Eleventh International Conference on CFD in the Minerals and Process Industries
CSIRO, Melbourne, Australia
7-9 December 2015

CUDA ACCELERATED LAGRANGIAN INTERPOLATION TO CARTESIAN GRID

Andrew D. LARSON[1*], Travis CARVER[1] and Paul ZHAO[1]

[1] CPFD Software LLC, Albuquerque, New Mexico, USA
*Corresponding author, E-mail address: andrew.larson@cpfd-software.com

ABSTRACT

Accelerated using the NVIDIA Compute Unified Device Architecture (CUDA), a framework for parallelized computation on a graphics processing unit (GPU), Barracuda Virtual Reactor® Software simulates large scale, industrial sized units using the Multi-Phase Particle-in-Cell (MP-PIC) methodology. The critical component to improved efficiency is a scalable and efficient algorithm for mapping discrete Lagrangian parcels to the Cartesian Eulerian grid. In each simulation time step, 10 to 20 separate interpolation mappings occur, motivating the importance of this algorithm. With parallel implementation of this algorithm, write contention increases as parcel count per cell increases. Naive approaches cannot realize the full potential of GPU acceleration, breaking down at higher particle loads and increased write contention. We present a CUDA specific implementation that scales and consistently achieves a 14x speed-up for mapping parcel properties to a Cartesian grid over the existing and basic CPU implementation.

NOMENCLATURE

CUDA
: Compute Unified Device Architecture is a parallel programming framework developed by NVIDIA specifically for GPGPU computation.

kernel
: CUDA routine executed on the GPU. Uses a CUDA grid dimension and CUDA block dimension to populate the multiprocessors.

warp
: a group of 32 threads executed simultaneously, all threads execute the same instruction.
The first level of thread aggregation in CUDA.

thread
: a single line of execution in CUDA model

block
: a group of 32-512 threads, has 3 dimensions used for algorithmic indexing and work assignment.
The second level of thread aggregation in CUDA.

global memory
: Main memory on the GPU, typically gigabytes and accessible by all threads in all blocks.

shared memory
: shared within a block, resides on multiprocessor, limited to about 48,000 bytes.

register memory
: accessible only by owning thread, resides on multiprocessor, very limited usage typically ranges from 0-60 at 32-bits per register.

For more about CUDA framework see (Nickolls et.al. 2008) and (NVIDIA 2015).

INTRODUCTION

Barracuda VR® utilizes the Multi-Phase Particle-in-Cell (MP-PIC) methodology to simulate transient gas-solids flow commonly found in many industrial processes. Able to model polydisperse particle flows, this computer aided engineering software focuses application in simulating industrial scale units (e.g. regenerators, risers, gasifiers, cyclones, strippers and pneumatic transport lines). Discrete elements, or computational particles, in the Lagrangian phase are used to represent a large number of physical particles (e.g. 10^{16}). More details about governing equations can be found in (Snider 2001).

Barracuda VR software has the unique capability to model the full range of solids PSD and calculate flows with any particle volume fraction from dense phase (packed) to the dilute phase within the same domain.

Generally, fine grid resolution is not required because the Lagrangian phase provides sub-grid resolution. However, due to common end-user model scale, engineering time constraints, particle counts in the millions and complicated chemistry, simulations regularly run for weeks to 1 or 2 months to reach a suitable 'quasi'-steady state solution. That's clear motivation for improved performance via parallelization.

What is parallelized in Barracuda VR?

For results presented in the paper, Barracuda VR series 16 was used. This was the first version to offer GPU parallelization with an accelerated pressure solver and acceleration of the major set of particle operations covering advection, wall interaction, interpolation calculations, drag, etc. Between underlying data structure changes and performance gains from GPU acceleration, significant reductions in wall time were achieved, from 15% to 50% of original run time, or 2 to 6 times faster.

This paper focuses on a very specific part of the extension implementation of the MP-PIC method for numerical approximation of these large scale units, Particle-To-Grid interpolation (on a Cartesian grid).

MOTIVATION FOR INTERPOLATION

Why is parallelizing the interpolation operator difficult?

