0.1 Claymore - Overview

Claymore is a recently developed, open-source, Multi-GPU, computer-graphics software [Wang et al.2020b] for running MLS-MPM [Hu et al.2018] (Sec. **??**). It boasts a 100x acceleration over optimized single-CPU codes and has been tested at particle counts in the ball-park of 100 million using 4-8 consumer GPUs in single-precision. We have since retooled the code into what is called ClaymoreUW, and validated it for double and mixed-precision engineering simulations with over a billion particles on just 4 GPUs.
Workflow in Claymore is similar to most MPM software, with a few key distinctions originating from its Multi-GPU oriented nature. The list below and subsequent sections identify the most prominent aspects of Claymore, including:

- **GPU oriented MPM Kernels.** All MPM algorithm steps are coded with respect to low-level GPU optimization. Most code is purely for squeezing out efficiency while the actual MPM algorithm is a minority of lines.
- **Multi-GPU Communication.** Simulations are partitioned across multiple GPUs. In "Halo-Regions", areas where GPU partitions overlap, the software must read/write information between devices without invalidating constraints of the MLS-MPM algorithm. Non-Halo tasks are done in parallel to Halo tasks, subsequently reduced as an aggregate each time-step.
- **Fused G2P2G Operation.** For extra efficiency, G2P + Advection + P2G are fused into one-step. This leads to a 2x improvement in speed for MLS-MPM and avoids storing the affine velocity matrix from APIC on particles (Sec. **??**). However, lack of global synchronization between G2P and P2G means certain things common in regular MPM codes are harder to implement here (e.g. any technique requiring particles to reference each other before P2G), and vice versa. Finite elements (Section **??**) are particularly unwieldy to implement, though our lab has added this functionality along with expanded advection schemes such as Affine-Separable Fluid-Implicit Particles (ASFLIP, Section **??**).
- **Data-Structure Design.** This code uses complex data-structures optimized for Multi-GPU use, fashioned similar to the approach Taichi uses. However, simply trying to retrieve information from a specific particle requires extra effort. Our lab has simplified user interaction with complex Multi-GPU data-structures in ClaymoreUW.
- **Computer Graphics Bias.** The Claymore software is not engineer friendly, though our project is improving this aspect. In its original version, graphical use was prioritized at every step, resulting in poorly validated physical behavior, engineering

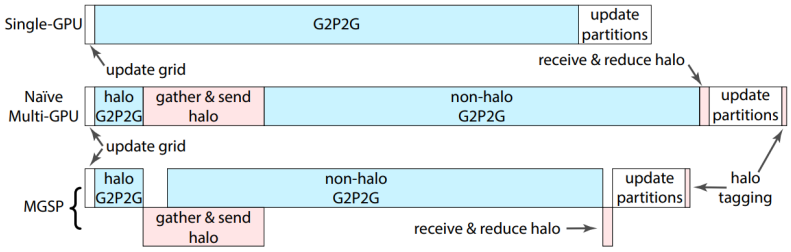

Fig. 1. Instructions for Single and Multi GPU Material Point Method - Claymore Software. Instruction pipe-line for both Single and Multi-GPU MPM in Claymore. Intelligent Multi-GPU MPM design uses CUDA asynchronous properties to hide latency, approaching Single-GPU speeds. Courtesy of [Wang et al.2020b].

method implementation difficulty, and unwieldy I/O. Basic errors have been corrected by our group and we add functionality to expand the software to engineering use.

0.2 From Graphics to Engineering: ClaymoreUW

The low-level code optimization required to build an effective Multi-GPU numerical tool is high, but the benefits are greater. Leveraging the work of skilled programmers (i.e. graphics researchers) allows us to tap into optimized Multi-GPU codes with minimal overhead. Retooling a graphical software for engineering use is not a small task, but it is easier than starting from scratch. Here a summary of changes made to Claymore [Wang et al.2020b] to make the software ClaymoreUW. We include reasons for doing so. Also, a more general guide for retooling is provided for developers.

We have made many changes to the ClaymoreUW software that span implementations of various algorithms, design of high-user-interfaces, optimization of low-level GPU instructions, and general bug-fixing. Novel contributions made by our lab to the code are: These changes are described in detail in Bonus 2023 PhD dissertation. These changes presented a technical challenge as they needed to be seamlessly implemented into an existing, highly-optimized Multi-GPU code that was designed primarily for animators using MLS-MPM. The original Claymore is fast, and it does run MLS-MPM, but it has very few features in the sense that any simulation that goes beyond loading in a provided list of particles and setting the material on one GPU would require changes to the source code. Further, it only ran MLS-MPM, which isn't enough for engineers on its own.

MPM advancements from the computer graphics literature (e.g. Sections **??** and **??**) are poorly validated by engineering standards. Animators are primarily concerned if a simulation looks good (i.e. roughly accurate particle positions and velocities) and accept visual results as validation of a mathematically sound derivation. However, math that is good on paper may not result in quality engineering simulations.

Computing decent positions and velocities of particles is simpler than getting physically sound results. Force, stress, and other engineering aspects (e.g. nonlinear material parameters) are far more difficult to simulate accurately and must be carefully validated against reality.

Compile-time constants are used extensively in Claymore. This is a problem for physical simulations since in its original form minor modifications of a simulation required edits in the source code and a full recompilation. In our expanded version of the code many pertinent simulation parameters (e.g. time-step, gravity, particles-per-cell, material model, etc.) have been changed to run-time, now set in a simple input script.

A common error in graphical codes is scene-scaling. Animators often scale just gravity and density, assuming that this will scale all physical laws in the scene appropriately. For instance, the Claymore MPM code creates erroneous stress values when the scene is scaled beyond a 1x1x1 box. This is due to APIC's stencil coefficient for quadratic B-Spline shape-functions, D_p^{-1}, not being adjusted in the code. Errors like this are likely missed by graphics researchers because they tend to focus on visual validation of their codes (i.e. are particle positions roughly correct). Errors in stress-strain behavior require further validation that necessitates a strong foundation in physics to undertake (i.e. engineers). For example, validating pressure-wave behavior of debris impacting rigid walls as elastic moduli varies in scaled scenes is simple for engineers but rarely in scope for animators. Small errors like this are not difficult to fix once identified, but the presence of multiple such errors in many open-source graphics codes makes the software give incorrect results "out-of-the-box". This may be why so few engineers use highly-optimized graphics codes, despite the fact that they run engineering algorithms, as test simulations will almost certainly not be up-to engineering standards.

0.3 ClaymoreUW - Compilation

The Claymore code's original source is at:
https://github.com/penn-graphics-research/claymore/
The modifications we've made in ClaymoreUW are kept at:
https://github.com/JustinBonus/claymore/
If you eventually publish research based on our modified Claymore, please cite the original authors, as well as our eventual publication on our open-source modifications. Original author citation as [Wang et al.2020b]
Here is a brief, but generally effective compilation guide for *NIX based systems:

1. Use modern versions of CMake (3.10+ typically). Use a modern compiler. We have used used GCC 7.5+, 8, 9, 10, and 11 but any above 7.5 should work. People on Windows systems have used MSVC successfully though we have not attempted a Visual Studio build with MSVC on windows.

CUDA version 10.2+ is required, we have compiled it with CUDA 10.2, CUDA 11.0, CUDA 11.4, and CUDA 12.2.

Ensure a modern GPU is used and know the "compute capability" of the GPU. ClaymoreUW should work for any NVIDIA Compute Capability 6.1+ (most cards after 2016). We have primarily run on 6.1, 7.5, 8.0, 8.6, and 9.0.

2. Go into ./setup_cuda.cmake and change the TARGET_CUDA_ARCH flag to apply to your GPU card. Go-to the line where it says: "set(TARGET_CUDA_ARCH -arch=sm_61)", and change sm_61 to sm_XY where XY is your compute capability. Advanced: Change the line so it compiles directly for a single GPU card architecture (the code will be faster) or have it compile for a range of cards (code will be slower the first time running, but will work on more GPU cards).

3. Check that ./Projects/FBAR/settings.h has some basic values set properly:

   g_device_cnt = number of GPUs you are using, e.g. 1 (do not use 2+ if there are not 2+ available GPUs).

   g_models_per_gpu = number of models per GPU, e.g. 3 if you want to have a fluid, granular material, and a solid model running on one GPU

   g_max_particle_num = max number of particles, e.g. 2000000 for 2,000,000 particles on a GPU. (resizes as simulation runs, must be correct at start)

   g_max_active_blocks = max number of active "blocks" on a GPU, e.g. 10000 for 10,000 blocks, which corresponds to 64 * 10,000 active grid-nodes (have mass) around the particles. (resizes within reason as simulation runs, must be correct at start). This has a big effect on memory/speed.

   DOMAIN_BITS X <- X is the "bits" for the grid, e.g. 10 = 2e10 grid-nodes in any direction (Total grid-nodes in 3D = 2e10 * 2e10 * 2e10). There are 8 buffer grid-nodes on either end of each dimension, so for a 25.2 x 4 x 7.5 meter simulation the max length (25.2) is the bottleneck for grid-cell spacing dx. i.e. 25.2 meters / (2e10 - 16) = smallest possible dx = 0.025 meters. Increase DOMAIN_BITS to use more memory but allow higher-res simulations, or the inverse to save memory.

   MAX_PPC X <- X is maximum particles-per-cell, e.g. 64. If you have a 8 PPC material that is compressible, you may expect as many as 64 PPC to happen in areas of high compression, so set it to something like that. Must be a power of 2 (...8, 16, 32, 64, 128...) in Claymore, ClaymoreUW allows non-power of 2 values. This number has a big effect on memory/speed.

4. Make sure CUDA is loaded on the computer, "nvidia-smi" will show available GPUs and the NVIDIA Driver version. Try "nvcc –version" to check your CUDA NVCC compiler version. On TACC systems (or any HPC), it may be neccesary to run something like "module load cuda/11.0" or "module load cuda/11.4" first to load the cor-

rect CUDA + NVIDIA drivers on a node before compiling. Again, use "nvidia-smi" to make sure GPUs are available on the node.

5. Compile using the steps given in https://github.com/penn-graphics-research/claymore/. Start by making a "build" folder in the claymore source folder. Starting from the ./claymore folder, you can run "cd ./build ; cmake .. ; cmake –build . ; cd ../Projects/FBAR" for GCC compilers.

6. To run the program use the command "./fbar" in the ./claymore/Projects/FBAR folder. This defaults to using the "scene.json" file in the same folder. To use a specific scene file you create, use "./fbar –file=scene_YourFile.json". The program will give you various info, warnings, and potential error messages as it initializes, the latter two require pressing ENTER on your keyboard to continue. Output files will be in the same directory that your terminal is in.

A set of tips that may help with getting the most out of ClaymoreUW and ensuring the set-up goes smoothly:

1. If using a GPU that is good at double-precision computation (e.g. NVIDIA Tesla series) expect a 8-32x performance boost over GPUs that are not good at double-precision (e.g. NVIDIA GTX/RTX/Workstation series), but they are all fairly fast. We recommend modern HPC systems such as TACC Lonestar6 if possible, it possesses 3 NVIDIA A100 GPUs per node (40 GB memory each, so 120GB total GPU memory). We have ran up-to 500 million particles on this system.

