing [CCGRID 2023]
  - Automatic Discovery of Collective Communication Patterns in Parallelized Task Graphs [HLPP 23]
  - …

Celerity

```
using Point = sycl::vec<double, 3>;

celerity::queue queue;
celerity::buffer<Point, 1> points_buffer(points.data(), points.size());
celerity::buffer<Point, 3> tile_points({width, height, points_per_tile});
celerity::buffer<int, 2> tile_point_count({width, height});

queue.submit([&](celerity::handler& cgh) {
  celerity::accessor point_acc{points_buffer, cgh, celerity::access::all{}, celerity::read_only};
  celerity::accessor tile_point_acc{tile_points, cgh, full_third_dim<3>{}, celerity::write_only, celerity::no_init};
  celerity::accessor points_per_tile_accumulator{tile_point_count, cgh, three_d_to_two_d<2>{}, celerity::write_only, celerity::no_init};

  auto tile_points_range = tile_points.get_range();
  celerity::range<3> range = celerity::range<3>{tile_points_range[0], tile_points_range[1], work_items_per_tile};
  cgh.parallel_for(celerity::nd_range<3>(range, celerity::range<3>(1, 1, work_items_per_tile)), [=](celerity::nd_item<3> item) {
    size_t points_per_local_item = ((points_size / work_items_per_tile) + 1);
    auto global_id = item.get_global_id();

    for(size_t i = 0; i < points_per_local_item; i++) {
      size_t idx = i * work_items_per_tile + item.get_local_id(2);

      Point p = point_acc[idx];

      int pos_x = (p.x() - x_min) / tile_size;
      int pos_y = (p.y() - y_min) / tile_size;

      if(point_is_in_current_tile) {
        sycl::atomic_ref<...> atomic_ref_count{points_per_tile_accumulator[{global_id[0], global_id[1]}]};

        int x = atomic_ref_count.fetch_add(1);

        tile_point_acc[{global_id[0], global_id[1], x}] = point_acc[idx];
      }
    }
  });
});
```

* code for demonstration purposes, some variables left out.
  Parts colored in green are abbreviated for the sake of simplicity.

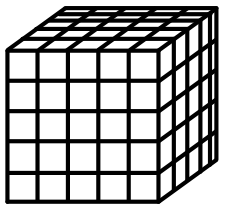
Celerity

```
1  using Point = sycl::vec<double, 3>;
2
3  celerity::queue queue;
4  celerity::buffer<Point, 1> points_buffer(points.data(), points.size());
5  celerity::buffer<Point, 3> tile_points({width, height, points_per_tile});
6  celerity::buffer<int, 2> tile_point_count({width, height});
7
8  queue.submit([&](celerity::handler& cgh) {
9    celerity::accessor point_acc{points_buffer, cgh, celerity::access::all{},
       celerity::read_only};
10   celerity::accessor tile_point_acc{tile_points, cgh, full_third_dim<3>{},
       celerity::write_only, celerity::no_init};
11   celerity::accessor points_per_tile_accumulator{tile_point_count, cgh,
       three_d_to_two_d<2>{}, celerity::write_only, celerity::no_init};
12
13   auto tile_points_range = tile_points.get_range();
14   celerity::range<3> range = celerity::range<3>{tile_points_range[0],
       tile_points_range[1], work_items_per_tile};
15   cgh.parallel_for(celerity::nd_range<3>(range, celerity::range<3>(1, 1,
       work_items_per_tile)), [=](celerity::nd_item<3> item) {
16     size_t points_per_local_item = ((points_size / work_items_per_tile) + 1);
17     auto global_id = item.get_global_id();
18
19     for(size_t i = 0; i < points_per_local_item; i++) {
20       size_t idx = i * work_items_per_tile + item.get_local_id(2);
21
22       Point p = point_acc[idx];
23
24       int pos_x = (p.x() - x_min) / tile_size;
25       int pos_y = (p.y() - y_min) / tile_size;
26
27       if(point_is_in_current_tile) {
28         sycl::atomic_ref<...> atomic_ref_count{points_per_tile_accumulator[{
       global_id[0], global_id[1]}]};
29
30         int x = atomic_ref_count.fetch_add(1);
31
32         tile_point_acc[{global_id[0], global_id[1], x}] = point_acc[idx];
33       }
34     }
35   });
36 });
```

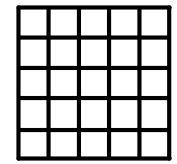
```
celerity::buffer<Point, 1> points_buffer(points.data(), points.size());
celerity::buffer<Point, 3> tile_points({width, height, points_per_tile});
celerity::buffer<int, 2> tile_point_count({width, height});
```

points_buffer

tile_points



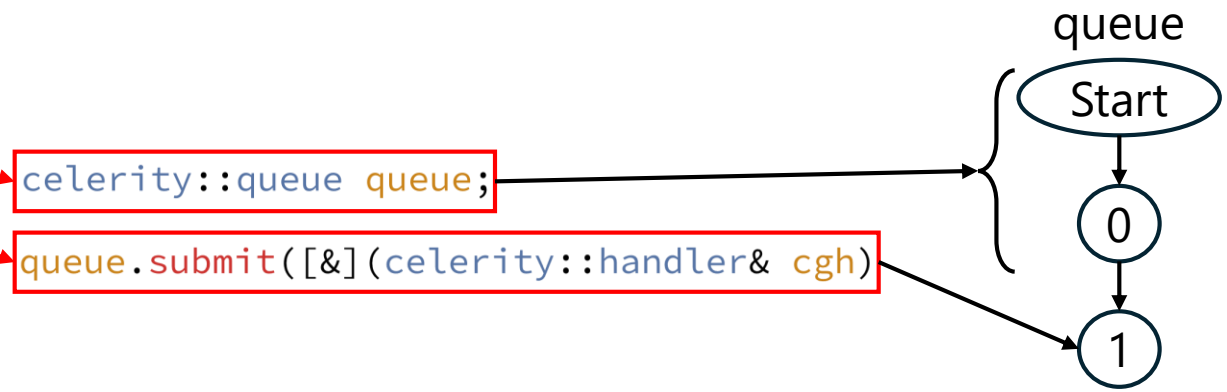
tile_points_count



Celerity
High-level C++ for Accelerator Clusters

Celerity

```
using Point = sycl::vec<double, 3>;

celerity::queue queue;
celerity::buffer<Point, 1> points_buffer(points.data(), points.size());
celerity::buffer<Point, 3> tile_points({width, height, points_per_tile});
celerity::buffer<int, 2> tile_point_count({width, height});

queue.submit([&](celerity::handler& cgh) {
  celerity::accessor point_acc{points_buffer, cgh, celerity::access::all{},
    celerity::read_only};
  celerity::accessor tile_point_acc{tile_points, cgh, full_third_dim<3>{},
    celerity::write_only, celerity::no_init};
  celerity::accessor points_per_tile_accumulator{tile_point_count, cgh,
    three_d_to_two_d<2>{}, celerity::write_only, celerity::no_init};

  auto tile_points_range = tile_points.get_range();
  celerity::range<3> range = celerity::range<3>{tile_points_range[0],
    tile_points_range[1], work_items_per_tile};
  cgh.parallel_for(celerity::nd_range<3>(range, celerity::range<3>(1, 1,
    work_items_per_tile)), [=](celerity::nd_item<3> item) {
    size_t points_per_local_item = ((points_size / work_items_per_tile) + 1);
    auto global_id = item.get_global_id();

    for(size_t i = 0; i < points_per_local_item; i++) {
      size_t idx = i * work_items_per_tile + item.get_local_id(2);

      Point p = point_acc[idx];

      int pos_x = (p.x() - x_min) / tile_size;
      int pos_y = (p.y() - y_min) / tile_size;

      if(point_is_in_current_tile) {
        sycl::atomic_ref<...> atomic_ref_count{points_per_tile_accumulator[{
    global_id[0], global_id[1]}]};

        int x = atomic_ref_count.fetch_add(1);

        tile_point_acc[{global_id[0], global_id[1], x}] = point_acc[idx];
      }
    }
  });
});
```