The difficult part to parallelizing the interpolation operator is the many-to-one relationship between particles and grid cells. This is further compounded by the need to do more than a local cell interpolation of values. In Barracuda, a 27 point stencil is avoided by using a support grid for interpolated values and an 8 point stencil. In the CUDA framework, one approach is to assign a thread to each particle, calculating interpolation

values and updating surrounding cells. However, this approach suffers from large amounts of write contention due to particle to cell ratios, drastically reducing concurrency, and thus, performance. A second approach is to assign a thread to each cell, where local cell values are maintained while working through a subset of particles that affect that cell. The second approach allows for more concurrency but still has many challenges for best performance. Implementation details of the second approach will be discussed and how it applies to Lagrangian interpolation to a Cartesian grid.

Interpolation is the Crux

In the Barracuda VR code base, other operations provide challenges, require significant effort or give better speed-up than the interpolation to a Cartesian grid. However, due to the highly coupled nature of the Lagrangian and Eulerian phases, Particle-To-Grid interpolation occurs 10 to 20 times within a single simulation time step. Compared to other routines found in a MP-PIC code, interpolation is the most challenging to parallelize and the most critical due to widespread use across a single simulation time step. In serial, interpolation does not account for much run time, but in parallel other routines with better performance require less run time highlighting interpolation as a critical routine in need of the best possible solution. Figure 1 helps motivate this.

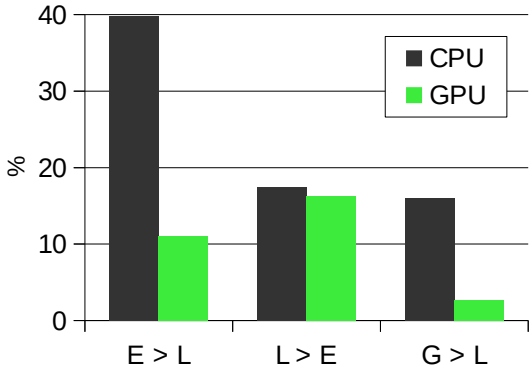

Figure 1: Percentage of total run time for 3 routine classes: Eulerian to Lagrangian mapping (E > L), Interpolation (L > E) and Geometry to Lagrangian a.k.a. ray tracing (G > L), using a typical customer sized problem. Note: CPU total is 73%, and GPU total is 30%.

Comparing to Related Work

Prior work investigating the use of CUDA to accelerate interpolation operators exists, reporting that Particle-To-Grid interpolation could achieve 17.4x speed-up for a single value using single-precision (Stantchev, Dorland and Gumerov, 2008). The current algorithm used in Barracuda VR provides a 14x speed-up for interpolation of 3 values in double precision. Certainly, when mapping many Lagrangian particles to a single Eulerian grid cell, write contention will exist. Because of this, naive approaches using atomic operations will not perform well at high load, which is vital for Barracuda VR, where, on average, 10 particles per 1 grid cell is a starting point for resolution. Particles collected together at close pack generate larger ratios, with 100:1 or larger possible.

IMPLEMENTATION

Barracuda VR MP-PIC Framework

Barracuda VR uses a Cartesian grid for its simplicity and robustness. The MP-PIC method makes this option possible via sub-grid resolution from the Lagrangian phase and the close coupling between the Lagrangian and Eulerian phases. The principle interpolation routine handles mapping particle volume to the Eulerian grid. A support grid and 8 point stencil is used. A particular cell in the grid might normally “see” 26 neighbors. However, the support grid, which is staggered, reduces the total volume of influence, effectively excluding particles more than a cell length away from the center of the cell in consideration. Figure 2 should help understanding.

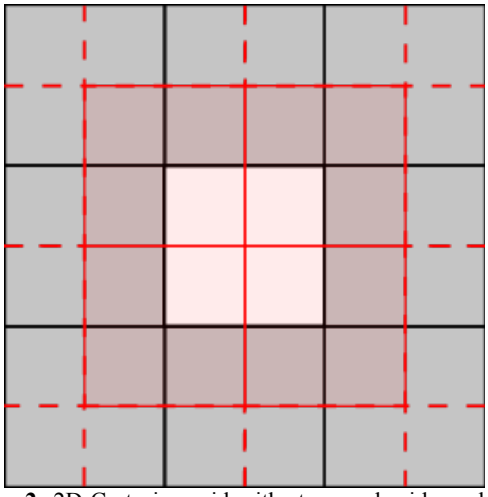

Figure 2: 2D Cartesian grid with staggered grid overlaid in red. The white cell in center receives interpolated particle values located in 4 supporting cells, colored red.

