DOE SBIR Phase-1 Report
on
Hybrid CPU-GPU Parallel Development of the Eulerian-Lagrangian Barracuda Multiphase Program

CPFD Software, LLC
10899 Montgomery Blvd NE, Suite A, Albuquerque, NM 87111
Principal Investigator: Dr. Dale M. Snider
Topic 23, Subtopic d

February 28, 2011

Table of Contents

Summary 3
Significance of the Problem and Technical Approach 4
Anticipated Public Benefits 5
Overview Barracuda Eulerian-Lagrangian Code 5
Technical Objective and Approach 6
Results from Phase-1 7
CPU Acceleration 7
1. Intel Compile 7
2. Inter-Procedural Optimization 8
3. Task-level Parallelism 8
4. Other Optimizations 8
5. Summary of CPU Acceleration 8
Code Restructuring 9
GPU Acceleration 10
1. Choosing a Function to Accelerate 10
2. Starting Codebase 11
3. Basic Port 12
4. Atomic Operations 12
5. Blockwise Synchronization 13
6. Collision-Free, Cell-Wise Algorithm 14
Result comparison program 18
Documentation and Code Cleanup 19
Plan for Phase-2 19

DOE SBIR Phase-1 Report
on
Hybrid CPU-GPU Parallel Development of the Eulerian-Lagrangian Barracuda Multiphase Program

CPFD Software, LLC
10899 Montgomery Blvd NE, Suite A, Albuquerque, NM 87111
Principal Investigator: Dr. Dale M. Snider
Topic 23, Subtopic d

February 28, 2011

This report does not contain any proprietary data or information

Summary

This report gives the result from the Phase-1 work on demonstrating greater than 10x speedup of the Barracuda computer program using parallel methods and GPU processors (General-Purpose Graphics Processing Unit or Graphics Processing Unit). Phase-1 demonstrated a 12x speedup on a typical Barracuda function using the GPU processor. The problem test case used about 5 million particles and 250,000 Eulerian grid cells. The relative speedup, compared to a single CPU, increases with increased number of particles giving greater than 12x speedup. Phase-1 work provided a path for reformatting data structure modifications to give good parallel performance while keeping a friendly environment for new physics development and code maintenance. The implementation of data structure changes will be in Phase-2. Phase-1 laid the ground work for the complete parallelization of Barracuda in Phase-2, with the caveat that implemented computer practices for parallel programming done in Phase-1 gives immediate speedup in the current Barracuda serial running code.

The Phase-1 tasks were completed successfully laying the frame work for Phase-2. The detailed results of Phase-1 are within this document. In general, the speedup of one function would be expected to be higher than the speedup of the entire code because of I/O functions and communication between the algorithms. However, because one of the most difficult Barracuda algorithms was parallelized in Phase-1 and because advanced parallelization methods and proposed parallelization optimization techniques identified in Phase-1 will be used in Phase-2, an overall Barracuda code speedup (relative to a single CPU) is expected to be greater than 10x. This means that a job which takes 30 days to complete will be done in 3 days.

Tasks completed in Phase-1 are:

Task 1: Profile the entire Barracuda code and select which subroutines are to be parallelized (See Section *Choosing a Function to Accelerate*)

Task 2: Select a GPU consultant company and jointly parallelize subroutines (CPFD chose the small business EMPhotonics for the Phase-1 the technical partner. See *Section Technical Objective and Approach*)

Task 3: Integrate parallel subroutines into Barracuda (See Section *Results from Phase-1* and its subsections)

Task 4: Testing, refinement, and optimization of parallel methodology (See Section *Results from Phase-1* and Section *Result Comparison Program*)

Task 5: Integrate Phase-1 parallel subroutines into Barracuda and release (See Section *Results from Phase-1* and its subsections)

Task 6: Roadmap of Phase-2 (See Section *Plan for Phase-2*)

With the completion of Phase 1 we have the base understanding to completely parallelize Barracuda. An overview of the work to move Barracuda to a parallelized code is given in *Plan for Phase-2.*

Significance of the Problem and Technical Approach

Phase-1 is the first step in converting the Lagrangian-Eulerian Barracuda® computational fluid dynamics software package from a serial program to a parallel program. The parallel conversion will be a hybrid solution which allows parallelization on a multiple core CPU computer and using graphics processing units (GPUs). This approach provides a robust parallel environment that will scale with increased number of computational units. The hybrid CPU-GPU approach has several orders of magnitude higher performance than a conventional CPU in certain applications. The complete work at the end of Phase-2 is expected to yield a minimum performance increase in the Barracuda program of an order of magnitude compared to a single CPU (Farber, 2008).

The complex three-dimensional solution provided by Barracuda can take days to weeks to reach a quasi-steady solution for industrial size problems. To provide more versatility and quicker solution turn-around time for current size problems and increased fidelity for future problems, the logical choice is a parallel computing environment. A parallel solution on a traditional computer with multiple general purpose CPUs with shared memory is limited by the number of cores in a single CPU (currently 6 cores) and the general complex instructions used by the general purpose CPU. Going to a larger number of CPUs requires alternative approaches than using the traditional computer cluster. The traditional computer-cluster is unwieldy, has severe bandwidth issues, and is difficult to run and costly to own and operate. The newer method of parallelization is to employ GPUs for computation giving speedups of an order of magnitude or larger. The combination of GPU and multi-core computer (hybrid) approach will be implemented in this work. The GPU has the potential of

orders-of-magnitude increase in computation speed without any loss in fidelity of results. GPUs are relatively inexpensive commodity processing units, and can be programmed by high level API (application program interface) directly from the C programming language.

A major challenge in the parallelization of Barracuda is the efficient and continuous providing of vector data to the vast number of process units. This will require expertise in parallelization techniques and restructuring of Barracuda data storage format to achieve the best performance. This work can also be implemented in such a manner to ensure all future Barracuda fluid-particle physics development will fit into the parallel-environment. Our technical approach to parallel development will be extensible and scalable as new computing hardware becomes available, e.g. multiple GPU cards.

