Claymore Documentation

Release 1.0

Xinlei Wang and Yuxing Qiu and Stuart R. Slattery and Yu Fang an

May 22, 2023

Overview

CHAPTER 1

Overview

This is the opensource code **Claymore** for the SIGGRAPH 2020 paper:

A Massively Parallel and Scalable Multi-GPU Material Point Method

page, pdf, supp, video

Authors: Xinlei Wang*, Yuxing Qiu*, Stuart R. Slattery, Yu Fang, Minchen Li, Song-Chun Zhu, Yixin Zhu, Min Tang, Dinesh Manocha, Chenfanfu Jiang (* Equal contributions)

1.1 Introduction

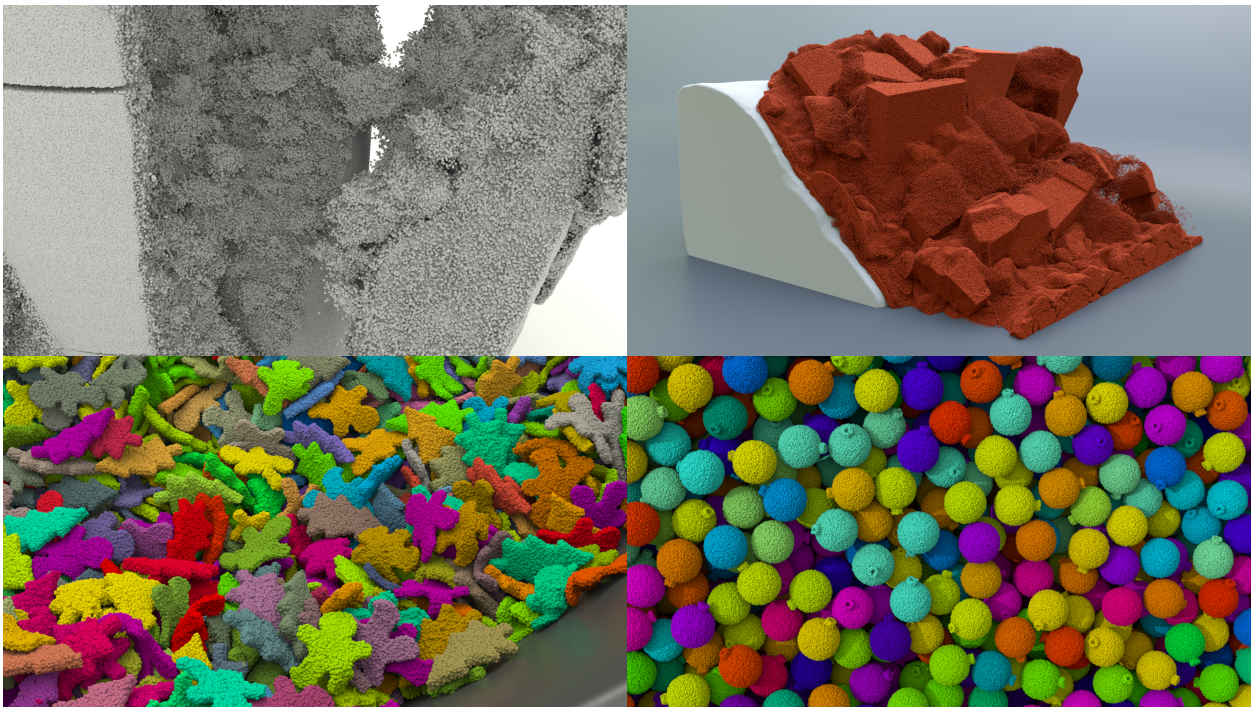
Harnessing the power of modern multi-GPU architectures, we present a massively parallel simulation system based on the Material Point Method (MPM) for simulating physical behaviors of materials undergoing complex topological changes, self-collision, and large deformations. Our system makes three critical contributions:

- Introduce a new particle data structure that promotes coalesced memory access patterns on the GPU and eliminates the need for complex atomic operations on the memory hierarchy when writing particle data to the grid.
- Propose a kernel fusion approach using a new Grid-to-Particles-to-Grid (G2P2G) scheme, which efficiently reduces GPU kernel launches, improves latency, and significantly reduces the amount of global memory needed to store particle data.
- Introduce optimized algorithmic designs that allow for efficient sparse grids in a shared memory context, enabling us to best utilize modern multi-GPU computational platforms for hybrid Lagrangian-Eulerian computational patterns.

We demonstrate the effectiveness of our method with extensive benchmarks, evaluations, and dynamic simulations with elastoplasticity, granular media, and fluid dynamics. In comparisons against an open-source and heavily optimized CPU-based MPM codebase on an elastic sphere colliding scene with particle counts ranging from 5 to 40 million, our GPU MPM achieves over 100X per-time-step speedup on a workstation with an Intel 8086K CPU and a single Quadro P6000 GPU, exposing exciting possibilities for future MPM simulations in computer graphics and computational science.

Moreover, compared to the state-of-the-art GPU MPM method, we not only achieve 2X acceleration on a single GPU but our kernel fusion strategy and Array-of-Structs-of-Array (AoSoA) data structure design also generalizes to multi-GPU systems. Our multi-GPU MPM exhibits near-perfect weak and strong scaling with 4 GPUs, enabling performant and large-scale simulations on a 1024x1024x1024 grid with close to 100 million particles with less than 4 minutes per frame on a single 4-GPU workstation and 134 million particles with less than 1 minute per frame on an 8-GPU workstation.

1.2 Gallery



1.3 Bibtex

Please cite our paper if you use this code for your research:

```
@article{Wang2020multiGMPM,
    author = {Xinlei Wang* and Yuxing Qiu* and Stuart R. Slattery and Yu Fang and
↪Minchen Li and Song-Chun Zhu and Yixin Zhu and Min Tang and Dinesh Manocha and
↪Chenfanfu Jiang},
    title = {A Massively Parallel and Scalable Multi-GPU Material Point Method},
    journal = {ACM Transactions on Graphics},
    year = {2020},
    volume = {39},
    number = {4},
    articleno = {Article 30}
}
```

CHAPTER 2

Compilation

This is a cross-platform C++/CUDA cmake project. The minimum version requirement of **CMake** is 3.15, although the latest version is generally recommended. The recommended version of **CUDA** is 10.2.

Currently tested C++ compilers (as the host compiler for **NVCC**) on different platforms include:

platform	Compilers
Windows	msvc142, clang-9
Linux	gcc8.4, clang-9

In short, the supported compilers should support **C++14** standard and be in compliance with **NVCC**. Since the future releases of **CUDA** are officially excluded on Mac OS, and there is a more suitable candidate **Metal** developed by **Apple**, the **Mac OS** platform is not discussed.