Detailed Description

Development of an effective CUDA implementation for Particle-To-Grid interpolation requires considerable effort compared to most other routines due to the write contention. Atomic operations give abysmal performance for ratios much more than 2:1. A factor here is the lack of hardware support for double-precision atomic operations in CUDA 5.0. Although, even with hardware support, atomic operations will not scale. The next step is a slightly less naïve approach where each cell is allocated a single CUDA thread. With particles sorted and a list of particle indexes per cell, each thread works on the particles in that cell, calculating 8 interpolation values for its support grid and storing them for later reduction. For small problems and smaller ratios, this may give a 5x speed-up over serial but degrades when subjected to real world problems, the issue being lack of focus of computational resources. Some cells have 0 particles, some 10 and some 100. Pair that with warp divergence due to mismatch in particle counts per cell and performance does not scale.

Extra Requirement – Sorting

Serial interpolation loops over particles and applies interpolated values directly. Our parallel implementations require that particles be sorted into cell bins. This is done using a standard sort routine from Thrust, a C++ STL style parallel algorithms library (Hoberock and Bell 2011). Particle sorting is required anytime particles move. Required once per time step, the extra cost of this operation can be amortized across all interpolation operations in that time step. The results presented take this into account, assuming that at least 10 interpolation operations happen per time step. In other words, 10% of

particle sorting time is included in results that present speed-up over CPU in Figure 4.

A method for 10x – Zhao's Method

A better implementation, by Paul Zhao of CPFD Software, uses 8 *threads* per cell to speed calculations. Particle interpolation values are calculated within the interpolation *kernel*. CUDA threads are allocated on a per cell basis with *blockDim(8, 16, 1)*. 8 *threads* per cell with 1 per particle, with each *block* processing 16 cells. Each *thread* handles 1/8th of the particles in a cell using its own *shared memory* to track the 8 interpolated values for particle area, mass and volume. Shared memory totals to 24,576 bytes per CUDA block and 1526 bytes / cell. Gathering the values requires writing to *shared memory* per particle, followed by 3 folding additions to tally all interpolated values for a cell. Finally, one thread (from each cell) writes the 8 interpolated values for each of the 3 particle properties to *global memory* for further collection and reduction in a separate *kernel*, which is included in timings results.

Pseudo-code for Zhao's Method

```
// Per cell, thread parallel in y
// tX denotes threads in x dim. of block
// tY denotes threads in y dim. of block
sip = firstSortedParticleIndex(cell)
eip = lastSortedParticleIndex(cell)
for(ip = sip + tX; ip < eip; ip += 8) {
  foreach(r in {0..7}) {
    // Sum to shared memory.
    s_a[tY][r] += InterpArea(ip, r)
    // Again for mass and volume.
  }
}
__syncthreads();
// Reduce shared memory (3 folds).
foreach(f in {4, 2, 1}) {
  // thread parallel in x
  if(tX < f) {
    foreach(r in {0..7}) {
      // Combine shared memory
      s_a[tY][r] += s_a[tY+f][r]
      // Again for mass and volume.
    }
    __syncthreads();
  }
}
// Write values to global cell memory
```

Corners Method

To improve on Zhao's method, *threads* are allocated for 8 supporting cells, a.k.a. Corners, with *blockDim(32,8,1)*. Each of the 8 *threads* work on a single particle and 4 groups of 8 *threads* are used per grid cell, each group processing 1/4th of the particles in that cell. *CUDA blocks* each process 8 cells. Several advantages then arise. The first, *register memory* is reduced. Each *thread* only tracks 3 values rather than 24. This then also reduces *shared memory* requirements because values to be gathered are more focused. *Shared memory* usage is important to note due to the effect on multiprocessor occupancy, which tends to indicate overall performance. Each *CUDA block* uses 6144 bytes of *shared memory*, which is 768 bytes per cell. Second, when gathering the results, only 2 folding additions are needed and reduction is synchronized at the *warp* level, which performs better than synchronization between multiple *warps*. Third, *global write* concurrency is increased by using 8 threads to write the 8 interpolated values for each of the 3 particle properties.