Anticipated Public Benefits

The Barracuda math-based computer program is a new generation software which provides complex three-dimensional solutions to fluid-solid flow. Barracuda provides solutions to the three-dimensional Navier-Stokes equation tightly coupled to the solid phase, the three-dimension solution of particle phase with coupling to gas phase, solutions to the energy equations for the fluid phase and for the solid phase, solutions to gas chemistry, solid chemistry, and various physics models. Through solution of all the physics, the behavior of chemical reactors, petrochemical reactors, silicon reactors, cyclones, etc., can be accurately calculated. With understanding, small and large gains are made on operation, maintenance, upgrades and understanding commercial plants. Even small performance gains in reactors or extended operation cycles directly relate to significant monetary success. Solution of the complex physics in Barracuda requires significant computer 'horsepower'. For large commercial units, a calculation can take 30 days to reach a quasi-steady condition. While the results are invaluable to plant-personnel, the results usually raise numerous questions of what is a better arrangement of reactor hardware or what can be expected with different flows, or a host of other questions. The challenge is to provide Barracuda solutions in a few days rather than several weeks. From the experience of others in math-based solutions, one or two orders of magnitude in computation speedup is realizable (Farber, 2008). A calculation which runs in 30 days can be done in less than 3 days. The ability to put science from a detailed, complex Barracuda solution behind day to day decisions in a chemical plant, or power plant or petrochemical plant will increase engineering productivity. This ability to move from art to science in decision making can be done on inexpensive commodity process units, and, for small companies, this may be a game changer.

Overview Barracuda Eulerian-Lagrangian Code

The Barracuda program is an Eulerian-Lagrangian method for calculating fluid particle flows. The software has been employed for solving a wide variety of problems in the chemical and petrochemical industry. The fluid-granular flow is predicted with a computational particle fluid dynamic (CPFD®) numerical method [Snider 2001]. Continuum or fluid models (Eulerian

reference frame) readily allow modeling of forces using spatial gradients of properties [Batchelor 1988, Gidaspow 1986]. However, modeling a distribution of types and sizes of particles complicates the continuum formulation because separate continuity and momentum equations must be solved for each size and type [Gidaspow 1986, Risk 1993]. While a continuous-fluid description of the solids phase has application in some solid-fluid flow regimes it is inaccurate in others. For dilute solid flows, closure models based on the assumption of high collision frequencies are questionable. In addition, the non-linear behavior of some solid flows is difficult to model with a Navier-Stokes type momentum equation. The Lagrangian or material description for the particle phase allows economical solution for flows with a wide range of particle types, sizes, shapes and velocities [O'Rourke 1981] and has no numerical diffusion associated with an advection operator. The CPFD method uses the MP-PIC scheme [Andrews and O'Rourke 1996, Snider, et al 1998, Snider 2001] for the motion of particles. The calculation method uses features from the Eulerian method and features from the material or Lagrangian method. The MP-PIC method models the fluid as a continuum and models the particles as discrete entities (material description). The MP-PIC method models enduring collision force on each particle as a spatial gradient. Dense particle flow collisions are modeled by a BGK-type collision model [O'Rourke and Snider 2010]. The body of this work is included in the Barracuda® commercial software.

All calculations are solved in three dimensions. The continuum phase and the discrete particle phase are tightly coupled. This requires continuum information to be mapped to particles and particle information to be mapped to the grid. The Eulerian conservation equations are calculated as a set of implicit sparse matrices tightly coupled to the Lagrangian phase.

Technical Objective and Approach

The objective of this work is to parallelize the Barracuda Eulerian-Lagrangian math-based program in the GPU-CPU (hybrid) parallel environment. The expected outcome of this work is the Barracuda program will run at least one-order of magnitude (10 times) faster than in the serial mode.

The migration of a serial programming code to a parallel code is always unique to the software program being parallelized. The computer science of parallelization requires unique talents. CPFD Software recognized the need for computer science experts to participate in the parallelization work. CPFD teamed with EMPhotonics (EMP) to assist in the moving to a parallelized code in Phase-1. EMP is a small business, on the same size as CPFD Software, and their staff is recognized as experts in parallelization by NVIDIA Corp. The computer scientists of EMP and the research engineers of CPFD Software give a strong team for parallelizing the Barracuda program.

Results from Phase-1

The work done in Phase-1 was beyond simply porting one function to the GPU, because the more important goal was the moving of the entire Barracuda code to a parallel environment which happens in Phase-2. The tasks to be done in the Phase-1 proposal are met and provide a base for the Phase-2 effort presented below.

The section of code chosen for parallelizing on the GPU for Phase-1 was one that consumed considerable computation time and one of the most difficult Barracuda functions to parallelize. The function chosen was the mapping of Lagrangian properties to the Eulerian grid. The difficulty arises because particles can move independent of the grid, changing their cell location during the transient, and particles in a given cell can be anywhere in particle memory (without a sort). This gives poor data caching, limiting piping data through parallel GPUs, and increases expensive memory accesses. An even bigger challenge with the chosen function is a race condition. A race condition occurs when many particles are simultaneously trying to update the same grid variable (same memory location). Phase-1 addressed these issues and others and the results are given below. The Barracuda function was parallelized to get the 12x speed gain.

CPU Acceleration

CPU Acceleration was the first step in Phase-1 because these optimizations would become available to all users of the codebase (not just those with GPUs) and within a much shorter time frame than would have been available with the GPU acceleration. Additionally, the code optimizations will complement the GPU solver, allowing the code to perform at a high level by coupling two effective solvers in a hybrid fashion. We attempted a number of CPU techniques, and will describe the results from each below.

1. Intel Compiler

Intel has provided a world-class compiler for its processors for many decades. Due to the Intel compiler's development team having full access to the architectural details of its own company's processors, they are able to generate code that can significantly outperform competing compilers on Intel processors. It is also among the best optimizing compilers for non-Intel processors, such as AMD. The Intel compiler often bests other compilers by 10% or more, and is often the best choice for high-performance numeric applications.

As such, we compiled the Barracuda code using the Intel compiler. There were minor code incompatibilities which required working around, and many warnings were generated. One of the warnings pointed to a minor defect in the source, which was corrected. Once we were correctly compiling, we observed performance gain of 15% beyond the code generated with the GCC compiler. CPFD has since been putting effort into eliminating *all* of the warnings generated by Intel, with an eye towards overall code improvement by clean compilation at the strictest warning level.