2. It is a very common issue for people that their CUDA NVCC version, CUDA Toolkit version, NVIDIA Driver version, and/or their GPU compute capability are mismatched in some way. This often happens when people install CUDA/NVIDIA Drivers incorrectly, in pieces, or improperly uninstall older versions before updating. This is not an issue with Claymore specifically, as it will prevent you from compiling any CUDA code. Check if you can compile and run NVIDIA's sample programs such as "/usr/local/cuda/samples/5_Simulations/fluidsGL/fluidsGL" (file-path may vary for you) before trying Claymore.

3. If using an IDE like Visual Studio Code (recommended) make sure your "Include" path is set properly so that the program can compile/link the library dependencies. Also make sure it can find your CUDA install location.

4. Ensure you have modern C++ versions, e.g. C++ 14, 17, and 20+ should work (compilers like GCC will typically install these for you) with C++ 17 and higher being preferred.

5. We would recommend setting up PartIO and/or SideFX Houdini for visualization. SideFX offers free apprentice licensing of their software.

6. The RapidJSON library dependency causes compilation complications for some users. I haven't had any, but you can update to newer RapidJSON versions in "./setup_externallibs.cmake" to potentially fix issues.

We have compiled on a personal laptop (NVIDIA 1060m Max-q), desktop (NVIDIA 780 ti, may now be deprecated), TACC Frontera (NVIDIA Quadro 5000), TACC Lonestar6 (NVIDIA A100 40GB), UW Hyak Mox (NVIDIA Tesla P100), UW Hyak Klone (NVIDIA RTX 2080 ti), Texas A&M University's ACES (NVIDIA H100 PCIe 80GB). For help compiling or running ClaymoreUW on these systems, or a comparable one using a Linux operating system, contact me at bonusj@uw.edu for reasonable assistance.
The original Claymore code is fast but has limitations in its accuracy and usability. We have built in a lot of user-interface changes (almost everything controlled by a JSON script similar to other engineering codes, and there is no need to recompile every time input is modified) and fixed physical inaccuracies (incorrect material law implementations, dimension scaling errors, general precision improvements, etc.) and CUDA issues (overwriting memory bounds, undefined behavior for some warp operations on newer architecture GPUs IIRC, not enabling full use of Shared memory from the L1 cache/carve-out for critical GPU kernels, etc.) have been mostly fixed. There are a couple of geotechnical material laws (Drucker-Prager, Non-associative Cam-Clay) that will need to be looked at a bit more closely (some "short-cuts" built into them that may need to be fixed).
Likewise we have added some materials like Neo-Hookean and another researcher, Dr. Javier Mroginski, is working on a Coupled U-P soil model. Multiple algorithms have also been added (FLIP, ASFLIP, F-Bar volumetric anti-locking, and basic coupling with Finite Elements). Additionally, multiple particle models per GPU of potentially different materials/algorithms are now supported in Multi-GPU usage (e.g. MPM sand with ASFLIP + FEM solids + MPM water with F-Bar anti-locking). We are currently looking to scale Claymore from Multi-GPU to Multi-Node + Multi-GPUs in the near-future. Nearly all features are currently accessible through a simple JSON script for Non-GPU programmers, the only instance that usually requires changes in the code is if adding a new feature or attempting to run a very memory intensive simulation.

0.4 ClaymoreUW - User Guide

Here we describe how a user can get started working with ClaymoreUW in their own custom simulations. This is not a technical task, as modifying input scene scripts in JSON is a very common and reasonably intuitive file format. The only true requirement is that the JSON "schema" is used, i.e. use the same variable names and structure as we will describe in this section.
Before continuing, make sure you have compiled and have a system capable of running ClaymoreUW by going through all steps in Section 0.3. Also, ensure that you can run an

example simulation provided in "scene.json" before struggling to debug any custom input scene.

Basic use of ClaymoreUW is entirely accessible through: **(i)** an input JSON script, and **(ii)** potentially minor changes of a few variables in file 'settings.h' in the appropriate Project folder. Construction of the input scene JSON script will be described here, along with common changes made to the settings file.

We will break-down each possible component in an input scene file, which must be formatted as JSON. By default, they should be placed within the same folder as the ClaymoreUW application, though this is not necessary if passing in the path to the input file when the application is called.

An example scene file, scene.json, is as follows:

```
{
    "simulation": {...},
    "bodies": [
        {
            "partition": {...},
            "material": {...},
            "algorithm": {...},
            "geometry": [{...}, ...],
            "output": {...},
        },
        ...
    ],
    "boundaries": [{...}, ...],
    "grid-sensors": [{...}, ...],
    "particle-sensors": [{...}, ...]
}
```

0.4.1 Input Parameters - Simulation

The `simulation` object contains basic simulation settings. For instance, the max domain size to simulate in, the gravity vector, the default time-step, and the default grid-cell size.

```
"simulation": {
    "domain": [8, 8, 8],
    "default_dx": 0.05,
    "gravity": [0,-9.81,0],
    "time": 0,
    "default_dt": 1e-3,
    "fps": 30,
    "frames": 10,
    "save_path": "./",
    "save_suffix": ".bgeo",
    "particles_output_exterior_only": false,
    "froude_scaling": 1.0
},
```

where variables are defined as:

- `domain`: Defines max dimensions of the simulation domain in each direction of a 3D simulation. Simulations that try to exceed domain will fail. Array with three elements. Each element must be a positive number greater than zero. [meters].
- `default_dx`: The default grid-cell length for the Material Point Method's background grid, Δx. Must be a positive number. [meters].
- `gravity`: The 3D gravity vector. Array with three elements. Each element must be a number. [meters / second^2]
- `time`: Initial time datum when the simulation starts. Must be a number. [seconds]
- `default_dt`: The default time-step for the Material Point Method, Δt. Will be overridden (i.e. decreased) if required to maintain stability for pressure wave speed of a material in the simulation. Must be a positive number greater than zero. [seconds]
- `fps`: Frames-per-second to output full particle files from the simulation. Must be an integer greater than zero. [seconds^{-1}]
- `frames`: Total number of frames to output during the simulation. Divided by the frames-per-second, this defines the total time duration of the simulation. Must be an integer greater than zero. [ ]
- `save_suffix`: File suffix to save the full particle output files with. Supports geometry (".geo") and binary geometry (".bgeo") currently, will be expanded to CSV, VTK, TXT, etc., later through use of PartIO library. Must be a string that begins with a period. [ ]
- `particles_output_exterior_only`: Whether or not to output only the particles near the exterior surface of a given full particle file. Saves a ton of memory in big simulations, but does not output all particles. Must be the Boolean true or false value. [boolean]
- `froude_scaling`: Whether or not to apply Froude similitude scaling to the simulation input scene. If true, reads in the scene file and scales length, velocity, time, mass, force, acceleration, etc. according to Froude similitude (e.g. λ^1, $\lambda^{0.5}$, $\lambda^{0.5}$, λ^3, λ^3, and 1). Must be a number greater than zero. [ ]

0.4.2 Input Parameters - Bodies

An array of `bodies` objects is used to create MPM and/or FEA objects. This includes specification of the device they each occupy (i.e. which GPU), the material properties, the algorithms to run (e.g. F-Bar antilocking, ASFLIP), the geometry to define them (e.g. a box, a loaded geometry from a SDF file), and any output settings specific to the model.

```
"bodies": [
    {
        "partition": {...},
        "material": {...},
        "algorithm": {...},
        "geometry": [{...}, ...],
        "output": {...},
    },
    ...
],
```

The `partition` object sets the Graphics Processing Unit (GPU) that holds the given model, as well as the model ID of the model on the GPU.

```
"partition": {
    "gpu": 0,
    "model": 0,
    "partition_start": [0, 0, 0],
    "partition_end": [1, 1, 1]
}
```

where variables are defined as:

- `gpu`: Graphics Processing Unit (GPU) ID to send this model to be simulated on. Must be a number that is between zero and the max number of supported GPUs on your system and compiled ClaymoreUW applications (set by g_device_cnt in `settings.h` file). [integer]
- `model`: Model ID on this GPU. Will eventually be done automatically for the user, but currently they must be set to be sequential integers starting at zero and increment according to appearance in the scene file. I.e., for GPU 0, first model 0 must appear in the scene, then model 1, model 2 (where each model is specific to the same GPU 0). Different model sequences for different GPU IDs may be interleaved in the scene file. [integer]

The `material` object is where material laws and parameters are set for the model. For instance, you may define water as:

```
"material": {
    "type": "particles",
    "ppc": 8,
    "constitutive": "JFluid",
    "CFL": 0.5,
    "rho": 1000,
    "bulk_modulus": 1e7,
    "gamma": 7.1,
    "viscosity": 0.001
}
```

where variables are defined as:

- `type`: Is the material represented as particles for MPM or as a mesh for FEA. Currently, must be a string that says either "particles" or "mesh". [string]
- `ppc`: Particle-per-cell for the model. If a power-of-two, will align to Newton-Cotes quadrature points. Must be a positive number greater than zero. [ ]
- `constitutive`: The material law to apply. Available options are "JFluid" (weakly-compressible isotropic fluid), "FixedCorotated" (hyper-elastic solid), "NeoHookean" (common hyper-elastic solid), "DruckerPrager" (Drucker-Prager, good for granular media), and "NACC" (Non-Associative Cam-Clay, good for clay, concrete, snow, and other basic topology changing solids). Must be a string stating one of the above key-words. [string]
- `CFL`: Courant-Friedrichs-Lewy condition number C for stability of the numerical method. May over-ride the default time-step `default_dt`, Δt, in the `simulation` object by forcing $\Delta t \leq C\Delta x / v_{\text{p-wave}}$ where p-wave velocity $v_{\text{p-wave}}$ is specific to a material, see Section **??**. Recommended to use a value between 0.1 and 0.6. Must be a positive number. [integer]
- `rho`: Density of the material at simulation start. [kilograms / meter^3]
- `bulk_modulus`: Elastic bulk modulus, B, of the material at simulation start. Measure of material incompressibility. Used for the "JFluid" constitutive model. [Pascals]
- `gamma`: Derivative of the bulk modulus w.r.t. initial pressure. Measure of nonlinear response to volume change. Used in the Tait-Murnaghan equation of state inside of the "JFluid" constitutive model, see Section **??**. [ ]
- `viscosity`: Dynamic viscosity of the material. Measure of stress response to time-dependent deformation. Used in the "JFluid" constitutive model. Must be a positive number. [Pascals * seconds]
- `youngs_modulus`: Elastic Young's modulus of the material at simulation start. Measure of inline stress-strain response. Used in "FixedCorotated", "NeoHookean", "DruckerPrager", and "NACC". Must be a positive number greater than zero. [Pascals].
- `poisson_ratio`: Poisson's ratio of the material at simulation start. Measure that relates axial deformation to 3D transverse deformation. Value of 0.5 is a truly incompressible material, 0 is fully compressible, and negative values are unusual expansive materials (e.g. cork). Must be a number less than 0.5. [ ]
- `friction_angle`: Friction angle to be used in the "DruckerPrager" material model. [degrees]