2.1 External Dependencies

These libraries are very helpful in developing this project and save a lot of efforts:

- CUB provides state-of-the-art, reusable software components for every layer of the CUDA programming model including many parallel primitives and utilities.
- fmt is an open-source formatting library for C++.

These libraries are used for particle data IO and initialization:

- partio is an open source C++ library for reading, writing and manipulating a variety of standard particle formats (GEO, BGEO, PTC, PDB, PDA).
- SDFGen is a simple commandline utility to generate grid-based signed distance field (level set) from triangle meshes, using code from Robert Bridson's website.

2.2 Build Commands

Run the following command in the *root directory*.

```
mkdir build
cd build
cmake ..
cmake --build . --config Release
```

The project can also be configured through other interfaces, e.g. using the *CMake Tools* extension in *Visual Studio Code* (recommended), in *Visual Studio Studio*, or in *CMake GUI*.

CHAPTER 3

Design Philosophy

3.1 Data-Oriented Design

Due to the increased overhead of memory over compute operations, **data-oriented design** philosophy has been widely adopted in software design. It focuses on the data layout that is efficient for certain patterns of memory access. Usually, the optimization is achieved through better utilization of caches.

Here are some of the most popular libraries providing efficient data structures.

- OpenVDB: OpenVDB is an open source C++ library comprising a novel hierarchical data structure and a large suite of tools for the efficient storage and manipulation of sparse volumetric data discretized on three-dimensional grids.
- SPGrid: SPGrid is a new data structure for compact storage and efficient stream processing of sparsely populated uniform Cartesian grids. It leverages the extensive hardware acceleration mechanisms inherent in the x86 Virtual Memory Management system to deliver sequential and stencil access bandwidth comparable to dense uniform grids.
- Cabana: As a generalization of SoA (Struct-of-Arrays) and AoS (Array-of-Structs) layouts, AoSoA (Array-of-Structs-of-Arrays), which is adopted by **Cabana**, appears to be a robust choice for both CPU and GPU

3.2 Zero-Overhead Principle

By C++ committee, the zero-overhead principle states that: What you don't use, you don't pay for (in time or space) and further: What you do use, you couldn't hand code any better. As Bjarne Stroustrup points out, it requires programming techniques, language features, and implementations to achieve such a zero-overhead abstraction. In C++ language, designing an almost zero-overhead abstraction is sophisticated and difficult. But it is worth working towards this goal. Meanwhile, such abstractions must provide more benefit than cost.

Despite all the differences in the interpretations of *cost* (See Chandler Carruth's talk), it is of great importance to reduce both the runtime cost as well as the build time cost (especially for C++). Sometimes there even has to be a trade-off. So we adopt the following practices in our codebase:

- Use constexpr if feasible

- Avoid recursive template instantiations
- Encapsulate reusable units into precompiled libraries
- ...

CHAPTER 4

Hierarchy Composition

4.1 Motivation

Following the principle **Data-Oriented Design**, Yuanming Hu introduces a high-performance programming language, Taichi, wherein dedicated data structures can be developed by assembling components of different properties in static hierarchies. Taichi provides a powerful and easy-to-use toolchain for developing a wide range of high-performance applications. It implements an abstraction to define multi-level spatial data structures and kernel functions through a user-friendly python front-end and a robust LLVM back-end that automatically handles memory, manages executions, and deploys to CPU or GPU.

Large-scale simulations are spatially sparse in general, which pose great challenges to develop efficient data structures for specific spatial queries. In MPM simulations, such queries involves grids in the Eulerian domain and particles in the Lagrangian domain. We intend to exploit the aforementioned utility from Taichi for designing data structures. Due to the current lack of support of fully using Taichi in a C++ front-end, a substitute is provided in native CUDA/C++ environment, which is implemented by C++ template meta-programming. Please refer to our tech doc or read the following sections for more details.

4.2 Components

The entire infrastructure consists of the four major components: *Domain*, *Decorator*, *Structural Node*, and *Structural Instance*. For more details, please refer to *Library/MnBase/Object* in the opensourced code.

4.2.1 Domain

Domain describes the range for the index of a data structure. It maps from multi-dimensional coordinates to a 1D memory span.

```
template<typename Tn, Tn Ns...>
struct domain {
    template<typename... Indices>
```

(continues on next page)

(continued from previous page)

```
    static constexpr Tn offset(Indices&&... indices);
};
```

4.2.2 Decorator

Decorator describes the auxiliary and detailed properties regarding the data structure it decorates.

```
enum class structural_allocation_policy : std::size_t {
    full_allocation = 0,
    on_demand = 1,
    ...
};
enum class structural_padding_policy : std::size_t {
    compact = 0,
    align = 1,
    ...
};
enum class attrib_layout : std::size_t {
    aos = 0,
    soa = 1,
    ...
};
template <structural_allocation_policy alloc_policy_,
    structural_padding_policy padding_policy_,
    attrib_layout layout_>
struct decorator {
    static constexpr auto alloc_policy = alloc_policy_;
    static constexpr auto padding_policy = padding_policy_;
    static constexpr auto layout = layout_;
};
```

4.2.3 Structural Node

Structural Nodes with particular properties is formed in a hierarchy to compose a multi-level data structure. Currently, we support three types of structural nodes (i.e., hash, dense, and dynamic). We are planning to support tree in future releases.

```
enum class structural_type : std::size_t {
    /// leaf
    sentinel = 0,
    entity = 1,
    /// trunk
    hash = 2,
    dense = 3,
    dynamic = 4,
    ...
};
```

