A Massively Parallel and Scalable Multi-GPU Material Point Method
Supplementary Technical Document

XINLEI WANG*, Zhejiang University and University of Pennsylvania
YUXING QIU*, University of California, Los Angeles and University of Pennsylvania
STUART R. SLATTERY, Oak Ridge National Laboratory
YU FANG, University of Pennsylvania
MINCHEN LI, University of Pennsylvania
SONG-CHUN ZHU, University of California, Los Angeles
YIXIN ZHU, University of California, Los Angeles
MIN TANG, Zhejiang University
DINESH MANOCHA, University of Maryland
CHENFANFU JIANG, University of Pennsylvania

ACM Reference Format:

1 COMPILe-TIME SETTINGS
To maximize the performance, we use compile-time constants for both controls- and material-related settings. Below, we provide additional details on compile-time settings for reproduction purposes.

We set the maximum number of particle-per-cell to be 64 for all the experiments. Note that this setting is more than sufficient for most MPM simulations since the typical particle-per-cell is 8 when initializing scenes. We have not observed any violations in any examples we used in this paper, and one can easily modify this setting in our code. However, the particles will be discarded if the particle number inside one cell exceeds the compile-time setting, leading to incorrect results.

We also preset the maximum number of particle blocks and grid blocks, as well as the maximum particle number for the experiments. These settings are adopted to enable the pre-allocation of all spatial data structures. Still, we periodically check the current demand for memory and dynamically resize to fulfill the need. Different scenes require different settings, and the program works as long as the whole memory allocated does not exceed the device memory limit.

Another assumption is that Courant–Friedrichs–Lewy (CFL) condition always holds during run-time, indicating that the particles would move at most one-cell distances in each time step. We use Courant-number 0.6 to compute the CFL-bounded default stepping time in all the experiments. The material stiffness is also considered when computing the default stepping time for stability requirements. During run-time, we compute the maximum of the grid velocity and calculate a stepping time to ensure particles do not move more than one-cell distance. The final stepping time is chosen as the minimum of the computed time among all devices and the default stepping time. The CFL condition is crucial for the correctness of the results, and the G2P2G kernel will crash if it is violated, leading to failures as would happen in traditional CPU and GPU solvers.

2 HIERARCHICAL DATA STRUCTURE COMPOSITION
Since the efficacy of the data structure is usually hardware- and algorithm-dependent, it often requires non-trivial engineering efforts to explore different choices. Therefore, the ability to quickly design and benchmark new data structures for a specific task can significantly reduce code complexity.

2.1 Data-Oriented Design Philosophy
Due to the increased overhead of memory operations, data-oriented design philosophy [A. 2014] has been widely adopted in HPC. Following this design principle, Hu et al. [2019] 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.
Still, there are two major issues in Taichi that prevents us from directly adopting it when developing multi-GPU-tailored MPM algorithms: 1) no access to low-level operations, including CUDA warp intrinsics, and 2) lack of multi-GPU support. Therefore, in our implementation, we refer to the data structure description described in Taichi as the mini-language and build up the infrastructure within our C++ codebase with the following improvements.

C++ Oriented Programming. Unlike developing a new compiler as in the Taichi programming language, we intend to develop a tool that can be directly used in both native C++ and CUDA C++. The latest standard supported by CUDA is C++14; thus it becomes our final choice. Function definitions are decorated with `constexpr` keyword whenever applicable on both the host- and the device-side.

Structural Composition. The C++ template meta-programming is adopted to implement the infrastructure. Most setups, including hierarchy, layout, the relationship of elements, etc., are known beforehand and can be statically specified as template parameters. Hence, the access interface and the internal composition of the customized data structure are specified.