- `cohesion`: Cohesive value for use in the "DruckerPrager" material model. Currently based on the logarithm of strain, i.e. adjust the yield surface to account for said amount of cohesive strain. To be updated. [ ]

The algorithms applied to a model are set in the `algorithm` object per model:

```
"algorithm": {
    "type": "particles",
    "ppc": 8.0,
    "use_ASFLIP": true,
    "ASFLIP_alpha": 0.3,
    "ASFLIP_beta_max": 0.01,
    "ASFLIP_beta_min": 0.0,
    "use_FBAR": true,
    "FBAR_psi": 0.8,
    "FBAR_fused_kernel": true,
    "use_FEM": false
}
```

where variables are defined as:

- `type`: Is the material represented as particles for MPM or as a mesh for FEA. Currently, must be a string that says either "particles" or "mesh" (NOTE: "mesh" currently deprecated). [string]
- `ppc`: Particle-per-cell for the model. If a power-of-two, will align to Newton-Cotes quadrature points. Must be a positive number greater than zero. [ ]
- `use_ASFLIP`: True/false for using Affine-Separable Fluid-Implicit-Particles. Must be true to use ASFLIP, FLIP, or PIC-FLIP mixing. [Boolean]
- `ASFLIP_alpha`: The ASFLIP and PIC-FLIP α varies velocity advection mixing between PIC ($\alpha = 0$) and FLIP ($\alpha = 1$). Higher number gives a more energetic simulation, but with greater risk for noise and typically smaller required time-step. Must be a number within 0 and 1, $\alpha \in [0, 1]$. [ ]
- `ASFLIP_beta_max`: The ASFLIP positional correction ratio maximum value, β_{max}. Larger number allow greater positional correction of particles in spurious tensile conditions and may improve the contact behavior between bodies and material. However, larger values risk greater noise and instabilities. Best to keep this small, e.g. $\beta_{max} \in [0, 0.2]$, though any value between 0 and 1 is permissible, $\beta_{max} \in [0, 1]$. [ ]
- `ASFLIP_beta_min`: The ASFLIP positional correction ratio minimum value, β_{min}. Larger number allow greater positional correction of particles in ambient conditions and may improve the contact behavior between bodies and material. However, larger values risk greater noise and instabilities. Best to keep this small, e.g. $\beta_{min} \in [0, 0.05]$, though any value between 0 and 1 is permissible, $\beta_{min} \in [0, 1]$. [ ]

- `use_FBAR`: True/false for using F-Bar volumetric antilocking (see Sections **??**, **??**, **??**, and **??**). Can greatly improve pressure field and reduce volumetric antilocking. [Boolean]
- `FBAR_fused_kernel`: True/false for using F-Bar volumetric antilocking in G2P2G fused kernel (true, faster) or G2P + P2G split kernels (false, slower). See section **??** for more details. [Boolean]
- `FBAR_psi`: Mixing ratio of F-Bar volumetric antilocking, ψ. Values of $\psi = 0$ and $\psi = 1$ reduce to no antilocking and full antilocking, respectively. In stiff simulations, we typically use between 0.5 and 0.9999, although all values between 0 and 1 are permissible, $\psi \in [0, 1]$. [ ]
- `use_FEM`: True/false for using finite element method. Pending update to improve user-interface for defining vertices and elements. [Boolean]

The geometries composing a model are set in the `geometry` array of objects per model:

```
"geometry": [
    {
        "object": "Box",
        "operation": "Add",
        "span": [0.25, 0.5, 1.0],
        "offset": [0, 1.6, 0],
        "rotate": [0,0,0],
        "fulcrum": [0, 0, 0],
        "array": [8, 4, 2],
        "spacing": [0.5, 1.0, 2.0]
    },
    ...
]
```

where variables are defined as:

- `object`: Type of geometry object to use. E.g., basic geometries like "box", "sphere", "cylinder". May also set it to "file' to load-in user-provided files (must provided file-path in different variables). [string]
- `operation`: Geometry operation to apply. E.g. "add", "subtract", and eventually "difference", "intersection", and "union". These are Boolean arithmetic operations that allow a user to define any geometry with multiple geometry objects using varying operations. Operation is applied in serial, e.g. "subtract" in the third geometry object subtracts from objects one and two, but "add" in the fourth will be impervious to the prior subtraction. [string]
- `span`: Dimensions of the geometry in each direction. Array of three elements. Elements must be numbers greater than zero. [meters]

- `offset`: Offset of the geometry's origin compared to the simulation origin. Array of three elements. Elements must be numbers. [meters]

- `rotate`: Euler angles of rotation to apply to the geometry. Array of three elements. Elements must be numbers. [degrees]

- `fulcrum`: Point to center rotations on. Array of three elements. Elements must be numbers. [meters]

- `array`: How many geometry objects to create in each direction (X, Y, Z). Array of three elements. Elements must be integers greater than or equal to 1. [integers]

- `spacing`: Spacing of geometry objects in each dimension (Sx, Sy, Sz). Array of three elements. Elements must be numbers, and numbers must be greater than or equal to the corresponding `span` element to avoid self-intersection between array bodies. [meters]

- `file`: If `object` selected is set to "file", you must provide the file-path in this variable. Relative to the ClaymoreUW AssetDir environment variable, default is "ClaymoreUW/Data/". [file-path]

- `padding`: If `object` is set to "file" and `file` was provided and SDF file-path, this declares the padding used in the SDF file. SDF file require a minimum padding of 1 to define an objects surface, so this should be an integer greater than or equal to 1 if using and SDF file geometry. [integer]

- `scaling`: If `object` is set to "file" and `file` was provided an SDF, CSV, BGEO, etc. file-path then this will scale the linear dimensions of that file's object by the value provided. I.e., 2 will double linear dimensions. Must be a positive number greater than zero. [ ]

NOTE: The following section on `output` is not currently implemented. Move all items within it to the global-scope of the element in `bodies` (i.e. adjacent to where `output` is currently in the JSON).
Output settings for a model are given in the `output` object per model:

```
"output": {
    "output_attribs": ["Pressure", "ID"],
    "track_attribs": ["Position_X", "Position_Y", "Position_Z",
    ↪  "Velocity_Magnitude", "Velocity_X",  "Velocity_Y", "Velocity_Z"],
    "target_attribs": ["Position_Y"]
}
```

where variables are defined as:

- `output_attribs`: List of attributes to output per particle in each full output file. ID, Mass, Volume, Position_X, Position_Y, Position_Z, Velocity_X, Velocity_Y, Velocity_Z, Velocity_Magnitude, DefGrad_XX, DefGrad_XY, DefGrad_XZ, DefGrad_YX, DefGrad_YY, DefGrad_YZ, DefGrad_ZX, DefGrad_ZY, DefGrad_ZZ, J, DefGrad_Determinant = J, JBar, DefGrad_Determinant_FBAR = JBar, StressCauchy_XX, StressCauchy_XY, StressCauchy_XZ, StressCauchy_YX, StressCauchy_YY, StressCauchy_YZ, StressCauchy_ZX, StressCauchy_ZY, StressCauchy_ZZ, Pressure, VonMisesStress, DefGrad_Invariant1, DefGrad_Invariant2, DefGrad_Invariant3, DefGrad_1, DefGrad_2, DefGrad_3, StressCauchy_Invariant1, StressCauchy_Invariant2, StressCauchy_Invariant3, StressCauchy_1, StressCauchy_2, StressCauchy_3, StressPK1_XX, StressPK1_XY, StressPK1_XZ, StressPK1_YX, StressPK1_YY, StressPK1_YZ, StressPK1_ZX, StressPK1_ZY, StressPK1_ZZ, StressPK1_Invariant1, StressPK1_Invariant2, StressPK1_Invariant3, StressPK1_1, StressPK1_2, StressPK1_3, StressPK2_XX, StressPK2_XY, StressPK2_XZ, StressPK2_YX, StressPK2_YY, StressPK2_YZ, StressPK2_ZX, StressPK2_ZY, StressPK2_ZZ, StressPK2_Invariant1, StressPK2_Invariant2, StressPK2_Invariant3, StressPK2_1, StressPK2_2, StressPK2_3, StrainSmall_XX, StrainSmall_XY, StrainSmall_XZ, StrainSmall_YX, StrainSmall_YY, StrainSmall_YZ, StrainSmall_ZX, StrainSmall_ZY, StrainSmall_ZZ, StrainSmall_Invariant1, StrainSmall_Invariant2, StrainSmall_Invariant3, Dilation = StrainSmall_Invariant1, StrainSmall_Determinant = StrainSmall_Invariant3, StrainSmall_1, StrainSmall_2, StrainSmall_3, VonMisesStrain, PorePressure, and logJp currently supported. Array of arbitrary number of elements. [strings]

- `track_attribs`: Attribute(s) on tracked particles to output. See `output_attribs` for supported particle attributes. Array of arbitrary number of elements. [string]

- `target_attribs`: Attribute(s) on particle-sensor to output. See `output_attribs` for supported particle attributes. Array of arbitrary number of elements. [strings]

There are a few remaining so-called global variables (within the scope of a model object) that require specification. Setting these outside of a geometry object will apply them to all geometry objects of the model. For instance:

```
"bodies": [
    {
        "velocity": [1, 0, 0],
        "track_particle_id": [0, 1000, 2000, 3000],
        "partition_start": [0,0,0],
        "partition_end": [4,4,4],
        "gpu": 0,
        "model": 0,
        ...
    },
]
```

where variables are defined as:

- `gpu`: Graphics Processing Unit (GPU) ID to send this model to be simulated on. Must be a number that is between zero and the max number of supported GPUs on your system and compiled ClaymoreUW applications (set by g_device_cnt in `settings.h` file). [integer]
- `model`: Model ID on this GPU. Will eventually be done automatically for the user, but currently they must be set to be sequential integers starting at zero and increment according to appearance in the scene file. I.e., for GPU 0, first model 0 must appear in the scene, then model 1, model 2 (where each model is specific to the same GPU 0). Different model sequences for different GPU IDs may be interleaved in the scene file. [integer]
- `velocity`: Initial velocity of all particles in the model. Array of three elements. Elements must be numbers. [meter / second]
- `track_particle_id`: List of global model IDs to track, i.e. particles with these IDs are specifically tracked for output. Attribute(s) to track on tracked particles should be specified in `track_attribs`. Array of arbitrary number of elements. Elements must be integers greater than or equal to zero. [integers]
- `partition_start`: Any particle that is closer to the origin than this point at simulation start is removed. Used to help split things across GPU partitions. Array of three elements. Elements must be numbers. [meters]
- `partition_end`: Any particle that is further to the origin than this point at simulation start is removed. Used to help split things across GPU partitions. Array of three elements. Elements must be numbers. [meters]

This concludes input parameters for the `bodies` object array.