Celerity

```
using Point = sycl::vec<double, 3>;

celerity::queue queue;
celerity::buffer<Point, 1> points_buffer(points.data(), points.size());
celerity::buffer<Point, 3> tile_points({width, height, points_per_tile});
celerity::buffer<int, 2> tile_point_count({width, height});

queue.submit([&](celerity::handler& cgh) {
  celerity::accessor point_acc{points_buffer, cgh, celerity::access::all{},
     celerity::read_only};
  celerity::accessor tile_point_acc{tile_points, cgh, full_third_dim<3>{},
     celerity::write_only, celerity::no_init};
  celerity::accessor points_per_tile_accumulator{tile_point_count, cgh,
     three_d_to_two_d<2>{}, celerity::write_only, celerity::no_init};

  auto tile_points_range = tile_points.get_range();
  celerity::range<3> range = celerity::range<3>{tile_points_range[0],
     tile_points_range[1], work_items_per_tile};
  cgh.parallel_for(celerity::nd_range<3>(range, celerity::range<3>(1, 1,
     work_items_per_tile)), [=](celerity::nd_item<3> item) {
    size_t points_per_local_item = ((points_size / work_items_per_tile) + 1);
    auto global_id = item.get_global_id();

    for(size_t i = 0; i < points_per_local_item; i++) {
      size_t idx = i * work_items_per_tile + item.get_local_id(2);

      Point p = point_acc[idx];

      int pos_x = (p.x() - x_min) / tile_size;
      int pos_y = (p.y() - y_min) / tile_size;

      if(point_is_in_current_tile) {
        sycl::atomic_ref<...> atomic_ref_count{points_per_tile_accumulator[{
    global_id[0], global_id[1]}]};

        int x = atomic_ref_count.fetch_add(1);

        tile_point_acc[{global_id[0], global_id[1], x}] = point_acc[idx];
      }
    }
  });
});
```

```
cgh.parallel_for(celerity::nd_range<3>(range, celerity::range<3>(1, 1,
   work_items_per_tile)), [=](celerity::nd_item<3> item) {
  size_t points_per_local_item = ((points_size / work_items_per_tile) + 1);
  auto global_id = item.get_global_id();

  for(size_t i = 0; i < points_per_local_item; i++) {
    size_t idx = i * work_items_per_tile + item.get_local_id(2);

    Point p = point_acc[idx];

    int pos_x = (p.x() - x_min) / tile_size;
    int pos_y = (p.y() - y_min) / tile_size;

    if(point_is_in_current_tile) {
      sycl::atomic_ref<...> atomic_ref_count{points_per_tile_accumulator[{
   global_id[0], global_id[1]}]};

      int x = atomic_ref_count.fetch_add(1);

      tile_point_acc[{global_id[0], global_id[1], x}] = point_acc[idx];
    }
  }
});
```

Kernel

Celerity
High-level C++ for Accelerator Clusters

Celerity

```
using Point = sycl::vec<double, 3>;

celerity::queue queue;
celerity::buffer<Point, 1> points_buffer(points.data(), points.size());
celerity::buffer<Point, 3> tile_points({width, height, points_per_tile});
celerity::buffer<int, 2> tile_point_count({width, height});

queue.submit([&](celerity::handler& cgh) {
  celerity::accessor point_acc{points_buffer, cgh, celerity::access::all{},
    celerity::read_only};
  celerity::accessor tile_point_acc{tile_points, cgh, full_third_dim<3>{},
    celerity::write_only, celerity::no_init};
  celerity::accessor points_per_tile_accumulator{tile_point_count, cgh,
    three_d_to_two_d<2>{}, celerity::write_only, celerity::no_init};

  auto tile_points_range = tile_points.get_range();
  celerity::range<3> range = celerity::range<3>{tile_points_range[0],
    tile_points_range[1], work_items_per_tile};
  cgh.parallel_for(celerity::nd_range<3>(range, celerity::range<3>(1, 1,
    work_items_per_tile)), [=](celerity::nd_item<3> item) {
    size_t points_per_local_item = ((points_size / work_items_per_tile) + 1);
    auto global_id = item.get_global_id();

    for(size_t i = 0; i < points_per_local_item; i++) {
      size_t idx = i * work_items_per_tile + item.get_local_id(2);

      Point p = point_acc[idx];

      int pos_x = (p.x() - x_min) / tile_size;
      int pos_y = (p.y() - y_min) / tile_size;

      if(point_is_in_current_tile) {
        sycl::atomic_ref<...> atomic_ref_count{points_per_tile_accumulator[{
    global_id[0], global_id[1]}]};

        int x = atomic_ref_count.fetch_add(1);

        tile_point_acc[{global_id[0], global_id[1], x}] = point_acc[idx];
      }
    }
  });
});
```

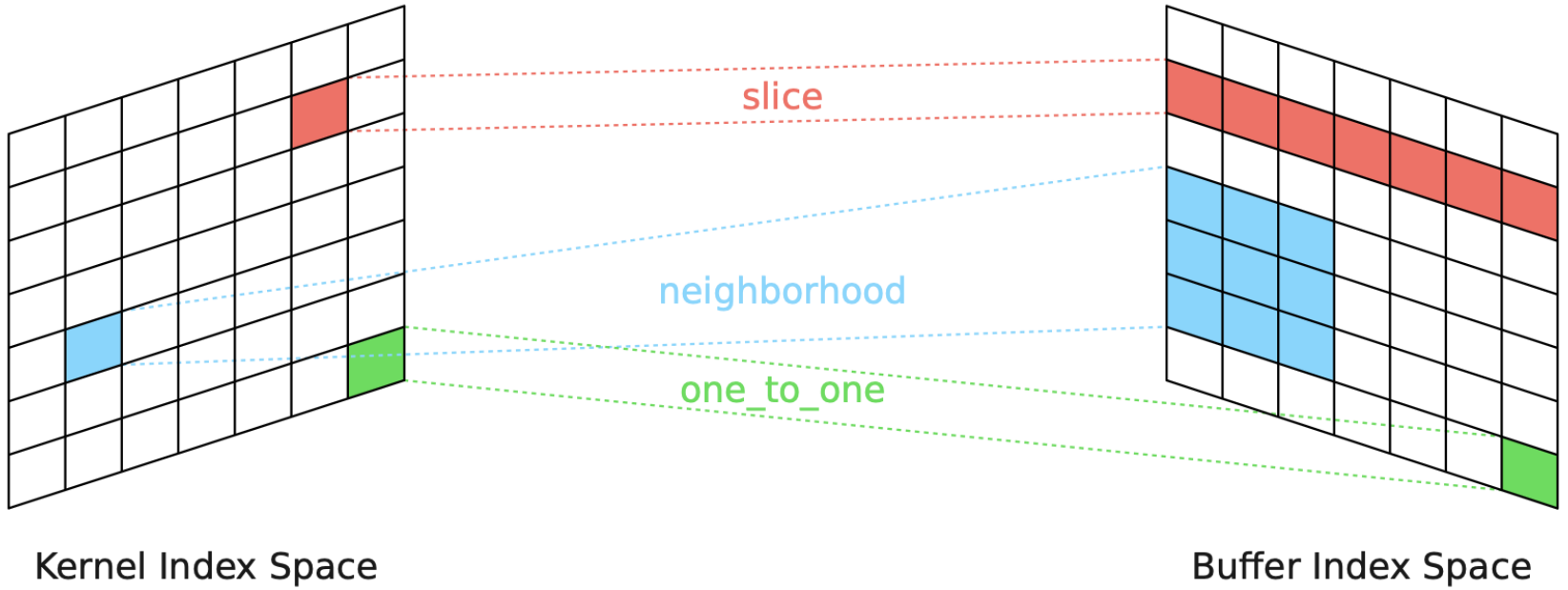
```
celerity::accessor point_acc{points_buffer, cgh, celerity::access::all{},
  celerity::read_only};
celerity::accessor tile_point_acc{tile_points, cgh, full_third_dim<3>{},
  celerity::write_only, celerity::no_init};
celerity::accessor points_per_tile_accumulator{tile_point_count, cgh,
  three_d_to_two_d<2>{}, celerity::write_only, celerity::no_init};
```

- Buffers
- Access mode
- Range mappers

Celerity
High-level C++ for Accelerator Clusters

Celerity

```
celerity::access::neighborhood{...}
celerity::access::one_to_one()
celerity::access::slice<...>(...)
```

