# GPU Acceleration of the Generalized Interpolation Material Point Method

Wei-Fan Chiang, Michael DeLisi, Todd Hummel,
Tyler Prete, Kevin Tew, Mary Hall
School of Computing, University of Utah
Salt Lake City, UT 84112

Phil Wallstedt
Dept. of Mechanical Engineering
University of Utah
Salt Lake City, UT 84112

James Guilkey
Schlumberger
Perforating Research
Salt Lake City, UT 84104

***Abstract*— This paper describes our experience rewriting a sequential particle-in-cell code so that its key computations are executed on a GPU. This code is well-suited to GPU acceleration, as it performs data-parallel operations on a regular grid. Key performance challenges are the need for global synchronization in mapping particles to grid nodes, and managing memory bandwidth to global memory. Performance results show overall speedups of 3.3x including the time to display the results of simulation, or 10.9x without the display I/O overhead.**

***Keywords-component; Particle-in-cell code, GPU acceleration***

## I. INTRODUCTION

There are many applications for rigid, soft body, and fluid simulations, including increased realism for fluids and smoke in games, accurate astrophysics simulations, and molecular dynamics. Such simulations are essential computations in scalable scientific applications for which the highest levels of performance are sought. This paper describes a parallel implementation of a specific particle-based method for materials simulation, targeting highly multithreaded graphics processors (GPUs), and written in the CUDA programming language. The Generalized Interpolation Material Point Method (GIMP) [1,2] is an extension of the Material Point Method (MPM) [3], which simulates the state of an object as it goes through physical transformations, e.g., the collision with a second object or an explosion.

In both MPM and GIMP, each particle keeps track of a part of the mass, velocity, acceleration and volume of the whole object. The particles then interact with a computation grid, which serves as a structure for collecting information about the stresses in different parts of the object. Figure 1 depicts how the set of particles are mapped to the discretized grid in a two-dimensional representation. The inset shows that a particle may contribute to multiple edges in the grid, up to 4 in a 2-dimensional representation for MPM.

After each time step, the grid performs interpolation to adjust the properties of the particles based on the stresses in the object. Particle stress, positions, and velocities are then updated from their associated grid nodes to serve as the initial data for the next time step calculation. The entire computation involves a series of such time steps, which captures the movement of the material over time resulting from properties of the material and the forces acting upon it.

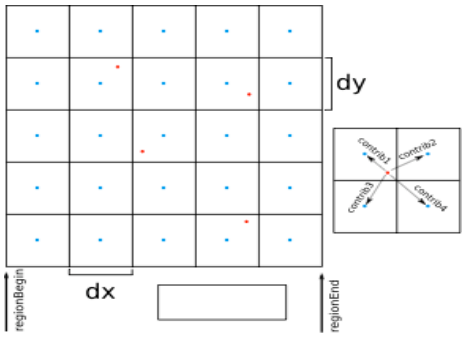


Figure 1. Particles mapped to regular grid.

Considering how to parallelize MPM/GIMP, we observe the computation exhibits a high degree of data parallelism and locality of data accesses within the grid, which should offer opportunities for significant performance gains on GPUs. Prior MPI and OpenMP implementations of MPM/GIMP offer some strategies for parallelization [4], but the highly threaded, global address space of GPUs offer new opportunities and challenges. MPM and GIMP can be thought of as a significant extension of the "particles" project from the nVIDIA CUDA Software Developers Kit (SDK) [5], which does not compute stresses and forces within an object. In this paper, we describe two GPU implementations of a code that computes both MPM and GIMP, derived from porting an existing C++ sequential implementation [6].

## II. GPU IMPLEMENTATION

GPUs are capable of executing computations with thousands of independent threads, operating in Single-Instruction-Multiple-Data (SIMD) mode. Thus, computations that perform roughly the same computation across a large independent data set are ideal candidates for GPU acceleration. MPM/GIMP exhibits significant data parallelism arising from performing the same calculations (interpolation, integration, divergence and derivatives) for all nodes in a grid or for all particles. In addition, data dependences are local and usually only extend to surrounding neighbors. Thus, most grid node and particle calculations can be isolated into independent GPU threads and executed in parallel.

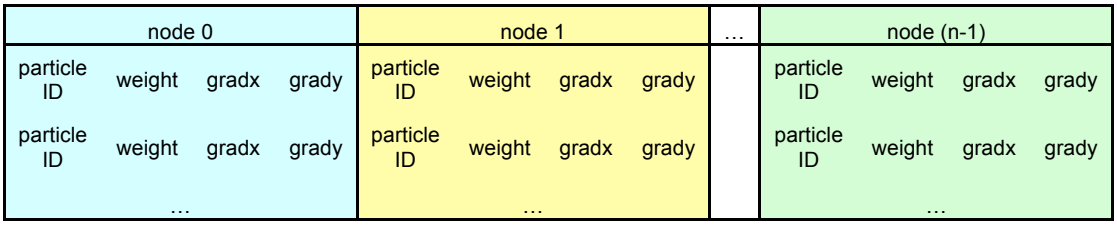


Figure 2. Structure of the reverse map.