2. Inter-Procedural Optimization

Inter-procedural optimization (IPO) is an advanced compiler feature that allows a compiler to see beyond the traditional boundaries of the software compilation model to optimize more effectively than it would otherwise. In a traditional compiler model, each source code file is analyzed independently, leading to each file being optimized independently. The traditional linking step combines these individual optimized objects together into an executable program. With IPO, the compiler analyzes and optimizes all files at once, allowing it to optimize with information not available in a single-file optimization. A good example is that IPO can inline functions residing in different source files, where traditional optimization and linking cannot. Compiling Barracuda using IPO gives a 5% speedup in overall run time.

Task-level Parallelism

Task-level parallelism refers to running different sections of code at the same time. This type of development can be labor intensive, as it requires an analysis by hand to determine which operations/tasks can be run in parallel without leading to data hazards, and so we completed only a partial proof of concept for these principles.

We experimented with two toolkits to achieve task-level parallelism: OpenMP and Intel's Threading Building Blocks (TBB) library. Each of these toolkits has their advantages but for this project we found TBB to be more effective and easier to use for development. One advantage of TBB is that it provides more direct parallelism control to the programmer than OpenMP, where the parallelism is largely implicit. We experimented with using TBB to run different parts of the codebase concurrently and found that even with the few functions we overlapped (parallelized), we achieved a 5% gain in performance.

Task-level parallelism will become increasingly important as the full codebase is ported to the GPU. The reason for this is that while the GPU is a computational powerhouse, it does not excel at every function. For some of these functions it is desirable to utilize the CPU, while leaving the GPU to more parallel and computationally intensive problems. This is often a beneficial scenario as further gains can be achieved as the CPU and GPU are used concurrently, resulting in a higher total system utilization and correspondingly a greater performance. This is the definition of the "hybrid" GPU computing model.

3. Other Optimizations

To complete this task, we implemented several common software optimizations such as loop reordering, link-time optimizations, and strength reduction. In total these optimizations achieved a 10% speedup beyond those realized by the efforts described above.

4. Summary of CPU Acceleration

In summary, the CPU acceleration, in this short phase of the project, yielded a

30% gain in performance for the entire codebase. As shown in Fig. 1, a Barracuda job completing in 7 days would complete in 5 days. This gain can be delivered to end-users today and will lead to increasingly larger gains as the second phase of the project is completed.

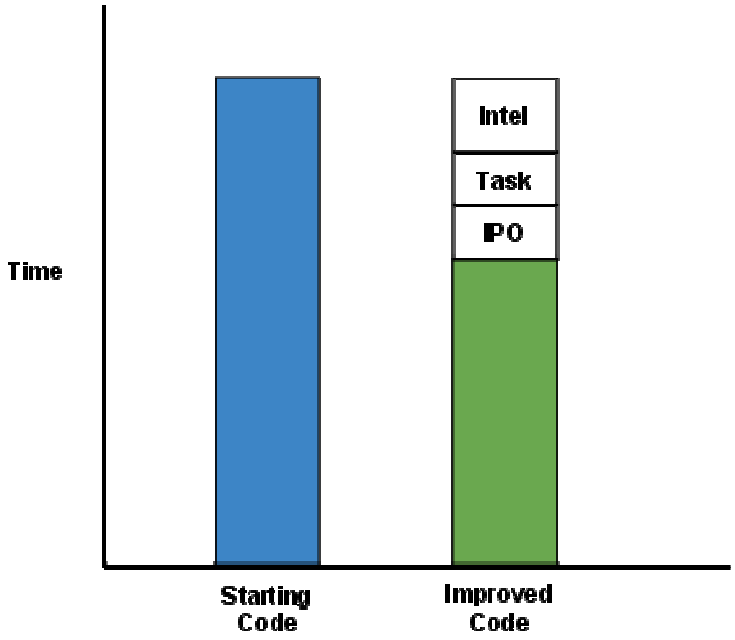


Figure 1 - Performance gains were achieved by adopting the Intel Compiler, using task-level parallelism, and inter-procedural optimization.

Code Restructuring

During the analysis of the Barracuda code, EM Photonics made numerous recommendations to CPFD on how to improve the quality, safety, and overall structure of the codebase. Some of these items were implemented in Phase-1, and progress in this area will continue throughout Phase-2.

One such example is the restructuring of the data storage. It was recognized from the onset that the current data format was not friendly to parallelization. The current array of structures limits the fast transfer of data to the GPU, and data are not presented in a cache friendly format to the GPU processors. Data are contained in several macro data structures whose address is passed throughout the code - in this format, they behave similarly to global variables. The Barracuda “block” data structure is a good example, weighing in at over 500 bytes. The blocks are stored in a list with the block address passed pervasively throughout the code. Any given function that references the block-structure tends use only a few variables from it. This large structure with a pointer reference is OK on a single CPU where all

data are available in one shared memory. However, when data reside on different processors, such as the CPU and GPU, with different memory, the entire block of data must be transferred between processors to access a few variables from the structure. For efficient data transfer and access, in a GPU environment, the current large data structures need restructuring in two ways:

1. Convert the majority of *structs* from array-of-*structs* to *struct*-of-arrays. The latter is more cache friendly.

2. Reduce the number of elements in any given structure. Breaking structures into smaller pieces allows for passing of blocks of only the relevant data between CPU and GPU. From a leaner new data structures, it becomes clear which data are used where, and this enables code analysis for task-level parallelism implementation.

In a similar vein, *const-correcting* (standard coding practice) will be implemented to make it obvious which data values are inputs and which are outputs to a function. This also improves code safety and reliability. Current mufti-dimension arrays will be made 1-dimensional arrays with a pointer to the multi-dimensional data in the array. The array memory allocation will use the C++ vector type which avoids memory leaks, allocates contiguous memory, and reduces the possibility of coding errors.

GPU Acceleration

In Phase 1, we completed a proof of concept of GPU acceleration by parallelizing one function in Barracuda. We took one of the more computationally expensive and complicated functions and accelerated it to achieve a speedup of 10-15x over its CPU counterpart. The design took several iterations, but the final result provides a template for many of the other functions in the codebase that operate similarly to the one accelerated, laying out a path for acceleration of similar algorithms in Phase-2.