No matter what the internal relationship of elements is within a structure (either contiguous- or node-based), we assume there is at least one contiguous chunk of physical memory to store the data; the size is a multiple of the extent of the Domain and the total size of all the attributes of an element (might be padded for alignment).

```
/// attribute index of a structural node
using attrib_index = placeholder::placeholder_type;

/// traits of structural nodes
template <structural_type NodeType, typename Domain, typename Decoration, typename...␣
↪Structurals>
struct structural_traits {
    using attribs = type_seq<Structurals...>;
    using self =
        structural<NodeType, Domain, Decoration, Structurals...>;
    template <attrib_index I>
    using value_type = ...;
    static constexpr auto attrib_count = sizeof...(Structurals);
    static constexpr std::size_t element_size = ...;
    static constexpr std::size_t element_storage_size = ...;
    /// for allocation
    static constexpr std::size_t size = domain::extent * element_storage_size;

    template <attrib_index AttribNo> struct accessor {
        static constexpr uintptr_t element_stride_in_bytes = ...;
        static constexpr uintptr_t attrib_base_offset = ...;
        template <typename... Indices>
        static constexpr uintptr_t coord_offset(Indices &&... is) {
        return attrib_base_offset + Domain::offset(std::forward<Indices>(is)...) *␣
↪element_stride_in_bytes;
        }
        template <typename Index>
        static constexpr uintptr_t linear_offset(Index &&i) {
        return attrib_base_offset + std::forward<Index>(i) * element_stride_in_bytes;
        }
    };

    // manage memory
    template <typename Allocator> void allocate_handle(Allocator allocator) {
        if (self::size != 0)
        _handle.ptr = allocator.allocate(self::size);
        else
        _handle.ptr = nullptr;
    }
    template <typename Allocator> void deallocate(Allocator allocator) {
        allocator.deallocate(_handle.ptr, self::size);
        _handle.ptr = nullptr;
    }
    // access value
    template <attrib_index ChAttribNo, typename Type = value_type<ChAttribNo>,␣
↪typename... Indices>
    constexpr auto &val(std::integral_constant<attrib_index, ChAttribNo>, Indices &&..
↪. indices) {
        return *reinterpret_cast<Type *>(_handle.ptrval + accessor<ChAttribNo>::coord_
↪offset(std::forward<Indices>(indices)...));
    }
    template <attrib_index ChAttribNo, typename Type = value_type<ChAttribNo>,␣
↪typename Index>
    constexpr auto &val_1d(std::integral_constant<attrib_index, ChAttribNo>,
                           Index &&index) {
        return *reinterpret_cast<Type *>(
            _handle.ptrval +
```

(continues on next page)

(continued from previous page)

```
            accessor<ChAttribNo>::linear_offset(std::forward<Index>(index)));
    }
    /// data member
    MemResource _handle;
};
/// specializations of different types of structural nodes
template <typename Domain, typename Decoration, typename... Structurals>
struct structural<structural_type::hash, Domain, Decoration, Structurals...> :␣
↪structural_traits<structural_type::hash, Domain, Decoration, Structurals...> {...};
```

We also define two special types of *Structural Nodes*, the root node and the leaf node, to form the hierarchy.

```
/// special structural node
template <typename Structural> struct root_instance;
template <typename T> struct structural_entity;
```

4.2.4 Structural Instance

A variable defined by the above **Structural Node** is an **Structural Instance** spawned given an allocator at the runtime. The instance is customizable (e.g. accessing the parent node requires additional data) as it is assembled from selected data components.

```
enum class structural_component_index : std::size_t {
    default_handle = 0,
    parent_scope_handle = 1,
    ...
};

template <typename ParentInstance, attrib_index, structural_component_index>
struct structural_instance_component;

/// specializations for each data component
template <typename ParentInstance, attrib_index>
struct structural_instance_component<ParentInstance, attrib_index, structural_
↪component_index::parent_scope_handle> {...};
```

Besides the data components, the **Structural Instance** also inherits from the **Structural Node** that specifies the properties of itself.

```
/// traits of structural instance, inherit from structural node
template <typename parent_instance, attrib_index AttribNo>
struct structural_instance_traits
    : parent_instance::attribs::template type<(std::size_t)AttribNo> {
    using self = typename parent_instance::attribs::type<(std::size_t)AttribNo>;
    using parent_indexer = typename parent_instance::domain::index;
    using self_indexer = typename self::domain::index;
};

/// structural instance, inherit from all data components and its traits (which is␣
↪derived from structural node)
template <typename ParentInstance, attrib_index AttribNo, typename Components>
struct structural_instance;
template <typename ParentInstance, attrib_index AttribNo, std::size_t... Cs>
struct structural_instance<ParentInstance, AttribNo,
                           std::integer_sequence<std::size_t, Cs...>>
```

(continues on next page)

(continued from previous page)

```
    : structural_instance_traits<ParentInstance, AttribNo>,
    structural_instance_component<ParentInstance, AttribNo, static_cast<structural_
↪component_index>(Cs)>... {
    using traits = structural_instance_traits<ParentInstance, AttribNo>;
    using component_seq = std::integer_sequence<std::size_t, Cs...>;
    using self_instance =
        structural_instance<ParentInstance, AttribNo, component_seq>;
    template <attrib_index ChAttribNo>
    using accessor = typename traits::template accessor<ChAttribNo>;

    // hierarchy traverse
    template <attrib_index ChAttribNo, typename... Indices>
    constexpr auto chfull(std::integral_constant<attrib_index, ChAttribNo>,
                          Indices &&... indices) const {
        ...
    }
    template <attrib_index ChAttribNo, typename... Indices>
    constexpr auto ch(std::integral_constant<attrib_index, ChAttribNo>,
                      Indices &&... indices) const {
        ...
    }
    template <attrib_index ChAttribNo, typename... Indices>
    constexpr auto chptr(std::integral_constant<attrib_index, ChAttribNo>,
                      Indices &&... indices) const {
        ...
    }
};
```

4.3 Usage

Here, we showcase the usages of the above interface by providing an example of **SPGrid**.

4.3.1 Basic Definitions

To simplify the usage, we define certain types and variables that are frequently used.

```
/// leaf node
using empty_ = structural_entity<void>;
using i32_ = structural_entity<int32_t>;
using f32_ = structural_entity<float>;

/// attribute index
namespace placeholder {
    using placeholder_type = unsigned;
    constexpr auto _0 = std::integral_constant<placeholder_type, 0>{};
    constexpr auto _1 = std::integral_constant<placeholder_type, 1>{};
    ...
}

/// default data components for constructing instances
using orphan_signature = std::integer_sequence<std::size_t,  static_cast<std::size_t>
↪(structural_component_index::default_handle)>;
```

4.3.2 Structural Node Definition

The following code defines the **SPGrid** used in our pipeline.

```
// domain
using BlockDomain = domain<char, 4, 4, 4>;
using GridBufferDomain = domain<int, g_max_active_block>;
// decorator
using DefaultDecorator = decorator<structural_allocation_policy::full_allocation,␣
↪structural_padding_policy::compact, attrib_layout::soa>;
// structural node
using grid_block_ = structural<structural_type::dense, DefaultDecorator, BlockDomain,␣
↪f32_, f32_, f32_, f32_>;
using grid_buffer_ = structural<structural_type::dynamic, DefaultDecorator,␣
↪GridBufferDomain, grid_block_>;
```

4.3.3 Create Structural Instance

After defining the internal structure, it still requires an allocator and the list of data components to get the instance.

```
template <typename Structural, typename Signature = orphan_signature>
using Instance = structural_instance<root_instance<Structural>, (attrib_index)0,␣
↪Signature>;

template <typename Structural, typename Componenets, typename Allocator>
constexpr auto spawn(Allocator allocator) {
    auto ret = Instance<Structural, Componenets>{};
    ret.allocate_handle(allocator);
    return ret;
}

auto allocator = ...;
auto spgrid = spawn<grid_buffer_, orphan_signature>(allocator);
```