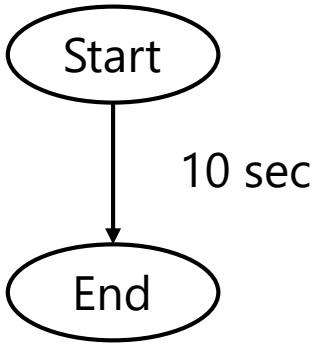


Compression API

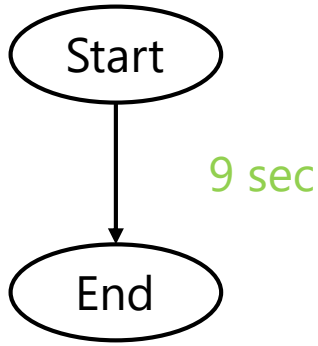
Compression API Design

Performance Ease of use Memory reduction Extensibility

without compression



with compression



Compression API Design

Performance **Ease of use** Memory reduction Extensibility

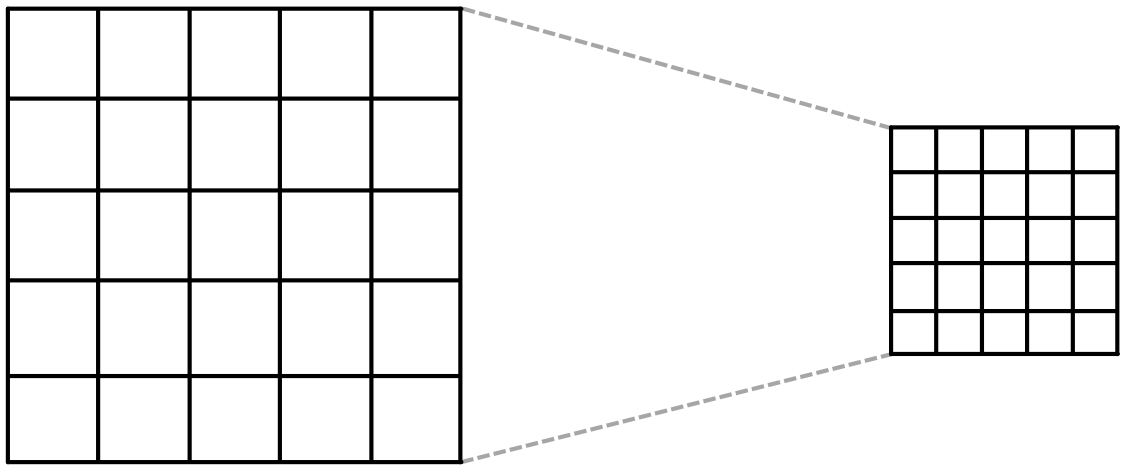
Simple declarative API at definition

```
1  using compression_type =
2    compressed<compression::compression_algorithm<...>>;
3
4  compression_type compress(...);
5
6  buffer<float, 2> matrix{{size, size}, compress};
7  buffer<float, 1> vector{{size}, compress};
8  buffer<float, 1> result{{size}, compress};
```

no changes required at the point of use

Compression API Design

Performance Ease of use **Memory reduction** Extensibility



32 bit 16 bit

Compression API Design

Performance Ease of use Memory reduction **Extensibility**

```
1  template <typename D, typename C>
2  class compressed<compression::algorithm<D, C>> {
3    ... // constructor and other functions
4    C compress(const D number) const {
5      return ...
6    }
7
8    D decompress(const C number) const {
9      return ...
10   }
11   ... // other functions
12 };
```