The work described in the following subsections represents the majority of the Phase-1 effort, and completes the work described in Task 3 of the Phase-1 proposal. Task 5 is also largely completed in the below described work, as the parallel implementation was integrated cleanly to the codebase as it was added. The GPU integration into Barracuda was implemented with an inefficient superfluous GPU data transfers. The inefficient data transfer was done because it allowed quick testing of parallelization options and techniques, and because direct transfer of Barracuda data to the GPU requires extensive data structure changes and code wide changes. The inefficient data transfer used in Phase-1 will be removed in Phase-2.

1. Choosing a Function to Accelerate

The first step towards accelerating the codebase on the GPU was to choose a section of code that would become the proof of concept for full acceleration of the

codebase. With this in mind, we chose a function that took a significant amount of the program's run-time and represented one of the hardest problems in transitioning the codebase to the GPU. Figure 2 shows a few of the functions considered for moving to the GPU. The function *CollisionParameters*, was chosen because it has loops over all of the particles (~5,000,000 particles) mapping particle data to a grid cell (~250,000 cells). Another factor that drove the choice of this problem was the similarity of this snippet of Barracuda to other sections of the code. There are on the order of 50 similar particle loops in Barracuda. With this in mind, an accelerated implementation of this section of code created a template for porting similar code to the GPU.

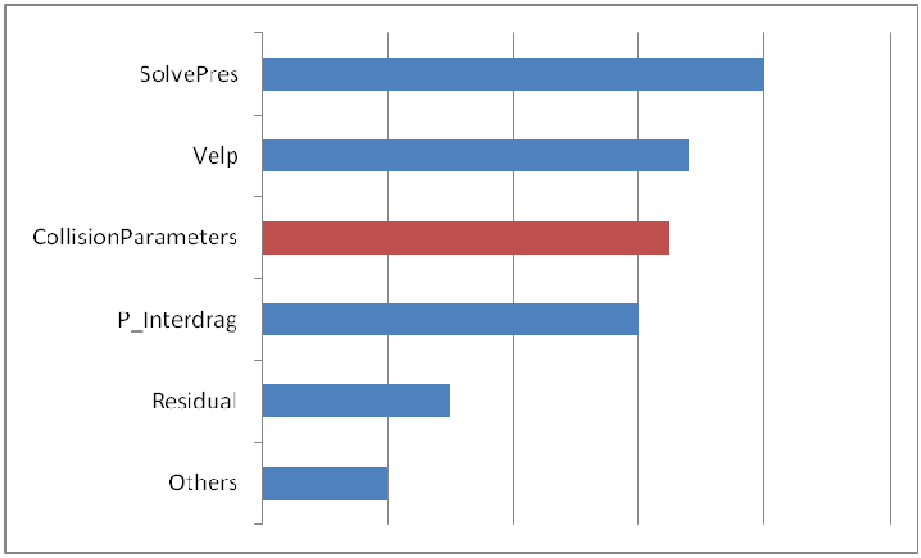


Figure 2. CollisionParameters was an ideal choice for acceleration because it took up a large percentage of the run-time and was representative of many other costly sections of code.

2. Starting Codebase

The starting codebase algorithm mapped particle properties to the grid in a serial fashion. Particles were organized into blocks, and although the particles were somewhat sorted in practice, there was no guarantee that a given particle would be spatially located anywhere near any of the particles in its set (see Fig. 3). What this resulted in was a code that was updating non-deterministic locations on a grid in an essentially stochastic fashion. This randomness prevented the code from taking advantage of the computer's memory hierarchy, as the caching system, meant to optimize nearby memory transactions, fell down as the memory transactions were scattered across the entire grid. The cache was essentially disabled.

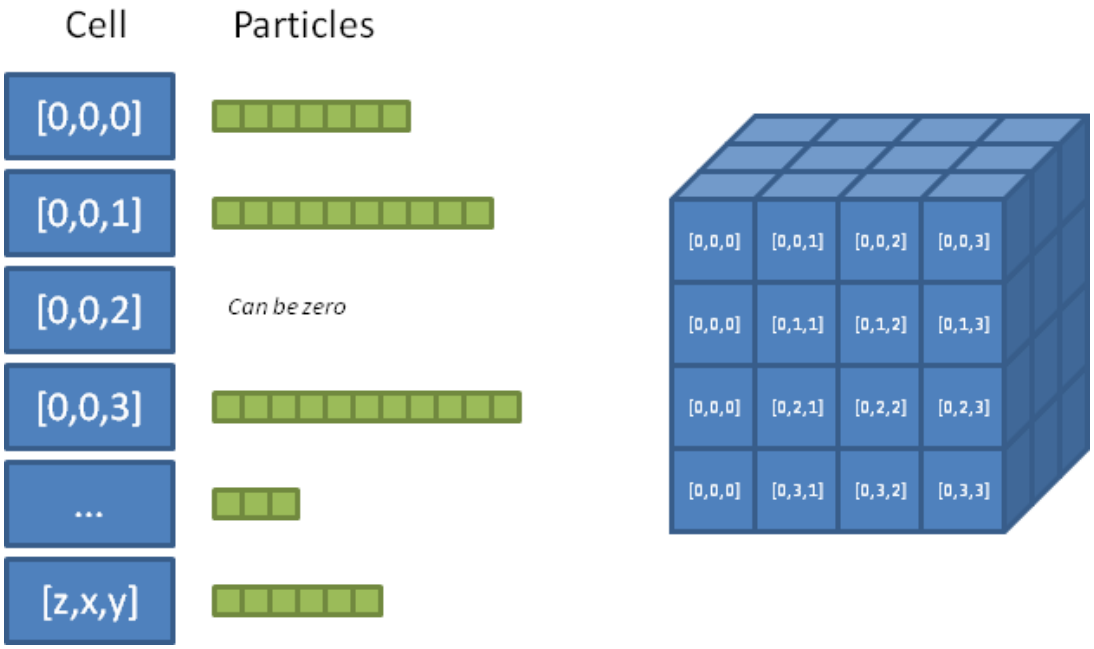


Figure 3. Particles are located within a cell on the grid. Because the code models a stochastic process, there is a non-deterministic number of particles in each particular cell.