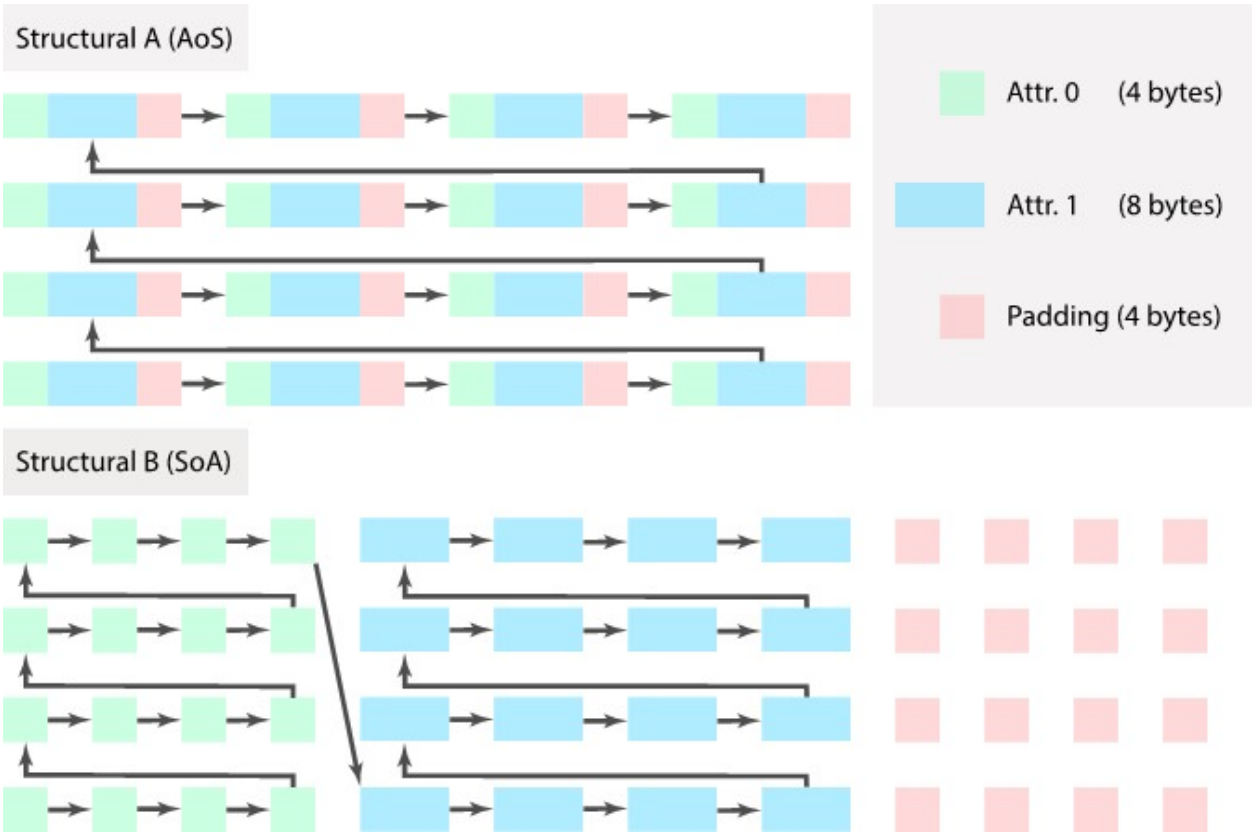
4.3.4 Access Interface

Generally, we need to provide both the attribute and child index to access a child of such an instance.

```
/// acquire blockno-th grid block
auto grid_block = spgrid.ch(_0, blockno);
/// access cidib-th cell within this block
grid_block.val_1d(_0, cidib);   // access 0-th channel (mass)
/// access cell within by coordinates
grid_block.val(_1, cx, cy, cz);   // access 1-th channel (velocity x)
```

4.4 Internal Layout

To gain a better insight into the internal layout, we here give another example.




```
using Attrib0 = structural_entity<float>;
using Attrib1 = structural_entity<double>;
using DecoratorA = decorator<
    structural_allocation_policy::full_allocation,
    structural_padding_policy::align,
    attrib_layout:aos>;
using DecoratorB = decorator<
    structural_allocation_policy::full_allocation,
    structural_padding_policy::align,
    attrib_layout:soa>;
using StructuralA = structural<structural_type::dense, DecoratorA, domain<int, 4, 4>,␣
↪Attr0, Attr1>;
using StructuralB = Structural<structural_type::dense, DecoratorB, domain<int, 4, 4>,␣
↪Attr0, Attr1>;

:caption: Two *structural nodes* are specified with different *decorators*. The␣
↪arrows connecting all elements indicate the ascending order in a contiguous chunk␣
↪of memory. The *structural node* can be used as an attribute of another *structural␣
↪node* to form a multi-level hierarchy. Elements displayed in the grid view are␣
↪accessed by an attribute index (marked with different colors) and a coordinate␣
↪within its domain. Note that the memory size of each *structural* is padded to the␣
↪next power of 2 due to the alignment decoration.
```

CHAPTER 5

Sparse Data Structures

In many graphics applications, e.g. ray tracing, collision detection, neighborhood search, etc., there exists a large amount of queries upon spatial data structures, and the involved volumetric data is often spatially sparse. Dedicated data structures like OpenVDB and SPGrid are introduced to provide efficient access to sparse data for specific operations. However, there is no best universal data structure for different types of sparsity and access patterns.

Through the previous **Hierarchy Composition** interface, we can compose and define sparse data structures very easily in order to experiment and identify an appropriate candidate for our need. In our experiences, the sparse data structures can be categorized in terms of their underlying memory resource.

- **Utilizing Virtual Memory**: rely on the virtual memory support from the underlying OS, driver and hardware.
- **Manually Managing Memory**: manually maintain the sparse data, sometimes including a mapping from the discrete identifiers (e.g. spatial coordinates) to a contiguous sequence of indices.

Then, providing allocators of specific memory resources to the **Structural Instance**, we complete defining the variable (instance), the internal structure of which is specified by the **Structural Node**. Here we illustrate two common key aspects.

5.1 Allocation

5.1.1 Utilizing Virtual Memory

The automatic management of the virtual memory can help relieve the burden of manual maintenance. By avoiding frequent page-faults, the access to the virtual memory can be as efficient as the heap or global memory (CUDA).

Depending on the location of the memory we want to allocate for the instance, the allocation APIs are different.

Allocation On Host

- Windows

```
void* ptr = VirtualAlloc(nullptr, totalBytes, MEM_RESERVE, PAGE_READWRITE);
```

- Linux

```
void* ptr = mmap(nullptr, totalBytes, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_
↪ANONYMOUS | MAP_NORESERVE, -1, 0);
```

Allocation On Device (CUDA)

```
void* ptr;
cudaMallocManaged((void **)&ptr, totalBytes);
```

5.1.2 Manually Managing Memory

The above method heavily relies on the support from the driver, and the granularity of the data unit should be on the same level as a page in the virtual memory system. Therefore it is often a more robust choice to manually manage the memory.

