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:
Xinlei Wang, Yuxing Qiu, Stuart R. Slattery, Yu Fang, Minchen Li, Song-Chun Zhu, Yixin Zhu, Min Tang, Dinesh Manocha, and Chenfanfu Jiang. 2020. A Massively Parallel and Scalable Multi-GPU Material Point Method Supplementary Technical Document. *ACM Trans. Graph.* 39, 4, Article 1 (July 2020), 4 pages. https://doi.org/10.1145/3386569.3392442

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.

*equal contributions

Authors' addresses: 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.

0730-0301/2020/7-ART1 $15.00
https://doi.org/10.1145/3386569.3392442

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:

```
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,
    structurals...>;
```

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.

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

```
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_;
};
```

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].

```
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.

```
/// 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 +
        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 define two 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;
```

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.

```
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...>>
    : 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 {
    ...
  }
};
```

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.

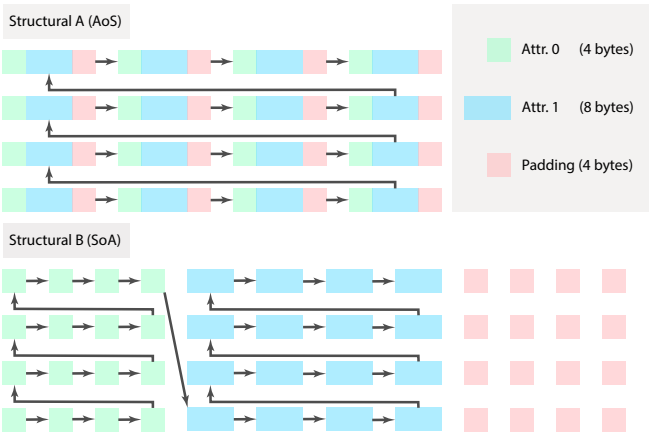
```
/// leaf node
using empty_ = structural_entity<void>;
using i32_ = structural_entity<int32_t>;
using f32_ = structural_entity<float>;

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

/// 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)>;
```

Definition of GPU SPGrid.

```
// 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.

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




```
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>;
```

Fig. 1. **Spatial structure specification.** Two *structurals* are specified with different *decorators*. The arrows connecting all elements indicate the ascending order in a contiguous chunk of memory. The *structural* can be used as a child of another *structural* to form a multi-level hierarchy. Elements displayed in the grid view are accessed by a child *structural* 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.

Access GPU SPGrid in a Function.

```
/// acquire blockno-th grid block
auto grid_block = grid.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)
```

Memory Layout. Two types of *Structural Nodes* with different *Decorators* are illustrated in Fig. 1 to explain the underlying memory layout.

REFERENCES

Mike A. 2014. CppCon 2014: Mike Acton "Data-Oriented Design and C++". https://www.youtube.com/watch?v=rX0ItVEVjHc.

Y. Hu, T. Li, L. Anderson, J. Ragan-Kelley, and F. Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–16.