Memory Management. The representations of memory handles vary across APIs for GPU computing. For CUDA C++, the memory handle of the device memory is simply a pointer on the host; the cost of copying is trivial. Thus, the memory handle can be value-copied to CUDA kernel functions from the host-device. The specific type of memory (e.g., unified virtual memory or device memory) that the variable is allocated with is determined by the allocator given at the run-time. The instance does not own the handle of the allocator so that its lifetime could be managed by programmers explicitly.

In our C++ codebase, we follow the same principle emphasized by the data-oriented design principle: the internal data structure should be highly compositional and shielded under a set of high-level access interfaces. Specifically, Structural Nodes can be associated with child nodes recursively for multi-level hierarchy composition, and the accompanying Decorator specifies the property of the node itself. For high performance, most specifications of the structure are performed at compile-time. We provide these utilities through C++ variadic templates in the following form:

```cpp
struct domain<
    Tn, Ns...>
    // Tn: Index Type, Ns: Multi-dimensional coordinates of type Tn
enum attrib_layout(aos, soa);
enum structural_type(entity, hash, dense, dynamic);
decorator<structural_allocation_policy, structural_padding_policy, attrib_layout>;
structural<structural_type, domain, decorator, structural_padding_policy, attrib_layout>;
```

2.2 C++ Implementation

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

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

```cpp
template<
    typename... Indices
    >
    static constexpr Tn offset(Indices&&... indices);
}
```

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

```cpp
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,
    ...
    };
```

```cpp
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_;  
};
```

2.2.3 Structural Node. Structural Nodes with particular properties are 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), same as in Hu et al. [2019].

```cpp
enum class structural_type : std::size_t
    { entity = 1,
    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.

```cpp
using attrib_index = placeholder::placeholder_type;
```

```cpp
// traits of structural nodes
```

```cpp
struct structural_traits {
    using attribs = type_seq<Structurals...>;  
    using self = structural<
        NodeType, Domain, Decoration,  
        Structurals...>;
    template <attrib_index I>
        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 = ...;
    static constexpr uintptr_t attrib_base_offset_in_bytes = ...;
    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(Alloc)
    Allocator allocator) {
    if (self::size != 0)
        _handle.ptr = allocator.allocate(self::size);
    else
        _handle.ptr = nullptr;
}

// access value
template <attrib_index ChAttribNo, typename Type = value_type<ChAttribNo>,
        typename... Indices>
    constexpr auto &chfull(std::integral_constant<
        attrib_index, ChAttribNo>, Indices &&... indices) const {
    return *reinterpret_cast<Type *>(_handle.ptrval +
            accessor<ChAttribNo>::coord_offset(std::forward<Indices>(indices))...);
}

2.2.4 Structural Instance. A variable defined by a Structural Node is an Structural Instance spawned given an allocator at the run-time. The instance is customizable (e.g., accessing the parent node requires additional data) as it is assembled from data components.

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

template <typename ParentInstance, attrib_index,
        structural_component_index>
    static constexpr uintptr_t element_stride_in_bytes = ...;
```

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

```cpp
// traits of structural instance, inherit from structural node
template <typename parent_instance, attrib_index AttribNo
        , typename Components>
    struct structural_instance_traits;
```

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

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

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

2.3 Examples

Here, we showcase usages of Structural in C++ by providing a set of examples that describes a GPU SPGrid.

Common Useful Definitions.

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

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

Definition of GPU SPGrid.

```cpp
/// 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_>
```

Create Instances.

```cpp
template <typename Structural, typename Signature = orphan_signature>
using Instance = structural_instance<root_instance<
    Structural>, (attrib_index)0, Signature>;
template <typename Structural, typename ComponentType,
    typename Allocator>
constexpr auto spawn(Allocator allocator) {
    auto ret = Instance<Structural, ComponentType>{
        allocator);
    return ret;
}
```

Access GPU SPGrid in a Function.

```cpp
auto grid = spawn<grid_buffer_, orphan_signature>(
    allocator);
```

Memory Layout. Two types of Structural Nodes with different Decorators are illustrated in ?? to explain the underlying memory layout.

REFERENCES