Allocation On Host

```
void* ptr = (void *)malloc(totalBytes);
```

Allocation On Device (CUDA)

```
void* ptr;
cudaMalloc((void **)&ptr, totalBytes);
```

5.2 Modeling Sparsity

The underlying memory resource intended for the sparse data structure largely influences the design of the data structure itself. Here we discuss multiple definitions of the *SPGrid-variants* using different strategies.

5.2.1 Utilizing Virtual Memory

The virtual memory system can save the efforts of modeling the sparsity information. Simply defining the grid like a dense grid is sufficient.

```
// domain
using BlockDomain = domain<char, 4, 4, 4>;
using GridDomain = domain<int, g_grid_size, g_grid_size, g_grid_size>;
// decorator
using DefaultDecorator = decorator<structural_allocation_policy::full_allocation,␣
↪structural_padding_policy::compact, attrib_layout::soa>;
// structural node
```

(continues on next page)

(continued from previous page)

```
using grid_block_ = structural<structural_type::dense, DefaultDecorator, BlockDomain,␣
↪f32_, f32_, f32_, f32_>;
using grid_ = structural<structural_type::dense, DefaultDecorator, GridDomain, grid_
↪block_>;
```

5.2.2 Manually Managing Memory

When adopting this strategy, the programmer should additionally maintain the mapping from spatial block coordinates to a contiguous sequence of indices. Usually we store this mapping through a (spatial) hash table or a lookup table. And the resulting data structure for the *SPGrid-variant* essentially becomes an array of grid blocks.

```
using GridBufferDomain = domain<int, g_max_active_block>;
using grid_buffer_ = structural<structural_type::dynamic, DefaultDecorator,␣
↪GridBufferDomain, grid_block_>;
```

CHAPTER 6

System

To conveniently reuse the code required for a specific type of tasks, we build a precompiled and customized library, i.e. **system**. It is essentially a **Singleton** class which can be selectively included and utilized by projects.

6.1 CUDA

CUDA system provides CUDA related utilities, including - setting up the GPU devices, constructing necessary resources and enabling certain features. - temporary memory pools of various memory type for intermediate computations. - context handles, each corresponding to one GPU, through which programmers launch kernels, synchronize among streams, manage memory, etc.

Also, a few helper functions that can be called on the host side or the device side are also included.

6.2 IO

IO system currently only provide what is necessary for the MPM project, e.g. reading geometry data and outputting generated particle data asynchronously.

CHAPTER 7

Usage

Q: Use the codebase in another cmake c++ project.

A: Directly include the codebase as a submodule, and follow the examples in the *Projects*.

Q: Develop upon the codebase.

A: Create a sub-folder in *Projects* with a cmake file at its root.

CHAPTER 8

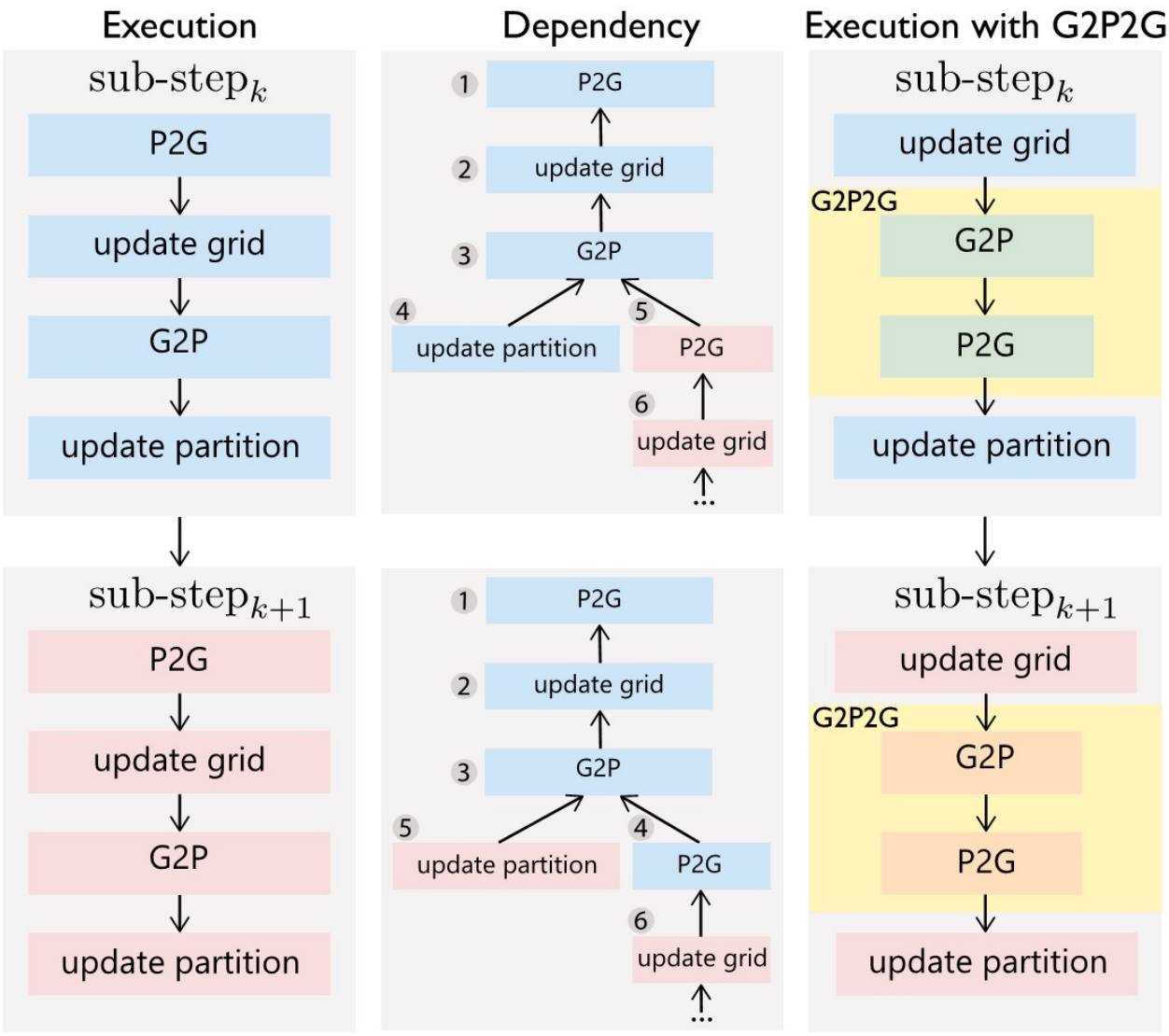
Simulator Pipeline

Our MPM simulator adopts the explicit time-integration scheme.