Nevertheless, a particle may contribute to multiple grid nodes, and each grid node contains multiple particles, so that any data decomposition across GPU processors either by nodes or particles will have some overlap with data accessed by other processors. The sequential code alternates between computations across grid nodes or across particles. For this reason, both a mapping from particles to nodes (the primary mapping) and a reverse mapping from nodes to particles are used to organize the computation. For the GPU implementation, we must derive the reverse mapping in parallel by iterating over the set of particles and the grid nodes to which each contributes. This reverse map is a key performance concern in the GPU version of MPM/GIMP.

In one implementation, building the reverse map uses atomicAdd to insert a particle's information into a global array, similar to the "particle" project in the CUDA SDK [5]. In this code, a 2-dimensional array in global memory represents the set of grid nodes in the columns and the maximum number of possible particles contributing to it in the rows, as shown in Figure 2 (up to 9 particles in GIMP). Each grid node has an integer associated with it which represents the current number of particles that have been inserted into its column. An atomicAdd is then used to increment this integer and identify the position in the grid node's list to which this particle should be inserted. This implementation in [5] can create a bottleneck when multiple threads are simultaneously invoking atomicAdd on a single location. As an optimization, we maintain a set of nine independent reverse maps to reduce the number of atomicAdd calls to a single location. Each reverse map includes only a subset of the particles that map to a node, so when the reverse map needs to be accessed, some portion of all 9 maps must be traversed. As compared to the CUDA SDK "particle" project, we see roughly a 2x speedup as a result of reducing global synchronization.

A second performance concern is the bandwidth of global memory accesses. When successive threads access successive locations in global memory, the memory accesses are *coalesced*, so that a single memory operation delivers several data elements, thus better utilizing memory bandwidth and reducing the high latencies to global memory (on the order of 100 cycles). We can organize the reverse mapping as in Figure 2, so that the information for each node is stored successively and can be coalesced when being read from global memory. Additional data structure modifications, such as using float2 and float4 structures that are aligned for coalesced access on the GPU. The need for such changes was determined from the measurements of coalesced and noncoalesced global memory accesses identified by the CUDA Visual Profiler.

The second GPU implementation conserves memory and avoids the potential bottleneck of atomicAdd, but requires one sort per time step (of 2*number of particle contributions elements) and uncoalesced memory accesses during the grid accumulation operation. Future work will compare these two strategies as the problem size scales. For the relatively small problem sizes we tested, the former implementation that uses replicated reverse maps was practical and performed better, but because it trades off performance for extra memory, may exceed storage availability for larger scales.

Additional optimizations to improve performance from both implementations include:

- Fusing multiple functions in the original application into a single kernel invocation to eliminate synchronization at the host.
- Maintaining data on the GPU, accessed from separate kernel functions, eliminating the need to copy intermediate results to/from host or to pass parameters. Constant memory is used for read-only input data.
- Reorganizing vector of vector objects into contiguous statically allocated arrays that can be more easily copied onto the GPU.
- Replacing divides by multiples of the inverse, cached so they need not be recomputed.

## III. Performance Results

We present results for one of the two implementations, as the approaches are similar. For measuring the results, a machine containing an Intel Core2 Duo E8400 (3.00 GHz) and an nVIDIA GeForce 9600 GT (8 SMs) was used. GPU software was developed using CUDA version 2.0. We use the sequential CPU code as a baseline, and look at both individual functions and the full simulation for three different problem sizes described in TABLE I.

| Cells | Particles | Grid Nodes |
|---|---|---|
| 32 | 1,352 | 2,553 |
| 64 | 5,356 | 9,177 |
| 96 | 12,012 | 19,897 |

TABLE I. Problem sizes for experiment.