0.4.3 Input Parameters - Grid Boundaries

Grid boundaries are set as an array of objects. Each will define a unique boundary on the grid with associated contact type, motion, etc., where appropriate.

```
"boundaries": [
    {
        "object": "Wall",
        "contact": "Slip",
        "friction_static": 0.0,
        "friction_dynamic": 0.0,
        "domain_start":[0,0,0],
        "domain_end": [8,8,8]
    },
    ...
]
```

where variables are defined as:

- `object`: Type of boundary object to use. E.g., Walls, Box, Plane, OSU_LWF_RAMP, OSU_LWF_PADDLE, USGS_RAMP, USGS_GATE, OSU_TWB_RAMP, OSU_TWB_PADDLE, WASIRF_PUMP, TOKYO_HARBOR, velocity_boundary. Note that some prefixes refer to boundaries specific to the digital twins of the Oregon State University Large Wave Flume (OSU_LWF, Chapter **??**), Oregon State University Directional Wave Basin (OSU_TWB, Section **??**), U.S. Geological Survey Debris-Flow Flume (USGS, Section **??**), University of Washington Flume Wind-and-Sea Interaction Facility (WASIRF, Section **??**), and Waseda University's Tsunami Wave Basin (TOKYO, Chapter **??**). [string]
- `contact`: Contact type for application of boundary condition. Includes "sticky", "slip", and "separable". See Section **??** for more details. [string]
- `friction_static`: Static friction coefficient to apply on boundary. Not all boundaries support this currently, except Plane, Floor, TOKYO_Harbor, USGS_RAMP. Coulomb friction model used. Must be a number greater than or equal to zero. [ ]
- `friction_dynamic`: Static dynamic coefficient to apply on boundary. Not all boundaries support this currently, except Plane, Floor, TOKYO_Harbor, USGS_RAMP. Coulomb friction model used. Must be a number greater than or equal to zero. [ ]
- `time`: Time that the boundary is active. Array of two elements, first element defines time that it appears, second element is the time that is disappears. Elements must be numbers. [seconds]
- `domain_start`: Where to begin the boundary domain, i.e. the point nearest to the origin. Array of three elements. Elements must be numbers. [meters]
- `domain_end`: Where to end the boundary domain, i.e. the point furthest from the origin. Array of three elements. Elements must be numbers greater than corresponding element in `domain_start`. [meters]
- `file`: File-path to the motion file for a moving boundary. Currently, only takes in CSV files with columns (no headers) specifying time, velocity in X, and position in X (relative to `domain_start` and `domain_end`). Used to define wave-maker piston motion. [file-path]
- `output_freq`: Frequency of sampling in the motion file. Subsequent entries assumed to be at increments of its inverse. [Hz]
- `velocity`: Constant velocity boundary. Array of three numbers. [meters / second]

The original Claymore also featured a means to load-in generalized boundaries defined by an SDF file. This was, however, very memory inefficient (had to define the SDF file over

and SDF discretization equal to the exact simulation grid domain) so it was deprecated. It will eventually be re-implemented in ClaymoreUW with a better user-interface and memory system (non-conformal discretization of SDF relative to grid). Will also include generalized application of friction on motion (translation and rotational) for ClaymoreUW.

0.4.4 Input Parameters - Grid Sensors

Sensors, i.e. instrumentation, to add to the simulation are split across the grid and the particles currently. Grid-sensor instruments, e.g. a load-cell that measures forces on a rigid MPM boundary using grid node forces, may be defined with:

```
"grid-sensors": [
    {
        "attribute": "Force",
        "operation": "Sum",
        "direction": "Z+",
        "output_frequency": 120,
        "domain_start": [-0.1, -0.1, 8],
        "domain_end": [8.05, 0.5, 8.1]
    },
    ...
]
```

where variables are defined as:

- `attribute`: Attribute on a grid-node that our sensor instrument will measure. "Force", "Acceleration", "Momentum", "Velocity", "Mass", "Volume", and "JBar" currently supported. [string]
- `operation`: Reduction operation to perform on the `attribute`. E.g. "Sum", "Max", "Min", "Count", "Average", and "STDEV". Just about any sensor should be possible using these fundamental reductions. [string]
- `direction`: The direction to measure with respect to if `attribute` concerns a vector quantity. E.g., "X" will measure vectors projected onto the X axis, "X-" will only take vectors that point opposite of the unit X vector after project onto the X axis, "X+" will only take vectors that point in the unit X vector after project onto the X axis. "Y" and "Z" (and their variants) also applicable. Will eventually use a vector instead, for now must provide a string. [string]
- `output_frequency`: Frequency to perform and output the reduction operation of the sensor instrument. I.e. if a load-cell that measures force has a 120 Hz operating frequency, use 120. [Hz]
- `domain_start`: Where to begin the sensor domain, i.e. the point nearest to the origin. Array of three elements. Elements must be numbers. [meters]

- `domain_end`: Where to end the sensor domain, i.e. the point furthest to the origin. Array of three elements. Elements must be numbers. [meters]

0.4.5 Input Parameters - Particle Sensors

Likewise, the particle-sensors, e.g. a free-surface elevation gauge (a.k.a. wave-gauge), are defined similarly to grid-sensors:

```
"particle-sensors": [
    {
        "attribute": "Elevation",
        "operation": "Max",
        "output_frequency": 120,
        "domain_start": [4.0, 1.6, 0.0],
        "domain_end":   [4.1, 8.0, 0.1]
    },
    ...
]
```

where variables are defined as:

- `attribute`: Attribute on a particle that our sensor instrument will measure. ID, Mass, Volume, Position_X, Position_Y, Position_Z, Velocity_X, Velocity_Y, Velocity_Z, Velocity_Magnitude, DefGrad_XX, DefGrad_XY, DefGrad_XZ, DefGrad_YX, DefGrad_YY, DefGrad_YZ, DefGrad_ZX, DefGrad_ZY, DefGrad_ZZ, J, DefGrad_Determinant = J, JBar, DefGrad_Determinant_FBAR = JBar, StressCauchy_XX, StressCauchy_XY, StressCauchy_XZ, StressCauchy_YX, StressCauchy_YY, StressCauchy_YZ, StressCauchy_ZX, StressCauchy_ZY, StressCauchy_ZZ, Pressure, VonMisesStress, DefGrad_Invariant1, DefGrad_Invariant2, DefGrad_Invariant3, DefGrad_1, DefGrad_2, DefGrad_3, StressCauchy_Invariant1, StressCauchy_Invariant2, StressCauchy_Invariant3, StressCauchy_1, StressCauchy_2, StressCauchy_3, StressPK1_XX, StressPK1_XY, StressPK1_XZ, StressPK1_YX, StressPK1_YY, StressPK1_YZ, StressPK1_ZX, StressPK1_ZY, StressPK1_ZZ, StressPK1_Invariant1, StressPK1_Invariant2, StressPK1_Invariant3, StressPK1_1, StressPK1_2, StressPK1_3, StressPK2_XX, StressPK2_XY, StressPK2_XZ, StressPK2_YX, StressPK2_YY, StressPK2_YZ, StressPK2_ZX, StressPK2_ZY, StressPK2_ZZ, StressPK2_Invariant1, StressPK2_Invariant2, StressPK2_Invariant3, StressPK2_1, StressPK2_2, StressPK2_3, StrainSmall_XX, StrainSmall_XY, StrainSmall_XZ, StrainSmall_YX, StrainSmall_YY, StrainSmall_YZ, StrainSmall_ZX, StrainSmall_ZY, StrainSmall_ZZ, StrainSmall_Invariant1, StrainSmall_Invariant2, StrainSmall_Invariant3, Dilation = StrainSmall_Invariant1, StrainSmall_Determinant = StrainSmall_Invariant3, StrainSmall_1, StrainSmall_2, StrainSmall_3, VonMisesStrain, PorePressure, and logJp currently supported. [string]

- `operation`: Reduction operation to perform on the `attribute`. E.g. "Sum", "Max", "Min", "Count", "Average", and "STDEV". Just about any sensor should be possible using these fundamental reductions. Note that the current build of ClaymoreUW is changing the way reductions are implemented, so not all may be viable (though Max and Sum should likely be operational). [string]

- `output_frequency`: Frequency to perform and output the reduction operation of the sensor instrument. I.e. if a wave-gauge that measures elevation has a 120 Hz operating frequency, use 120. [Hz]

- `domain_start`: Where to begin the sensor domain, i.e. the point nearest to the origin. Only measure particles past this point. Array of three elements. Elements must be numbers. [meters]

- `domain_end`: Where to end the sensor domain, i.e. the point furthest to the origin. Only measures particles before this point. Array of three elements. Elements must be numbers. [meters]

0.5 ClaymoreUW - Developer Guide

With an understanding of the MPM code implementation in Multi-GPU ClaymoreUW, this section summarizes the actual file-structure of a ClaymoreUW application for the convenience of developers.
The Claymore open-source project is well written but not at all intuitive to those accustomed to engineering software. Below is a brief synopsis of the file structure and constituent functions so one may better navigate and add to the code-bases of either Claymore or ClaymoreUW.

- `scene.json` - Simulation run-time variables are set here. This JSON script includes file locations for models, specification of material properties, setting of force recorders, etc.

- `settings.h` - Simulation global compile-time variables are set here. This file includes max particle counts, max particles per cell, domain length, domain resolution, default material properties.

- `project.cu` - Here is where the simulation begins. This file handles the run-time input from an input script (`scene.json`) for the simulation. Initializes particle models, among other things, for passing into the actual simulation (found in `mgsp_benchmark.cuh`).

- `mgsp_benchmark.cuh` - This includes the simulation workflow. Multi-GPU consensus is organized here. This file serves as a wrapper of sort for launching GPU kernels. Output of entities is organized here.

- `mgmpm_kernels.cuh` - Here is where the majority of the GPU kernel functions reside. This file includes the full Material Point Method Algorithm (Grid update, G2P2G, etc.), as well as data-structure management and input/output schemes for device arrays. Many functions are material specific. Changes to the MPM algorithm are implemented here and called from the wrapper (`mgsp_benchmark.cuh`).
- `constitutive_models.cuh` - This file contains GPU device functions for the material laws (e.g. Fixed-Corotated, Weakly-Compressible Fluid a.k.a. JFluid).
- `particle_buffer.cuh` - This file organizes the data-structures for the particle models to be used in GPU kernels. Sets material specific properties.
- `grid_buffer.cuh` - This file organizes the data-structures for the grids to be used in GPU kernels. Sets scheme specific grid-node attributes (e.g. ASFLIP requires more grid-node velocity information than standard MPM). Dual-grids, staggered grids, etc. can be implemented here.
- `halo_buffer.cuh` - This file organizes the data structures for the halo regions to be used in GPU kernels.
- `hash_table.cuh` - Rarely need to edit this file unless trying to improve memory usage/change data-structure behavior. This file provides a hash-table mapping for fast access of sparse, complex GPU data-structures.
- `halo_kernels.cuh` - This file contains GPU device functions for operating the halo-regions between GPUs.
- `partition_domain.h` - This file partitions the domain. Not used in current state of code.
- `utility_funcs.hpp` - This file includes utility functions for convenience, such as finding the dyadic product for quadratic B-spline kernels. Not always utilized.