Pseudo-code for Corners Method

```
// Per cell, thread parallel in x & y
// tX denotes threads in x dim. of block
// tY denotes threads in y dim. of block
sip = firstSortedParticleIndex(cell)
eip = lastSortedParticleIndex(cell)
bgn = tX / 8
stp = blockDim.x / 8
for(ip = sip + bgn; ip < eip; ip += stp) {
  // Sum to thread local memory
  l_area += InterpArea(ip, r)
  // Again for mass and volume.
}

i = tX + blockDim.x*tY

// Write to shared memory.
s_a[i] = l_area // mass and volume

// Reduce shared memory (2 folds).
foreach(f in {16, 8}) {
  if(tX < f) {
    s_a[i] += s_a[i+f]
    // Again for mass and volume.
  }
}
// Write values to global cell memory
```

RESULTS

Results obtained were sufficient to focus on other routines for acceleration. In typical end-user applications, GPU acceleration exhibits a 10x speed-up over the existing implementation executed serially on a CPU with level 2 optimization. Zhao's method gives 10x speed-up for typical commercial end-user usage. The Corners method significantly increases acceleration in the lower range of particle counts and overall performs better than Zhao's method on the particle counts tested.

A simple CPU comparison is used due to development time constraints, to compare against existing end-user experience in the released product and to provide a common point of comparison for future accelerations.

Below are the raw numbers followed by easier to digest graphs, each with a short discussion.

Millions of Particles	(M1) CPU	(M1) GPU Zhao	(M2) GPU Zhao	GPU sort	(M2) GPU Corners
3.8	187.2	16.1	17.5	11.5	10.1
5.1	184.6	19.7	22.5	14.0	12.9
5.9	213.0	21.7	25.4	16.0	14.5
7.0	258.5	23.3	28.3	17.5	16.8
9.1	310.7	28.3	33.7	22.5	22.2
10.2	341.4	30.1	36.9	27.5	25.1
11.7	390.4	33.0	40.0	28.0	29.6

Table 1: Particle-to-Grid interpolation timings. All times are in milliseconds.

Multiple machines were used for the comparison as a result of hardware configuration, denoted by (M1) and (M2). Machine 1 uses an Intel i7-4770 @ 3.5 GHz CPU and a NVIDIA GeForce GTX Titan GPU (with double-precision on). Machine 2 uses an Intel i7-X980 @ 3.3 GHZ CPU and a NVIDIA Tesla K40c GPU. All CPU timings use the faster processor of M1. Results come from a typical end-user problem with complicated internal geometry with 330,000 grid cells and much larger particle counts. The simulation is isothermal and non-reacting.

In Figure 3, the Corners method is plotted against the Zhao method. The difference between the methods as run on Machine 2 are used to correlate the CPU interpolation to the Corners method.

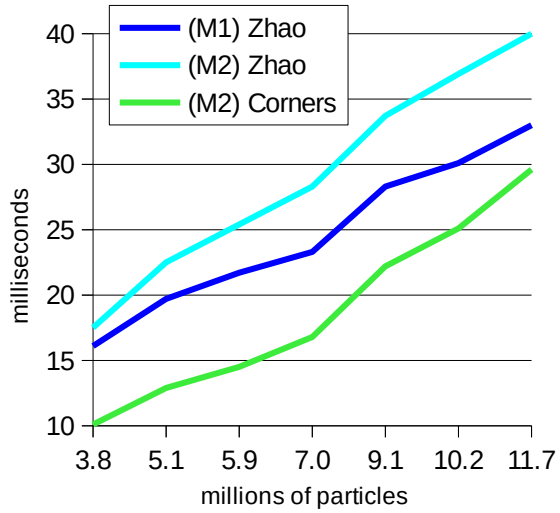

Figure 3: Particle-to-Grid interpolation timings of CUDA kernels using Zhao's method and the Corners method.