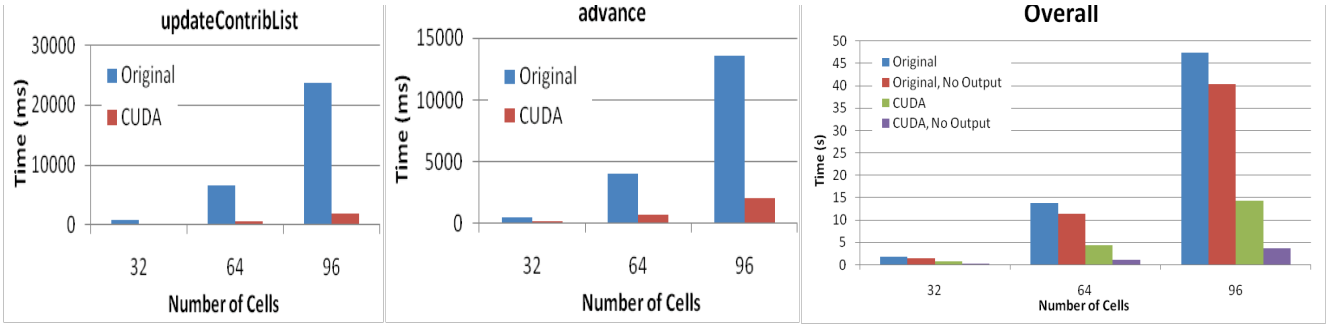


Figure 3. Performance comparison for key computations and overall, between original sequential code and CUDA version.

Figure 3 presents the speedup comparison between the original sequential code and the CUDA versions for the numbers of cells from TABLE I. The first two graphs represent the two key computations in the sequential code. The computation *updateContribList* determines the set of grid nodes to which a particle contributes given its current placement, and the computation *advance* computes the various calculations of a grid node from its particles. Using 128 threads per block, *updateContribList* and *advance* were 12.5x and 6.6x faster, respectively. The overall speedup was measured with output on and off. File I/O is currently used to visualize the simulation; this file I/O cannot be parallelized and significantly constrains our potential gains to a 3.3x speedup. With more work, the problem could be rendered in real time on the GPU, yielding a speedup similar to the 10.9x speedup achieved with no output.

## IV. Conclusion and Future Work

This paper has described a GPU re-implementation of MPM/GIMP, a particle-in-cell code used for rigid, soft body and fluid simulations. We demonstrate an overall speedup of 10.9x as compared to a sequential CPU implementation, when i/o to display the simulation results are disabled. With this work, we show that it is possible to achieve large gains in performance while using a relatively inexpensive GPU device.

The two key performance concerns in the GPU implementation were the potentially high cost of synchronization in building the reverse map from grid nodes to particles, and managing memory bandwidth in accessing global data structures. As we port this application to future generations of GPU hardware and later versions of CUDA, these performance challenges may become less of a concern as support for global synchronization and better global memory access coalescing is already available in newer hardware/software environments.

In the future, we see three areas for improvement of our implementation. Because the GPU we used did not efficiently support double precision floating point, we reverted to single precision for these results. While single precision was sufficiently accurate for this problem, in some cases double precision is essential to correct simulation results. Fortunately, higher end GPUs offer better support for double precision arithmetic. Second, we can eliminate the I/O overhead for displaying results by using Open GL buffers and displaying directly from the results calculated on the GPU. As a final point, we used a constant 128 threads per block for these experiments, and in some cases, this was too small a number of threads to hide the latency to memory. The 8192 registers in a streaming multiprocessor are shared among all threads in a block. We were unable to increase the number of threads due to limits on available registers; higher-end GPUs have more registers, but beyond this, we would need to utilize them more efficiently. Therefore, all three future improvements will benefit from new features of GPU hardware and software, already available in higher-end platforms.

## Acknowledgment

The authors wish to thank Lin Ha for early suggestions on how to rewrite MPM/GIMP for a GPU. This work has been sponsored in part by NSF awards CSR-0615412 and OCI-0749360 and by hardware donations from NVIDIA Corporation.

## References

[1] *The generalized interpolation material point.* **Bardenhagen, S. and Kober, E.** 2004, CMES: Computer Modeling in Engineering and Sciences, Vol. 5, pp. 477-495.

[2] *Examination and analysis of implementation choices within the material point method (MPM).* **Steffen, M., et al.** 2008, CMES: Computer Modeling in Engineering and Sciences, Vol. 31, pp. 107-127.

[3] *A particle method for history-dependent materials.* **Sulsky, Deborah, Chen, Zhen and Schreyer, Howard L.** 1994, Computer Methods in Applied Mechanics Engineering, Vol. 118, pp. 179-196.

[4] *Shared Memory OpenMP Parallelization of Explicit MPM and Its Application to Hypervelocity Impact.* **Huang, P., et al.** 2008, CMES: Computer Modeling in Engineering and Sciences, Vol. 38.

[5] *CUDA Particles*. **Green, Simon**. [nVIDIA Cuda SDK] s.l. : nVIDIA, 2008.

[6] *An evaluation of explicit time integration schemes for use with the generalized interpolation material point method.* **Wallstedt, Phil and Guilkey, Jim.** 22, s.l. : Academic Press Professional, Inc., November 2008, Journal of Computational Physics, Vol. 227, pp. 9628-9642.