The conventional MPM pipeline looks like:

- **Particles-to-Grid (P2G)**. Transfer mass and momentum from particles to grid nodes: $\{m_p, m_p v_p^n\} \rightarrow \{m_i, m_i v_i^n\}$
- **Grid Update**. Update grid velocities with either explicit or implicit time integration: $v_i^n \rightarrow v_i^{n+1}$
- **Grid-to-Particles (G2P) and Particle Advection**. Transfer velocities from grid nodes to particles, evolve particle strains, and project particle deformation gradients for plasticity (if any). Update the particle positions with their new velocities: $\{v_i^{n+1}\} \rightarrow \{v_p^{n+1}, F_p^{n+1}\}$, $\{p_p^n, v_p^{n+1}\} \rightarrow \{p_p^{n+1}\}$
- **Partition Update**. Maintain the sparse data structure topology by updating the active-block array and the mapping from block coordinates to array indices.

We propose a novel pipeline with a fused **G2P2G** kernel as a substitute for the original **P2G** and **G2P**. It allows less memory footprint for each particle (although in total doubled due to double-buffering required by G2P2G) and a better performance in general.


Execution
Dependency
Execution with G2P2G
sub-step$_k$
P2G
update grid
G2P
update partition
1
P2G
2
update grid
3
G2P
4
update partition
5
P2G
6
update grid
G2P2G
sub-step$_{k+1}$

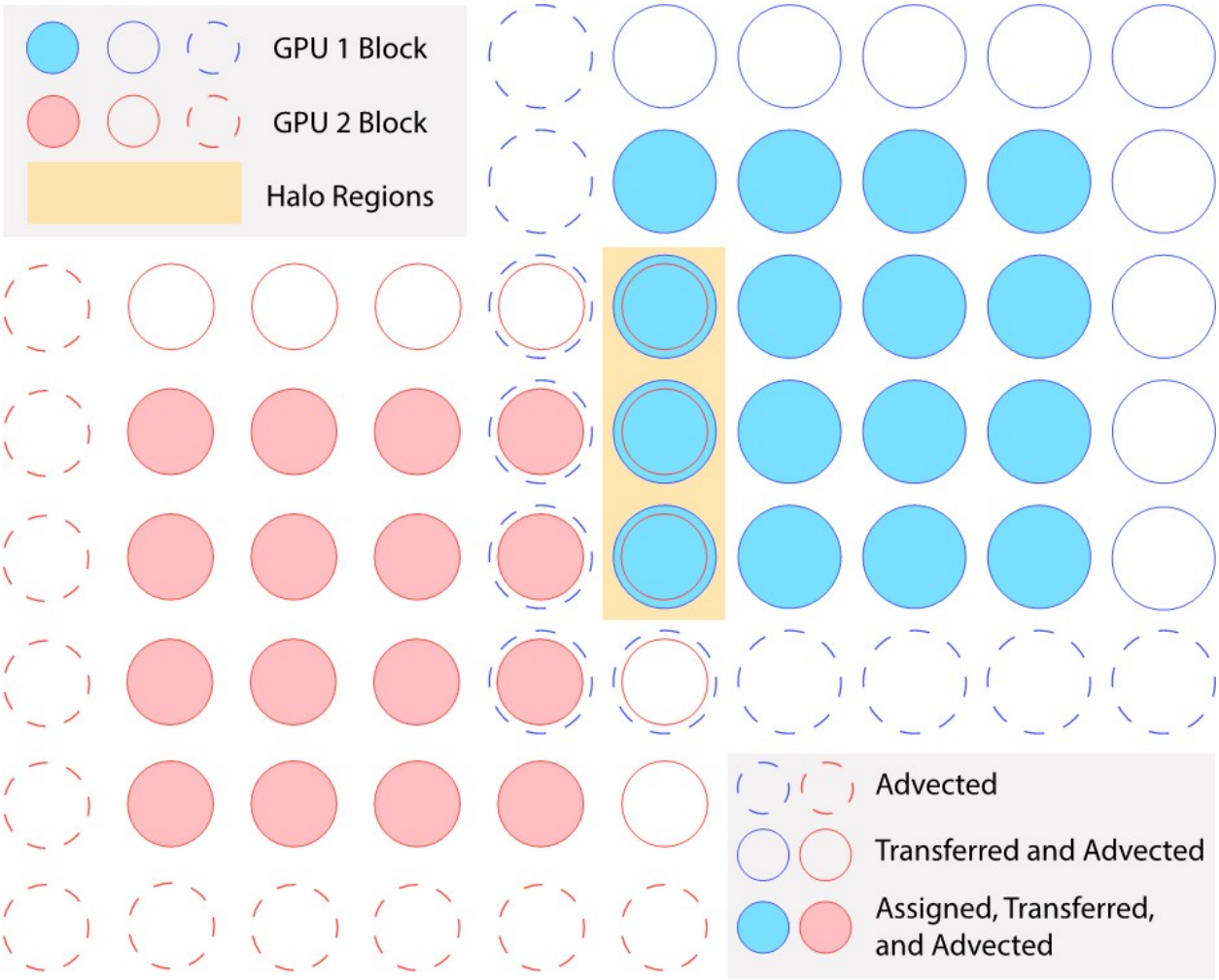
CHAPTER 9

Data Structure

Data structures in a MPM simulation usually need to model the particles, the grids and the mappings between them. Correspondingly, we encapsulate these in *Partition*, *Particle Buffer* and *Grid Buffer*.

9.1 Partition

Partition maintains the active elements, i.e. *block* in this context, and the mapping between block coordinates and linear indices through spatial hashing in the current timestep. By *active element*, it means blocks that particles are assgined to, advect to or transfer to as shown below.



It consists of an array to hold all the block coordinates (sparse information), hash tables for spatial hashing and particle buckets for holding particle indices grouped in blocks. In multi-GPU cases, it additionally stores halo tags.