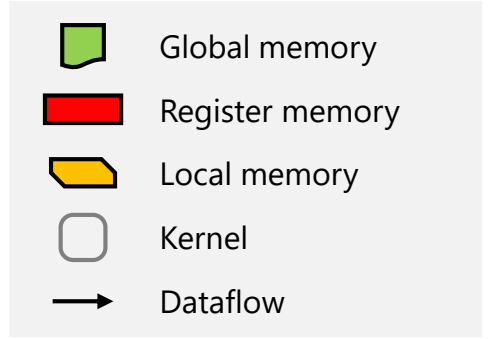
Compression Categories

Element-wise

Kernel

Global memory

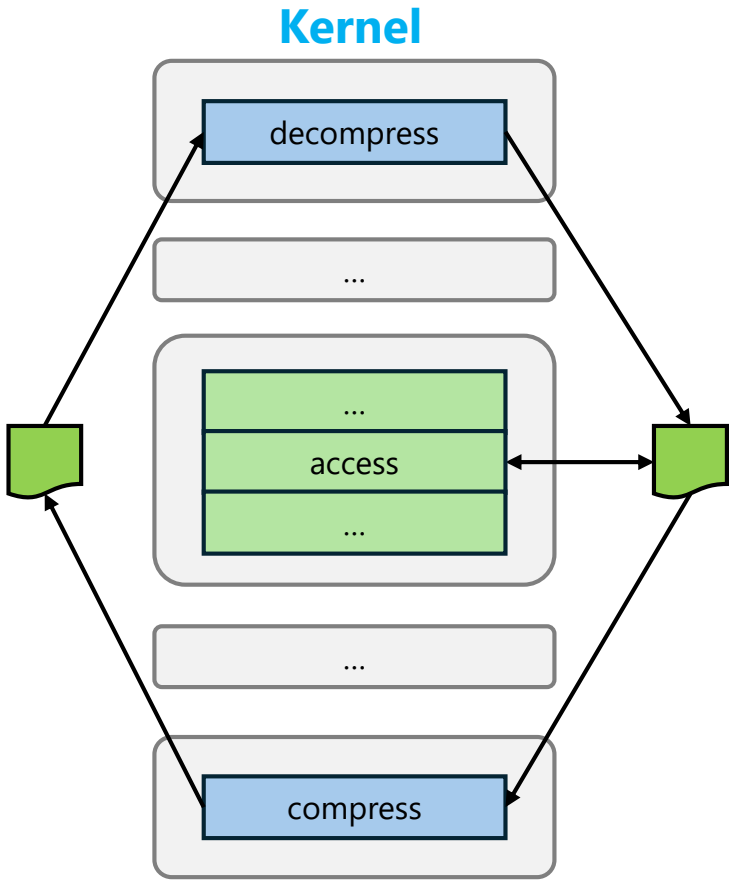
Local memory



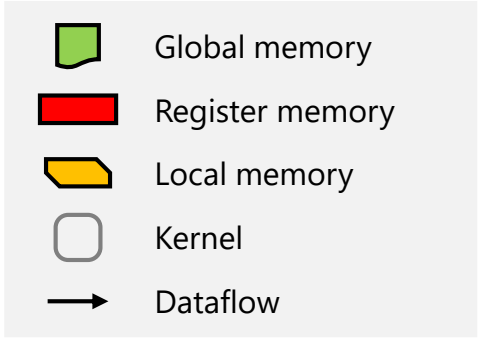
Compression Categories

Element-wise

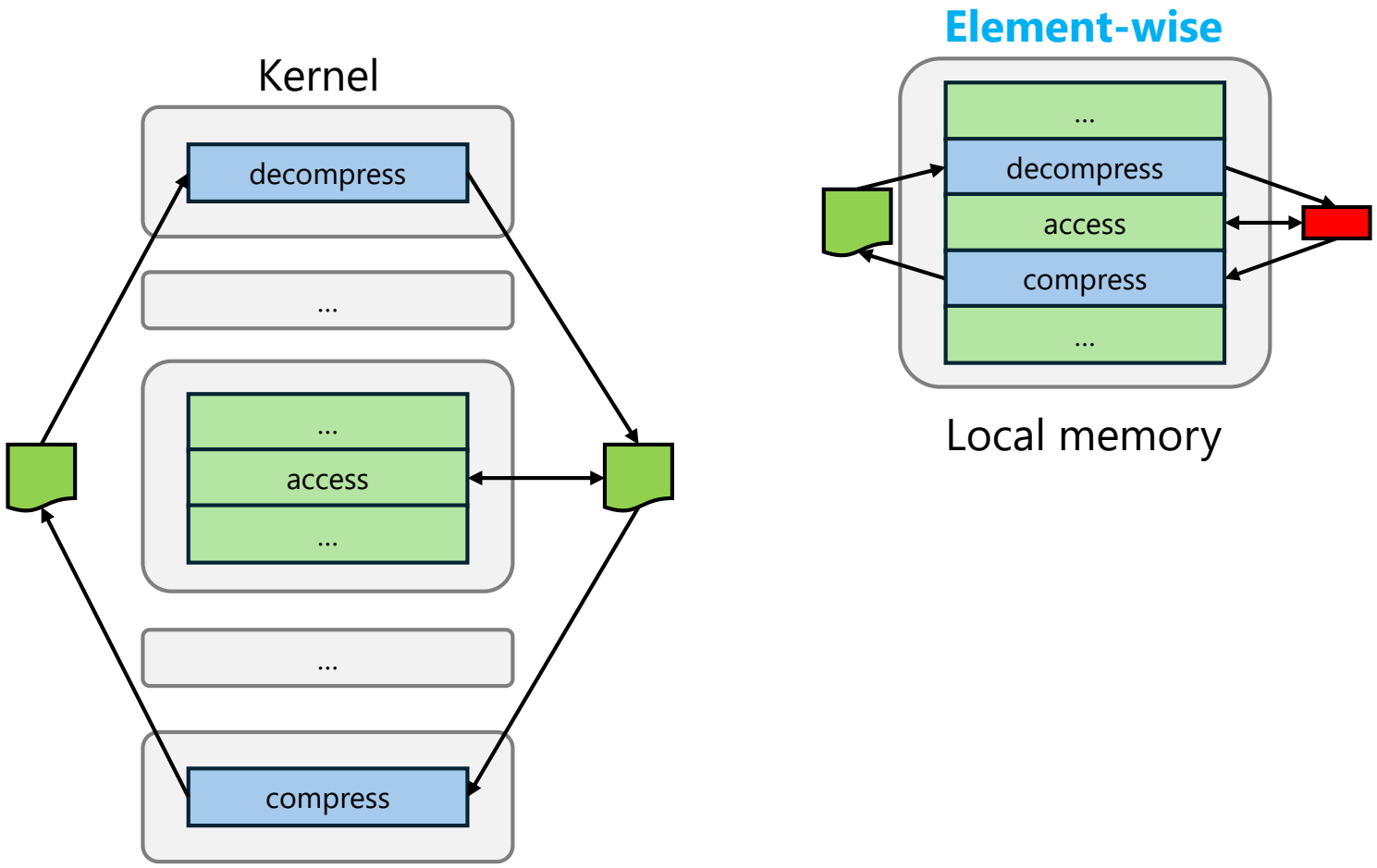
Global memory



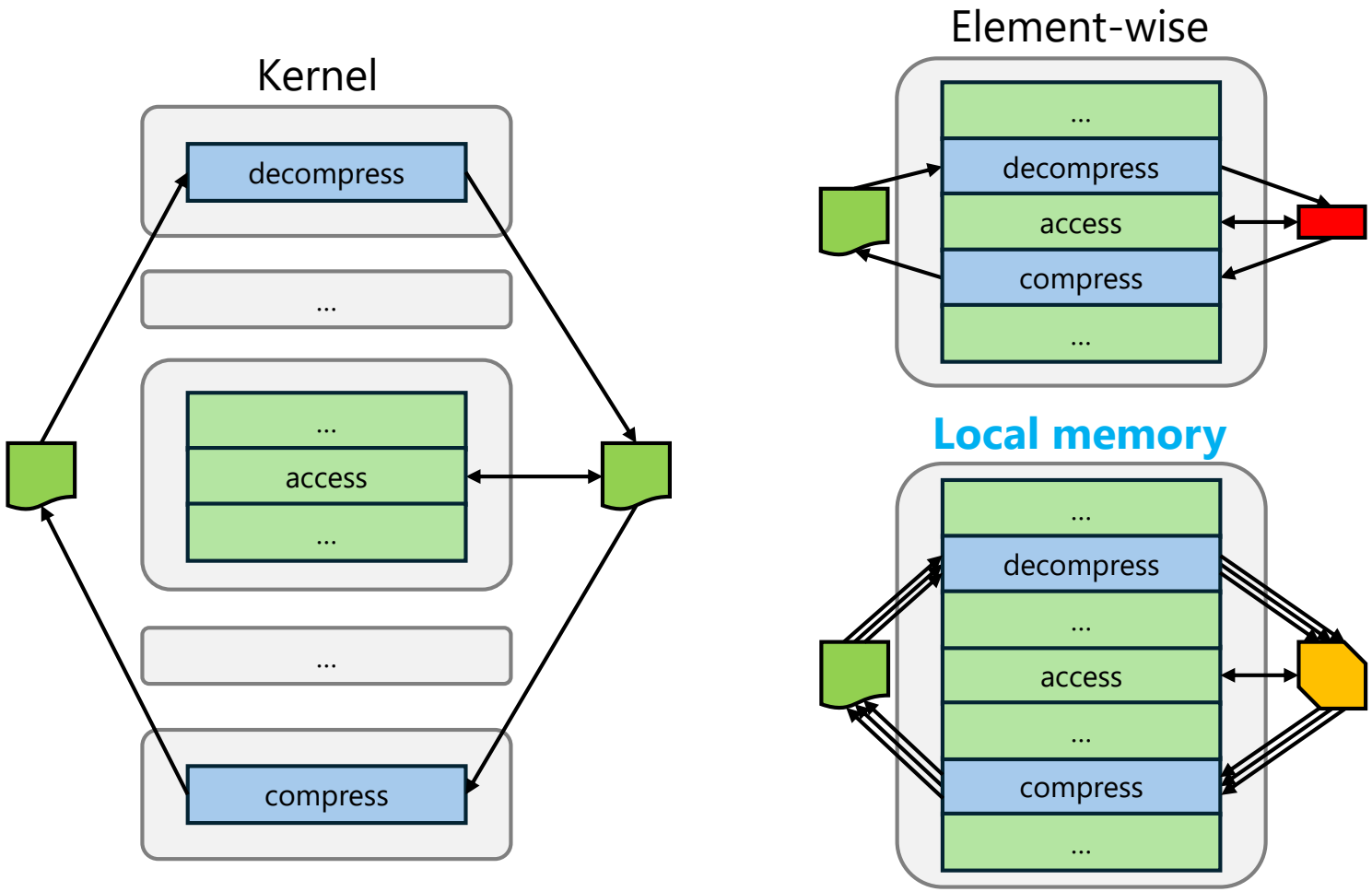
Local memory



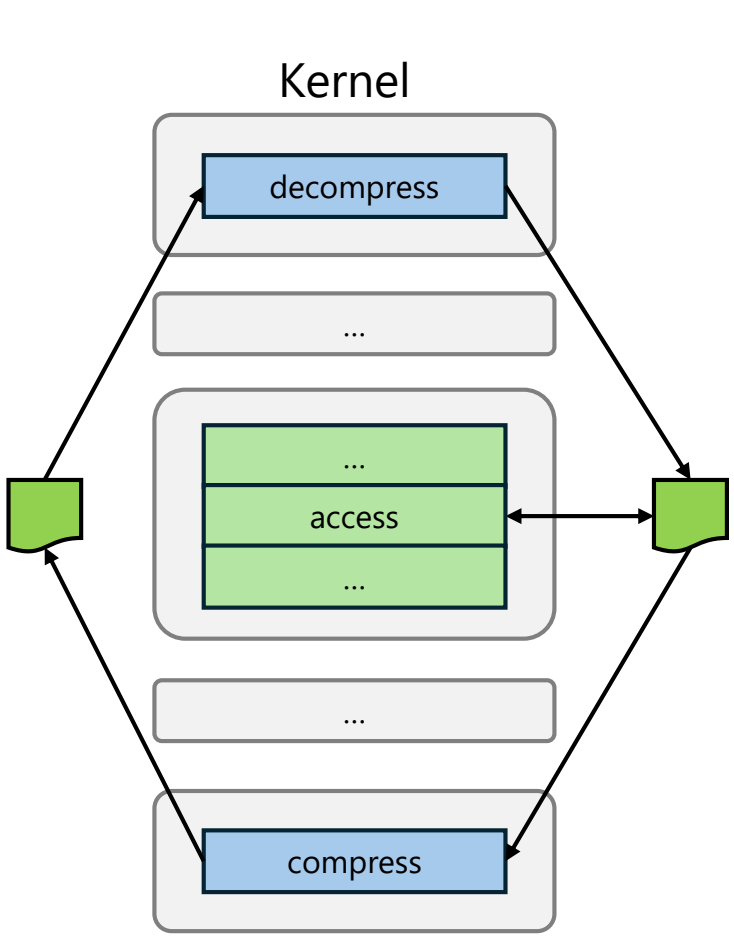
Compression Categories

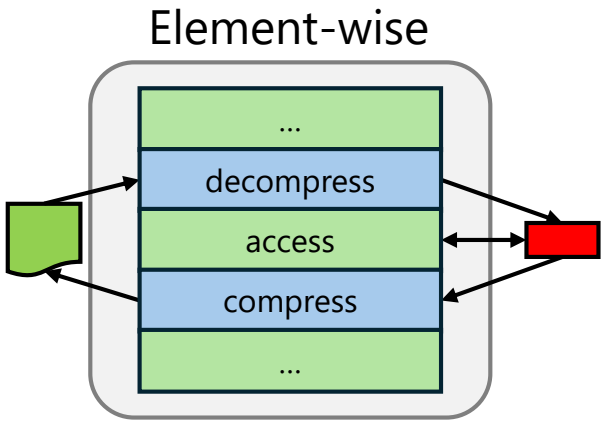


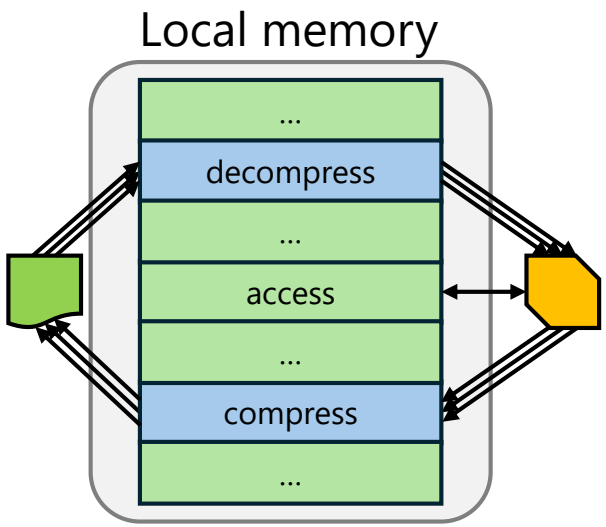
Compression Categories

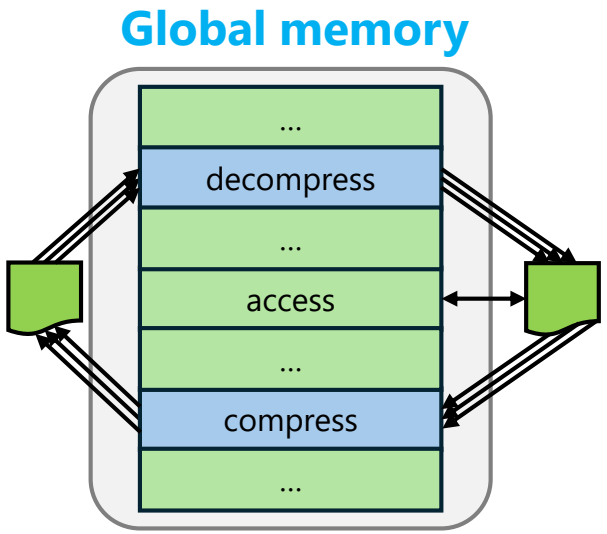


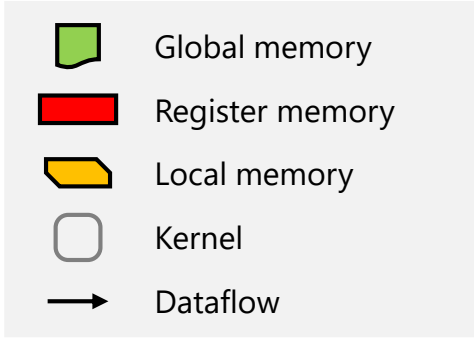
Compression Categories



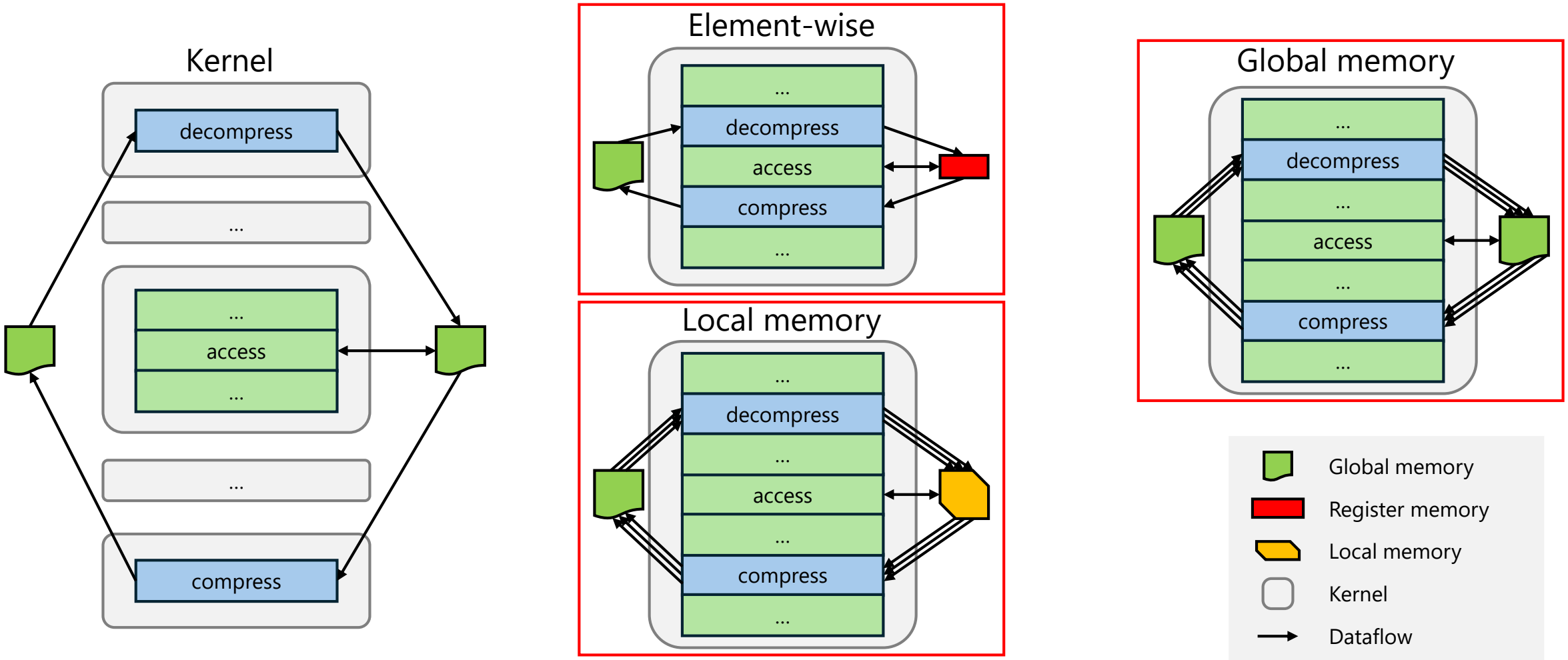






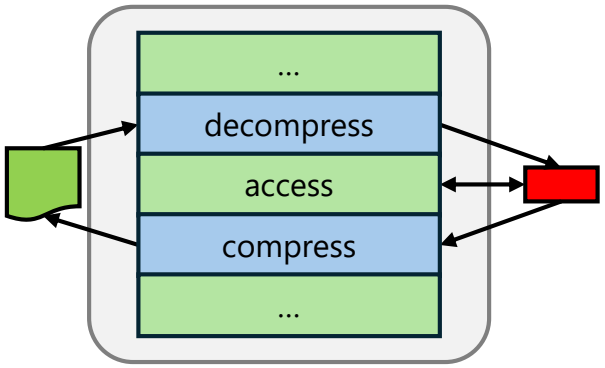


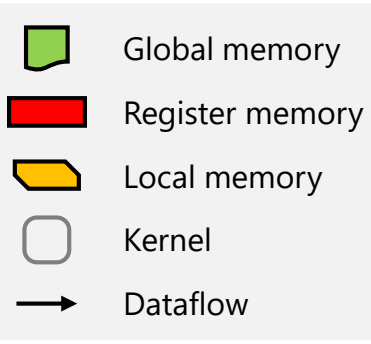
Compression Categories



Compression Categories

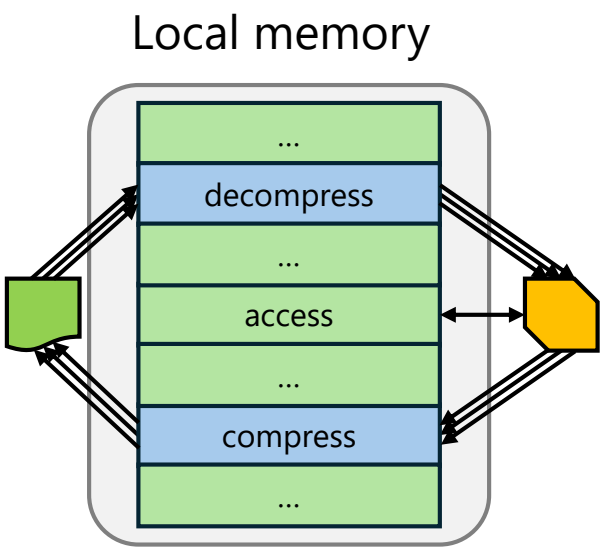
Element-wise



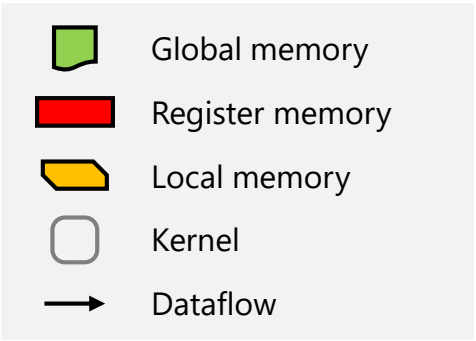



```
1  using Point = sycl::vec<double, 3>;
2  using CompressedPoint = sycl::vec<sycl::half, 3>;
3  using compression_tile_type = celerity::compressed<celerity::compression::
       point_cloud<Point, CompressedPoint>>;
4  ...
5  compression_tile_type compression_tile{{x_min, y_min, 0}, tile_size};
6  celerity::buffer<Point, 3, compression_tile_type> tile_points({width, height,