0.6 ClaymoreUW - Algorithm

Claymore uses a specific Moving-Least-Squares Material Point Method (MLS-MPM, Section **??**) formulation with explicit time-integration and Quadratic B-Spline shape-functions. In this section, the grid-to-particle-to-grid fused kernel (G2P2G) [Wang et al.2020b] used in Claymore is briefly described relative to the MPM algorithm execution. The following steps are used and show in Figure 2:
Update Grid. The grid state is updated. Velocity evolves with respect to boundary conditions. $\mathbf{v}_i^n \rightarrow \mathbf{v}_i^{n+1}$
G2P2G. Fused step involving G2P, Particle Advection, and P2G: *G2P* is when all relevant grid velocity is transferred onto particles to reconstruct particle velocities and

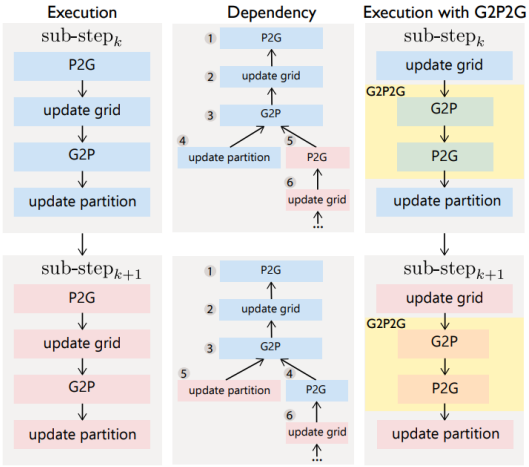

Fig. 2. G2P2G MPM Algorithm - Claymore Software. Grid-to-particle-to-grid fused kernel for Material Point Method in Claymore software. Courtesy of [Wang et al.2020b].

particle affine velocity matrices. $\mathbf{v}_i^n \rightarrow \mathbf{v}_p^{n+1}, \mathbf{C}_p^{n+1}$. *Particles Advection* occurs. $\mathbf{v}_p^{n+1} \rightarrow \mathbf{x}_p^{n+1}$. *Particles Evolve* deformation gradients and stress. $\mathbf{F}_p^n \rightarrow \mathbf{F}_p^{n+1}, \boldsymbol{\sigma}_p^{n+1}$. *P2G* transfers particle mass, momentum, affine momentum, and internal force to relevant grid-nodes. $m_p, \mathbf{v}_p^{n+1}, \mathbf{C}_p^{n+1}, \boldsymbol{\sigma}_p^{n+1} \rightarrow m_i, m_i \mathbf{v}_i^{n+1}$.

Update Partition. Data-structures are updated for the next time-step. Unoccupied grid-blocks are deactivated. This is essentially software overhead.

Claymore's parallel code implementation uses the MLS-MPM algorithm pipeline and G2P2G step described above. A few notes on the GPU kernel implementation is presented next.

0.7 ClaymoreUW - GPU Kernels

The high-level MPM workflow in Claymore and ClaymoreUW (Sec. 0.6) is organized as a series of kernels implemented in CUDA C++. Said code may be accessed in the respective open-source github repositories. This section presents the steps of the parallel MLS-MPM algorithm of ClaymoreUW used in each GPU kernel and how they are optimized to run on Multi-GPU systems. Only kernels relating to the numerical algorithm are described, we do not include all book-keeping functions used for managing back-end data-structures. Thread/Block/Grid design, shared memory usage, Multi-GPU MPM consensus, and other aspects necessary for understanding the software are covered. For an introduction to GPUs and CUDA review see Chapter **??**. The instruction pipeline of Claymore is shown in Figure 3 for single, naive Multi-GPU (non asynchronous Halo/Non-Halo), and Multi-GPU Static Partitioning (MGSP, i.e. partition scene across GPUs by undeformed particle location).

Update Grid. This function updates the grid state (i.e. velocities) and solves the equation of motion on the shared MPM grid. Kernel `update_grid_velocity_query_max()` is called

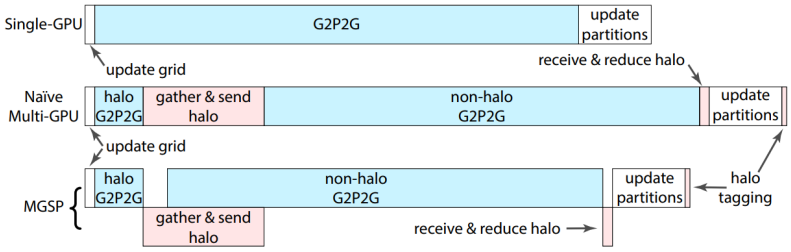

Fig. 3. Instructions for Single and Multi GPU Material Point Method - Claymore Software. Instruction pipe-line for both Single and Multi-GPU MPM in Claymore. Intelligent Multi-GPU MPM design uses CUDA asynchronous properties to hide latency, approaching Single-GPU speeds. Courtesy of [Wang et al.2020b].

here. CUDA blocks contain one or more grid-blocks each. Threads pull in data from a grid node (corresponding to thread index) in the grid block (corresponding to the block index) from the grid buffer. Basic boundary conditions are determined and applied to the grid node velocities before writing results to the next grid buffer. Each CUDA block receives a shared memory array to hold max velocity values. These atomically reduce to a single max velocity per warp. Warp max values are reduced to a single max value, used for the CFL condition.

G2P2G. This function is the grid-to-particle-to-grid fused algorithm, as shown in Figure 2, that runs the bulk of MLS-MPM computations. Kernel `g2p2g()` is called on a material basis (i.e. unique `g2p2g()` exists and is optimized for each material). Shared memory per CUDA block is reserved for two arrays, representing the prior and posterior grid-arena state relative to a particle-block as shown in Figure 6. The first is the `g2pbuffer` which uses 4 x 3 x 512 bytes, the second is the `g2pbuffer` which uses 4 x 4 x 512 bytes. Respectively, this is for three and four floats per grid node in a grid arena (i.e. velocity vector in G2P, mass and momentum vector in P2G). `g2pbuffer` is populated efficiently by threads pulling values from grid nodes in the correct grid blocks in the grid buffer. This is done in a manner designed to avoid bank conflicts. `p2gbuffer` is zeroed initially. A shared pool of memory representing all needed grid information for G2P is now available in `g2pbuffer`. Each thread will now represent a particle in a particle bin of the particle block being executed by the CUDA block of `g2p2g()`. The thread pulls in particle data (e.g. position, deformation gradient) from global memory into the register. MLS-MPM is performed now, using particle data and the shared `g2pbuffer`. Particles advect velocity and position. Particles evolve deformation and stress. Updated particle values are written back into the global memory particle buffer. P2G now starts. Particle mass, momentum, and internal force contributions atomically add into the shared `p2gbuffer` of the CUDA block. When all threads complete of the block complete, the `p2gbuffer` is written (atomically added) into the global grid buffer in a coalesced manner for efficiency.

Update Partition. In this function overhead tasks and consensus are performed. This includes Multi-GPU communication (lock, send, receive, unlock), particle bin reorganizing, grid block activation/deactivation, and halo block tagging.

Input/Output. These function ares used to input and output data into the MPM simulation. This is where input of new data on the host (CPU, RAM, HDD, SSD) to be written into the device (GPU) is done, such as updated boundary conditions. For this purpose, data is typically arranged as basic host arrays which asynchronously copy into a basic device array. Asynchronous operation prevents this process from slowing down the Multi-GPU program. Then, if needed, a kernel is launched to map basic device arrays into more complex data-structures described in Section 0.8. Output entails the same process in reverse. Our lab has added extensive functionality to output options so engineers may better understand the simulations they are performing without manually having to find a way to pull out data from complex Multi-GPU data-structures (e.g. summed force on a specified surface at a frequency).

0.8 ClaymoreUW - Data Structures

Data-structures are the largest barrier preventing engineers from enjoying the benefits of Multi-GPU numerical simulations. The arrangement of numerical bodies (particles, grid-nodes, elements), the means to access them, and the properties they possess are all managed by data-structures. Maintenance and use of data-structures on Multi-GPUs for hundreds of millions of numerical bodies is costly in time and memory. For programmers without traditional training in data-structures they are also incredibly unwieldy to use. Often times they are the singular obstacle preventing the implementation of numerical features (e.g. implicit time-integration, Poisson equation solvers) as these features may require data-usage that is difficult to organize on a Multi-GPU system.

To accommodate a typical user's development time, complex data-structures should be interacted with at only the highest, most abstracted level. However, at the lowest level these data-structures should be optimized for memory and computation usage on multiple GPUs. This is a tall order.

Modern advancements in numerical simulations, especially in the field of computer graphics, have aimed to abstract away a users interaction with complex GPU data-structures. Taichi is an excellent high-performance programming language for numerical simulations and visualizations [Hu et al.2019a] that, among other things, allows users to design complex data-structures (e.g. hierarchical, sparse, dynamic, etc.) with a front-end that requires minimal to no CUDA C++ and/or GPU expertise. In addition, Taichi makes the access to these structures convenient and their use well optimized. Claymore [Wang et al.2020b] follows a similar design philosophy, as described in their technical supplement [Wang et al.2020a] which is described in Section 0.8 as it is used in ClaymoreUW. However, neither code is

specifically intended for engineering applications, and thus they present a learning curve that we hope to alleviate in some small part here.

A few terms will be used throughout this section to describe the data moving through a Multi-GPU MPM Claymore simulation. Some of the specifications are specific to supporting quadratic B-spline shape-functions and may grow or shrink with cubic B-splines or tri-linear shape-functions respectively. At a high level, data is organized as follows.

Particles, also referred to as "material points" in MPM, are the smallest unit of the material domain. They serve as quadrature points numerically and are free to move through the domain. Particles hold on to whatever independent information they need to update. Figure 4 shows how a simple fluid or solid particle may store information differently. An isotropic fluid may hold position (3x1 vector, 3 floats) and volume change ratio (scalar, 1 float). A linear elastic solid may hold position (3x1 vector, 3 floats) and deformation gradient (3x3 tensor, 9 floats). Shared, static particle properties are not saved on each particle (e.g. Poisson's Ratio for linear elastic material). Therefore, memory usage is material/scheme dependent. For certain applications, particles may need to hold velocity (3x1 vector, 3 floats) or unique material parameters (e.g. for plasticity). For example, Fixed-Corotated solids using a PIC-FLIP advection scheme requires position (3 floats), velocity (3 floats), and deformation gradient (9 floats). In Claymore, particles are executed by an individual CUDA thread.

Grid Nodes are the smallest unit of the background, "scratch-pad" domain. They are used to solve the equation of motion (EOM) and to apply boundary conditions. Grid nodes hold whatever information they need to solve the EOM, as well as other values used for boundary conditions or augmented G2P transfers. Typically they hold mass (scalar, 1 float) and momentum/velocity (3x1 vector, 3 floats). Grid nodes are arranged in a uniform Cartesian pattern which resets to its initial position each time-step so they do not need to hold position information. Some schemes (e.g. FLIP/ASFLIP) require grid-nodes to hold an extra velocity vector, while advanced boundary conditions may require a surface normal (3x1 vector).