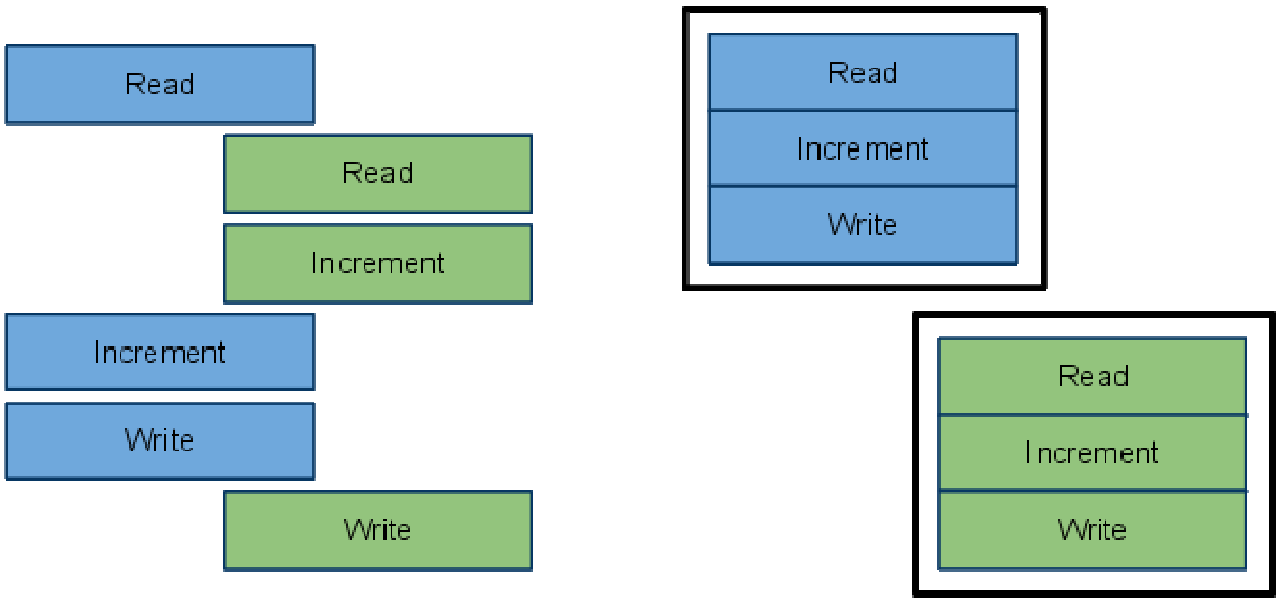
3. Basic Port

The first step in GPU acceleration was to complete a basic port to the GPU. The first step of this task is to set up data structures on the GPU and write functions that transfer data to and from the GPU. Once complete, we created a CUDA kernel that operates on this data that resides on the GPU. As is typical of most GPU development, in this early part of the project we created a kernel that was quite naive, as the main goal of the task was to include the GPU in the program flow, albeit with very low performance. After implementing this basic kernel, we took the first steps towards its correctness by using atomic operations to handle data races between different processing units.

4. Atomic Operations

During the course of this work we explored the usage of atomic operations for implementing various sections of code in the GPU solver. Atomic operations are traditionally hardware-intrinsic functions for reading, updating, and writing a location in memory with one uninterrupted operation. Because none of the steps in this operation can be broken, this avoids any non-deterministic ordering of operations between different parallel processing units that may attempt to operate on this memory location at the same time. CUDA GPUs have supported atomic operations since the earliest models, yet only recently were built-in, double-precision operations included in the

Fermi GPU line. This capability was highly desired in this project, as many of the intensive kernels used double-precision arithmetic.



Without synchronization, read, update, and write steps may be interleaved in unexpected orders, leading to four possible results from these two threads.

With an atomic operation, read, update, and write are joined into one unbreakable operation. There is only one possible outcome for this code.

Figure 4. Example of race condition

While studying this problem, we determined that atomic operations in global memory were far too costly for the particle accumulation loops. The reason for this is that atomics are traditionally meant to handle memory collisions between a small number of threads, on the order of less than 10. Within the particle code, however, a given cell could have several hundred particles located within it, leading to several hundred threads contending for access to one memory location, which led to a significant slowdown in the code. Additionally, different threads writing to scattered locations throughout memory prevented the GPU from achieving any economies of scale in its memory transactions. After completing this phase of the research, we sought to develop a version of the kernel that could avoid the limitations of the global memory and scattered global memory writes.

5. Blockwise Synchronization

Once we determined that atomic operations in global memory were too costly to be used with a non-deterministic number of threads writing to the same location, we experimented with introducing some determinism to the contention by sorting particles into blocks of nearby particles. The theory behind this approach was that instead of using global atomic operations, which were significantly expensive, we could utilize cheaper atomic operations in a shared workspace that is local to each microprocessor. These cheaper operations could minimize the most significant limitation of the first approach to the solver. Additionally, by co-locating particles by tying them to a particle cell, the GPU could achieve coalesced memory writes by writing out these contiguous memory locations in a completely parallel manner. A diagram of this approach is shown in Fig 4.

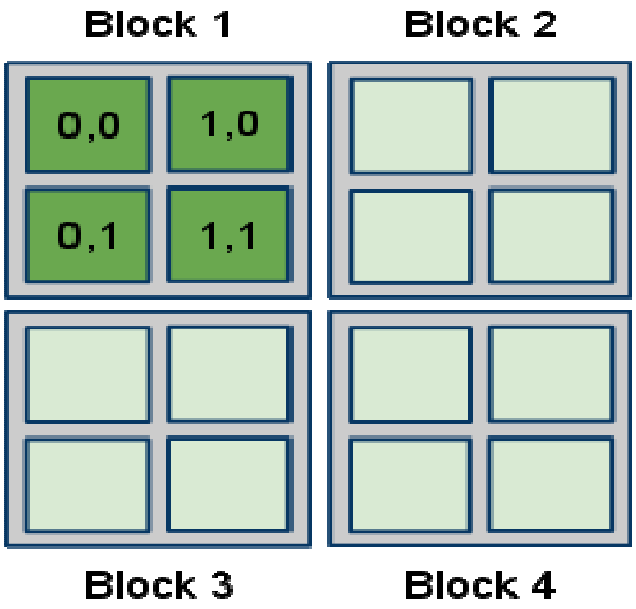


Figure 5. Neighboring cells are combined to form a block. Each of these blocks is processed in parallel, using atomic operations to handle collisions.