         points_per_tile}, compression_tile);
7  ...
8  queue.submit([&](celerity::handler& cgh) {
9    ...
10   celerity::accessor tile_point_acc{tile_points, cgh, full_third_dim<3>{},
       celerity::write_only, celerity::no_init};
11   ...
12   cgh.parallel_for(celerity::nd_range<3>(range, celerity::range<3>(1, 1,
       work_items_per_tile)), [=](celerity::nd_item<3> item) {
13     ...
14     for(size_t i = 0; i < points_per_local_item; i++) {
15       ...
16       if(point_is_in_current_tile) {
17         ...
18         tile_point_acc[{global_id[0], global_id[1], x}] = point_acc[idx];
19       }
20     }
21   });
22 });
```

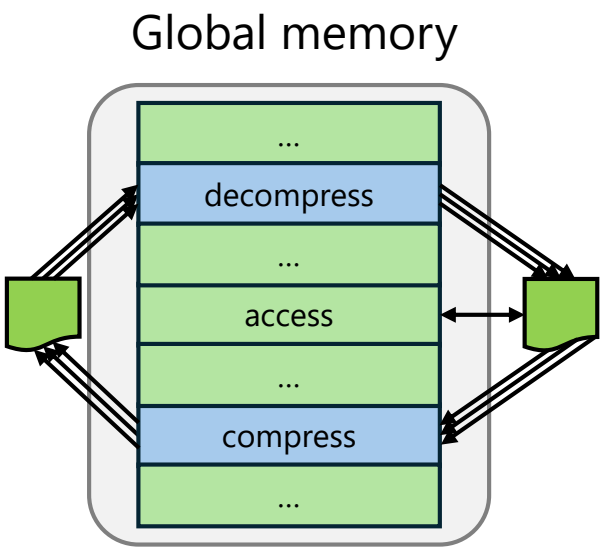
Compression Categories



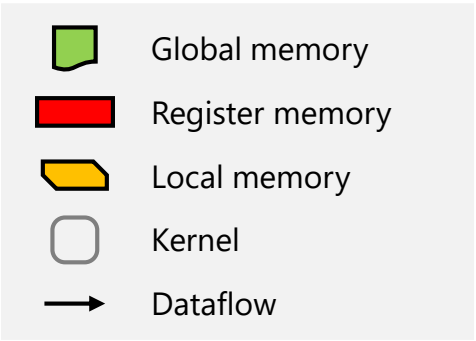

```
1  queue.submit([&](celerity::handler& cgh) {
2    ...
3    celerity::accessor tile_point_acc{tile_points, cgh, full_third_dim<3>{}, celerity::write_only,
       celerity::no_init};
4    ...
5    cgh.parallel_for(celerity::nd_range<3>(range, celerity::range<3>(1, 1, work_items_per_tile)),
       [=](celerity::nd_item<3> item) {
6      auto current_tile = tile_point_acc.decompress_data(item, points_per_tile_accumulator,
        points_per_tile, {x_min, y_min, 0}, tile_size);
7      ...
8      for(size_t i = 0; i < points_per_local_item; i++) {