9.2 Particle Buffer

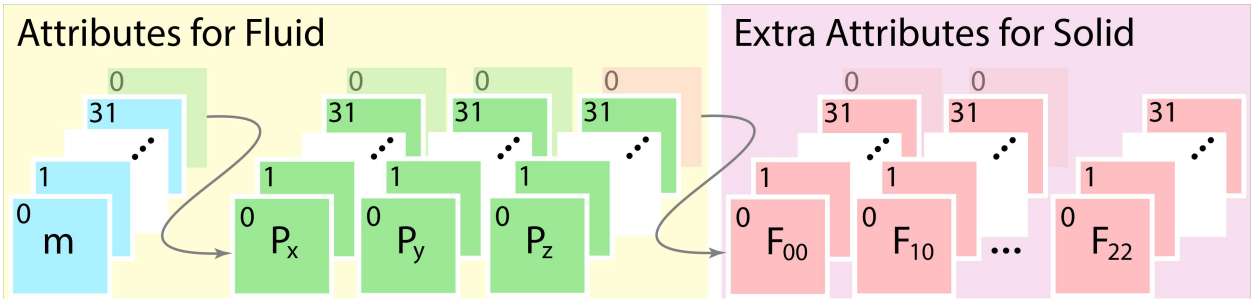
Particle Buffer consists of particle bins from all the blocks that particles belong to. This novel particle data structure is essentially in AoSoA layout similar to Cabana.

```
using f32_ = StructuralEntity<float>;
using Decorator = decorator<structural_allocation_policy::full_allocation, structural_
↪padding_policy::sum_pow2_align>;
using ParticleBinDomain = aligned_domain<char, config::g_bin_capacity>;
using particle_bin4_ = structural<structural_type::dense, Decorator,␣
↪ParticleBinDomain, attrib_layout::soa, f32_, f32_, f32_, f32_>;

using ParticleBufferDomain = compact_domain<int, config::g_max_particle_bin>;
using particle_buffer_ = structural<structural_type::dynamic, Decorator,␣
↪ParticleBufferDomain, attrib_layout::aos, particle_bin4_>;
```

The memory layout of each particle bin specified above is SoA with a total number of 4 attributes (arrays) for fluid particles and a length that is a multiple of 32 (warp size) for coalesced access. Note that the particle buffer is updated

in a delayed fashion according to the particle index buckets in the previous substep. Since the particle buckets are known before *g2p2g*, writing to global memory is generally coalesced.



9.3 Grid Buffer

Grid Buffer is composed of grid blocks that are currently visible to particles.

```
using BlockDomain = compact_domain<char, config::g_blocksize, config::g_blocksize,␣
↪config::g_blocksize>;
using grid_block_ = structural<structural_type::dense, Decorator, BlockDomain, attrib_
↪layout::soa, f32_, f32_, f32_, f32_>;

using GridBufferDomain = compact_domain<int, config::g_max_active_block>;
using grid_buffer_ = structural<structural_type::dynamic, Decorator, GridBufferDomain,
↪ attrib_layout::aos, grid_block_>;
```

This is the same GPU-tailored SPGrid variant for grid structure used in an earlier GPU implementation of MPM.

Note: For *particle buffer* and *grid buffer*, we store them in the device-local memory. If replaced with the virtual memory (*unified virtual memory* in CUDA), then the *ParticleBufferDomain* and *GridBufferDomain* could be replaced with *compact_domain<int, config::g_grid_size, config::g_grid_size, config::g_grid_size>* for a more straightforward access through block coordinates. However, we encountered unidentified issues utilizing the *unified virtual memory* on Window 10, and the difference between the two resulting performances is trivial. Thus we choose the former scheme which is relatively more robust and controllable.

CHAPTER 10

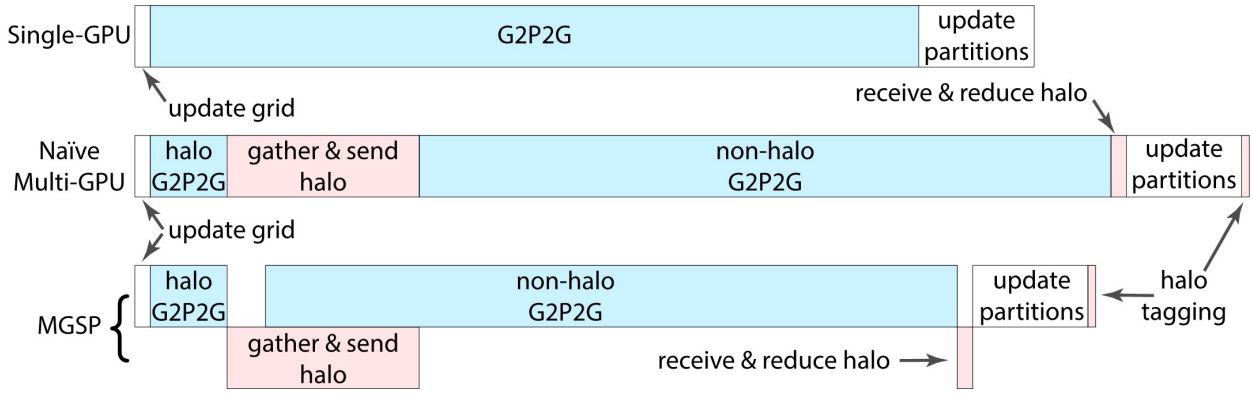
Multi-GPU Design

10.1 Reduce Memory Overhead

To extend from using a single GPU to running on multi-GPUs, we divide the whole simulation domain into partitions (maintained through a list of activated blocks covering all particles) according to the number of devices and assign one partition to one GPU device. An efficient utilization of multi-GPUs for MPM needs to consider the following tasks:

- **Halo Block Tagging**: tag the blocks that overlap partitions on other devices (i.e., the halo blocks).
- **Halo Block Merging**: share block data in the halo region with other devices after executing the G2P2G kernel, for grid reduction and/or particle migration depending on partitioning strategies.

The latency of the memory transfer among GPU devices (required by the above tasks) relies highly on the underneath hardware setup. In most consumer-level machines, multi-GPU devices are connected via the slow PCI-Express x16 Gen 3, which may lead to high communication latency. Fortunately, nearly all CUDA devices with compute capability of 1.1 or higher can concurrently perform the memory copies and computing kernels. Therefore, it is viable and critical to hide the latency by overlapping data transfers with computations for a better performance scalability on multi-GPUs.



10.2 Partitioning Methods

Depending on the dynamics of the simulation, the same partitioning scheme could result in drastically different performances on various scenes. Therefore we introduce detailed designs of two MPM-tailored variations of the most widely adopted partitioning methods, i.e. the *static geometric (particle) partitioning method* and the *static spatial partitioning method*.

CHAPTER 11

Benchmark

11.1 Settings

Benchmark settings of each project are set within its directory (e.g. *Projects/MGSP*) independently.

11.1.1 General setup

Most configurations regarding the simulation are set in *settings.h*.

Data structure related setup:

- **DOMAIN_BITS**: the resolution of the background grid in each dimension, e.g. 8 refers to a $256 \times 256 \times 256$ grid.
- **MAX_PPC**: the maximum number of particles held within a cell.
- **g_bin_capacity**: the number of particles in each particle bin.
- **g_max_particle_num**: the maximum number of particles in each GPU.
- **g_max_active_block**: the maximum number of blocks in each GPU. This affects the memory consumption of most important data structures (*particles*, *grid*, *partition*, etc).