In Figure 4, the final results show that Zhao's method is generally 10 times faster and the Corners method should consistently be 16 times faster than the baseline CPU for Particle-to-Grid interpolation. Sort times are not ignored; speed-up was calculated using the following:

$$SpeedUp = \frac{CPUTime}{MethodTime + 0.1 * sort}$$

Sort times on the GPU differed little between the Titan and Tesla, hence a single column in Table 1. Another less interesting comparison is between the sort times and the Corners method timings, which are fairly close together. This was an unexpected surprise that interpolating 3 double-precision values for all particles to the grid might take less time than sorting those particles.

CONCLUSION

Parallelization of any commercial code base is daunting. During our efforts, profiling code provided keen insight into the next course of action, which is very important because parallelization does not happen overnight -even with directives. Barracuda VR 17, which was released in Q2 of 2015, has improved acceleration for several more routines, reducing total wall time and further highlighting the need for a top notch parallel implementation of the Particle-to-Grid interpolation. Pair this with the highly coupled Lagrangian and Eulerian phase and any possible algorithmic improvement to interpolation results in a noticeable improvement in performance.

The Corners method for interpolating Lagrangian values to a Cartesian grid is presented. The speed up over a basic CPU implementation is consistently 14 times faster.

Speculation

To further improve the Corners method, increasing *global write* concurrency by spreading work to idle *threads* may be possible. Using *warp* specific operations for quickly swapping values between *threads* in a *warp* could speed calculation by eliminating the need to use *shared memory*.

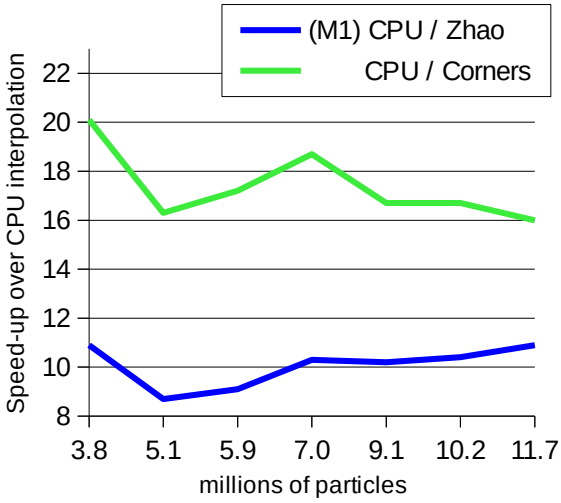

Figure 4: Observed speed-ups for Zhao's methods and Corners method both include 10% of the time needed to sort particles included.

ACKNOWLEDGMENTS

-- Partial funding of work embodied in the paper was supplied by U.S. Department of Energy National Energy Technology Laboratory (NETL).
-- Dr. Chris Guenther at NETL for support of the research and development needed for this work.
-- Stan Posey at NVIDIA for promoting our work via the latest hardware and via other technical resources.

REFERENCES

ANDREWS M. J. and O'ROURKE P. J., (1996) "The multiphase particle-in-cell (MP-PIC) method for dense particle flow" *Int. J. Multiphase Flow*, **22**, 379-402.

SNIDER D. M., (2001) "An Incompressible three dimensional multiphase particle-in-cell model for dense particle flows" *J. of Computational Physics,* **170**, 523-549.

SNIDER D.M., CLARK S.M. and O'ROURKE P.J., (2011) "Eulerian–Lagrangian method for three-dimensional thermal reacting flow with application to coal gasifiers" *Chemical Engineering Science*, **66**, 1285–1295.

STANTCHEV G., DORLAND W. and GUMEROV N., (2008) "Fast Parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU" *J. of Parallel Distributed Computing,* **68**, 1339-1349.

NICKOLLS J., BUCK I., GARLAND M. and SKADRON K., (2008) "Scalable Parallel Programming with CUDA" *ACM Queue*, vol. 6 no. 2, 40.

HOBEROCK J., and BELL N., "Thrust: A parallel template library" 2011. Version 1.4.0

NVIDIA "CUDA C Programming Guide" July 2015 http://docs.nvidia.com/cuda/cuda-c-programming-guide