9        ...
10       if(point_is_in_current_tile) {
11         ...
12         current_tile[{global_id[0], global_id[1], x}] = point_acc[idx];
13       }
14     }
15   });
16 });
```



Compression Categories



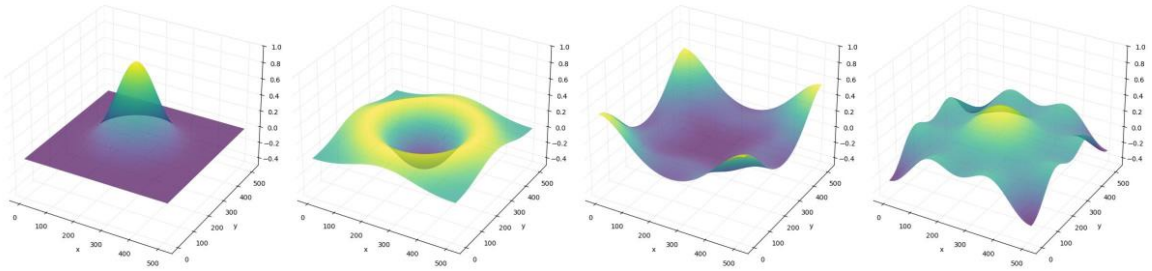

```
1  queue.submit([&](celerity::handler& cgh) {
2    ...
3    celerity::accessor tile_point_acc{tile_points, cgh, full_third_dim<3>{}, celerity::write_only,
       celerity::no_init};
4    ...
5    cgh.parallel_for(celerity::nd_range<3>(range, celerity::range<3>(1, 1, work_items_per_tile)),
       [=](celerity::nd_item<3> item) {
6      auto current_tile = tile_point_acc.decompress_data(item, points_per_tile_accumulator,
        points_per_tile, {x_min, y_min, 0}, tile_size);
7      ...
8      for(size_t i = 0; i < points_per_local_item; i++) {
9        ...
10       if(point_is_in_current_tile) {
11         ...
12         current_tile[{global_id[0], global_id[1], x}] = point_acc[idx];
13       }
14     }
15   });
16 });
```



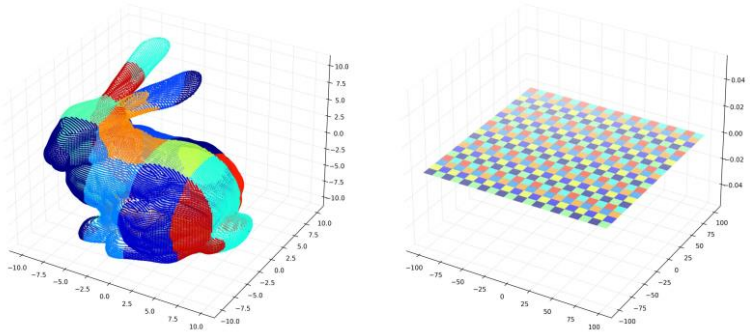
Evaluation

Pilot codes

Wave Sim

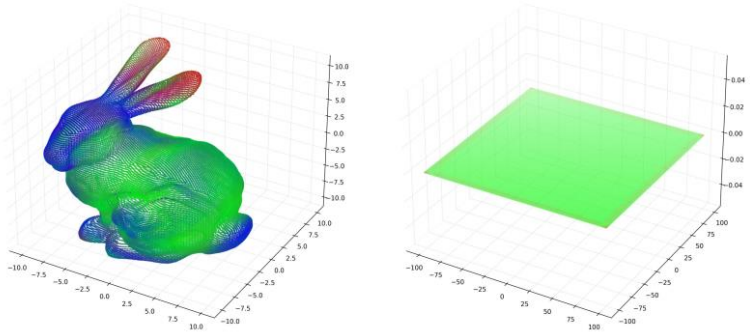


Point Cloud Tiling



RSim

Shape Factor Calculation



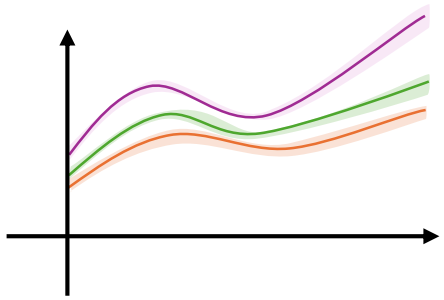
Celerity
High-level C++ for Accelerator Clusters