Particle Bins are groups of 32 nearby particles. 32 specifically optimizes for GPUs (See Chapter **??**). Bins are executed in parallel by a CUDA warp. Particle bins reconstruct every time-step to maintain the spatial condition of 32 particles within a 4 x 4 x 4 grid-node domain (for Quad. B-Splines shape-functions). They are selected in a way that reduces the chance that any two particles in a bin occupy the same grid-cell, minimizing bank conflicts when particles in the warp read/write from/to shared grid-nodes for G2P/P2G transfers. Figure 4 shows how a particle bin organizes particle data as structures-of-arrays (SoA) in GPU global memory.

Particle Blocks are dynamic groups of Particle Bins that are located in the same 4 x 4 x 4 grid-node group (for Quad. B-Spline shape-functions). Their dynamic quality allows for better memory usage in simulations where particles are not evenly distributed. Figure 5

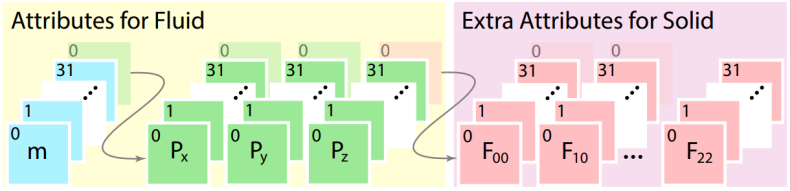

Fig. 4. Particle Bin Attributes by Material - Claymore Software. Attributes on particles in Multi-GPU MPM software Claymore visualized. Different material models require differing information, e.g. 4 floats per fluid particles and 13 floats per fixed-corotated solid. Arranged in bins of 32 to match NVIDIA warp size. Courtesy of [Wang et al.2020b].

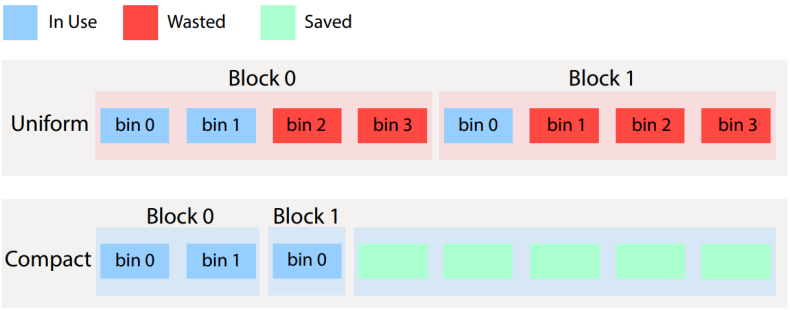

Fig. 5. Particles Binning in Particle Block - Claymore Software. Particle Blocks (4 x 4 x 4 cell size) dynamically resize to fit Particle Bins (32 particles) in its domain. Saves memory but has overhead. Courtesy of [Wang et al.2020b].

visually shows that the applied dynamic resizing buffer of compact particle bins scheme saves memory compared to a non-dynamic approach where no resizing occurs.

Grid Blocks are 4 x 4 x 4 adjacent grid-nodes (for Quad. B-Spline shape-functions). These are the smallest unit of grid-nodes that are actually used in most kernels, as G2P and P2G operations require many grid-nodes to support particle blocks. Grid blocks are considered active if any interior grid-node possesses mass and inactive if no grid-node has mass (i.e. memory freed for the next time-step).

Grid Arenas are groups of 2 x 2 x 2 adjacent grid blocks (8 x 8 x 8 grid-nodes = 512 grid-nodes). Grid arena size is set to maintain the off-by-two requirement for a particle block execution in the G2P2G kernel. Figure 6 show a schematic of a grid arena for a 2D case. Each grid-node holds a minimum of 3 floats for G2P operation, and 4 floats for P2G operation. They are the smallest set of grid-nodes a full particle block will read/write from/to during the G2P2G operation (Sec. 0.7). Grid arenas are an array of grid-block information pulled into shared memory for optimized particle reads and writes within a CUDA block's scope. At the end of G2P2G, they hold inter-block atomically reduced particles contributions from G2P2G, which then intra-block reduce to achieve consensus on the full, shared grid. Memory for grid arenas is manually reserved as shared memory for the kernel.

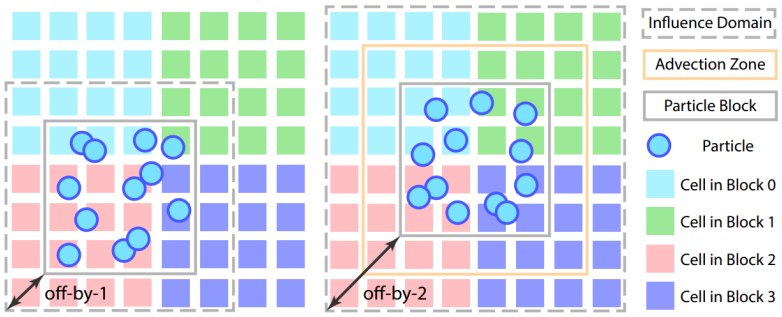

Fig. 6. Off-By-2 - Claymore Software. G2P2G kernel requires off-by-2 scheme, i.e. a given particle block entering G2P2G needs read/write access to a Grid Arena (2 x 2 x 2 Grid-Blocks) which is reserved in shared memory. It assumes CFL condition and Quadratic B-Spline shape-functions. Particle Block must be centered which creates a 2 grid-node buffer relative to the Grid Arena, hence "Off-By-2". Courtesy of [Wang et al.2020b].

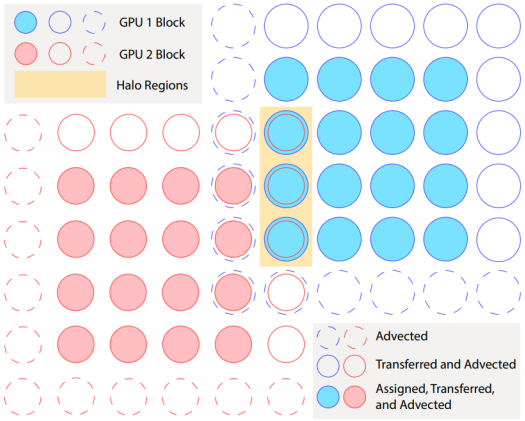

Fig. 7. Organizing MPM Grid Blocks for Multi-GPU - Claymore Software. Halo-Blocks are tagged for Multi-GPU transfer. Courtesy of [Wang et al.2020b].

Halo Blocks are grid and/or particle blocks in overlap regions between GPU devices. A halo-tagging process is undertaken to determine which blocks are relevant and which devices they must be sent to. These are the smallest unit for Multi-GPU communication. Transferring/receiving grid-nodes and/or particles in blocks, which are already optimized for CUDA reads/writes, is far more efficient than sending information from individual particles and grid-nodes. Multi-GPU MPM tasks are divided between operations that influence only halo blocks and operations influencing only device specific blocks (non-halo blocks). Halo block regions are seen as a yellow rectangle in Figure 7 for a dual-GPU simulation. Results in these regions are shared between the GPUs.

Non-Halo Blocks are blocks that do not overlap in regions between partitions of the GPU device. They do not require special treatment. MPM operations on non-halo blocks do not need to communicate across Multi-GPUs so they are usually performed after halo block execution, but asynchronous of Multi-GPU halo block communication. Non-halo blocks are all blocks not in the yellow rectangle of Figure 7 and are unique to their GPU.

Particle Buffers are the full collection of particle blocks that constitute an object (e.g. a water body, a debris model) on a device. They collect particle bins (SoA) into a global memory array, making an array-of-structures-of-arrays (AoSoA) data-structure. They are material specific, holding basic material information (e.g. stiffness, density) and functions (e.g. for updating properties) common in all composing particles. Multiple objects can be on a device so there can be multiple particle buffers per device. Devices can transfer particle blocks to and from particle buffers of other devices depending on the partitioning scheme (Section 0.11) using marked halo blocks in the buffer.

Grid Buffers are the full collection of active grid-blocks on a device. Devices can transfer grid blocks to and from grid buffers of other devices depending on the partitioning scheme (Section 0.11) using marked halo blocks in the buffer.

Partition is a data-structure for management of the grid and the relationship between particles and the grid. It also facilitates Multi-GPU organization. The partition reserves the largest amount of memory in most Claymore simulations. It keeps track of active grid-blocks to maintain grid sparsity, tags halo-blocks, provides a mapping from a scalar grid-block index to a 3D index (considering sparsity). It records which particles are in a grid-cell, number of particles in grid-cells, which particles are in a grid-block, and number of particles in a grid-block. The partition domain is set to encompass the entire simulation domain so it does not have to resize (as the grid buffer does), but this is at the cost of memory usage for sparse simulations. It pre-allocates enough memory to write 2 + `MAX_PPC` * 2 integers for each grid-cell in the full domain.

Vertices are a novel addition by our lab in Claymore to represent vertices in our coupled implementation of Finite Elements (see Section **??**). Vertices are held in basic device arrays. The structure does not employ sparsity, hierarchy, etc., and does not reorder. Vertices hold their position (3 floats), internal force (3 floats), mass (1 float), volume (1 float), and optionally velocity (3 floats) or surface normal (3 floats). Each vertex is positioned in an array according to its unique integer ID. MPM particles tagged as "meshed" write advected positions to their corresponding vertex by knowing their ID, transcribing directly into the Vertice array index. This works despite MPM particles reorganizing throughout particle-bins and particle-blocks in Claymore. It is also compatible with Multi-GPU simulations. Meshed MPM particles read information from vertices (e.g. internal force) after FEM elements updates.

Elements are a novel addition by our lab to represent element-wise organization of our coupled implementation of Finite Elements (Sec. **??**). Elements are basic device arrays, holding integer IDs of all its vertices. They also hold precomputed edge vectors and volume of the undeformed state (i.e. done at the start of a simulation). For a 4-point linear tetrahedron element, this corresponds to 4 integers for vertex IDs, 9 floats for precomputed edge matrix, 1 float for precomputed volume. Elements read the information of its vertices from the Vertice array and use it along with its own information to compute the updated deformation and stress at element Gauss points. Stress is then mapped as internal

force back to vertices, done as a reduction using atomic addition to vertices in a Vertice array. This prevents data-races, but further optimization could of course be found through warp-reduction within blocks before a global reduction with atomic addition.
Element Buffer is a novel addition by our lab, serving as a superstructure that organizes FEM elements. It is similar to a Particle Buffer as it holds material information that is shared across elements (e.g. stiffness). It allows for declaration of different material and element types with different information requirements (e.g. 4-point linear-elastic tetrahedron vs 8-point non-associative Cam-Clay brick).
This covers the broad data-structures used in our modified ClaymoreUW software. If a user becomes comfortable with them, and minds best practices of CUDA design in Chapter **??** and references [NVIDIA2022] and [NVIDIA2023], then implementing on Multi-GPUs is not much different to a CPU code. However, the results are orders of magnitude faster because of low-level optimization scaled across massively parallel systems.
In the next section, we provide an adapted technical supplement by [Wang et al.2020a] for developers interested in adding to or repurposing the Claymore or ClaymoreUW code.