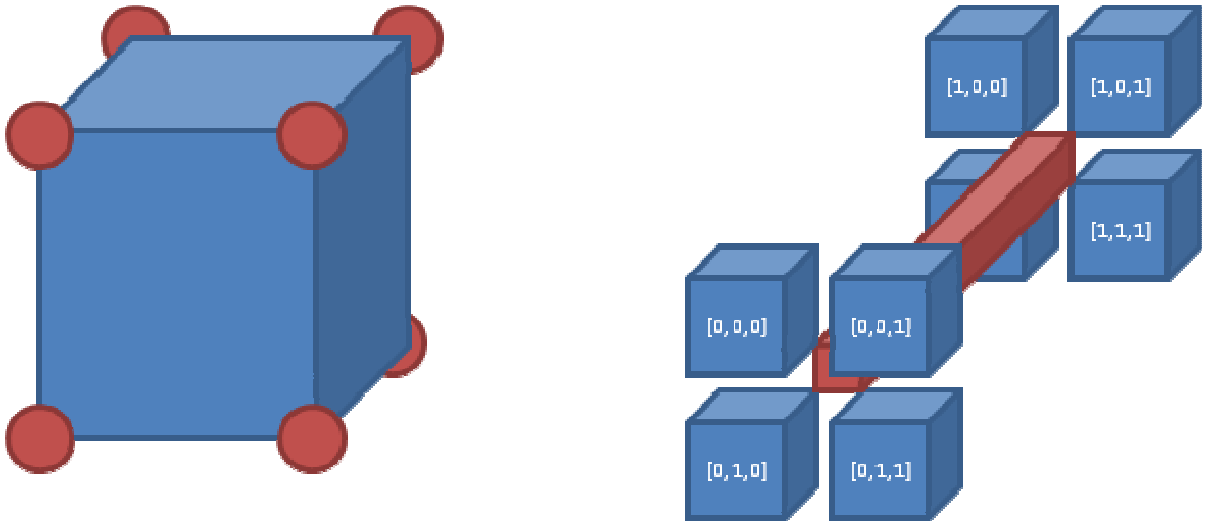
We implemented this approach but found that atomic operations were still far too costly. Although this had a speedup when compared with the first iteration, the contention was far too great to allow the GPU to perform at its full computational potential.

6. Collision-Free, Cell-Wise Algorithm

The implementation above highlighted the need to move away from GPU

atomics for synchronization. Evolving the algorithm implemented above, we decided that rather than processing a block of cells in parallel, we would instead process each cell in parallel. The clear advantage of this approach is that instead of using atomic operations, we could allow each cell to have a private workspace in shared memory for its collision free accumulations. Once all of the threads processed each of their cells, the temporary values for each cell could be reduced in a deterministic and collision-free manner.

The key for processing each cell is to ensure that there are no conflicts between corners for a given cell interpolation. One factor at play, that leads to possible data-collisions, is the interpolation operator in mapping particle properties to the grid. A particle in a cell maps data to 8 cells surrounding the particle. As illustrated in Fig. 7, part of the 8-corner stencil is shared by other particles in a neighbor cell. In parallel computing, there is the possibility that multiple particles can change a cell variable at the same time (step on each other's toes). We need to ensure that contributions from particles in neighboring cells are respected when we calculate the final result.



Each cell is surrounded by 8 corners.

Analogously, each corner is shared by 8 cells.

Figure 6. The 8 point edge stencil for interpolating particle data to grid cells.

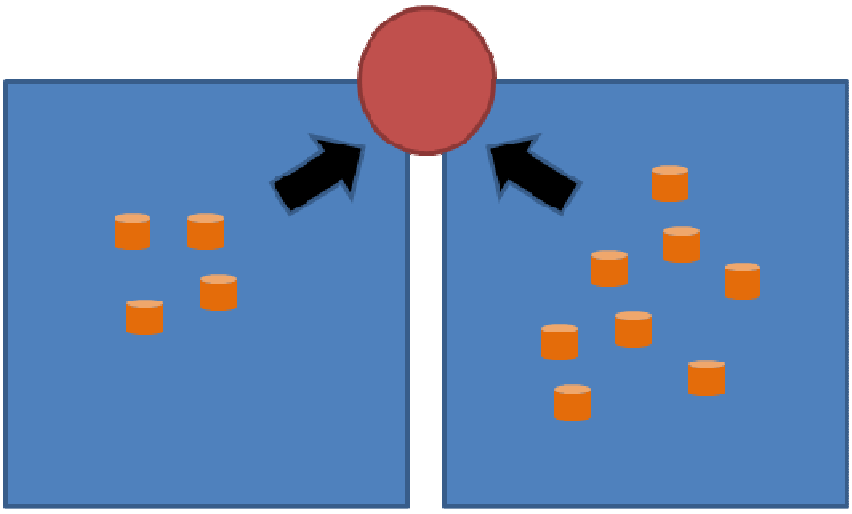


Figure 7 - Particles within a cell contribute to each corner in the cell, and these corners are shared between multiple cells.