Experiment setup

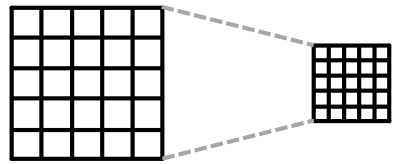
System	GPU & CPU	Memory	Software stack	Connect	Sycl
Leonardo	4x NVIDIA A100-SXM 1x Intel Xeon 8358	64 GB 512 GB	CUDA 12.1 RedHat 8.6	InfiniBand (200 Gbps)	ACPP v24.10.0
NV3090	4x NVIDIA 3090 2x AMD EPYC 7763	24 GB 1008 GB	CUDA 12.6 Ubuntu 24.04.1	16x PCIe 4.0 (32 Gbps)	

Metric

Performance

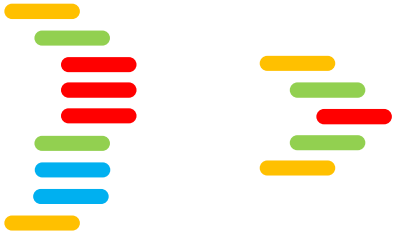


Memory reduction



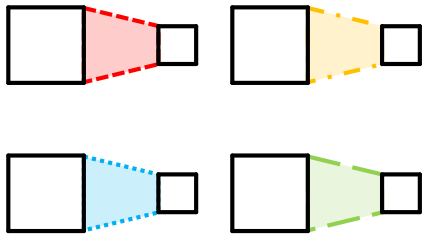
32 bit 16 bit

Ease of use



9 loc 5 loc

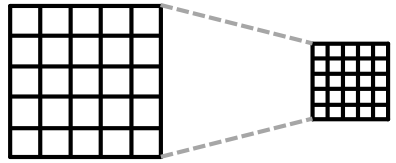
Extensibility



Ease of use

type	implementation	Tiling		Shape Factors	
		added lines	*places to change*	*added lines*	*places to change*
direct	*without API*	9	4	17	6
	with API	4	3	2	2
local	*without API*	67	6	64	10
	with API	7	3	8	7
global	*without API*	81	6	50	12
	with API	7	3	8	7

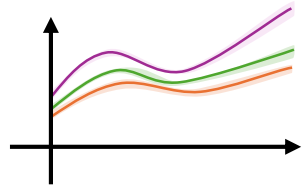
Memory reduction



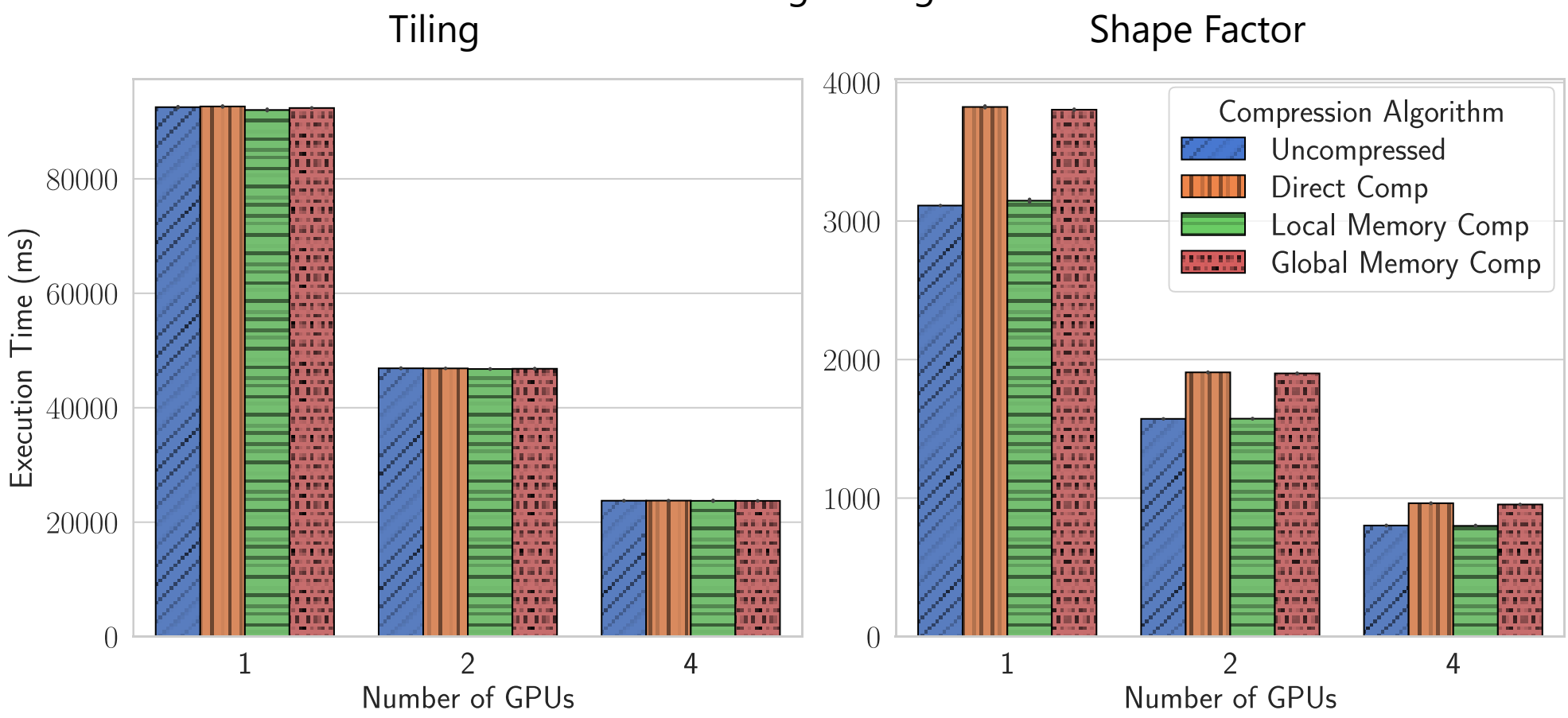
	size	Wave Sim		RSim		Tiling		Shape Factor	
		S_U	S_C	S_U	S_C	S_U	S_C	S_U	S_C
measured	*small*	2312	1351	390	320	338	312	373	318
	medium	8867	4572	862	484	535	434	671	458
	large	-	17457	5158	1692	1323	918	1864	988
theoretical	*small*	2147	1074	99	25	34	9	69	13
	medium	8590	4295	530	132	136	34	271	51
	large	34360	17180	4844	1211	540	135	1081	203