Physical parameter setup:

- **g_gravity**: the gravity constant.
- **MODEL_PPC**: particle-per-cell for sampling particles from a SDF model. per-particle-volume is computed from this and the grid resolution.
- **DENSITY**: the density constant of particles.
- **get_material_type(int did) -> material**: the type of particles for each GPU. Assume each GPU handles only one category of particles.

Multi-GPU related setup:

- **g_device_num**: the number of GPUs used.

- **get_domain(int did) -> domain**: the spatial partition domain for each GPU. It can be set time-dependently.

11.1.2 Additional material setup

In *particle_buffer.cuh*, there are currently four types of particles supported. They are named *JFluid*, *FixedCorotated*, *Sand*, and *NACC*. Within each **ParticleBuffer**, there are various material-dependent constant parameters that could be configured.

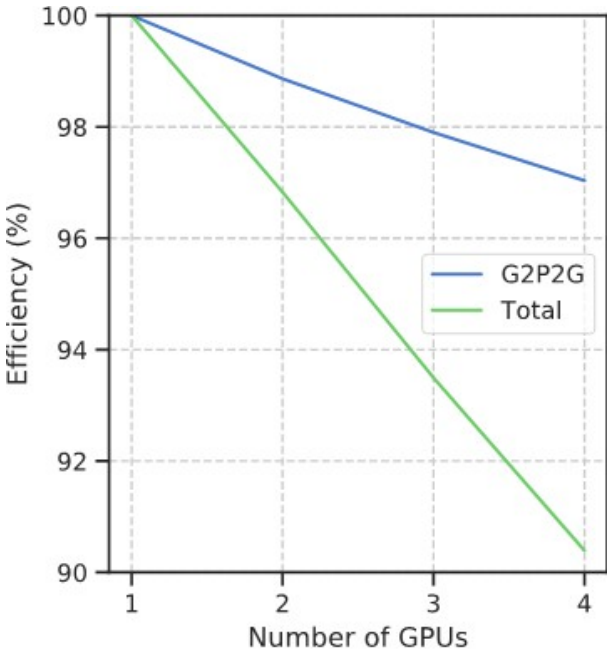
11.1.3 Initial model setup

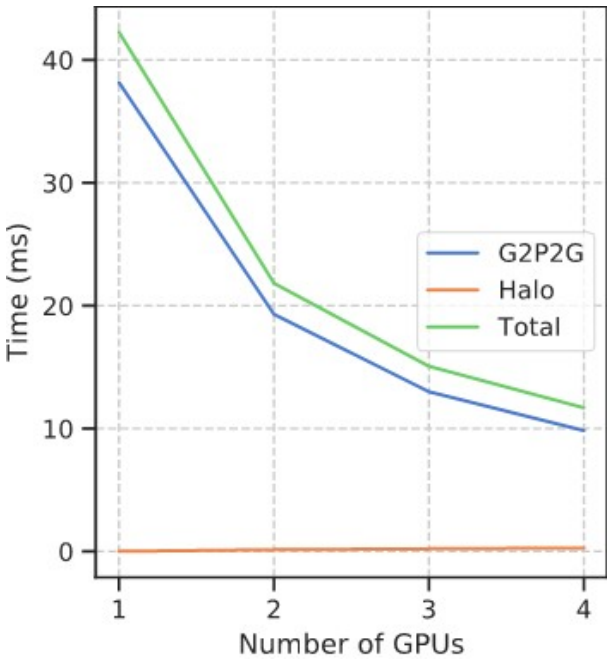
Initial particle models for all GPUs are set in the **init_models** function in *main.cu*.

11.2 Results

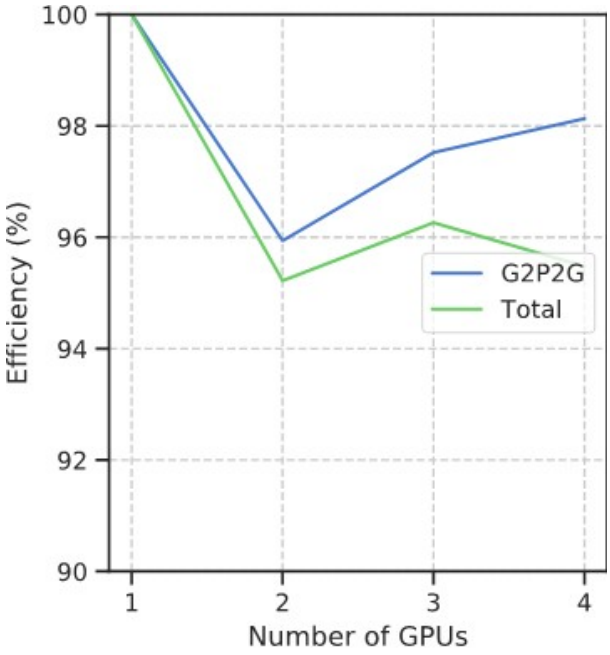
11.2.1 Multi-GPU scalability

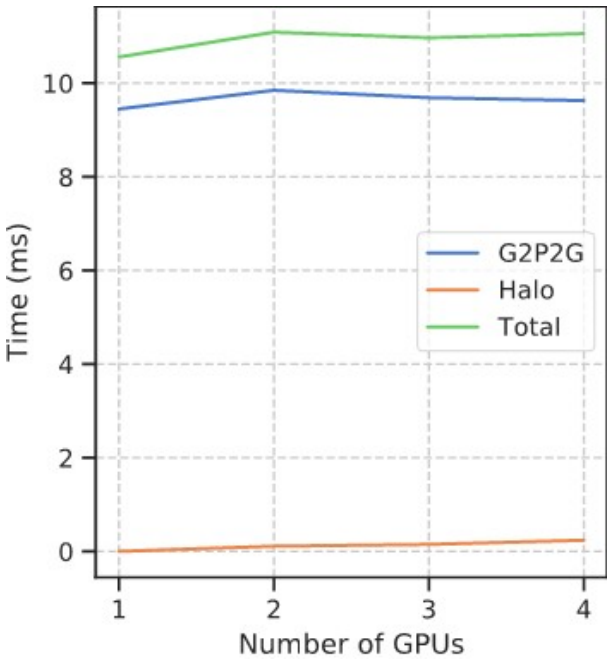
The following shows the strong scalability of our multi-GPU pipeline.





The following shows the weak scalability of our multi-GPU pipeline.




Time (ms)
10
8
6
4
2
0
1
2
3
4
Number of GPUs
G2P2G
Halo
Total

CHAPTER 12

Acknowledgement

We thank Yuanming Hu for useful discussions and proofreading, Feng Gao for his help on configuring workstations. We appreciate Prof. Chenfanfu Jiang and Yuanming Hu for their insightful advice on the documentation.