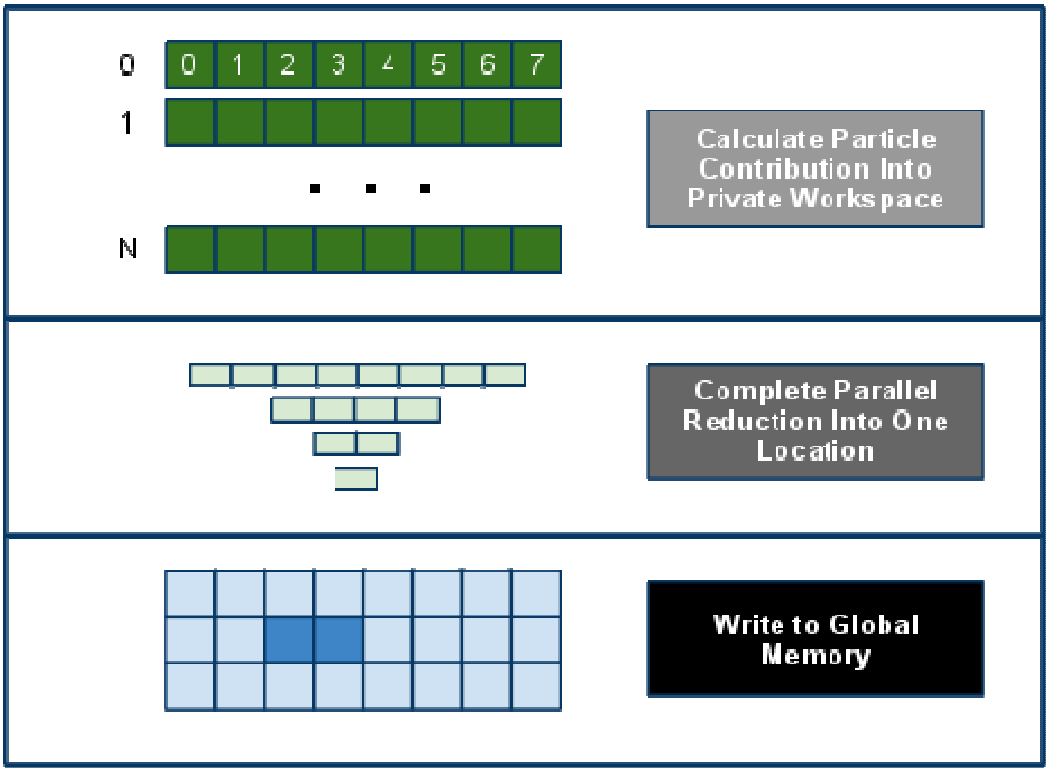


Figure 8 - In the first kernel, each thread completes a three step process that includes looping over all of the particles to which it has been assigned, reducing these particles

in partnership with its neighbors, and finally writing out the final result to global memory.

We handle these dependencies by splitting the processing into two steps, the corner contribution and the corner reduction. Within the kernel, a set of threads looped over all of the particles that were located in its cell, updating a private workspace location in the GPU's shared memory. Once all the particles in a particle cell have been processed, each block combines (reduces) the private workspace locations into one cell. With the reduction complete, the full contribution of the particles to all 8 corners of a particular cell have been calculated, and all that remains is to write these values out to global memory.

The second step of this process is to push the contribution of particles from a given cell to its neighboring cells. The advantage of this approach is that each cell is completely independent and can proceed without need to perform any synchronization with any other cells. A diagram of this process is shown in Fig. 9.

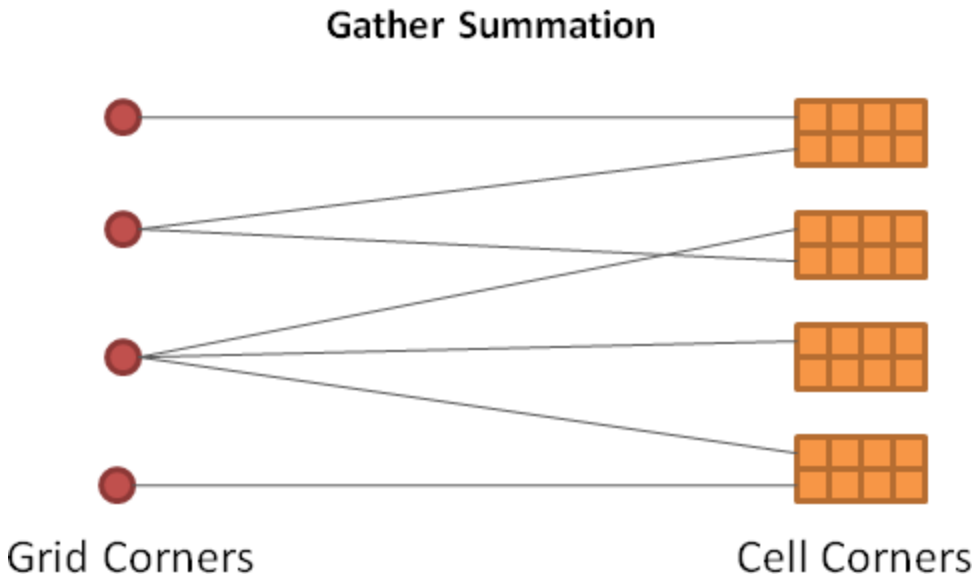


Figure 9 - A given corner is read by multiple cells because this corner is shared between 8 different cells. Each of these reads can be done in parallel because the cell information is kept in a different array than the corner information, leading to no data hazards.

```
 // Neighbor                Corner of neighbor  Offset to neighbor
if( hasLeftFrontBotNeighbor ) m += massCorners[ CORNER(rightBackTop,  x , y , z  ) ];
if( hasLeftFrontTopNeighbor ) m += massCorners[ CORNER(rightBackBot,  x , y , z+1 ) ];
```

```
if( hasLeftBackBotNeighbor  ) m += massCorners[ CORNER(rightFrontTop, x , y+1, z  ) ];
if( hasLeftBackTopNeighbor  ) m += massCorners[ CORNER(rightFrontBot, x , y+1, z+1 ) ];
if( hasRightFrontBotNeighbor) m += massCorners[ CORNER(leftBackTop,  x+1, y , z  ) ];
if( hasRightFrontTopNeighbor) m += massCorners[ CORNER(leftBackBot,  x+1, y , z+1 ) ];
if( hasRightBackBotNeighbor ) m += massCorners[ CORNER(leftFrontTop, x+1, y+1, z  ) ];
if( hasRightBackTopNeighbor ) m += massCorners[ CORNER(leftFrontBot, x+1, y+1, z+1 ) ];
```

Figure 10 - Example of this process represented in code. Each cell looks to the corners of each of its 8 neighbors to determine the total contribution to the given cell. The desired corner mirrors the direction of its neighbor's location; for example the 'Right, Back, Top' corner is read from the 'Left, Front, Bot' neighbor.

Within each of these statements, a given cell will check if it has a neighbor in a specific direction, and if so, will read the contribution of its neighboring cell to the corresponding corner. The reduction step completes the particle-to-grid interpolation algorithm. By avoiding memory contention that the first two iterations of the solver exhibited, we were able to achieve a speedup of 10-15x over the CPU code. Analyzing and benchmarking the speedup completes the work described in Task 4 of the Phase-1 proposal.