S_U = uncompressed
S_C = compressed
all numbers in MB

Performance

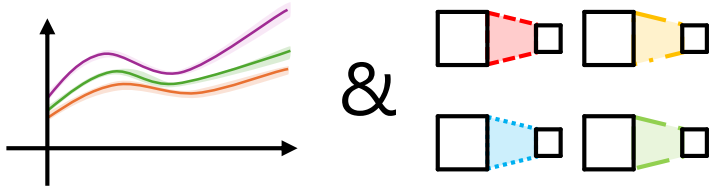


Point Cloud – Strong scaling – NV3090

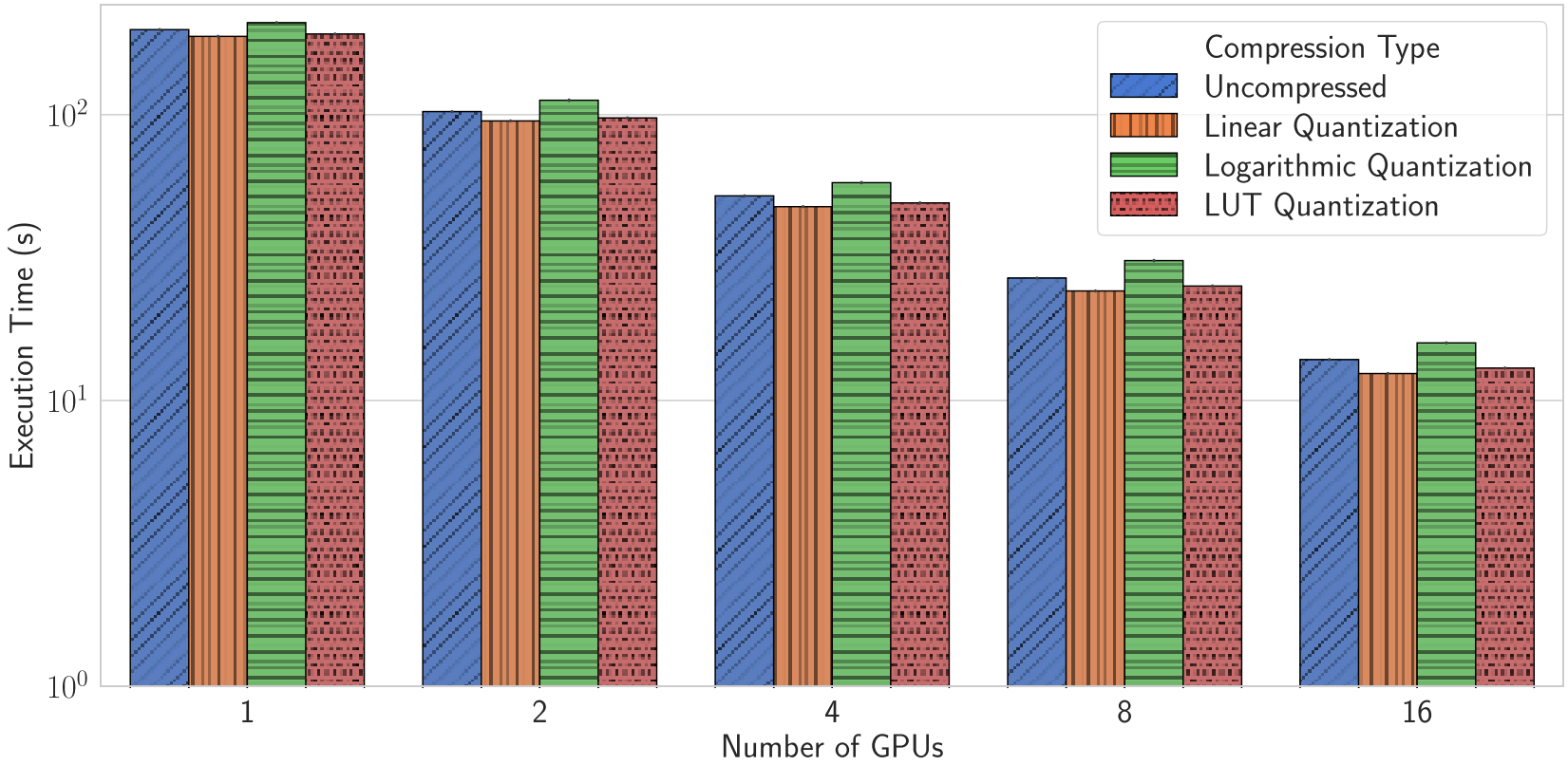


Celerity
High-level C++ for Accelerator Clusters

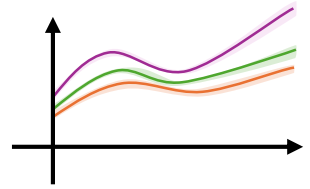
Performance

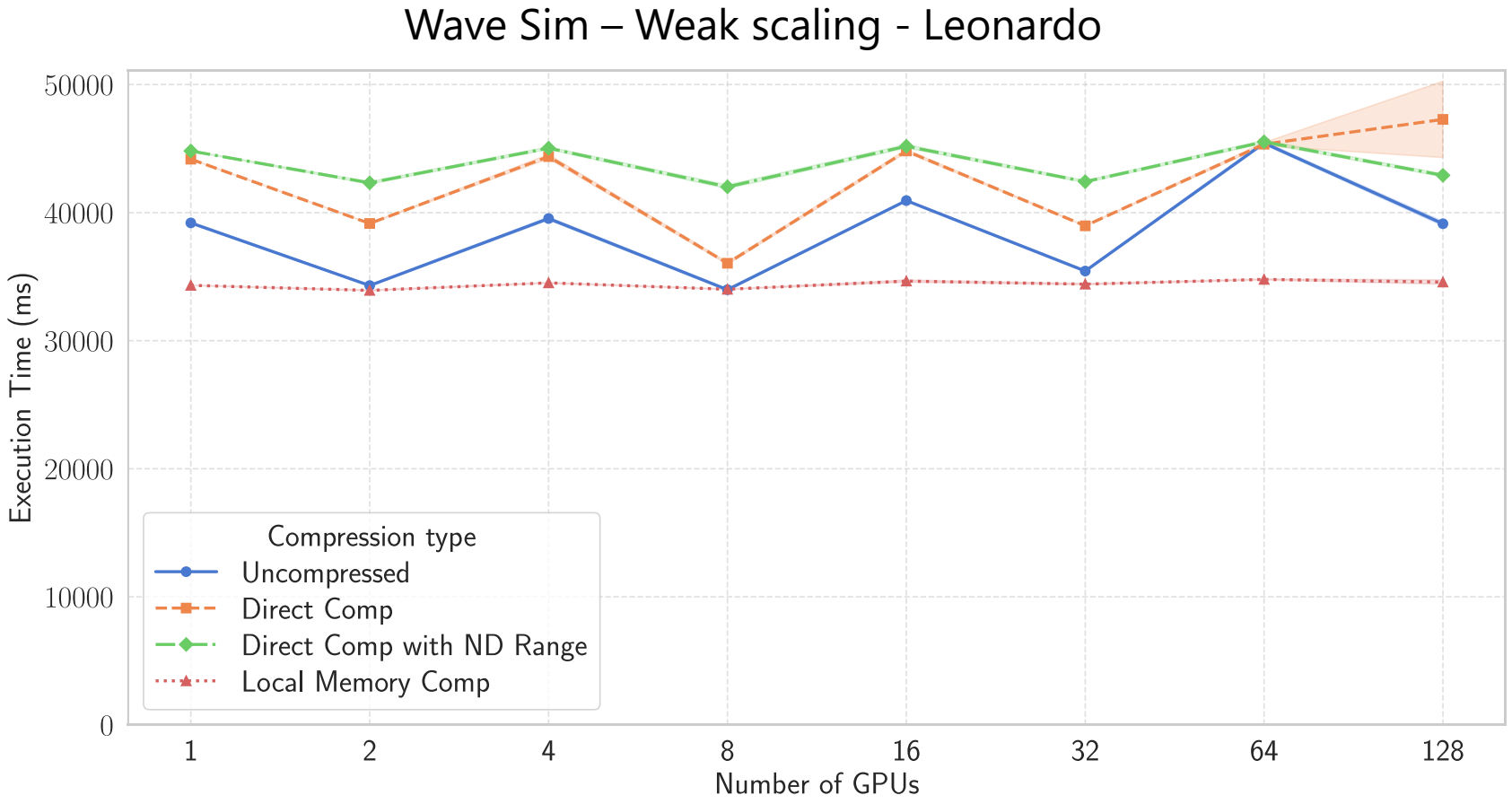


RSim – Strong scaling - Leonardo



Performance





Closing Thoughts

Summary & Conclusion

- Good performance
- Acceptable memory reduction
- Extensible
- Easy to use

Future Work & Limitations

- Improve extensibility of local and global memory compression
- Include API fully into Celerity
- Include traditional kernel compression
- Auto selection of compression algorithms
- Improve performance

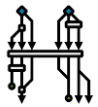
Thank you for your attention!

https://github.com/GagaLP/celerity-runtime-hlpp-2025

https://celerity.github.io/

This research is supported by the Austrian Research Promotion Agency (FFG)
via the UMUGUC project (FFG #4814683).
Compute time on Leonardo was provided by an AURELEO grant.

universität innsbruck



HLPP 2025