0.9 ClaymoreUW - Technical Supplement

For a more low-level view of the underlying data-structure system in ClaymoreUW, Claymore, Taichi, and to an extent, a variety of other hierarchically assembled GPU data-structure libraries written in CUDA C/C++, we provide a technical documentation written by the authors of the original Claymore code [Wang et al.2020b], which was published in ACM conference proceedings by [Wang et al.2020b], all right reserved.
In this section, we provide additional exposition so that the technical supplement by [Wang et al.2020b] may be more easily understood. This is because for those not accustomed to CUDA and C++ data-structure design and optimization the technical implications of this supplement will not be apparent. We also include minor alterations to expository text that does not fully apply to ClaymoreUW, and direct readers back to [Wang et al.2020a] for the original format.
However, we emphasize this is not our original work, beyond minor commentary, and we have not significantly altered this portion of the Claymore code in ClaymoreUW. While we use this system to alter the actual data structures that are used and to introduce new ones, the underlying system for construction is their original work and should be cited as such if publishing work based on it or if redistributing said work. Our reason for providing it below is purely for the convenience of readers who would like to better understand how to develop in the open-source code-bases of Claymore and ClaymoreUW.

0.9.1 Compile-Time Settings

To maximize the performance, Claymore uses compile-time constants for both simulation controls and material related settings. ClaymoreUW seeks to eliminate the majority of

interaction with these settings for users without a performance hit through template specialization (though this increases compile times). Below, we provide additional details on compile-time settings for user convenience and reproduction purposes. For example in [Wang et al.2020b], the paper in which Claymore was introduced, they set the maximum number of particle-per-cell to be 64 for all example simulations. This setting is more than sufficient for most graphical MPM use since the typical particle-per-cell (PPC) is 8 when initializing scenes, and material are reasonably compressible but unlikely to exceed 8x the initial PPC in all but the most chaotic scenes. However, the particles will be discarded if the particle number inside one cell exceeds the compile-time setting, leading to incorrect results, though graphical applications are not usually concerned with these errors when lost particles are not visual gaps in geometries and their subsequent errors are not visually dominant after post-processing. For engineering, a more modest MAX_PPC values is often applicable and saves significant amounts of GPU memory for large-scale simulations, e.g. between 2 to 4x the initialized PPC if scenes feature nearly-incompressible materials. For highly compressible scenes, a ratio of 8x will almost always prevent particle loss, but higher values can be selected for safety. In the future, we may pursue a dynamically resizing MAX_PPC value in ClaymoreUW so that only portions of the domain that need large values will reserve large pools of memory, but it remains static as of now. We do note that ClaymoreUW has expanded the setting of MAX_PPC from using integers of base-2 to any 32-bit integer (e.g. 12, 31, 64, 100, etc.), which gives much finer control of how much memory is pre-allocated. This is an important back-end feature for exa-scale simulations, and entailed undoing hard-coded bit-wise operations that relied on the assumption of base-2 integers to avoid more intensive but typical integer operations (e.g. modulo).

Claymore and ClaymoreUW preset the maximum number of particle blocks and grid blocks (g_max_grid_blocks), as well as the maximum particle number (g_max_particle_num). These settings are adopted to enable the pre-allocation of all spatial data structures. Still, the run-time application periodically check the current demand for memory and dynamically resizes to fulfill the need. If it is not possible to resize (i.e. more memory requested of a GPU than available) then the simulation will crash with an output to the C++ standard error stream.

Different scenes require different settings, and the program works as long as the whole memory allocated does not exceed the device memory limit. In general, we recommend not using more than 50% of a GPU total memory, as the application will run faster if their is ample room for resizing data-structures when needed and it will be far less error prone.

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. Claymore uses a Courant-number of 0.6 to compute the CFL-bounded default stepping time in all their benchmarks. However, we recommend use of ClaymoreUW with a CFL number between 0.5 and 0.3, with our own simulation typically at 0.45. This is because the added features (e.g. PIC-FLIP mixing, F-Bar antilocking) and tendencies

for engineers to use much stiffer materials undergoing more complex constitutive laws necessitate more stringent time-stepping to avoid introduction of even small errors and poor resolution of pressure waves in the medium. We provide run-time control of CFL numbers specific to any particle object's material model in ClaymoreUW's input script. [Wang et al.2020b] did consider material stiffness when computing the default stepping time for stability requirements, however, ClaymoreUW introduces an automatic calculation of this time appropriate to selected material parameters. This is for user convenience, as determining the required time-step manually in every simulation is tedious and prone to error. We refer readers to [Bai and Schroeder2022] for a deeper understanding of potential time-step conflicts in explicit MPM and MLS-MPM (e.g. near-boundary and lone particle instabilities). During run-time, Claymore computes the maximum of the grid velocity and calculates 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, which is material stiffness considerate. 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. This occurs because the G2P2G kernel must write particles back to the grid after it advects them, but it only has access to a limited view of the grid (namely, the 8 x 8 x 8 grid-cell arena around the centered 4 x 4 x 4 grid-cell particle-block) so any particle whose advection moves it further than one grid-cell will cause significant errors. For instance, the particle will not contribute to the grid (a fundamental MPM step) and its registration within the particle cell buckets data-structure will fail (holds the relative particle ID used to map particles between particle bins) so the particle simply disappears in the next step.

0.9.2 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.

0.9.3 Data-Oriented Design Philosophy.

Due to the increased overhead of memory operations, data-oriented design philosophy has been widely adopted in HPC. Following this design principle, [Hu et al.2019b] 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 tool-chain 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: **(i)** no access to low-level operations, including CUDA warp intrinsics, and **(ii)** 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++ code-base with the following improvements:

0.9.4 C++ Oriented Programming

Unlike developing a new compiler as in the Taichi programming language [Hu et al.2019b], 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 is the minimum requirement for compilation. However, it is recommended, and likely the default if configuring a system in the past four years, to use C++17 compatible compiler version. Function definitions are decorated with constexpr keyword whenever applicable on both the host and the device-side.

0.9.5 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.

0.9.6 Memory Management

The representations of memory handles vary across APIs for GPU computing. For CUDA C++ [NVIDIA2023], 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++ code-base, 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:

```
1 domain<Tn, Ns...>; // Tn: Index Type, Ns: Multi− dimensional coordinates of type Tn
2 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...>;
```

0.9.7 C++ Implementation.

For fine details, i.e. the full code, please refer to the open-sourced Claymore code-base by [Wang et al.2020b]. The same back-end data-structure code is preserved in ClaymoreUW, with added comments to assist in navigating the various library files employed. The fundamental data structure composition infrastructure consists of four major components: Domain, Decorator, Structural Node, and Structural Instance, which are described as follows:

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);
};
```

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_;
};
```

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.2019b]. Hash is appropriate for hash-table type data-structures. Dense is for densely packed data of a given size (e.g. a grid-block structure). Dynamic is for dynamically resizing data-structures (e.g. the grid-buffer super-structure which contains dynamically changing counts of the dense grid-block structure as a simulation progresses).

```
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;
```

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 {
        ...
    }
};
```

0.9.8 Example Usage

To showcase the usages of Structural in C++ this section provides a set of examples that describes a GPU SPGrid. Firstly, the definitions below are useful short-hands:

```
/// leaf node
using empty_ = structural_entity<void>;
using i32_ = structural_entity<int32_t>; // single-precision signed integer
using f32_ = structural_entity<float>; // single-precision float
using i64_ = structural_entity<int64_t>; // double-precision signed integer
using f64_ = structural_entity<double>; // double-precision 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)>;
```

The following code defines the SPGrid data structure within Claymore and ClaymoreUW:

```
Definition of GPU SPGrid.
// domain
using BlockDomain = domain<char, 4, 4, 4>; // block-domain in 3D, 4x4x4
using GridBufferDomain = domain<int, g_max_active_block>; // grid-buffer domain,
↪  element per max grid-blocks
// 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_>; // single-precision 4x4x4 grid-block
using grid_buffer_ = structural<structural_type::dynamic, DefaultDecorator,
↪  GridBufferDomain, grid_block_>; // single-precision grid-buffer
```

After defining the internal structure, it still requires an allocator and the list of data components to get the instance and actually have it occupy GPU memory in a simulation:

```
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>;
```

To access hierarchical elements of the GPU SPGrid in a function, it is neccesary to index as follows:

```
/// acquire blockno-th grid block
auto grid_block = grid.ch(_0, blockno);
/// access cidib\-th cell within this block with 1D coordinate
grid_block.val_1d(_0, cidib); // access 0-th channel (mass)
/// access cell within by 3D coordinates
grid_block.val(_1, cx, cy, cz); // access 1-th channel (velocity x)
```

Memory layout can be visually interpreted for data-structures for simpler comprehension. Two types of Structural Nodes with different Decorators (Array-of-Structures and Structure-of-Arrays, i.e. AoS and SoA, respectively) are illustrated in Figure 8 to explain the underlying memory layout. Both are very important organization schemes to understand in high-performance computing. Further, they are often seen in Array-of-Structures-of-Arrays (AoSoA) format in recent years, i.e. a combination of AoS and SoA to promote hierarchical access of groups of particles, nodes, etc. which may then be loaded into GPU memory in a coalesced, efficient manner.

0.9.9 Concluding Remarks

This completes our brief summary of both a high-level view on the data-structures used in ClaymoreUW, as well as the low-level configuration in CUDA C++ that originates from Claymore, by [Wang et al.2020b], which is based on the Taichi programming language by [Hu et al.2019b].

Note that there is still significant room to optimize these structures. Namely, for the hash-map which is not currently sparse (it creates an entry per each grid-block within the maximum 3D domain that can be very large for very large max domains), and for the cell and

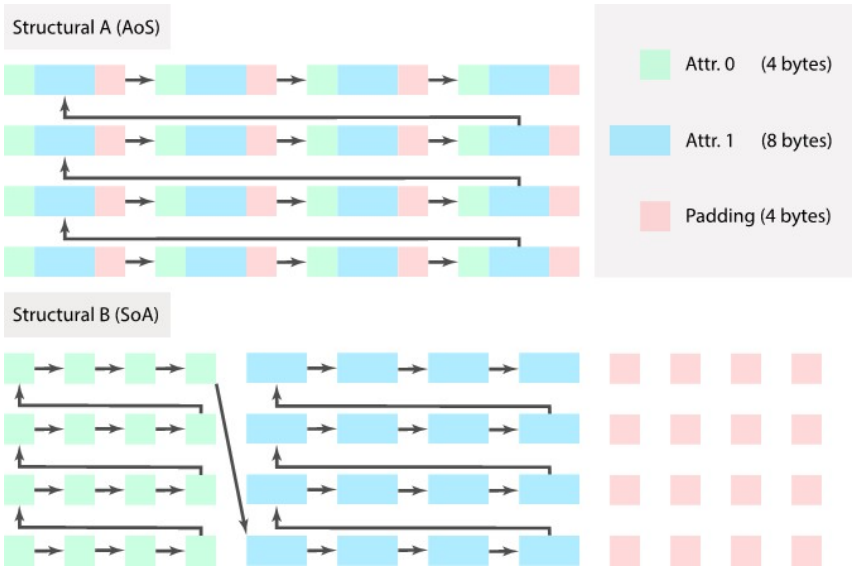

Fig. 8. Spatial structure specification in Claymore's hierarchical composition scheme for C++. Two data-structures are specified with different decorators. The arrows connecting all elements indicate the ascending order in a contiguous chunk of memory. The structures can be used as a child of another 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 object is padded to the next power-of-2 due to the alignment decoration. Figure adapted from the supplementary technical document provided by [Wang et al.2020b]. Their work was based on that of Taichi [Hu et al.2019b]. This system is directly imported into ClaymoreUW. All rights of original authors and publishers are reserved.

block particle buckets (which hold relative IDs of particles for a cell which aggregate into blocks, and are not currently sparse or dynamically resized so all cells in all active grid-blocks will reserve MAX_PPC entries for themselves. This takes a lot of memory if MAX_PPC is large and many grid-blocks are active).

0.10 ClaymoreUW - Frequently Asked Questions

A few common tasks a user of Claymore-for-Engineers (i.e. ClaymoreUW) may need to know are listed here. This is not an exhaustive user-manual but serves as a starting point for an otherwise high learning-curve software.

Adding a material model. A couple of basic material models (weakly-compressible fluid, fixed-corotated solid, Drucker-Prager, modified Non-Associative Cam-Clay) are already implemented. Implementing a new material is a basic capability of an MPM software. Here we provide a walk-through for this task. Most difficulty is from navigating Claymore's file-structure and making new GPU functions to support a material, but this essentially amounts to copy-pasting specific code that is highlighted here. To add a material, go to `particle_buffer.cuh` and check if any currently defined particle representations apply to your particle (i.e. if your material needs one more float on particles than any previous model you will need to copy-paste existing code, e.g. `particle_bin8` as `particle_bin9`, and add an extra float value (f32). The means to do this is fairly apparent in the file. Then, define the material itself at the bottom of the file where the particle buffer is defined as a variant of possible materials, e.g. "Test-Material", and have it reference the appropriate particle data structure you just created. Copy code for any similar material Particle Buffer and alter it to reference "Test-Material", as well as changing it to include any material parameters (e.g. shear modulus). Go to `settings.h` and add "Test-Material" to the material list. Definition of the material in terms of memory allocation and structure is complete. Now go to `constitutive_models.cuh` and add the constitutive model that defines the material, i.e. given a deformation gradient and needed material parameters what is the resultant stress. This is no different than a material model in any other software, but it is written as an inline device function so be mindful that it operates on GPU data. Finally go to `mgmpm_kernels.cuh` and copy one of every material specific function, changing its header to use "Test-Material". The contents of the functions may or may not need to change, it will be material specific, but at the minimum it should account for any change in particle data usage as set by the particle bin structure defined at the start. For example, `g2p2g()` should call the appropriate constitutive model function for your material and it should read and write to the appropriate memory addresses in the particle bin structure you defined. Despite a fairly large number of steps, you mostly just copy-paste with minimal modifications, leveraging highly optimized Multi-GPU code with low-overhead.

Input data. Currently the expanded Claymore code supports *.sdf, *.csv, and *.pos binary files to create MPM particles. The former will actually sample particles from a signed-

distance-field according to a given PPC, while the latters just read in particle positions directly. Vertices and Elements for our FEM coupling use *.csv files only. Claymore does not natively support reading in stress, velocity, etc. on a particle basis, but we have built in functionality so we may read in *.bgeo files (a standard output format for particles and their attributes in ClaymoreUW) directly to imbue particles with initial attributes, i.e. a "checkpoint" simulation state. This can be useful for reusing previous frame outputs from other simulations (i.e. picking up a simulation where it left off).

Making a Signed-Distance-Field input file. Open-source command-line tool SDFGen translates common triangle mesh based *.obj files into signed-distance-field *.sdf files. As a signed-distance-field, each cell in a grid covering the object knows its distance from the manifold. This format is the preferred way to generate MPM particles and apply non-conforming boundaries because it is scale invariant if the SDF resolution is fine enough for the object, it provides distance from manifold (i.e. body) which is valuable for determining the surface/near-surface/body of an object, and it can provide surface normals easily which is often challenging for MPM particles and non-conforming boundaries.

Output particle data. If a user wants to pull, for example, particle Cauchy stress out of an object in a simulation, how would they do this? First, ensure that each particle in the object (i.e. Particle Buffer) is equipped with memory space to store a stress tensor (additional 3x3 floats). This is not set by default in `particle_buffer.cuh`. Second, tell particles to write their stress tensors to their memory within the material specific G2P2G kernel in `mgmpm_kernels.cuh`, this is just a copy-paste as particles already write values into their memory. Third, augment the current particle attribute retrieval functions `retrieve_particle_attributes()` to copy this new data from each particle in the Particle Buffer, going through it block by block, to a basic device array. Fourth, copy the device array information to a host array asynchronously, already done in `mgsp_benchmark.cuh`. Fifth, `mgsp_benchmark.cuh` will output the host array containing the particle stresses to a desired output format (e.g. *.csv, *.bgeo) by calling output functions. PartIO, an open-source particle I/O toolbox, is used for this. Output functions are found in Claymore/Library/MnSystem/IO/ParticleIO.hpp and are easily modified as they are standard C++.

Output grid data. Our lab implemented two methods to output grid-data. The first method outputs grid-blocks only, with summed mass and momentum from composing cells. The second method outputs grid-cell information within defined domains (e.g. surfaces, volumes) in the simulation, usually providing the mass, momentum, and force on grid-nodes. The former is useful for observing overall grid behavior efficiently while the latter is good for high-resolution, targeted information. We also allow output to *.csv files for aggregated quantities, like summed force on a surface or max surface elevation in a set domain. Respectively, these replicate load-cells and wave-gauges in experiments and can be set by a user-specificed output frequency rate. Our implementation works on Multi-GPUs and the sparse data-structure of the grid in Claymore. It also leverages shared memory and warp level atomics for computing aggregate quantities. Asynchronous writes are used

to go from device to host so the simulations are not interrupted when outputting large amounts of data.

Visualize data. Claymore output is provided in binary geometry (*.bgeo) format by default, though the PartIO library can convert to other common file types. To visualize *.bgeo files you can quickly use PartView (command in PartIO) or SideFX Houdini. The latter has a free license for non-commercial use and provides a massive toolbox for animation and visualization capabilities. Simply import the *.bgeo as geometry data and it will show position data in a viewing window for every available frame. Particle attributes (e.g. pressure) can be shown by opening a Scene Option in the Scene Viewer toolbar, clicking the Visualize tab, adding a new visualizer, setting it to reference the name of the attributes, and setting any other specifications for visualization (e.g. visualize as color, vector, color-mapping, etc.). Advanced rendering (e.g. subsurface scattering, ray-tracing, etc.) can be done here as well.

0.11 ClaymoreUW - Multi-GPU Domain Decomposition

With scaling laws (Section **??**), kernel design (Section 0.7), data-structure difficulties (Section 0.8), and Multi-GPU memory limits (Sections **??** and **??**) in mind, one must consider a variety of partitioning schemes to optimize MPM simulations on multiple GPUs. Schemes considered by the author are:

- **Partition-by-Space.** Favorable to scenarios of predictable, even spatial occupation. The grid-block halo region is static and designated. Fluids, especially ones that mix and occupy a fixed domain volume, are usually done in this way.

- **Partition-by-Particles.** Ideal for particles that are expected to maintain their rough configuration relative to one-another throughout the simulation. Stiff solids behave well in this scheme.

- **Partition-by-Material.** Useful when there are multiple materials in a scene that require very different time-steps or undergo unique algorithms to advance through time. Incompressible fluids and compressible solids in the same scene may be best performed in this way.

- **Partition-by-Method.** Allows for different partitions to handle separate numerical methods. For instance, an MPM-FEM coupling may split FEM meshed bodies from purely MPM bodies to simplify design. In our code, the FEM implementation is not as high-performance as the MPM implementation, but there are typically far more particles in a scene than elements so having one GPU handle FEM and the rest running MPM results in a time-balanced scene.

- **Mixed Partitions.** Fitting for complex scenes. For instance, we partition FEM meshed debris on one GPU, while partitioning the MPM water across multiple GPUs spatially. This leverages both Partition-by-Method and Partition-by-Space.
- **Adaptive Partitions.** Adapt throughout the simulations course. For instance, partitions may rebalance themselves according to the deformation occurring in the scene. This can account for nonlinear materials models which may slow-down as deformation grows, thus GPUs with too many particles undergoing nonlinear deformation may offload some to other, less taxed GPUs to balance computations.

0.12 Final Remarks

In this chapter we cover the basics of bringing the Material Point Method (MPM) onto multiple graphics processing units (Multi-GPUs). An open-source software by [Wang et al.2020b] is evaluated for this task, but was not built for practicing engineers as the intended user-base. After notable modification by our group, we arrive at an augmented version of the open-source code-base, ClaymoreUW, that is retooled for engineers.
A developer and user's guide is provided for our open-source project, with the file-structure, data-structures, underlying algorithm, and input script construction all described. Modifications made to the original Claymore [Wang et al.2020b] by our lab, including user-interface changes, new antilocking methods, introduction of multiple advection schemes, improved computational precision, and coupling finite elements, among dozens of other changes, are described.

Bibliography

[Bai and Schroeder2022] Bai, S. and Schroeder, C. (2022). "Stability analysis of explicit mpm." *Computer Graphics Forum*, 41, 19–30.

[Hu et al.2018] Hu, Y., Csail, M., Fang, Y. U., Ge, Z., Zhu, Y., Fang, Y., Qu, Z., Prad-Hana, A., and Jiang, C. (2018). "A moving least squares material point method with displacement discontinuity and two-way rigid body coupling method with displacement discontinuity and two-way rigid body coupling." *ACM Trans. Graph*, 37, 146.

[Hu et al.2019a] Hu, Y., Li, T.-M., Anderson, L., Ragan-Kelley, J., and Durand, F. (2019a). "Taichi: a language for high-performance computation on spatially sparse data structures." *ACM Transactions on Graphics (TOG)*, 38(6), 201.

[Hu et al.2019b] Hu, Y., Li, T.-M., Anderson, L., Ragan-Kelley, J., and Durand, F. (2019b). "Taichi: a language for high-performance computation on spatially sparse data structures." *ACM Transactions on Graphics (TOG)*, 38, 201.

[NVIDIA2022] NVIDIA (2022). "CUDA Math API Reference Manual (1).

[NVIDIA2023] NVIDIA (2023). "CUDA C++ Programming Guide Release 12.1 NVIDIA.

[Wang et al.2020a] Wang, X., Qiu, Y., Slattery, S. R., Fang, Y., Li, M., Zhu, S., Zhu, Y., Tang, M., Manocha, D., and Jiang, C. (2020a). "A massively parallel and scalable multi-GPU material point method supplementary technical document." *ACM Trans. Graph*, 39, 4, Article 1, 4.

[Wang et al.2020b] Wang, X., Qiu, Y., Slattery, S. R., Fang, Y., Li, M., Zhu, S.-C., Zhu, Y., Tang, M., Manocha, D., and Jiang, C. (2020b). "A massively parallel and scalable multi-GPU material point method." *ACM Trans. Graph.*, 39.