Result comparison program

A result comparison utility program was built to assist in testing code-correctness during the development of the parallel code. The purpose of this program was to compare output from two different versions of the code that have run the same test problem. When specified tolerances are exceeded between the two code versions, a diagnostic is printed showing the time and physical quantity out-of-bounds. In this utility program, code errors are easily identified and corrected. See Fig. 9 for an example output.

```
Data files report ======================================
  List of variables
     1  Time
     2  Pressure
     3  Pressure
     4  Fluid x-vel
     5  Fluid y-vel
     6  Fluid z-vel

  Specified 77 data lines were processed.

Comparison report ======================================
  Number of data lines                  : 77
  Number of data columns                : 6
  Last data read from first file        : -2.3179370165e-01
  Last data read from second file       : -2.3179370165e-01
  Total number of data points processed  : 462
  Total number of data points compared   : 448
  ------------------------------------------------------
  Average relative error                     : 1.924034e-08
  Max relative error                         : 7.810188e-06
    Max error location 1st file (line, col)  : 88, 3
    Max error location 2nd file (line, col)  : 88, 3
    Max error variable                       : Pressure
    Variable value from 1st file             :  4.9330562592e+01
    Variable value from 2nd file             :  4.9330947876e+01
  ------------------------------------------------------
  Number of data points with relative error > 1.e-3  : 0
  Number of data points with relative error > 1.e-4  : 0
  Number of data points with relative error > 1.e-5  : 0
  Number of data points with relative error > 1.e-6  : 1
  Number of data points with relative error > 1.e-7  : 3

Comparison result (Equivalence of two data files)=======
      **** WARNING ****
```

Figure 11 - The comparison program compares all of the variables that are collected in a given run and reports which of these variables exceed error thresholds. This provides a means to compare the output of different versions of the solvers.

Documentation and Code Cleanup

Once the accelerated port was complete, we completed a documentation and code-cleanup effort. This effort simplified merging the optimized code with other code modifications that were made while the branch was being implemented. Additionally, this effort served to provide the most solid foundation for the Phase-2 work.

Plan for Phase-2

The acceleration of the Barracuda code in the upcoming Phase-2 effort, was demonstrated in Phase-1. The first step in Phase-2 is reformatting existing data structures to make data caching and data flow compatible with high degree of parallelization will be done. Current data structures will be broken into smaller structures, and structures will be switched from array of structures (AOS) to structure of arrays (SOA). The grouping of data in arrays will

be based on the code architecture, such that the blocks of data which will be moved to and from the GPU will contain only data needed on the GPU. This grouping will also be cache friendly in the hybrid parallelization.

The next step is to define the code which will reside on the GPU, which code is parallelized on the CPU and which code remains serial. Currently there is a natural separation of having the particle algorithms reside on the GPU. It is also known that computation intensive operations, such as the conjugate gradient solver, are candidates for the speed that the GPU gives. A detailed code “call tree” will identify which routine calls other routines, and identify which data are required by which routines. The tree-map will assist in parallelizing Barracuda in Phase-2 by providing an obvious usage of data by function and the interconnection of functions. By knowing such taxonomy, we can more easily identify where algorithms and each piece of data will reside, and identify times when data must be transferred between devices.

Serial code functions will be ported to GPU-kernel functions. The order of migrating Barracuda functions to the GPU will be based on profiling and identifying hot-spots of high computation time. Sections of code which lend themselves to parallelization and have significant computation time will be ported to a GPU-kernel. Other parts of the code, such as i/o functions, input processing, etc., will be left in serial mode. In the parallelization process, less significant code sections which are used by the GPU code sections will need to be ported to the GPU. There are approximately 750 functions in Barracuda. It estimated that 500 of the functions will be ported to GPU or parallelized on the CPU. This effort is expected to represent the top 95% of Barracuda’s run time.

Continual testing is required during the parallelization effort. Parallel programs are notorious for giving inconsistent calculation results, to a large extent from contention when updating a variable from multiple sources (race condition). The parallel code results will be compared to serial code results, as sections of parallel code are completed. Testing will also check that the section of code parallelized gives effective speedup. In Phase 2, testing and validating the parallel code to the serial code will be a significant effort throughout the project. The work will be heavily documented within Barracuda (coding comments), and standard coding practices will be enforced for ease of maintenance and giving an extensible code.

In the Phase-2 effort, the CPU codepath will be kept 100% operational, and it will be possible to compile the solver completely excluding GPU support, or to disable it at runtime. During this time, the CPU codepath will likely see additional improvements from completed Phase-2 tasks, such as from reorganization of data structures for the GPU. In some cases, the CPU code will be overlapped with GPU code in the hybrid mindset to avoid slowing the simultaneous GPU execution path.

At the end of the Phase-2, a GPU-accelerated Barracuda will meet the performance target of an order of magnitude improvement in run time. Future work beyond this Phase-2 may include the use of multi-GPU processing or multi-node (i.e. cluster) processing to extend the performance of the code even further beyond this level.

References

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

G. K. Batchelor, 1988, "A new theory of the instability of a uniform fluidized bed," *J. Fluid Mech*., 193, 75-110

D. Gidaspow, 1986, "Hydrodynamics of fluidization and heat transfer supercomputer modeling," Appl. Mech. Rev. 39, 1-22

P. J. O'Rourke, 1981, *Collective drop effects on vaporizing liquid sprays*, PhD. Thesis, Princeton University

P.J. O'Rourke ans D.M. Snider, 2010, “An improved collision damping time for MP-PIC calculations of dense particle flow with application to polydisperse sedimenting beds and colliding particle jets,” *Chemical Engineering Science*, 65, 6014-6028

M. A. Risk, 1993, "Mathematical modeling of densely loaded, particle laden turbulent flows," *Atomization and Sprays*, 3, 1-27

D.M. Snider, 2001, “An Incompressible three dimensional multiphase particle-in-cell model for dense particle flows”, *Journal of Computational Physics*, 170, 523-549.

D.M. Snider, P. J. O’Rourke, and M. J. Andrews, 1998, “Sediment flow in inclined vessels calculated using multiphase particle-in-cell model for dense particle flow,” *Int. J. Multiphase Flow*, 24, 1359-1282

R. Farber, 2008, “CUDA, Supercomputing for the Masses: Part 1”, *Dr. Dobb's,* http://www.ddj.com/cpp/207200659